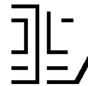
# Topics in

# Computer Organization

(based on old class notes)



**D. Salomon**

**Feb 2008**

(619) 443-6528

dsalomon@csun.edu

http://www.davidsalomon.name/

Modern computer programming is a race between
programmers striving to write bigger and better
idiot-proof programs, and the Universe trying to produce
bigger and better idiots. So far, the Universe is winning.

—Rich Cook

# Preface

The dictionary definition of the term "organizing" is "to put together into an orderly, functional, structured whole." This term applies twice to this text because it tries to "put together into an orderly, functional, structured whole" the main components of computers; the way computers are organized.

More and more people use computers, but to most computer users the computer is a "black box," and writing a program is akin to uttering magical incantations. This text tries to clear up these mysteries. It attempts to give the reader a robust framework to which new knowledge can later be added by more studying. Specifically, the text answers the question "how do computers work?" It describes the main components of the computer and their interrelationships. It includes material on the control unit, memory, and input/output processing, as well as topics such as logic devices, microprocessor components, microprogramming, machine instructions, assembly language and assembler operation, and a survey of the main features of some existing computers.

The title of this text is "Computer Organization." This term refers to a conceptual understanding, rather than an engineering understanding, of the inner workings of computers. A more formal definition of this term is "the logical arrangement of the hardware components and how they are interconnected."

Another important term commonly used with computers is "computer architecture." This term answers the question "how is a computer designed?" rather than "how does the computer work?". An alternative definition of computer architecture is "the components or attributes of the computer that are visible to the assembler language programmer."

As a result of these definitions, features such as the instruction set, the word size, addressing modes, and interrupt handling are architectural. Features such as memory technology, bus organization, and handshaking signals are organizational. A typical example is input/output (abbreviated I/O). The I/O instructions in the instruction set are an architectural feature. The physical implementation of those instructions (by way of a DMA device, a channel, or something else) is an organizational feature.

The distinction between organization and architecture can be traced back to 1964. In that year, IBM released its system/360, the first *family* of computers. The different models in the family had different performances (and prices) and, as a result, different organizations. However, they were upward compatible, which means they had the same architecture. The success of the 360 family encouraged IBM and other manufacturers to design many more families of computers, thereby establishing the distinction between organization and architecture.

Here are a few more general terms used with computers.

- "Software" refers to how the computer is used.
- "Hardware" refers to how the computer is constructed (its physical building blocks).
- The general term "computing" refers to problem solving on computers by means of programming.
- The computer itself is a machine that performs tasks and solve problems by storing and manipulating information.
- Information stored in computers is digital (it consists of discrete digits).

These notes are based on experience gained from teaching computer organization over many years. Much of the material found here was originally included in response to questions and requests from students.

The exercises constitute an important part of the notes and should be worked out! The answers are provided, but should be consulted only as a last resort. Remember what the Dodo said:

> "Why," said the Dodo, "the best way to explain it is to do it."
> —Lewis Carroll (1832–1898)

The text is divided into three parts as follows:

Part I consists of the first six chapters and covers the main concepts of computer organization. Chapter 1 introduces the control unit, instruction fetch and execution, memory, ALU, microprogramming, and interrupts. Chapter 2 is a review of machine instructions and data types. It discusses instruction formats, opcodes, addressing modes, instruction types, floating-point, BCD, and fixed-point numbers, and carry and overflow. Chapter 3 is devoted to the important topic of input/output (I/O). It consists of three parts, the I/O processor, I/O codes, and computer communications. The first part discusses register I/O, memory-mapped I/O, DMA, and other types of I/O processors. The second part introduces the ASCII and Unicode codes, continues with a coverage of reliable (i.e., error-detecting and error-correcting) codes, and of data compression methods, and concludes with a description of important cryptographic methods (secure codes). Part three explains serial I/O and how computers are connected in local-area and wide-area networks.

Microprogramming is the topic of chapter 4. The chapter includes a review of the basic concepts and a detailed example of a simple, microprogrammed computer. Parallel computers are discussed in chapter 5. The discussion concentrates on MIMD computers, but SIMD and data flow computers are also mentioned. The short chapter 6 introduces the reader to the concept of reduced instruction-set computers (RISC).

Part II consists of chapters 7 through 9. This is an introduction to digital devices and computer design. Chapter 7 introduces the main digital devices used in computers. It starts with the logic gates and covers latches, counters, decoders, multiplexors, and other devices. Chapter 8 is an introduction to computer memory. The main types of memory and their internal operations are discussed. The ALU and its circuits is the topic of chapter 9. Digital circuits for integer arithmetic, for shifts, and for comparisons are explained in detail.

Part III consists of appendixes that provide information on the history of computers, Unicode, and CD-ROMs.

> Hating, as I mortally do, all long unnecessary preface, I shall give you good quarter in this, and use no farther apology, than to prepare you for seeing the loose part of my life, wrote with the same liberty that I led it.
> —John Cleland, 1749, *Fanny Hill*

# Contents


**Preface** — i

**1. Introduction** — 1

1 Terminology 1
2 The Control Unit 2
3 The Memory 5
4 Instruction Execution 11
5 CPU Bus Structure 17
6 Microprogramming 18
7 Interrupts 20
8 I/O Interrupts 26
9 Interrupts in a Microprocessor 28
10 The ALU 30

**2. Machine Instructions** — 33

1 Instruction Formats 33
2 The Opcode Size 34
3 Addressing Modes 36
4 The Direct Mode 37
5 The Relative Mode 37
6 The Immediate Mode 38
7 The Index Mode 38
8 The Indirect Mode 38
9 Multilevel or Cascaded Indirect 39
10 Other Addressing Modes 40
11 Instruction Types 41
12 Data Movement Instructions 41
13 Operations 42
14 $N$-Operand Instructions 48
15 Actual Instruction Sets 48
16 The VAX Instruction Set 49
17 The RISC I Instruction Set 52
18 Non-Numeric Data Types 53
19 Numeric Data Types 54
20 Signed Integers 54
21 Floating-Point Numbers 57
22 Fixed-Point Numbers 63
23 Decimal (BCD) Numbers 64
24 Other BCD Codes 65
25 Rational Numbers 67
26 Carry and Overflow 68

**3. Input/Output ——— 71**

| | | |
|---|---|---|
| 1 | The I/O Processor | 71 |
| 2 | Polled I/O | 73 |
| 3 | Interrupt I/O | 75 |
| 4 | DMA | 75 |
| 5 | I/O Channels | 78 |
| 6 | I/O Codes | 79 |
| 7 | ASCII and Other Codes | 80 |
| 8 | Information Theory and Algebraic Coding | 83 |
| 9 | Error-Detecting and Error-Correcting Codes | 84 |
| 10 | Data Compression | 93 |
| 11 | Variable-Size Codes | 94 |
| 12 | Huffman Codes | 95 |
| 13 | Facsimile Compression | 97 |
| 14 | Dictionary-Based Methods | 100 |
| 15 | Approaches to Image Compression | 102 |
| 16 | Secure Codes | 107 |
| 17 | Transposition Ciphers | 114 |
| 18 | Transposition by Turning Template | 116 |
| 19 | Columnar Transposition Cipher | 118 |
| 20 | Steganography | 126 |
| 21 | Computer Communications | 131 |
| 22 | Serial I/O | 131 |
| 23 | Modern Modems | 143 |
| 24 | ISDN and DSL | 144 |
| 25 | T-1, DS-1 and Their Relatives | 145 |
| 26 | Computer Networks | 148 |
| 27 | Internet Organization | 152 |
| 28 | Internet: Physical Layout | 153 |
| 29 | CSUN in the Internet | 160 |
| 30 | ICANN and IANA | 163 |
| 31 | The World Wide Web | 163 |

**4. Microprogramming ——— 167**

| | | |
|---|---|---|
| 1 | Basic Principles | 167 |
| 2 | A Short History of Microprogramming | 168 |
| 3 | The Computer Clock | 169 |
| 4 | An Example Microarchitecture | 169 |
| 5 | The Microinstructions | 170 |
| 6 | Microinstruction Timing | 172 |
| 7 | The Control Path | 175 |
| 8 | The Machine Instructions | 177 |
| 9 | The Microcode | 178 |
| 10 | Final Notes | 184 |
| 11 | A Horizontal Example | 187 |

**5. Parallel Computers** — **189**

| | | |
|---|---|---|
| 1 | Introduction | 189 |
| 2 | Classifying Architectures | 189 |
| 3 | Parallel Processing Methods | 190 |
| 4 | Design Criteria | 190 |
| 5 | The Hypercube | 191 |
| 6 | Array Processors | 193 |
| 7 | Example: MPP | 195 |
| 8 | Example: The Connection Machine | 197 |
| 9 | MIMD Computers | 199 |
| 10 | Parallel Algorithms | 200 |
| 11 | The Intel iPSC/1 | 202 |
| 12 | Vector Processors | 204 |
| 13 | The Von-Neumann Bottleneck | 205 |
| 14 | Associative Computers | 206 |
| 15 | Data Flow Computers | 209 |

**6. Reduced Instruction Set Computers** — **219**

| | | |
|---|---|---|
| 1 | Reduced and Complex Instruction Sets | 219 |
| 2 | A Short History of RISC | 221 |
| 3 | The RISC I Computer | 222 |
| 4 | Conclusions | 226 |

**7. Digital Devices** — **227**

| | | |
|---|---|---|
| 1 | Combinational and Sequential Devices | 228 |
| 2 | Multivibrators | 228 |
| 3 | Counters | 232 |
| 4 | Registers | 236 |
| 5 | Multiplexors | 236 |
| 6 | Decoders | 237 |
| 7 | Encoders | 237 |

**8. The Memory** — **241**

| | | |
|---|---|---|
| 1 | A Glossary of Memory Terms | 241 |
| 2 | Static RAM | 242 |
| 3 | ROM | 243 |
| 4 | PLA | 245 |

**9. The ALU** — **247**

| | | |
|---|---|---|
| 1 | Integer Addition and Subtraction | 247 |
| 2 | The Accumulator | 250 |
| 3 | Integer Multiplication | 252 |
| 4 | Integer Division | 260 |
| 5 | Shifts | 261 |
| 6 | Comparisons | 263 |

**10. Assemblers** — **267**

| | | |
|---|---|---|
| 1 | Introduction | 267 |
| 2 | A Short History of Assemblers | 270 |
| 3 | Types of Assemblers and Loaders | 272 |
| 4 | Assembler Operation | 272 |
| 5 | The Two-Pass Assembler | 276 |
| 6 | The One-Pass Assembler | 280 |
| 7 | Absolute and Relocatable Object Files | 283 |
| 8 | Absolute and Rel. Address Expressions | 286 |
| 9 | Local Labels | 288 |
| 10 | Multiple Location Counters | 289 |
| 11 | Literals | 293 |
| 12 | Attributes of Symbols | 294 |
| 13 | Assembly-Time Errors | 295 |
| 14 | The Symbol Table | 296 |

**11. Macros** 301

| | | |
|---|---|---|
| 1 | Introduction | 301 |
| 2 | Macro Parameters | 304 |
| 3 | Pass 0 | 308 |
| 4 | MDT Organization | 310 |
| 5 | Other Features of Macros | 311 |
| 6 | Nested Macros | 315 |
| 7 | Recursive Macros | 318 |
| 8 | Conditional Assembly | 318 |
| 9 | Nested Macro Definition | 324 |
| 10 | Summary of Pass 0 | 334 |

**A. History of Computers** 337

| | | |
|---|---|---|
| 1 | A Brief History of Computers | 337 |
| 2 | First Generation. Vacuum Tubes | 340 |
| 3 | Second Generation: Transistors | 342 |
| 4 | Third Generation: ICs | 342 |
| 5 | Later Developments | 342 |

**B. Unicode** 345

**C. The Compact Disc** 353

| | | |
|---|---|---|
| 1 | Capacity | 353 |
| 2 | Description | 354 |
| 3 | Error-Correction | 354 |
| 4 | Encoding | 356 |
| 5 | The Subcode | 357 |
| 6 | Data Readout | 357 |
| 7 | Fabrication | 357 |
| 8 | The CD-ROM Format | 358 |
| 9 | Recordable CDs (CD-R) | 359 |
| 10 | Summary | 359 |
| 11 | DVD | 360 |

**D. ISO Country Codes** 363

**References** 367

**Answers to Exercises** 371

**Index** 385

# 1 Introduction

Perhaps the main reason why computers are so prevalent in modern life is the fact that a single computer can perform a wide variety of tasks and can solve problems in many fields. The reason for that is that anything that the computer does is driven by software. When we want to use a computer, we have to write a program (or use an existing program). Without a program, the computer can do nothing. With the right program, the computer can perform the right task. Therefore, the most important computer feature is the way it executes a program. The bulk of this chapter is thus devoted to the execution of instructions. The part of the computer that's responsible for instruction execution is called the control unit (CU).

This chapter starts with a discussion of the control unit and its operations. This topic is illustrated by several examples of instruction execution. A simple register transfer notation is used to describe the execution of instructions by the control unit. The execution of many instructions requires a memory access, so the next topic of this chapter is the memory unit, its main features and operations. The discussion concentrates on the interface between memory and the control unit, and the way the control unit performs memory operations. The internal operations of the memory unit itself are not discussed.

The important concept of *interrupts* is treated in Section 1.7, because the hardware aspects of interrupts are part of the control unit. Vectored interrupts are explained, followed by the two main problems of a multi-interrupt system, namely simultaneous interrupts (and interrupt priorities), and nested interrupts (and interrupt enabling/disabling). The section concludes with a discussion of software interrupts and their applications to time slices and privileged instructions.

Section 1.8 discusses I/O interrupts. Notice that input/output is fully treated in Chapter 3.

Section 1.10 is a short introduction to the ALU and its operations. Simple circuits for adding integers are described. Chapter 9 is a detailed treatment of many ALU circuits.

## 1.1 Terminology

Computers consist of many components so, in order to concentrate our discussion, standard terminology is needed. The most general concept discussed here is that of a *computer installation*. An installation consists of a computer (or several computers), I/O devices such as modems, printers, and disk drives, personnel, and secondary features such as air conditioning, cables, and raised floors. Notice that a personal computer can also be considered a computer installation since it includes I/O devices.

From the point of view of organization, the computer is divided into three components, the processor, the memory, and the I/O processor. The memory is a passive component. It simply stores the information written in it. The function of the I/O processor is to interface the computer to the outside world. Advanced I/O processors can perform the actual input and output operations, thereby releasing the processor from this slow task. The processor (or *central processing unit*, CPU) is the main part of the computer. Its task is to execute the program instruction by instruction. The processor itself is functionally divided into three components, the control unit (CU), the arithmetic and logical unit (ALU), and the registers. The registers

are fast storage units, where a few pieces of information can be stored temporarily until they are needed by the program. The ALU and the registers are sometimes combined and are called the RALU. Figure 1.1 illustrates the relations between these components.

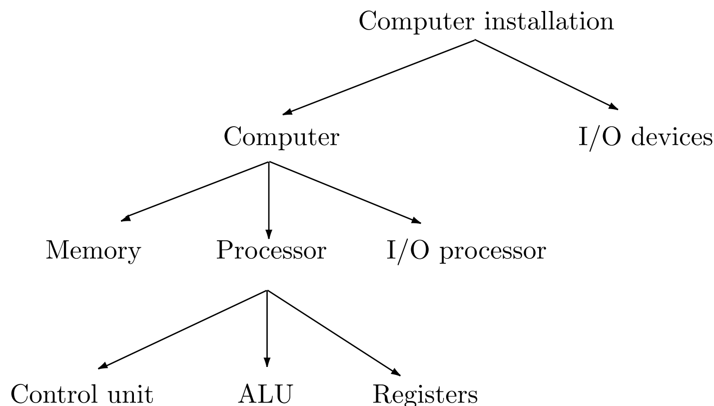



**Figure 1.1:** Basic computer terminology

▸ **Exercise 1.1:** Why is input/output a slow task?

## 1.2 The Control Unit

The computer is an information processing machine and the processor (CPU) is that part of the computer where information is processed. More specifically, the processor executes the program.

The main feature of the computer, the feature that makes it so different from other machines, is the *internal control*. The computer is controlled by a *program*, which specifies the operations to be performed. The program is located in memory, outside the processor, and the processor executes the program by reading (fetching) it from memory, instruction by instruction, and executing each instruction. This task is performed mainly by the control unit (CU), which uses the ALU and the registers as tools. The ALU is that part of the processor where operations are performed on numbers. The registers, as mentioned earlier, are fast, temporary storage.

The control unit is the main part of the processor. It performs its task by repeatedly cycling through the following steps:

1. Read (fetch) the next instruction from memory.
2. Examine the instruction to determine what it is and to find any possible errors.
3. Execute the instruction.

These steps are known as the fetch-execute cycle. In order to read the next instruction from memory, the control unit needs to know the location (memory address) of the instruction. This address is kept in a special register called the *program counter* (PC). The PC is a special register and is normally located in the control unit. It always contains the address of the *next* instruction, not the current one. When the next instruction is fetched, it is stored in the *instruction register* (IR), another special-purpose register, that always contains the current instruction. Note that as soon as the next instruction is fetched and is stored in the IR it becomes the *current* instruction (Figure 1.2).

▸ **Exercise 1.2:** Can the PC be one of the general-purpose registers (GPRs)?

The fetch-execute cycle can now be rewritten as:

1. Read the instruction that's pointed to by the PC from memory and move it into the IR.
2. Increment the PC.
3. Decode the instruction in the IR.
4. If the instruction has to read an operand from memory, calculate the operand's address (this is called the *effective address*, or EA) and read the operand from memory.
5. Execute the current instruction from the IR.

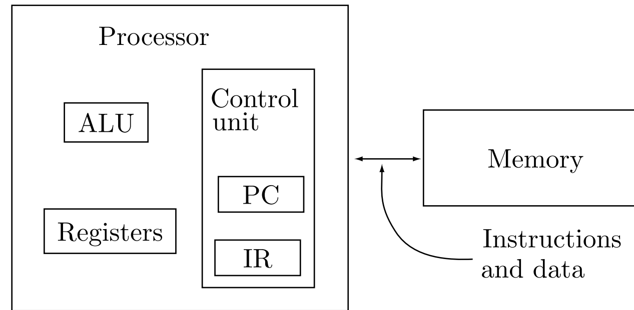



**Figure 1.2:** The main components used in the fetch-execute cycle

Step 2 is important since the PC should always point to the next instruction, not to the current one (the reason for this is discussed below). The PC is incremented by the size of the instruction (measured in words). In modern computers, instructions have different sizes. The control unit therefore has to determine the size of the instruction, make sure that all parts of the instruction have been fetched, then increment the PC. Step 2 is, therefore, not as simple as it sounds, and it is combined with Step 3. Instruction sizes are discussed in Section 2.1.

In Step 3, the instruction is examined (decoded) to determine what it is and how it is executed. Various errors can be detected at this step, and if an error is detected, the control unit does not proceed to Step 4 but instead generates an *interrupt* (Section 1.7).

▸ **Exercise 1.3:** What errors can be detected when an instruction is decoded?

Step 4 is executed only if the instruction needs to read an operand from memory or if it uses an addressing mode. Addressing modes are discussed in detail in Section 2.3.

In Step 5, the control unit executes the instruction by sending signals (control signals) to the ALU, to the memory, and to other parts of the computer. Step 5 is the most complex part of the control unit, since the instructions are different, and each should be executed differently. The control unit has a different circuit for executing each instruction and, since a modern computer might have more than 100 different instructions, the control unit is a large, complex circuit.

On considering the steps above, the reader may wonder why they are arranged in this order. Specifically, why does the control unit perform *fetch—increment PC—execute*, rather than *fetch—execute—increment PC*. In fact, the latter sequence may seem more natural to most.

It turns out that for most instructions, the two sequences above are identical and the control unit can use any of them. However, there is one group of instructions, those affecting the flow of control, where the *fetch—increment—execute* sequence is the only one that works.

Such instructions are the various jumps and branches. They are used to change the normal flow of control, the way execution proceeds within a program, and they themselves are very easy to execute. Executing an instruction such as `JMP 846` is done simply by resetting the PC to 846 (to the jump address). Since the PC is supposed to point to the next instruction, resetting the PC guarantees that the next instruction will be fetched from location 846. Figure 1.3 compares the operations of the control unit in both the *fetch—increment—execute* (part a) and the *fetch—execute—increment* (part b) sequences. It is easy to see that the latter sequence ends up with the PC being reset to address 847, which is obviously wrong.

▸ **Exercise 1.4:** What other instructions change the flow of control?

The upper part of Figure 1.3 shows memory locations 189–191 with the `JUMP` instruction stored in location 190. The PC contains 190, so the instruction at memory location 190 is the next one, implying that the current instruction is the one at location 189. In part (a2) the instruction from 190 has been fetched into the IR. In (a3), the PC is incremented to point to the next instruction, the one in location 191. In (a4), the instruction in the IR is executed by resetting the PC. It is reset to the jump address, 846.

In parts (b2), (b3), and (b4), the `JUMP` instruction is fetched, it is executed by resetting the PC to 846, and, finally, the PC is incremented to 847, which is wrong.

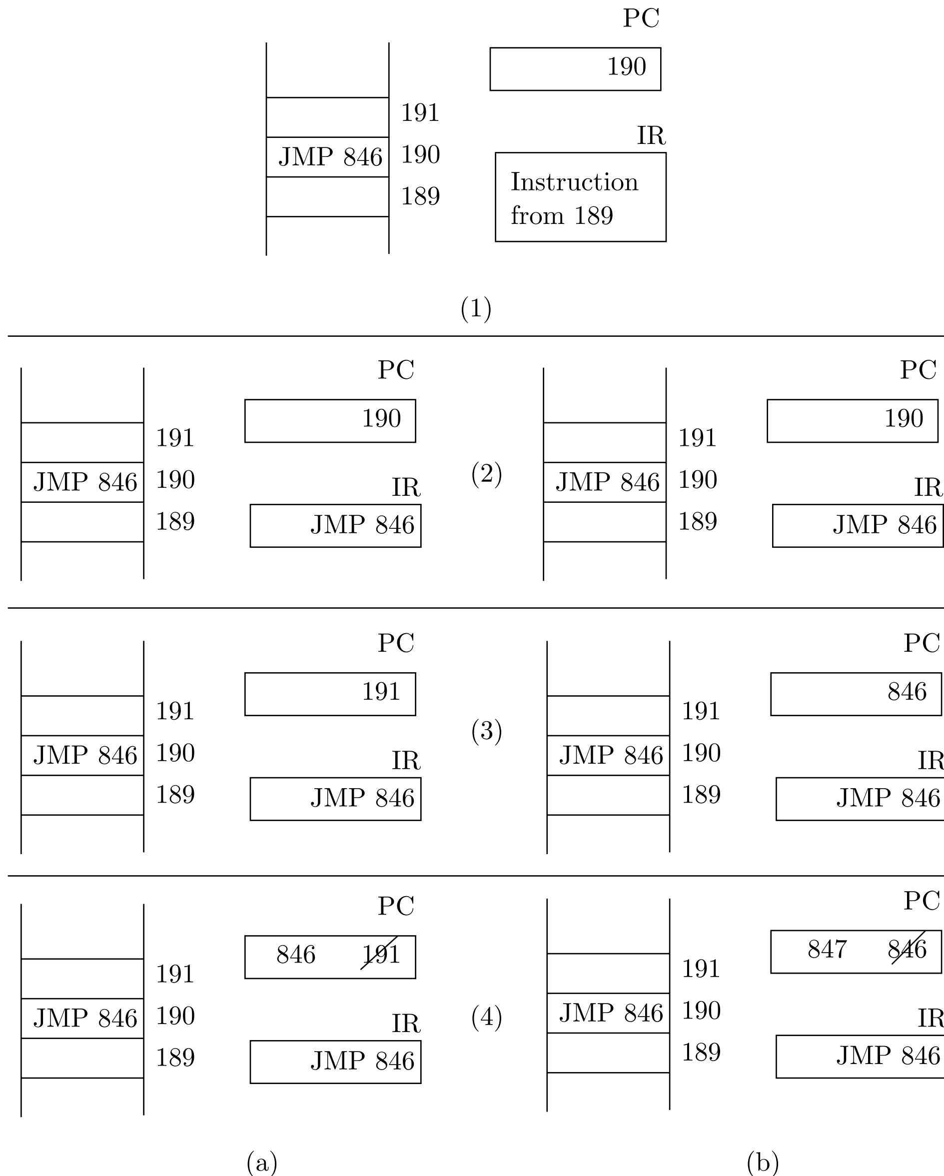



**Figure 1.3:** Two alternative fetch-execute sequences

### 1.2.1 Control Unit Start and Stop

How does the fetch-execute cycle start? It starts as soon as power to the computer is turned on, which is why it is important to make sure that the PC is set to the correct value as soon as the computer is started. This is done by a special circuit that's invoked when power is turned on (and also every time the computer is reset). The problem is that when the computer is turned on there is nothing in memory, since no program has been written or compiled yet. To what address should this circuit set the PC?

The answer requires an understanding of the interaction between the user (a person), the operating system (a piece of software) and the control unit (hardware). The operating system is a program (rather, a set of programs) whose task is to service the user (in a multiuser computer, the operating system also supervises the users). For example, the various compilers are part of the operating system and they, of course, provide an important service. When a computer is manufactured, a small part of the operating system, called the resident, is permanently written into the lowest area of the computer's memory. The resident (actually, its first part, the *bootstrap loader*) is the first program that's executed each time the

computer is turned on or is reset. (See the discussion of bootstrapping in Section 1.3.1.)

Here is a summary of what happens when the computer is started. The special circuit resets the PC to the start address of the resident part of the operating system (in many computers this is address zero). The control unit starts its cycle, so it fetches and executes instructions from the start of the resident operating system (i.e., the bootstrap loader). This loads the rest of the operating system from disk. The operating system then goes into a loop where it waits for the first user command or action.

▸ **Exercise 1.5:** What would happen if the PC does not get set when the computer starts?

▸ **Exercise 1.6:** Why not store the entire operating system permanently in memory?

The control unit stops when power to the computer is turned off. This seems a "rough" way to stop the computer, but it makes sense when we consider the way the user, the operating system, and the control unit interact. The user turns the computer on, which starts the resident operating system, which in turn loads the rest of the operating system. The operating system then executes user actions and commands, such as "open a window," "delete a file," "execute a file," "move the cursor," "pull down a menu," "insert a floppy disk into the disk drive," etc. At the end of the session, the user has no more commands/actions, so he turns off the computer. At this point the control unit is still running at full speed, fetching and executing instructions, but we don't mind shutting it off abruptly since those instructions are executed by the operating system while looking for user actions. From the point of view of the user those instructions are effectively useless.

The most important task of the control unit is to execute instructions. However, before this can be discussed in detail, the reader must know something about memory and its operations. The next section discusses memory, and Section 1.4 returns to the control unit and instruction execution.

## 1.3 The Memory

The inner working of computer memories are treated in Chapter 8. This section discusses the basics features of memory, memory operations, and how the control unit executes them. Memory is the second of the three components of the computer. It is used for storage and is therefore a passive component. A piece of data stored in memory is simply saved there and is not operated on. To operate on data, it is necessary to read it from memory and move it to the processor (specifically, to the ALU). Information stored in memory can be either machine instructions or data. Computer memories exhibit the following features:

1. Memory is organized in equal-size units called *words* (or cells). If there are $M$ words in memory, each consisting of $N$ bits, then memory is said to be of size $M \times N$. The quantities $M$ (number of words) and $N$ (word size) are two of the most important parameters of the architecture of the computer.

2. Only binary information (bits, zeros and ones) can be stored in a computer memory. In particular, memory cannot be empty. There is no such thing as a blank word in memory. If nothing has been stored in a memory word, it will contain random bits. This is why data has to be stored in memory before it can be read back.

3. There are only two memory operations, read and write. Reading a word from memory brings in a copy of the data. The original stays in the word and may be read again and again. Writing into a word destroys its original data.

4. Memory is accessed on a word basis, so each word should be uniquely identified. This is done by assigning an *address* to each word. Addresses are numeric and run from 0 to $M-1$. When memory is accessed, an address has to be specified first. The word associated with that address is accessed, and can either be read from or written to.

It is important to understand that exactly one word can be accessed at any time. It is impossible to read less than a word in memory. To access two words, each has to be accessed individually, a process that requires two separate memory operations. Also, even though modern memories are very fast, a memory unit can only do one thing at a time; it does not operate in parallel (although parallel memories can be made for special applications).

Next, we discuss typical values of $M$ and $N$. It turns out that certain values are better than others, and an understanding of $M$ and $N$ is important for an overall understanding of computer organization (see also Table 1.4 for standard names of powers of 10).

| Prefix | Symbol | Power | Value |
|---|---|---|---|
| yotta | Y | $10^{24}$ | 1 000 000 000 000 000 000 000 000 |
| zetta | Z | $10^{21}$ | 1 000 000 000 000 000 000 000 |
| exa | E | $10^{18}$ | 1 000 000 000 000 000 000 |
| peta | P | $10^{15}$ | 1 000 000 000 000 000 |
| tera | T | $10^{12}$ | 1 000 000 000 000 |
| giga | G | $10^{9}$ | 1 000 000 000 |
| mega | M | $10^{6}$ | 1 000 000 |
| kilo | K | $10^{3}$ | 1000 |
| hecto | h | $10^{2}$ | 100 |
| deca | da | $10^{1}$ | 10 |
| deci | d | $10^{-1}$ | 0.1 |
| centi | c | $10^{-2}$ | 0.01 |
| milli | m | $10^{-3}$ | 0.001 |
| micro | (mu) | $10^{-6}$ | 0.000 001 |
| nano | n | $10^{-9}$ | 0.000 000 001 |
| pico | p | $10^{-12}$ | 0.000 000 000 001 |
| femto | f | $10^{-15}$ | 0.000 000 000 000 001 |
| atto | a | $10^{-18}$ | 0.000 000 000 000 000 001 |
| zeppo | z | $10^{-21}$ | 0.000 000 000 000 000 000 001 |
| yocto | y | $10^{-24}$ | 0.000 000 000 000 000 000 000 001 |

**Table 1.4:** International standard names of powers of ten

Certain familiar values for $M$ are 8K, 64K, 256K, 1M, (pronounced 'one mega'), and 32M words. The quantity K (the letter K stands for "Kilo") is normally defined as $10^3 = 1000$ but, since computers use binary numbers, the quantity K that's used in computers is defined as the power of 2 that's nearest 1000. This turns out to be $2^{10}$ or 1024. Thus, 8K is not 8000 but $8 \times 2^{10} = 8192$, and 64K equals $2^6 \times 2^{10} = 65536$ rather than 64000. Similarly M (Mega) is not defined as a million ($10^6$) but as $2^{20} = 1024K$, which is a little more than a million. Notice that $M$ specifies the number of memory *words*, where a word is $N$ bits long. When $N = 8$, the word is called a *byte*. However, many modern computers have different values of $N$.

▸ **Exercise 1.7:** Express a petabyte (PB) in terms of terabytes (TB), gigabytes (GB), megabytes (MB), kilobytes (KB), bytes, and bits.

Why are such values used for $M$ instead of nice round numbers like 1000, 8000, or 64000? The answer is that 8192 *is* a nice round number, but in binary, not in decimal. It equals $2^{13}$. Some of the values mentioned earlier are

$$
\begin{aligned}
64\text{K} &= 64 \cdot \text{K} = 2^6 \cdot 2^{10} = 2^{16},\\
256\text{K} &= 2^8 \cdot 2^{10} = 2^{18},\\
32\text{M} &= 2^5 \cdot 2^{20} = 2^{25}.
\end{aligned}
$$

In general, $M$ is always assigned values of the form $2^j$ for some positive integer $j$. The reason is that such values make best use of the range of addresses. It turns out that in a memory with $M = 2^j$ words, every address is a $j$-bit number, and every $j$-bit number is a valid address.

To see why this is true, consider the familiar decimal numbers. In a memory with 1000 words, decimal addresses range from 0 to 999. These addresses are therefore 3-(decimal) digit numbers. What's more, *every* 3-digit number is an address in such a memory.

As a simple binary example, consider $M = 8 = 2^3$. Binary addresses range from 0 to $7 = 111_2$ and are therefore 3-bit numbers. Also, every 3-bit number is an address, since $111_2$ is the largest 3-bit number.

In general, if $M = 2^j$, addresses vary from 0 to $M - 1$. But

$$M - 1 = 2^j - 1 = \underbrace{1\ldots11}_{j}{}_2,$$

so addresses in such a memory are $j$-bit numbers.

▸ **Exercise 1.8:** How many words are there in a 32M memory?

Old computers typically had memory sizes of 32K or 64K words. The first 8-bit microprocessors also had memories with no more than 64K words. Modern computers, however, both large and small, support much bigger memories and have values of $M$ in the range of 1M to several G (where G, Giga, is defined as $2^{30}$) or even a few T (Tera, or $2^{40}$). Such computers have long addresses, typically in the range of 20 to 40 bits.

It turns out that the address size is an important organizational feature of the computer. Chapter 2 has more to say on the relation between the address size and the design of the instruction set.

The word size, $N$, also needs to be discussed. Again certain values are preferable while other values make no sense. Many modern computers use $N = 8$. Such a word is called a *byte*. In general, the value of $N$ is determined by what will be stored in the memory words when the computer is running. Memory words contain either machine instructions or data. Data stored in a word in memory can be of several types (all described in Chapter 2). Two data types, namely characters and decimal digits, are briefly mentioned here in connection with the values of $N$.

It is possible to store text in memory by assigning a code to each character (letters, digits, and punctuation marks) and storing the codes. The size of the code of a character is one parameter affecting the choice of the word size. Typical character codes have sizes of 7–8 bits. With 7-bit codes, it is possible to code $2^7 = 128$ characters. With 8-bit codes, the computer can handle a set of up to 256 characters.

It sounds strange, but decimal digits are also stored in memory. We already know that memory can contain only bits, so how can decimal digits be stored in it? The idea is to translate each digit into bits. Since a single decimal digit cannot exceed 9, at most 4 bits are needed. For example, the number 90,762 is translated into the five 4-bit groups 1001, 0000, 0111, 0110, and 0010. Such numbers can be stored in memory and can be operated on, and this fact suggests that the word size should be a multiple of 4. This, combined with the size of a character code, is the reason why the word size of modern computers is a multiple of 8.

It should be mentioned that old computers—most notably the large CDC computers popular in the 1960s and 1970s—had word sizes that were multiples of 6 (Section 3.6). They used 6-bit character codes, allowing for only 64 different characters.

In many modern computers, a memory word consists of several bytes. The individual bytes of a word can be ordered in one of two ways termed *big endian* and *little endian*. The big endian convention arranges the individual bytes of a word such that the last byte (the end) is the biggest (most significant). This is used on processors made by Motorola. Thus, the big endian order is byte0, byte1,... (the least-significant byte is located at the highest address). In little endian, the bytes are arranged such that the last byte (the end) is the "littlest" (least significant). This convention is used on Intel processors. Thus, the little endian order is ...,byte1, byte0 (the least-significant byte is located at the lowest address).

▸ **Exercise 1.9:** Given the four-byte word 00001111|00110101|01011010|00000001, show how it is laid out in memory in big endian and in little endian.

### 1.3.1 Other Features of Memories

First-generation computers had memories made of mercury delay lines or of electromechanical relays. They were bulky, slow, and unreliable. Second-generation computers had magnetic core memories which were smaller, faster, and more reliable. Modern computers use semiconductor memories which are fabricated on integrated circuits, use very little electrical power, and are as reliable as the processor itself (and almost as fast). One important feature of these memories, namely volatility, is discussed in this section.

The two main types of semiconductor memories are ROM and RAM. The acronym ROM stands for "read only memory." The computer can read from this type of memory but cannot write into it. Data can be written in ROM only during its manufacturing process. Some types of ROM can be written to (programmed) using a special device located outside the computer. RAM is Read/Write memory and is of more general use. Incidentally, the name "RAM" stands for "random access memory," but this is an unfortunate name since both ROM and RAM are random access. A better name for RAM would have been RWM (read write memory).

The term “random access” is defined as the case where it takes the same amount of time to read any memory location, regardless of its address. It also takes the same time to write into any location. However, the read and write times are not necessarily the same.

If ROM cannot be written to by the computer, why is it used at all? It turns out that ROM has at least two important applications. It is used for dedicated applications and also for the bootstrap loader.

We are used to computers where different programs are compiled, loaded, and executed all the time. However, many computers spend their “lives” executing the same program over and over. A computer controlling traffic lights, a computer in a microwave oven, and a computer controlling certain operations in a car are typical examples of computers that execute *dedicated applications*. In such computers, the programs should be stored in ROM, since ROM is nonvolatile, meaning it does not lose its content when power is turned off. RAM, on the other hand is volatile.

When a computer is first turned on, there should be a program in memory for the computer to execute. Without such a program it would be impossible to use the computer, since there would be no way to load any programs into memory. This is why computers have a small program, the *bootstrap loader*, in permanent ROM to start things up by loading the rest of the operating system. It should be pointed out that there is another way to bootstrap a computer. Some old computers had switches that made it possible to manually load a small bootstrap loader into memory. With volatile semiconductor memories, however, it is more convenient to have a bootstrap ROM.

To increase memory reliability, some computers use a *parity bit* in each memory word. Each time new information is written to the word, the parity bit is updated. Each time the word is read, parity is checked. Since no memory is completely reliable, bits may get corrupted while stored in memory, and it is important to detect such errors.

The idea of a parity bit is to complete the number of 1’s in the word to an even number (even parity) or to an odd number (odd parity). For example, suppose the number 010 is written in a 3-bit word and even parity is used. The parity bit should be set by the memory unit to 1, to complete the number of 1’s to an even number. When the word is read, the memory unit checks parity by counting the number of 1’s. If the number of 1’s is odd, memory notifies the processor of the problem by means of an interrupt (Section 1.7). A parity bit is thus a simple device for error detection. However, a single parity bit cannot tell which bit (or bits) is bad.

A single parity bit also cannot detect every possible error. The case where two bits change their values in a memory word cannot be detected by simple parity. There are ways of adding more parity bits to make data more reliable, some of which are discussed in Section 3.9.

▸ **Exercise 1.10:** What if the parity bit itself goes bad?

### 1.3.2 Memory Operations

This section treats memory as a black box and does not go into how memory operates internally. We only discuss how memory is accessed by the control unit (later we see that the DMA device can also access memory). As mentioned earlier, there are only two memory operations, namely memory read and memory write. Each operation accesses one word, so at any given time it is only possible to read a word from memory, or to write into a word in memory. When reading from a word, only a copy of the content is read and the original content of the word is not affected. This means that successive reads will read the same data. This behavior is a special case of a general rule that says that when data is moved inside the computer, only a copy is moved and the original is unaffected.

▸ **Exercise 1.11:** What are exceptions to this rule?

The “memory write” operation, on the other hand, writes new information in the word, erasing its old content. This is the only way to erase information in memory, there being no “erase” operation. After all, a memory word may contain only bits, so there is no such thing as a blank word. As a result, even words that have not been initialized with any information, have something in them, either remnants from an old program, or random bits.

The first step in either memory operation is to send an address to memory. The address identifies the memory word that should be accessed (read from or written into). From that point, the “read” and “write” operations differ. The “memory read” operation consists of the following four steps:

1. Send an address to memory.
2. Send a “read” signal to memory.
3. Wait for memory to complete its internal operations and prepare a copy of the data to be read.
4. Move the content from memory to where it should go.

The address is sent, in Step 1, to a special register called the *memory address register* (MAR). The MAR is a special register in the sense that each time it is accessed from the outside, memory senses it and considers the new content of the MAR an address. The “read” signal in Step 2 is necessary because there can be two operations, read and write. Step 3 is unfortunate—since the control unit has to idle during this step—but it is unavoidable, because memory operates at a finite speed and it is slower than the processor. During this step, the data is moved by memory into a special register called the *memory buffer register* (MBR). In Step 4, the MBR is moved to the processor and can be stored in any of its three components, namely the control unit, the ALU, or the registers.

The four steps above can now be written as follows:

1. MAR←Address.
2. “R”
3. Wait.
4. Destination←MBR.

The time it takes to complete these four steps is called the *memory read cycle time*, since it is the minimum time between the start of one memory read and the start of the next memory operation. The time it takes to complete the first three steps is called the *memory read access time*, since this is the time memory is actually busy with the read operation. There are, of course, also a *memory write cycle time* and a *memory write access time*, which may be different from the memory read times.

How long is the wait time in Step 3? In modern semiconductor memories, this time is about 50–100 nanoseconds long, where a nanosecond (ns) is defined as $10^{-9}$ of a second, or one thousandth of a microsecond. These are incredibly short times, considering that even light, traveling at about 186,000 miles/s, advances only about a foot in one nanosecond! Yet memory is accessed so often that the wait times add up and can count for a significant percentage of the total execution time consumed by a program.

The signals needed to perform a memory access are normally sent by the control unit. Section 3.4 shows that the DMA device can also send such signals. Thus, the circuit that performs a memory access is called the *memory requestor*. The requestor has to know how long to wait in Step 3, and this may not be simple. The computer owner may physically replace some memory chips in the computer, and it is practically impossible to change the wait time in the processor each time memory is replaced. Also, a computer may have several memory chips, some faster than others. A simple solution is to let memory send a feedback signal, announcing to the outside world that the wait period is over. The memory requestor should proceed with Step 4 upon receiving that signal.

Figure 1.5 summarizes the steps of a memory read. It shows how the signals move between the CPU and memory on groups of wires called *buses*. The bus is an important architectural concept that has not been mentioned yet.

A computer bus is a group of wires (or printed connections) carrying signals of the same type. Three buses are shown in Figure 1.5. The address bus, carrying the $j$ bits of an address, the data bus, carrying the $N$ bits of a word or a register, and the control bus, with R, W, and Feedback lines. Computer buses are discussed in Chapter 3.

Memory write is, of course, different from memory read, but not significantly. The steps are:

1. MAR←address.
2. MBR←data.
3. “W”
4. wait.

The steps are different and a “W” (write) signal is necessary. However, the main difference between the read and write sequences is the wait period.

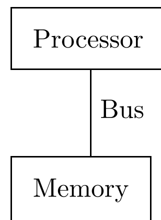
During memory read, the memory requestor must wait. During a memory write, however, since the wait period is the last step, the requestor can—at least in principle—do something else. The idea is that in Step 4, while memory is busy with its internal operations, no new memory access can start. However, If the requestor does not need another memory access, it can continue and can perform an internal operation. For

### The von Neumann Bottleneck

A conventional computer, traditionally called a *von Neumann machine*, consists of an ALU—where numbers are operated on—a control unit, responsible for instruction fetch and execute, and a memory—arranged as a linear array of fixed-size cells or words—where data and instructions are stored.

This is an elegant, powerful, and general-purpose design, which has proved its usefulness countless times in the last fifty years. During that period, however, both designers and users of computers came to realize that the simplicity and generality of the von Neumann machine are not only advantages but are also the cause of the limitations of the design. We now recognize two specific disadvantages of the von Neumann machine. The first is its excessive dependence on addresses; the second is the fact that the program control is sequential and centralized.

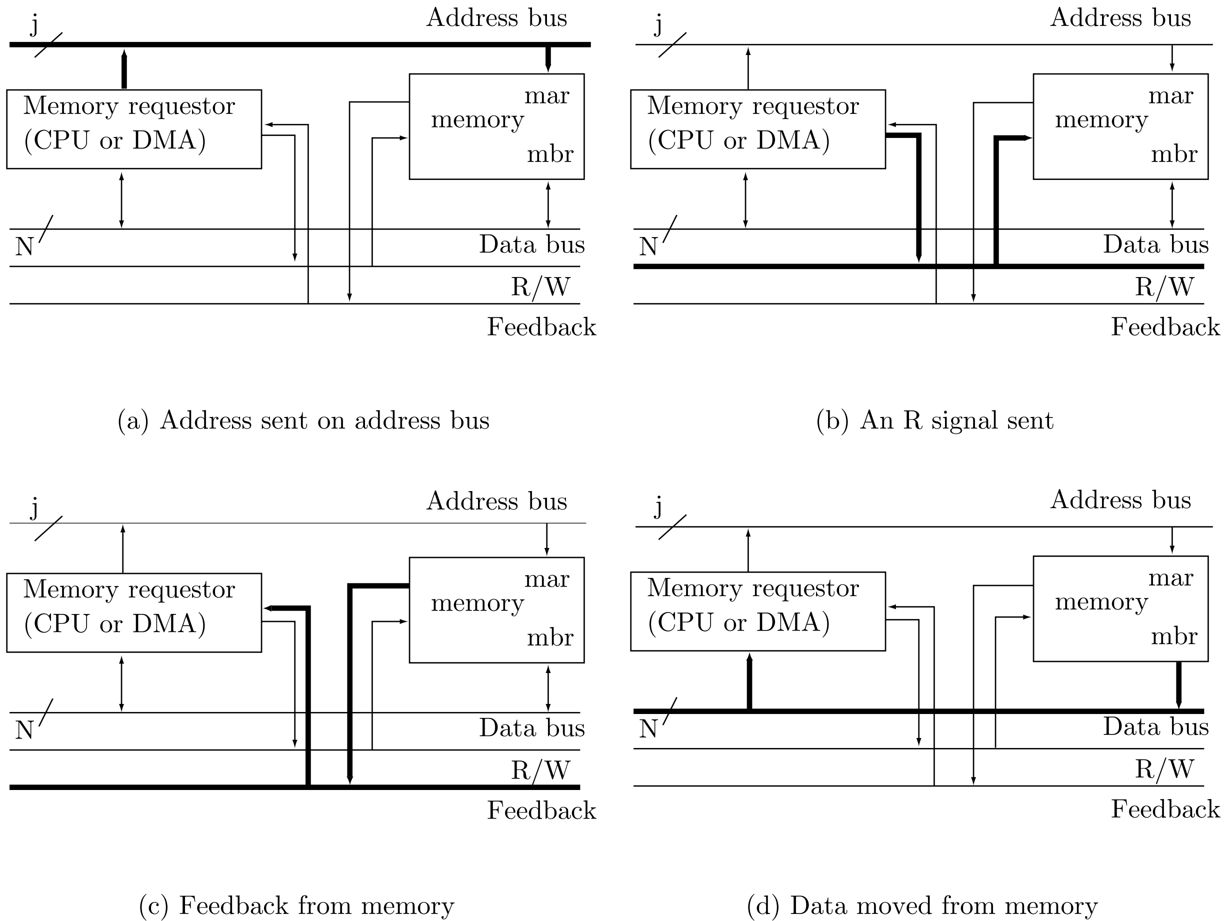
Here we discuss the first point, traditionally called the *von Neumann bottleneck* (Section 5.13). It is a result of the fact that every memory access must start by sending an address to memory. We can view the computer as consisting of two components, the processor and the memory, connected by a single communications channel, the memory bus. At any given time, a single word can be sent, on the bus, between the processor and memory, and it must be preceded by its address, sent on the same bus. This means that in order to read or write a piece of data from memory, the processor has first to obtain the address of that data, then send the address to memory. The address can be obtained in three different ways:




1. It can come from a register in the processor.
2. It can be part of an instruction. In that case, the instruction should first be fetched from memory, which means that *its address* has to be sent to memory through the bus.
3. It can be obtained as a result of some calculations. In this case, *several* instructions have to be fetched and executed, which implies that *their addresses* have to be pushed through the same bus, which now justifies the name *von Neumann bottleneck*. Note also that the program itself has to be fetched—instruction by instruction—from memory, and an address has to precede each such fetch.

As a result, we can now picture the von Neumann machine as a device that spends a large part of its time pumping addresses and data through the single bus connecting the processor and memory. Not a very efficient picture.

The realization that the von Neumann bottleneck places a limitation on the performance of present computers has led many designers to search for computer architectures that minimize the number of addresses, or that completely eliminate addresses. Two interesting approaches are the concepts of *associative computers* (Section 5.14) and *data flow computers* (Section 5.15), that are discussed in computer architecture texts and have had limited applications in practice.




(a) Address sent on address bus

(b) An R signal sent

(c) Feedback from memory

(d) Data moved from memory

**Figure 1.5:** Details of memory read

example, suppose that the control unit has initiated a memory write. During Step 4, the control unit can perform a register shift, which is an internal operation and does not involve memory.

Figure 1.6 summarizes the steps of a memory write operation.

With this knowledge of the memory unit and its two operations, we are now ready to look at the details of how the control unit fetches and executes instructions.

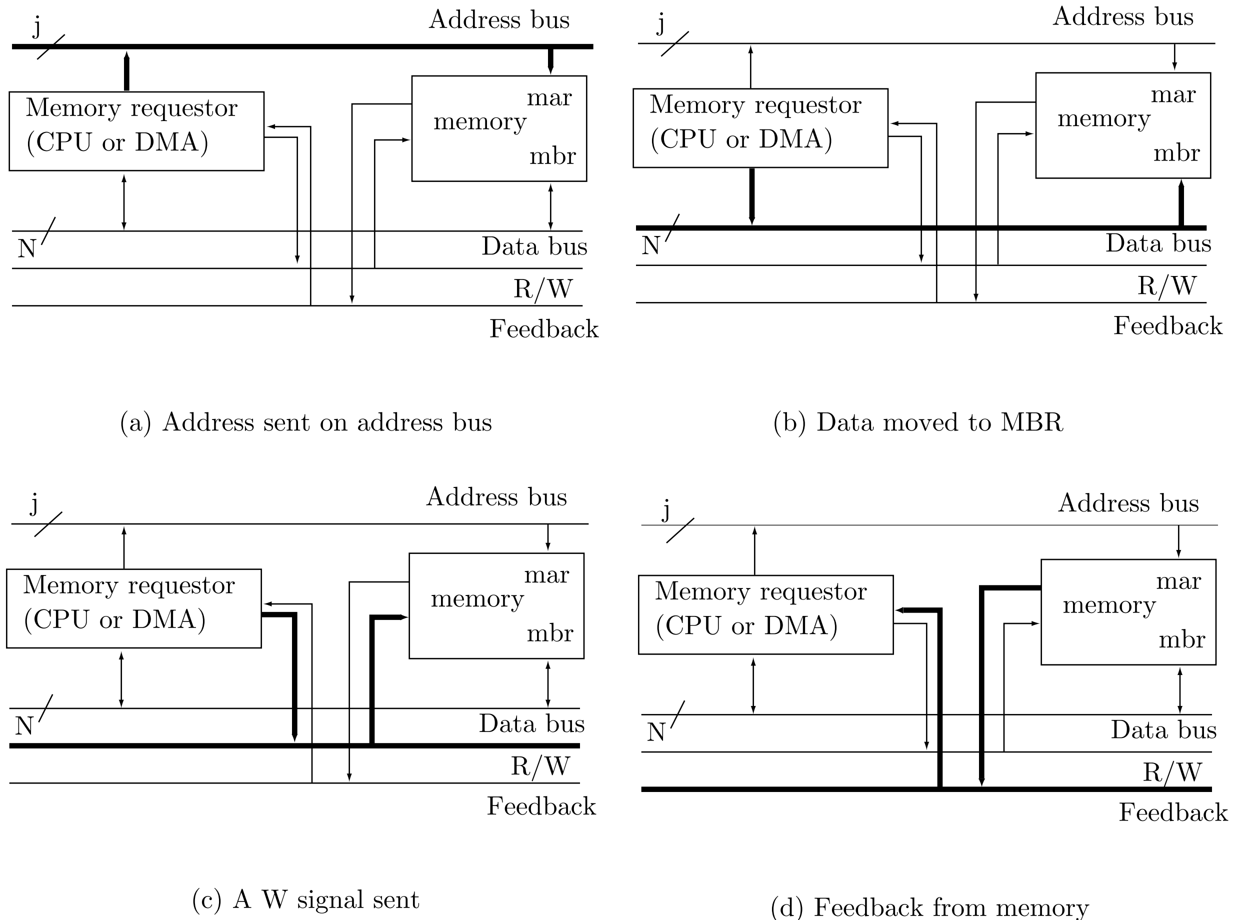
## 1.4 Instruction Execution

Fetching an instruction is a "memory read" operation. All instructions are fetched from memory in the same way, the only difference being the size of the instruction. In modern computers, some instructions are long. They occupy more than one memory word, and therefore have to be fetched in two or even three steps. In supercomputers, memory words are long (typically 64 bits each) and instructions are packed several to a word. In such a machine, reading a word from memory fetches several instructions, of which only the first can be immediately moved to the IR and executed. The rest are stored in an instruction buffer, waiting to be executed.

Decoding the current instruction, in Step 3, is done by passing the opcode through a decoder (Figure 1.7).

The decoder outputs are connected to the various execution circuits of the control unit and each output activates "its" execution circuit to execute an instruction. An execution circuit executes an instruction by generating a sequence of control signals that are sent to various parts of the computer. It should also be mentioned that in modern computers, opcodes have different lengths, so the control unit has to figure out the length of the opcode before it can decode it. Variable size opcodes are discussed in Section 2.2.

The opcode decoder is also the point where invalid opcodes are detected. Most computers have several




(a) Address sent on address bus

(b) Data moved to MBR

(c) A W signal sent

(d) Feedback from memory

**Figure 1.6:** Details of memory write

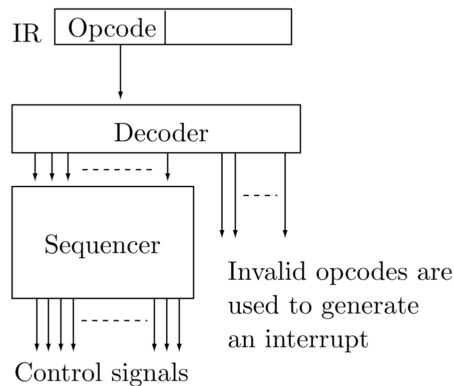



**Figure 1.7:** Decoding the opcode

unused opcodes (to be used in future versions of the machine), and as a result, some of the decoder outputs do not go to any execution units. Those outputs are connected together and, when any of them becomes active, that signal is used to generate an interrupt.

Instruction execution, however, is different from fetching and decoding and is much more complex than the operations in Steps 1–4 of the control unit cycle. Modern computers can have a large instruction set with as many as 200–300 instructions. Some instructions are similar and are executed in very similar ways.

Others are very different and are executed in completely different ways. In general, the control unit has to treat each instruction individually, and therefore has to contain a description of the execution of each instruction. The part of the control unit containing those descriptions is called the *sequencer*. This is the collection of all the execution circuits.

The sequencer executes an instruction by sending a sequence of control signals to the different parts of the computer. Most of those signals move data from one location to another. They perform *register transfers*. Other signals trigger parts of the computer to perform certain tasks. For example, a "read" signal sent by the sequencer to memory, triggers memory to start a "read" operation. A "shift" signal sent to the ALU triggers it to start a shift operation. The examples below list the control signals sent by the control unit for the execution of various instructions.

---

### The Three Steps

Following are a few examples of the execution sequences for certain common instructions. Notice that each example is presented in **three steps**. We first show how the programmer writes the instruction in assembler language (the precise notation depends, of course, on the particular assembler used; the notation used here is "generic"). Next, we show how the assembler assembles the instruction. The result is several numbers that together constitute a machine instruction (they are the *fields* of the instruction). The third step is the actual execution. It is important to realize that the three steps are distinct. They are performed by different entities and at different times. The first step is done by the programmer (a person) at the time the program is written. The second step is performed by the assembler (software) at assembly time, where instructions are assembled but not executed. The third step is done by the control unit (hardware) at run time (i.e., when the program is executed). However, all three steps are important and should be carefully studied for a complete understanding of the control unit and its operations.

---

**Example 1.** `JMP 158`. The programmer writes the mnemonic `JMP` and the jump address `158`. The assembler assembles this as the two-field instruction | `JMP` | `158` |. The control unit executes a jump instruction by resetting the PC to the jump address (address 158). Figure 1.8 shows the components involved in executing the instruction.

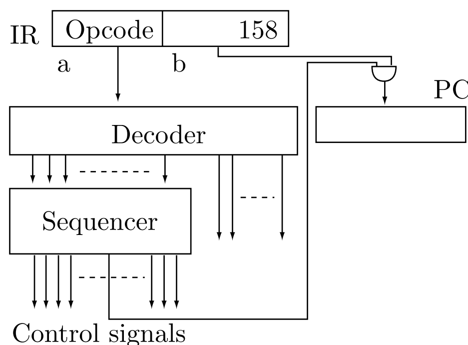



**Figure 1.8:** Executing a `JMP` instruction

It seems that the only operation necessary in this case is PC←158. This notation seems familiar since it is similar to that used in higher-level languages. In our case, however, this notation indicates operations performed by the control signals, and such operations are simple. A control signal can cause the transfer of register `A` to register `B`, but cannot move a constant to a register. This is why we need to have the jump address in a register. That register is, of course, the IR, where the instruction is located during execution. Since our example instruction has two parts, we assume that the IR is partitioned into two fields $a$ and $b$ such that $\mathrm{IR}_a$ contains the opcode and $\mathrm{IR}_b$ contains the address. The only control signal necessary to execute the instruction can now be written as: $\mathrm{PC} \leftarrow \mathrm{IR}_b$.

**A useful rule**

When an instruction is executed by the control unit, it is located in the IR.

Execution follows the fetch and decode steps, so when an instruction is executed, it has already been fetched into the IR and been decoded. When the control unit needs information for executing the instruction, it gets it from the IR, because this is where the instruction is located during execution (some information is obtained from other sources, such as the PC, the SP, and the general-purpose registers, GPRs).

As a result, many of the control signals used by the control unit access the IR.

**Example 2.** `LOD R1,158`. The programmer writes the mnemonic `LOD`, followed by the two operands (register and address) separated by a comma. The assembler generates the 3-field machine instruction

| LOD | 1 | 158 |
|---|---|---|

. We label these fields a, b, and c (where field $IR_b$ is the register number and field $IR_c$ contains the address). This instruction loads the content of memory location 158 into `R1` (one of the GPRs, in the processor). Before executing the instruction, the control unit has to calculate the effective address. The way this is done in practice depends on the addressing mode used. In this example we assume that the instruction contains the effective address and that no special addressing mode is used. Executing the instruction involves reading the operand from memory and moving it to the destination register. The steps are:

1. MAR $\leftarrow IR_c$
2. "R"
3. wait
4. R1 $\leftarrow$ MBR

However, Step 4 cannot really be written in this form since these steps should be valid for any `LOD` instruction. Step 4 should therefore be general. Instead of using the explicit name of the destination register, it should get that name from field c of the IR. We can write this step $R_{IR_b} \leftarrow$MBR. The four steps are illustrated in Figure 1.9a.

**Example 3.** `ADD R1,158`. This is similar to the previous example, and is illustrated in Figure 1.9b. Executing this instruction involves the steps:

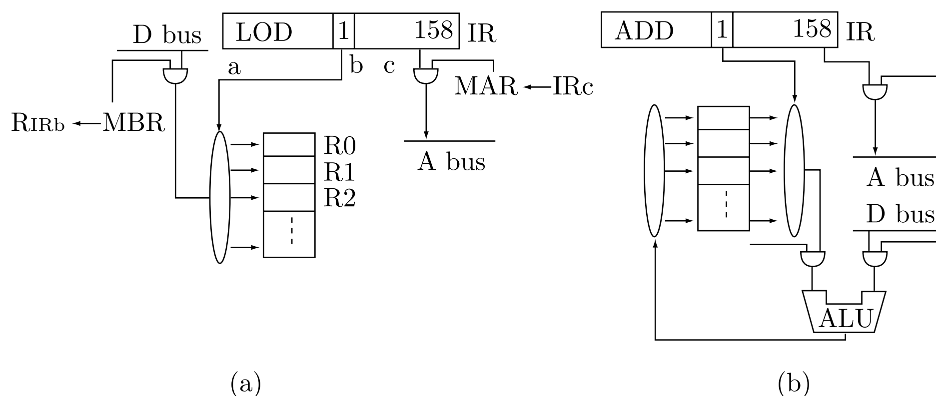



**Figure 1.9:** Executing `LOD` and `ADD`

- Read one operand from memory and send it to the ALU.
- Move the other operand from the register to the ALU.

- Send a signal to the ALU to start adding.
- Wait for the ALU to finish.
- Move the result (the sum) from the ALU output to its destination (the register).

The control signals are:

1. MAR ← $\mathrm{IR}_c$
2. "R"
3. wait for memory
4. left-ALU←MBR
5. right-ALU← $\mathrm{R}_{\mathrm{IR}_b}$
6. "add"
7. wait for the ALU
8. $\mathrm{R}_{\mathrm{IR}_b}$ ←ALU-output

Notice the wait in Step 7. The processor is waiting for a feedback signal from the ALU, indicating that the ALU has completed the operation. This is similar to the feedback signal from memory.

This example merits careful study for two reasons. One reason is that it illustrates how the task of executing a typical instruction is divided between the control unit and the ALU. The control unit is responsible for moving data between registers, for sending control signals to the various parts of the computer, and for waiting for feedback signals. The ALU, on the other hand, performs operations on numbers. It does nothing else, nor does it know where the numbers came from or where the result is going to.

The other reason this example is interesting is that many instructions are executed by the control unit in a similar (sometimes almost identical) way. In fact, all the instructions that operate on two numbers fall in this category. Thus, instructions such as `SUB R2,268`, `MUL R3,278`, and `COMPR R4,288` are executed by the control unit in almost the same way. Those instructions are obviously different, but the differences between them are reflected in the way they are executed by the ALU, not by the control unit. Even many single-operand instructions are executed in a similar way.

**Example 4.** `LSHIFT R3,4`. The content of register 3 is to be shifted four positions to the left. The way the programmer writes this instruction is obvious. The assembler assembles it as the 3-field number

| LSHIFT | 3 | 4 |
|---|---|---|

. To execute this instruction, the control unit should move `R3` to the ALU, start the ALU on the shift operation, wait for a feedback signal, and move the result back to the register. The control signals needed are (Figure 1.10):

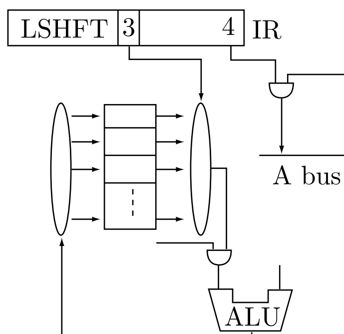



**Figure 1.10:** Executing a shift instruction

1. left-ALU← $\mathrm{R}_{\mathrm{IR}_b}$
2. right-ALU← $\mathrm{IR}_c$
3. ALU-function← 'left shift'
4. wait for ALU
5. $\mathrm{R}_{\mathrm{IR}_b}$ ←ALU-output

Note that the details of this process are transparent to the programmer. The programmer thinks of the shift as taking place in `R3`. Notice also the similarity between this example and the previous one. The difference between the execution of an add and a shift is in the ALU operation, the signals sent by the control unit are very similar.

**Example 5.** `COMPR R4,288`. This is a "compare" instruction, comparing its two operands. Writing and assembling this instruction are similar to the previous example. The comparison is done in the ALU and the results (see the discussion of comparisons in Section 2.13.4) are stored by the ALU in the status flags. We can consider the flags part of the ALU, with the control unit having access to them.

The control signals are:

1. MAR←$\text{IR}_c$
2. 'read'
3. wait
4. left-ALU←MBR
5. right-ALU← $\text{R}_{\text{IR}_b}$
6. ALU-function← 'compare'
7. wait for ALU

This should be compared to the control signals issued for executing the `ADD` instruction above. Note that Step 7 is normally unnecessary. In principle, the control unit should wait for the ALU only if it has to use it again immediately. In most cases, however, the next step of the control unit is to fetch the next instruction, so it does not need to use the ALU for a while. As a result, a sophisticated control unit—of the kind found in modern, fast computers—looks ahead, after Step 6, to see what the next operation is. If the next operation does not use the ALU, Step 7 is skipped. On the other hand, the control unit in a cheap, slow microprocessor—designed for simplicity, not speed—always executes Step 7, regardless of what follows.

**Example 6.** `CALL 248`. Calling a procedure is an important programming tool, and every computer includes a `CALL` instruction in its instruction set. The instruction is executed like a `JMP`, but the return address is saved. The return address is the address of the instruction following the `CALL`, so it is found in the PC at the time the `CALL` is executed. Thus, executing the `CALL` involves two steps:

1. Save the PC.
2. PC← $\text{R}_{\text{IR}_b}$.

Where is the PC saved? Most computers use one of three techniques. They save the PC either in a register, in the stack, or in memory.

Saving the PC in a register is the simplest method. The user should specify an available register by writing an instruction such as `CALL R5,248`, and the control unit saves the PC in R5. The downside of this method is that procedure calls may be nested and there may not be enough available registers to hold all the return addresses at the same time.

Saving the PC in the stack (stacks are discussed in Section 2.13.5) is a good method since procedure calls are often nested and the *last* procedure to be called is the *first* one to return. This means that the last return address to be saved in the stack is the first one to be used, which conforms to the way a stack is used. Modern computers have stacks and use them extensively. The few old mainframes still in use, generally do not have stacks. Also, some special-purpose computers that have just ROM cannot have a stack.

The third method is to save the PC in memory. This makes sense since there is an ideal memory location for saving each return address, namely the first location of the procedure. It is ideal since (1) it is different for different procedures, (2) it is known at the time the `CALL` is executed, and (3) it can easily be accessed by the procedure itself when the time comes for it to return to the calling program. An instruction such as `CALL 248` is therefore executed by (1) saving the PC in location 248, and (2) resetting the PC to 249. The procedure starts at location 248, but its first executable is located at 249. When writing a procedure (in assembler) on a computer that uses this method, the programmer should reserve the first location for the return address. If the program is written in a higher-level language, the compiler automatically take care of this detail. When the procedure has to return, it executes an indirect `JNP` to location 248. The use of the indirect mode mean that the control unit reads location 248 and resets the PC to the content of that location. The indirect mode, as well as other addressing modes, is discussed in Section 2.3.

The problem with this method is recursive calls. Imagine a procedure $A$ being called by the main program. The first location of $A$ contains the return address, say 1271. If $A$ calls itself before it returns, the new return address gets stored in the first location of the procedure, thereby erasing the 1271.

As a result, the PC is normally saved in the stack. The control signals are:

1. MAR←SP.
2. MBR←PC.
3. write.
4. wait. SP←SP+1.
5. PC← $IR_b$.

Notice that the control unit uses the wait period to increment the stack pointer.

**Example 7.** `BPL 158`. This is a "Branch on PLus" instruction. It is a conditional branch that branches to address 158 if the `N` status flag (sometimes called the `S` flag) corresponds to a positive sign (i.e., has a value of zero). More information on the status flags can be found in Sections 2.13.5 and 2.13.5. The only control signal necessary is "`if N=0 then PC`← $IR_b$".

Initially, it seems that this conditional signal is neither a straightforward signal nor a register transfer, but a new type of control signal. However, Figure 1.11 shows how the control unit can execute this instruction by sending a normal, unconditional control signal "Execute BPL" that passes conditionally through the AND gate depending on the state of the `N` status flag. Thus, the control unit can execute any conditional branch by sending a normal control signal; the only special feature needed for this type of instruction is an AND gate.

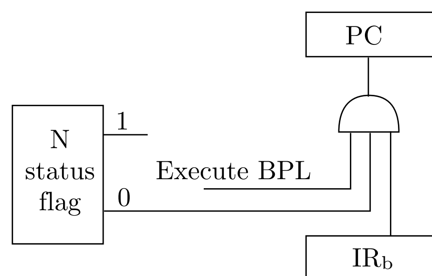



**Figure 1.11:** Executing a BPL.

This example is important since it is the basis of all the decisions and loops performed by the various programming languages. Any `if`, `for`, `while` or similar statements are compiled to machine instructions that include either a `BPL` or another conditional branch instruction. To execute such an instruction, the control unit has to be able to test any of the flags and perform an operation only if the flag has a certain value.

## 1.5 CPU Bus Structure

The discussion in the previous section implies that data can be moved freely inside the CPU between the control unit, registers, and ALU. This section shows the details. Data is moved inside the CPU along *buses* that connect the three components, and there are two ways of organizing these buses, referred to as a single bus and triple bus (the reader should not confuse those buses with the buses, described in Section 3.1, connecting the CPU to the memory and I/O devices).

A single-bus CPU is shown in Figure 1.12a. Every piece of data is moved over the single data bus, and the control unit sends register-transfer control signals on the control bus to open the right gates and move the data from source to destination. A point to note is that the ALU requires two inputs. One is sent to the auxiliary register and the other stays on the data bus during the ALU operation. Thus, sending input to the ALU requires two steps. The ALU result is latched in a result register and is sent to the data bus when the ALU has finished.

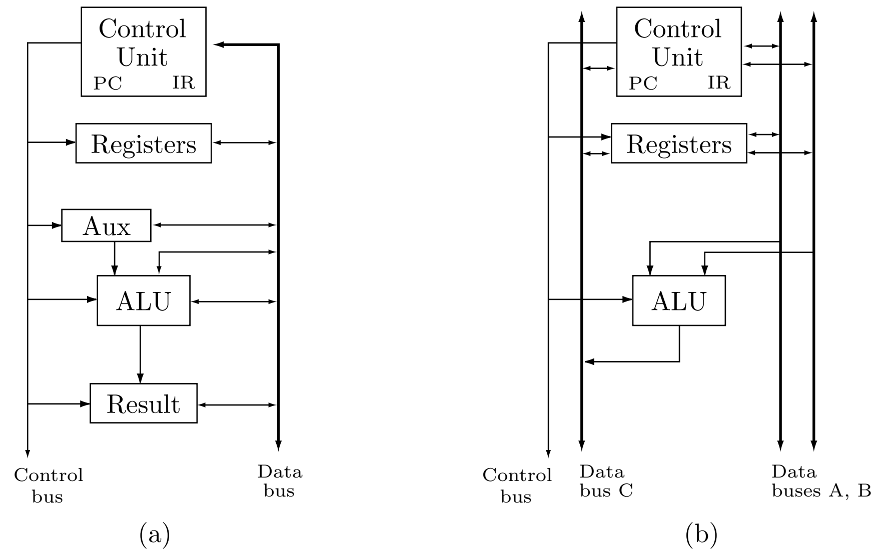



**Figure 1.12:** CPU bus structures

The advantage of this CPU organization is that only one bus is needed. However, some auxiliary registers are required and some operations require more than one step.

The triple-bus CPU design is shown in Figure 1.12b (compare this with Figure 4.1). The control unit and the registers can send data to, and receive data from, any of the three data buses. The ALU receives its two operands from data buses A and B, and sends its output to data bus C. There is no need for any auxiliary registers but three buses are needed.

## 1.6 Microprogramming

The control unit has a separate circuit for the execution of each instruction. Each circuit sends a sequence of control signals, and the total of all the circuits is called *the sequencer*. The advantage of this arrangement is that all the information about instruction execution is hardwired into the sequencer, which is the fastest way to execute instructions. There is, however, another way of designing the control unit. It is called *microprogramming* and it results in a microprogrammed control unit. This section is a short introduction to microprogramming. Chapter 4 discusses this topic in detail.

The basic idea of microprogramming is to have all the information on instruction execution stored in a special memory instead of being hardwired. This special memory is called *the control store* and it is not part of the main memory of the computer. Clearly, microprogramming is slower than hardwired execution because information has to be fetched constantly from the control store. The advantages of microprogramming, however, far outweigh its low speed, and most current computers are (fully or partly) microprogrammed. Most computer users, even assembler language programmers, do not know that the computer is microprogrammed or even what microprogramming is. Programs are written and compiled normally, and the only difference is in the way the control unit executes the instructions. This is why microprogramming is considered a low-level feature. The term “low level” means closer to the hardware and microprogramming is as close to the hardware as one can get.

What exactly is stored in the control store? Given a typical sequence for the execution of an instruction:

a ← b
“read”
c ← d
wait for memory feedback
e ← f

we can number each signal and store the numbers in the control store. The sequence above could be stored as, say 122, 47, 84, 115, 7; the sequence below could perhaps be stored as 84, 48, 55, 115, 7.

c ← d
"write"
x ← y
wait for memory feedback
e ← f

Note that signals 84 and 155 mean the same in the two sequences.

To design a microprogrammed control unit, the designers have to number every signal that the control unit can generate, decide on the execution sequence for each instruction, and store each sequence as a string of numbers in the control store.

Executing an instruction thus involves fetching its execution sequence, number by number, from the control store and executing each number in the sequence by sending the signal that corresponds to that number. The number is sent to a decoder that generates a unique output for each number that it receives. The output is the control signal that corresponds to the number. The sequence of numbers for the execution of an instruction $S$ is called the *microprogram* of $S$.

We therefore conclude that a hardwired control unit knows how to execute all the instructions in the instruction set of the computer. This knowledge is built into the sequencer and is permanent. In contrast, a microprogrammed control unit does not know how to execute instructions. It knows only how to execute microinstructions. It has hardware that fetches microinstructions from the control store and executes them. Such a control unit has a micro program counter (MPC) and a micro instruction register (MIR). The execution sequences are now stored in the control store, each as a sequence of numbers. This makes it easy to modify the instruction set of the computer and it constitutes one of the advantages of microprogramming. The numbers in the control store are much simpler to execute than machine instructions, so a microprogrammed control unit is simpler than a hardwired control unit. Figure 1.13 shows the main components and paths of a microprogrammed control unit.

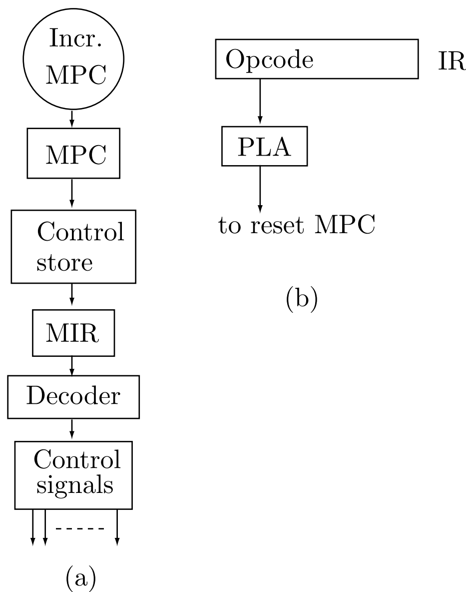



**Figure 1.13:** A microprogrammed control unit

It turns out that more than just signal numbers can be stored in the control store. In practice, each control store location contains a *microinstruction* with several fields. The fields can specify several control signals that should be generated, and can also indicate a jump to another microinstruction.

To fully understand microprogramming, two more points need to be clarified: (1) How does the control unit find the start of the sequence of microinstructions for a given instruction and (2) what does it do when it is finished executing such a sequence.

The answer to the first question is: When an instruction is fetched into the IR, the control unit uses its opcode to determine the address in the control store where the microinstruction sequence for that instruction starts. The address may be the opcode itself, it may be a function of the opcode or, what is most common, the address may be obtained from the opcode by means of a small ROM. In this method, the opcode is used as an address to this ROM and the content of that address is the start location of the microinstruction sequence in the control store. In practice, such a small ROM is constructed as a *programmable logic array* (PLA) (Section 8.4).

The answer to the second question is: When the control unit has executed a complete sequence of microinstructions, it should go back to Step 1 of the fetch-execute cycle and fetch the next machine instruction as fast as possible. Here we see one of the advantages of microprogramming. There is no need for any special circuit to fetch the next instruction. A proper sequence of microinstructions in the control store can do this job. As a result, there should be at the end of each sequence a special microinstruction or a special number, say 99, indicating the register transfer MPC $\leftarrow$ 0. This register transfer moves zero into the MPC, causing the next microinstruction to be fetched from location 0 of the control store. Location 0 should be the start of the following sequence:

1. MAR $\leftarrow$ PC
2. "read"
3. wait for memory feedback
4. IR $\leftarrow$ MBR
5. Increment PC

that fetches the next instruction.

More details, as well as a complete example of a simple, microprogrammed computer, can be found in Chapter 4.

## 1.7 Interrupts

Interrupts are a very important feature of the computer, so much so that even the smallest, most primitive microprocessors can handle at least one interrupt. Unlike many other computer features that are nebulous and hard to define, interrupts are easy to define.

**Definition:** Interrupts are the way the computer responds to urgent or unusual conditions.

Examples of such conditions are:

1. Invalid opcode. When the control unit decodes the current instruction (in Step 3 of the fetch-execute cycle), it may find that the opcode is invalid (i.e., unused). On many computers, not all opcodes are used. Some opcodes do not correspond to any instruction and are considered invalid. When the control unit identifies such an opcode, it generates an interrupt, since this is both an unusual and an urgent condition. It has to be handled immediately and the program cannot continue as usual.

2. Zero-divide. When the ALU starts executing a "divide" instruction, it first tests the divisor. If the divisor is zero, the ALU generates an interrupt since this is an urgent condition; the division cannot take place.

3. Timer. On many computers, there is a timer that can be set, by software, to any time interval. When execution time reaches this interval, the timer senses it and generates an interrupt.

4. Software interrupt. When the program wants to communicate with the operating system, it can artificially generate an interrupt. This interrupt is called a *break*, a *software interrupt*, or a *supervisor call*. It is an important feature of the computer and is discussed later in this section.

5. Many hardware problems, such as a parity error (in memory or during I/O) or a drop in the voltage can cause interrupts, if the computer is equipped with the appropriate sensors to sense those conditions and to respond accordingly.

6. I/O interrupts. Those are generated by an I/O device or by the I/O processor. Such an interrupt indicates that an I/O process has completed or that an I/O problem has been detected. Those interrupts are discussed in Section 1.8.

A good way to understand interrupts is to visualize a number of *interrupt request lines* (IRQs) arriving at the control unit from various sources in the computer (Figure 1.14). When any interrupt source decides

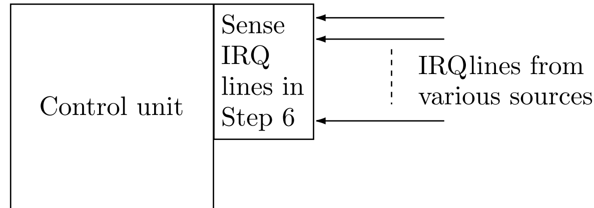



**Figure 1.14:** The control unit with IRQ lines

to interrupt the control unit, it sends a signal on its IRQ line. The control unit should test those lines from time to time to discover all pending interrupts, and respond to them.

Testing the IRQ lines is done in a new step, Step 6 of the fetch-execute cycle. The cycle becomes:

```
1. Fetch.
2. Increment PC.
3. Decode.
4. Calculate EA and fetch operand.
5. Execute.
6. Test all IRQ lines. If any of them is high, perform the following:
  6.1 Save the PC.
  6.2 Set the PC to a memory address, depending on the interrupt.
  6.3 Disable all interrupts.
  6.4 Set processor mode to ‘system’.
```

After Step 6.4, the control unit continues its cycle and goes back to step 1 to fetch the next instruction. However, if any IRQ line was tested high in step 6, the next instruction will be fetched from the address sent to the PC in Step 6.2. In fact, Steps 6.1 and 6.2 are equivalent to a procedure call (compare them with the example of executing a procedure call instruction). The conclusion is that the control unit, upon sensing an interrupt, performs a *forced procedure call.* All interrupts are treated by the control unit in the same way, the only difference being the address used in Step 6.2. Different addresses are used for different interrupts, resulting in different procedures being called.

It is now clear that the computer handles an interrupt by *artificially calling a procedure*. This mechanism is also called a *forced procedure call*. The control unit forces a call to a procedure each time an interrupt occurs. Figure 1.15 shows how control is switched from the interrupted program to the procedure and back. We now have to have a number of *interrupt handling procedures*, one for each interrupt. Where do those procedures come from? Obviously, they have to be written by someone, loaded into memory, and wait there. Those procedures are never explicitly called by any program. In fact, most computer users are not even aware of their existence. They lie dormant, perhaps for a long time, and are invoked only when something unusual happens in the computer and an interrupt signal is issued and is sensed by the control unit in Step 6 of its cycle. We say that the interrupt handling routines are called *automatically*, which means the calls are forced by the control unit.

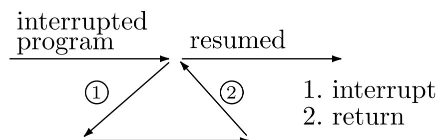



**Figure 1.15:** Flow of control in an interrupt

An interrupt handling routine is a piece of software and can do anything that a program can do. For example, the handling routine for an invalid opcode should print an appropriate error message, should also print the saved value of the PC (giving the user an indication of where the error occurred), and should probably terminate the program, returning control to the operating system in order to start another program.

The handling routine for a zero-divide interrupt should also print an error message, print the saved value of the PC, then try to communicate with the user. In some cases the user should have a good idea of what happened and may be able to suggest a value for the divisor, or tell the routine to skip the division altogether. In such cases the interrupted program may be resumed. To resume a program, the handling routine should use the saved value of the PC.

In summary, a good computer design calls for various sensors placed in the computer to sense urgent and unusual conditions. Upon sensing such a condition, the sensor generates an interrupt signal. The control unit tests the IRQ lines in Step 6, after it has finished executing the current instruction (or after it has decided not to execute the current instruction because of a problem). If any IRQ line is high, the control unit forces a procedure call to one of the interrupt handling routines. The routine must be prewritten (normally by a systems programmer) and is only invoked if a particular interrupt has occurred. Each handling routine should only handle one interrupt and it can do that in any way. It can print an error message, change any values in memory or in the registers, terminate the user program or resume it, or do anything else.

Note that the IRQ lines are only tested in Step 6, i.e., between instructions. It is important to service an interrupt only between instructions and not immediately when it occurs. If the control unit stops in the midst of an instruction to service an interrupt, it may not be able later to complete the execution of the instruction, and thus may not be able to resume the program.

If each interrupt routine is devoted to one interrupt, we say that the interrupts are *vectored*. In many old microprocessors, there is only one IRQ line and, as a result, only one interrupt handling routine. All the interrupt sources must be tied to the single IRQ line and the first task of the interrupt handling routine is to identify the source of the interrupt. This is time consuming and is the main reason why vectored interrupts are considered important.

Most computers support several interrupts and have to deal with two simple problems. The first is simultaneous interrupts, a problem easily solved by implementing interrupt priorities. The second is nested interrupts, solved by adding instructions (i.e., hardware) for enabling and disabling interrupts.

Simultaneous interrupts do not have to be issued at the same time. If several IRQ lines go high at different times during Steps 1–5, then in Step 6, when the IRQ lines are tested, the control unit considers them simultaneous. Simultaneous interrupts are a problem because the computer can handle only one interrupt at a time. The solution is to assign a priority to each interrupt. If Step 6 finds several IRQ lines high, the control unit responds to the one with the highest priority and lets the others wait.

The problem of nested interrupts has to do with the interrupt handling routine. The routine is a piece of software, so it can operate only by executing instructions. While the routine is running, many kinds of interrupts may be issued. The problem is: Should a given interrupt be allowed to nest the handling routine or should it wait? The most general answer is: It depends. For example, the interrupt handling routine for interrupt 6 may not want interrupts 1, 4, and 7 to nest it. It may decide, however, that interrupts 2, 3, and 5 are important and deserve fast response. In such a case, it may let those interrupts nest it.

The general solution is therefore to add instructions (i.e., hardware) to disable and enable interrupts. A computer may have two such instructions, one to enable and another to disable, individual interrupts. Examples are '`EI 5`' (Enable Interrupt 5), '`DI 3`' (Disable Interrupt 3), `EA` (Enable All interrupts). Step 6.3, where all interrupts are disabled, is also part of the extra hardware. After this step, the control unit goes to step 1, where the next instruction is fetched. It is fetched from the handling routine, which means that when this routine is invoked, all interrupts have been disabled and the routine cannot be nested by any interrupts. If the handling routine decides that, for example, interrupts 2, 3, and 5 are important and should be allowed to nest it, it can start by executing '`EI 2`', '`EI 3`', and '`EI 5`'.

▸ **Exercise 1.12:** This discussion suggests that there is no need for instructions that disable interrupts. It that true?

When the handling routine completes its task, it should enable all the interrupts and return. This is a simple task but it involves a subtle point. We want to enable all interrupts and return, but if we execute the two instructions `EA` and `RET`, the control unit will sense the freshly enabled interrupts at Step 6 of the `EA` instruction, and will invoke a new handling routine before it gets to the `RET`. Here are the details.

The control unit fetches and executes the `EA` instruction in Steps 1–5. However, when the `EA` is executed, in Step 5, all interrupts become enabled. When the control unit checks the IRQs, in Step 6, it may find some IRQ lines high. In such a case, the control unit executes substeps 6.1–6.4, then goes to fetch the next

instruction. This next instruction comes from the new interrupt handling routine, and the routine starts its execution. All the while, the `RET` instruction hasn't been executed, which means that the original interrupt handling routine did not return. It has effectively been nested at the last moment.

To prevent this situation, the computer has to have a special instruction that enables all interrupts *and* also returns (by popping the stack and moving this data to the PC).

We now turn to the details of Step 6.2 above. How does the control unit know the start addresses of all the interrupt service routines? Here are three approaches to this problem:

1. The start addresses of the service routines are permanent and are built into the control unit. This is a bad choice, since the operating system is updated from time to time, and each update may move many routines to new locations.

2. Certain memory locations are reserved for these addresses. Imagine a 16-bit computer with 32-bit addresses. An address in such a computer is therefore two words long. If the computer has, say, 10 interrupts, the first 20 memory locations are reserved. Each time the operating system is loaded into memory, it loads the start addresses of the ten interrupt service routines into these 20 locations. Whenever the operating system is modified, the routines move to new start addresses, and the new operating system stores their new addresses in the same 20 locations. If the control unit senses, in step 6.1, that $\text{IRQ}_i$ is high, it fetches the address stored in locations $i$ and $i+1$ and moves it into the PC in step 6.2. This method of obtaining the start addresses is called *vectored interrupts*.

3. The control unit does not know the start addresses of the service routines and expects to receive each address from the interrupting device. In step 6.2, the control unit sends an *interrupt acknowledge* (IACK) signal to all the interrupting devices. The device that has sent the original IRQ signal responds to the IACK by dropping its original interrupt request and placing an address on the data bus. (Normally, of course, addresses are placed on the address bus, not on the data bus. They are also placed there by the control unit, not by any device, so this is an unusual situation. Also, in the case of a 16-bit computer with 32-bit addresses, the address should be placed on the 16-bit data bus in two parts.) After sending the IACK signal, the control unit inputs the data bus and moves its content to the PC. This method too is referred to as *vectored interrupts*.

Regardless of what method is used to identify the start address in step 6.2, the interrupt acknowledge signal is always important. This is because the interrupting device does not know how long it will take the control unit to respond to the interrupt, and thus does not know when precisely to drop its IRQ signal and to place the address on the data bus. The control unit may take a long time to respond to an interrupt if the interrupt is disabled or if it has low priority.

Ideally, each interrupt should have its own line to the control unit, but this may not always be possible for the following two reasons:

1. If the CPU is a microprocessor (a processor on a chip), there may not be enough available pins on the chip to accommodate many interrupts. A typical microprocessor has just one pin for interrupt request lines.

2. A computer may be designed with, say, 16 interrupt request lines but may be used in an application where there are many interrupting devices. An example is a computer controlling traffic lines in many intersections along several streets. At each intersection, there are four sensors buried under the street to detect incoming cars. When a sensor detects a car moving over it, it sends a pulse to the computer. Since the computer is much faster than any car, we can expect the computer to execute many thousands of instructions between receiving consecutive pulses. It therefore makes sense to send the pulses to the computer as interrupts. Such a computer can easily have hundreds of sensors sending it interrupts. It is possible to design a special-purpose computer for this application, but it makes more sense to use a general-purpose computer. A general-purpose computer is readily available, cheaper, and already has an operating system with a compiler and useful utilities.

We therefore consider the extreme case of a computer with just one interrupt request line. All the interrupt sources send their interrupt signals to this single line, and the interrupt system (i.e., the interrupt hardware in the control unit and the interrupt service routines) should be able to identify the source of any interrupt. We discuss two ways to design such an interrupt system, one using a *daisy chain* and the other using a *priority encoder*.

In a daisy chain configuration, interrupt signals from all the interrupt sources are ORed and are sent to the control unit. The acknowledge signal (IACK) sent back by the control unit arrives at the first device. If this device has sent the interrupt signal, then it drops its IRQ and places the address of its service routine on the data bus; otherwise, it forwards the IACK to the next device on the chain (Figure 1.16).

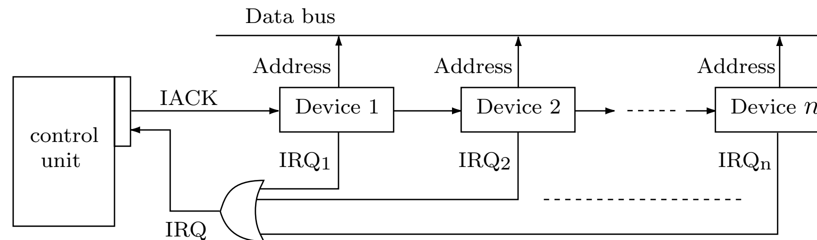



**Figure 1.16:** A daisy chain of interrupts

An advantage of daisy chaining is that it implicitly assigns *priorities* to the various devices. Imagine a case where device 3 sends an interrupt signal at a certain point in time, followed a little later by an interrupt signal from device 2. When the IACK arrives from the control unit, device 2 sees it first and responds to it by (1) dropping its interrupt request signal and (2) placing the address of its service routine on the data bus. The control unit moves this address to the PC in step 6.2, and disables all the interrupts in step 6.3. The service routine for device 2 executes, then enables the interrupts and returns. At that point, the control unit starts checking interrupts again in step 6.1 (recall that they had been disabled while the service routine executed) and discovers that there is an interrupt request pending (the original IRQ from device 3). The control unit sends an IACK that is now forwarded all the way to device 3, which responds to it.

In a priority encoder configuration (Figure 1.17), interrupt signals from the currently-enabled interrupt sources are sent to a priority encoder (Section 7.7.1) and are also ORed and sent to the control unit. The IACK signal from the control unit is used to enable the priority encoder which then places the start address of the interrupt with the highest priority on the data bus. There is still the problem of making the highest-priority device drop its interrupt request signal. This can be done either by the priority encoder sending this device a fake IACK signal, or by the service routine. If the device is sophisticated enough, its interrupt service routine can send it a command to drop its interrupt request signal.

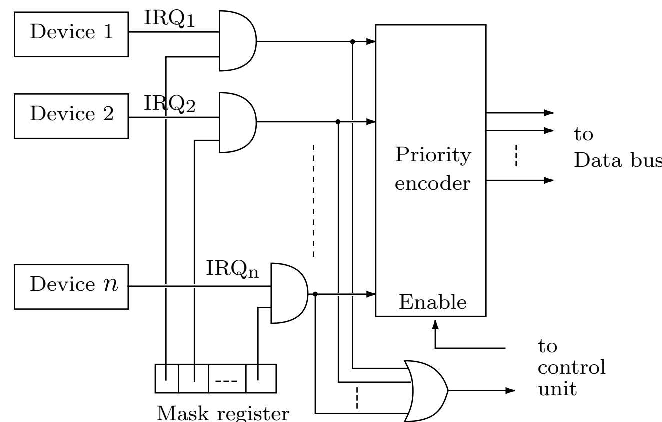



**Figure 1.17:** A priority-encoder interrupt system

While the service routine is executing, all interrupts are disabled, so the control unit does not check the (single) IRQ in step 6. As soon as the routine enables interrupts and returns (these two tasks must be done by one instruction), the control unit checks the IRQ line in step 6. If it is high, the control unit sends an IACK, which enables the priority encoder, and the entire process repeats.

Notice how the interrupt request signals from the various devices can be disabled by the mask register. Each bit in this register corresponds to one interrupting device. Any bit can be set or cleared by special instructions. If bit $i$ is zero, an interrupt request signal from device $i$ will not reach the priority encoder. Device $i$ will either drop its request signal eventually or wait until the program sets bit $i$ of the mask register.

Software interrupts have already been mentioned. This is a standard way for a user program to request something from, or to transfer control to, the operating system. The user program may need to know the time or the date, it may need more memory, or it may want to access a shared I/O device, such as a large disk. At the end of its execution, a user program should notify the operating system that it has finished (a normal termination). When the program discovers an error, it may decide to terminate abnormally. All these cases are handled by generating an artificial interrupt and a special code that describes the case. This interrupt is called a *break* or a *software interrupt*. It is generated by the program by executing a special instruction (such as `BRK`) with an operand that's a code. The interrupt is handled in the usual way and a handling routine (which is part of the operating system) is invoked. The routine handles the interrupt by examining the code and calling other parts of the operating system to perform the requested services.

### 1.7.1 Processor Modes

In a multiuser computer, the operating system performs important tasks such as protecting each user's area from all other users, allocating the processor to users, and supervising all I/O operations. We know, from long experience that computer users cannot be trusted. Given half a chance, some users will try to destroy someone else's data or even damage the operating system.

In order for the operating system to perform tasks that the users are not allowed to perform, it has be "more powerful" than the users' programs. However, the operating system is software. It can perform tasks only by executing instructions. Thus, the computer should have certain "privileged" instructions that only the operating system is allowed to execute. When a user tries to execute a privileged instruction, the control unit should sense it and interrupt the user program. Therefore, the hardware needs to know, at any time, whether the currently running program is a user program or an operating system routine. This is done by adding a hardware feature called the *processor mode*. This is a single bit (a flag), usually part of a special register called the *program status* (PS). A flag of zero indicates the system mode, and a flag of one, the user mode.

When the control unit decodes the current instruction, it checks the status of the instruction. If the instruction is privileged, and the processor mode is 1, the control unit issues an interrupt and skips the execution of the instruction (i.e., it goes directly to Step 6). This is how a user is prevented from executing a privileged instruction. The problem is to make sure that the mode flag always contains the correct value and that no user can change it from the user mode (flag of 1) to the system mode (zero flag). Here is how this is achieved:

When the computer is turned on (and also each time it is reset), the hardware sets the mode to "system" because the first program to execute is always one of the operating system's routines. When the operating system decides to start a user program, it changes the mode from "system" to "user" by means of an instruction. While the user's program is running, it cannot change the mode back to "system", because there is no instruction to do that. This is why a user cannot pretend to be the operating system. The mode changes to 'system' when the user program returns to the operating system. This can be done *only* by means of an interrupt, and this is why Step 6.4 sets the mode to "system."

A simple example of a privileged instruction is `HLT`. On a small, personal computer, where only one user program is loaded in memory and executed at any time, this instruction may exist and it is generated by the compiler as the last executable instruction in the program. It does not have to be privileged, since the user is allowed to stop his computer. On a large computer, however, with many user programs loaded in memory and competing for computer time, a user program is not supposed to stop the computer when it terminates, but should instead return control to the operating system. This is done by means of a software interrupt (a break) and it guarantees smooth, quick transition between programs. On such a computer, the `HLT` instruction is privileged so that it can be executed only by the operating system. It is, in fact, rarely executed. It is only necessary when a catastrophic error occurs, requiring manual intervention by the operator, or when the computer is stopped for scheduled maintenance.

Another example of a privileged instruction is any I/O instruction on a multiuser computer. Such a computer normally has large disk units that should be accessed only by the operating system, since they are shared by all users. A user program that needs I/O can only place a request to the operating system to perform the I/O operation, and the operating system is the only program in the computer that can access the shared I/O devices.

▸ **Exercise 1.13:** The operating system sets the mode to "user" by means of an instruction. Does this instruction have to be privileged?

> We interrupt this program to increase dramatic tension.
> —Joe Leahy (as the Announcer) in *Freakazoid!* (1995).

### 1.7.2 Time Slices

In a multiuser computer, several (or even many) users may be connected to the computer at any time, all expecting quick response. The computer has only one control unit, so it can do only one thing at a time. On the other hand, the computer is fast, so it can be switched between user programs many times each second. This switching is done by the operating system, using interrupts. The mechanism for switching employs the concept of *time slices*. A user program is allocated a time slice, lasting typically a millisecond, following which the processor is switched by the operating system to another user program. This way, in one second the computer can handle a thousand programs devoting a millisecond to each (or 100 programs, allocating 10 time slices to each program).

Figure 1.18 shows a typical allocation of time slices on a multiuser computer. At time 0, when the computer starts, the operating system is executed. It performs diagnostics, then loads user programs into memory, and finally starts the first user program on a time slice. At the end of the time slice control returns to the operating system, which spends a short amount of CPU time deciding who the next user will be, and switches the processor to that user program. The diagram assumes that three user programs, 1, 2, and 3, are initially loaded in memory and are assigned time slices. User program 2 finishes during its second time slice and is terminated, creating room in memory for user program 4. It is clear that the dark areas on the diagram, representing CPU times used by the operating system, constitute overhead and should be minimized.

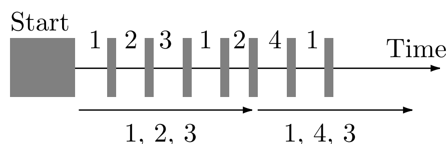



**Figure 1.18:** Time slices

The only problem is: Once a user program takes control, the operating system cannot stop it, because the operating system itself is a program, and it needs the use of the processor in order to do anything. The solution is to use interrupts. The hardware of a multiuser computer must include a timer, which is a special memory location whose content is decremented by special hardware typically once every microsecond. When the timer reaches zero, it issues an interrupt.

When the operating system decides to start a time slice, it loads the timer with the value 1000, and executes a jump to the user program. After 1000 microseconds (one millisecond) the timer reaches zero, a timer interrupt occurs, and control is switched to the interrupt handling routine, which is part of the operating system. The system then saves the return address (the address where the user program should be restarted in its next time slice) and all the registers, and allocates the next time slice to another user program.

Processor modes and time slices are just two examples of important features that are implemented by means of interrupts. They are two reasons why interrupts are so important. Interrupts provide a uniform mechanism for handling many different situations.

## 1.8 I/O Interrupts

The I/O processor is the third part of the mainframe, the part responsible for communications between the mainframe and the outside world (the I/O devices). There are several types of I/O processors, and they are all discussed in detail in Chapter 3. Here we discuss the use of interrupts for I/O.

Interrupt I/O is used for small quantities of I/O and is slow. The idea is to read a piece of data (usually one byte worth) from an input device into one of the registers in the processor. A register can also be output to an output device. This process is done under program control, it is therefore slow, and it also ties up the entire processor.

In the case of register I/O, The computer must have a special group of I/O instructions. The two most important ones are `IN` and `OUT`. To input a piece of data into register 6, the program should execute an instruction such as `IN 5,R6` where 5 is the number of the input device. Similarly, the instruction `OUT 15,R2` outputs the content of register 2 into device 15.

Since the processor is responsible for executing the machine instructions, it has to execute the I/O instructions too. These instructions, however, use a lot of processor time; they are slow. Register I/O is slow because of two reasons, the speed of the I/O devices and the need for verifying device status. I/O devices have moving parts and are, therefore, slow compared to the computer. Verifying device status is important and is a result of (1) the difference in speed between the processor and the I/O devices and (2) the nature of many input devices.

To understand the concept of device status. let's look at an example of a dot matrix printer. Assuming that the printer's device number is 15, an instruction such as `OUT 15,R2` will output one piece of data (the code of one character) to the printer. However, before the program sends the next character to the printer, it has to check status to make sure that the printer is ready for the next character. A typical dot matrix printer can print 100–150 characters per second. This is a very high speed, considering that the printing head has to strike the paper and then move the next printing position more than a hundred times a second. Still, compared to the processor speed, this is very slow. A typical processor can easily execute more than a million instructions per second, which translates to more than 10000 instructions in the time it takes to print one character.

As a result, the program has to wait until the printer completes printing the current character before the next character can be sent. This is done by a special instruction that tests the status of a device. The result may look something like:

```
WAIT: BNR 15,WAIT
      OUT 15,R2
```

The `BNR` instruction (Branch on Not Ready) branches if device 15 has a status of 'not ready'. Since the `BNR` branches to itself, this single instruction constitutes a loop. The loop is repeated until the printer returns a 'ready' status, at which point the next instruction, `OUT`, is executed. The discussion above shows that such a loop can iterate 10000 times or more, which explains why register I/O is so slow. It should be emphasized that—during such a loop—the processor has to execute the `BNR` instruction and is, therefore, tied up and cannot do anything else.

▸ **Exercise 1.14:** Laser printers are much more common today than dot matrix printers. A laser printer does not print individual characters but rather rows of small dots [typically 600 dots per inch (dpi)]. In such a case, is there still a need to check printer status?

A similar example is keyboard input. A keyboard is a slow device since few people can type more than 10 characters per second. It is also used for small quantities of input. Before issuing an `IN` instruction to the keyboard, the program has to check keyboard status to make sure that there is something in the keyboard to input. After all, the user may be sitting at the keyboard daydreaming, thinking, or hesitating. Even assuming the user to type at maximum speed (about 10 characters/second), the computer can easily execute 100000 instructions between the typing of two consecutive characters. Again the result is a tight loop where the processor has to spend a lot of time for each character being input:

```
LP: BNR 5,LP
    IN 5,R6
```

▸ **Exercise 1.15:** If register I/O is so slow, why is it ever used?

An interesting point is that there is no need to have special instructions, such as `BNR`, to check the status of an I/O device. In practice, a device such as a keyboard has two device numbers, one for status and the other for data. Assuming that the keyboard has device numbers 4 (for status) and 5 (for data), the program starts by reading the keyboard status (from device 4) and checking the status by means of a conditional branch. If the status is wrong, the program loops back, inputs status and and checks it again. Otherwise, it inputs the data from device 5. The following example assumes that the status is located in the leftmost bit of the number sent from device 4, and that bit can be either 0 (not ready) or 1 (ready).

The program fetches the status by an instruction of the form "`LP: IN 4,R2`". It then checks the status by examining the leftmost bit of register 2. If that bit is zero (not ready), the program jumps to label `LP` and fetches status again. This is done by a `BPL` (Branch on PLus) instruction of the form "`BPL LP`". If the bit is one, the program inputs the data from device 5 with an instruction such as "`IN 5,R3`".

To overcome some of the disadvantages of register I/O, computers use interrupt I/O. Returning to the case of the keyboard, if the keyboard can interrupt the computer, the program does not have to spend so much time on checking status. Instead, the program can run without worrying about the input, assuming that, when the keyboard has something to send, it will interrupt the computer. When the user presses a key on the keyboard, the keyboard senses it, prepares the character code (normally ASCII) for the key, and interrupts the computer. This transfers control from the program to a special interrupt handling routine. That routine should be prewritten but is only invoked when an interrupt occurs. When the routine is invoked, it assumes that the keyboard is ready and it executes an `IN` instruction without checking status. The routine stores the character in a buffer in memory and returns to the interrupted program.

This way of operation may result in a lot of processor time saved. There is, of course, the overhead associated with invoking the routine, returning to the main program, and with saving and restoring registers. Also, sometimes the program needs the input and cannot continue until it receives the entire message from the keyboard. In such a case the program may have to idle until the interrupt handling routine has received the full input and has stored it in memory.

Many large computers solve this problem in an elegant way. In such a computer, there may be several user programs stored in memory at any time, waiting to be executed. Since the computer has only one processor, it can only work on one program at a time. When such a program (program $A$) needs input and has to wait for it, the processor is switched by the operating system to another program (program $B$) and program $A$ waits in memory, in a state of hibernation (it is taken off time slices). Program $B$ is executed and, from time to time, when a keyboard interrupt occurs, the interrupt handling routine is invoked and reads one character.

Much later, when the entire input for $A$ has been read, the processor can be switched back to program $A$ and resume its execution. Such a way of operating a computer is called *multiprogramming*. It is a complex process and the operating system must be, in such a case, complex enough to control the switching between processes, the response to interrupts, the storing of input in memory, and many other tasks.

Interrupts can also be used in the case of output. The printer mentioned earlier can interrupt the processor as soon as it is completes printing the current character. The interrupt handling routine is invoked and should execute an `OUT` instruction, without checking status, to send the next character to the printer.

## 1.9 Interrupts in a Microprocessor

Interrupts are a very important feature of the computer. So much so that even the smallest microprocessor supports at least one interrupt. Typically, the control unit is designed with $n$ inputs for interrupt request lines, so the user can connect up to $n$ different sources of interrupts to the computer (Figure 1.19a). If there are more than $n$ interrupt sources, two or more interrupt request lines have to be connected to the same control unit input (Figure 1.19b). In such a case, the interrupt handling routine, when invoked, should first identify the source of the interrupt by interrogating all the interrupt sources connected to its input. This, of course, is time consuming and is avoided in a well-designed computer. The case where each input has at most one interrupt request line connected to it is called *vectored interrupts*.

Normally, a computer has a value of $n$ as large as possible. In a microprocessor, however, the situation is different. A microprocessor is a processor (a CPU) fabricated on one small silicon chip. The chip is placed in a plastic or ceramic package, and is connected to the outside world by means of pins coming out of the

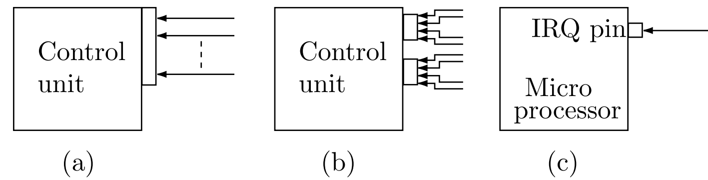



**Figure 1.19:** Connecting interrupt request lines

package. The number of pins is limited and the typical microprocessor is limited to just one or two interrupt request pins (Figure 1.19c).

The process of interrupting a microprocessor is summarized in Figure 1.20a. The process starts when an interrupt request (IRQ) signal is sent from an interrupting source to the IRQ pin in the microprocessor (1). The microprocessor completes the current instruction and then, if interrupts are enabled inside the microprocessor, sends an acknowledge signal through the IACK pin. The acknowledge signal arrives at all the interrupting devices (2) and it causes the one that has interrupted to drop the IRQ signal (3).

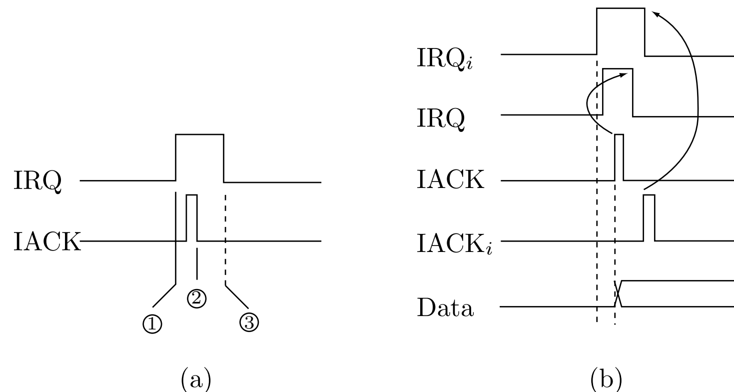



**Figure 1.20:** Timing diagrams for interrupting a microprocessor

## A PIC on a chip

In order to implement vectored interrupts in a microprocessor-based computer, an additional chip, a support chip, is used. It is called an interrupt controller or, if it can be programmed, a Programmable Interrupt Controller (a PIC). The name PIC sometimes stands for a Priority Interrupt Controller.

Only a programmable PIC is discussed here. Such a device has four main functions:

1. It has enough pins to accept vectored interrupts from $n$ sources.
2. Once it is interrupted, it interrupts the microprocessor and causes the right handling routine to be invoked.
3. If two or more sources interrupt the PIC at the same time, it decides which source has the highest priority, starts the microprocessor on that interrupt, and saves the other ones, to be serviced as soon as the microprocessor is done with the current interrupt. The interrupt priorities are not built-in and can be changed under program control.
4. If the program decides to ignore (disable) certain interrupts, it can program the PIC to do so.

The main advantage of the PIC is point 2 above. The PIC identifies the specific interrupt source and decides which handling routine should be invoked. To actually invoke the routine, the PIC sends its start address to the microprocessor. Since the handling routines may be written by the user, they can start anywhere in memory, so the PIC cannot know their start addresses in advance. The program should, therefore, send the start addresses of all the handling routines to the PIC. This is done by means of commands sent to the PIC early in the program.

The PIC commands are sent by the microprocessor as regular output. The PIC therefore should have a device select number, much like any other I/O device. Since the commands represent a low-volume I/O, they are sent to the PIC using register I/O.

Figure 1.21 shows a typical PIC connected in a microcomputer system. The address lines carry the device select number to the PIC in the same way they carry it to all other I/O devices. The commands are sent to the PIC on the data bus and the start addresses are sent, by the PIC, on the same bus, in the opposite direction. The PIC interrupts the microprocessor through the IRQ pin and then waits for an acknowledge from the microprocessor on the IACK pin. At that point, the PIC sends an acknowledge pulse to the interrupting device. Figure 1.20b is a timing diagram summarizing the individual steps involved in a typical interrupt.

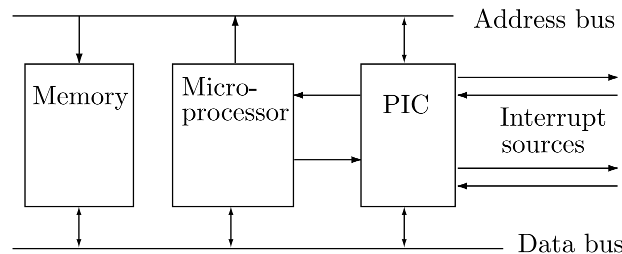



**Figure 1.21:** Interfacing a PIC to a microprocessor

A good example of a common, simple PIC is the Intel 8259. In addition to the features described here, the 8259 has a master-slave feature, allowing up to nine identical PICs to be connected to one microprocessor in a configuration that can service up to 64 interrupts. For more information on the 8259, see [Intel 87] and [Osborne and Kane 81].

## 1.10 The ALU

As mentioned earlier, the ALU (Chapter 9) is a collection of circuits that perform various operations on numbers. There are three types of operations: Arithmetic, Logical, and Shifts. They are all discussed in chapter 2 in connection with machine instructions. In this section we examine some of the simple ALU circuits to get an idea of how they operate.

The ALU has two main inputs and one main output. It also has an input for a function code, and it can access the status flags (figure 1.22). The two main inputs (usually called left and right) are the numbers to be operated on (the operands). Note, however, that some operations only require one operand.

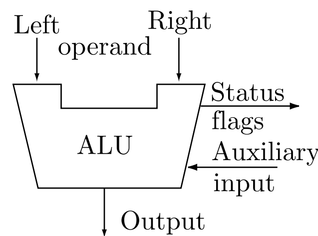



**Figure 1.22:** ALU inputs and outputs

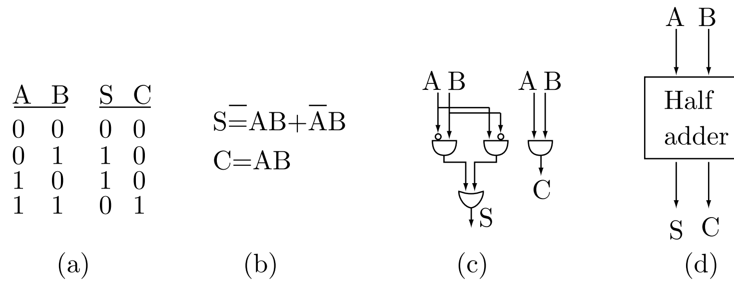
The first example is addition, which is one of the most important and common operations. To add two binary numbers, we first have to know how to add two bits, which involves the following steps;

1. The truth table should be written (figure 1.23b). It shows the output for every possible combination of the inputs.

2. The truth table is analyzed and the outputs are expressed as logical functions of the inputs.

3. The circuit is designed out of logical elements such as gates and flip-flops. Such a circuit is called a *half adder*.

The half-adder is a very simple circuit but can only add a pair of bits. It cannot be used for adding numbers. Adding binary numbers involves two operations, adding pairs of bits and propagating the carry.

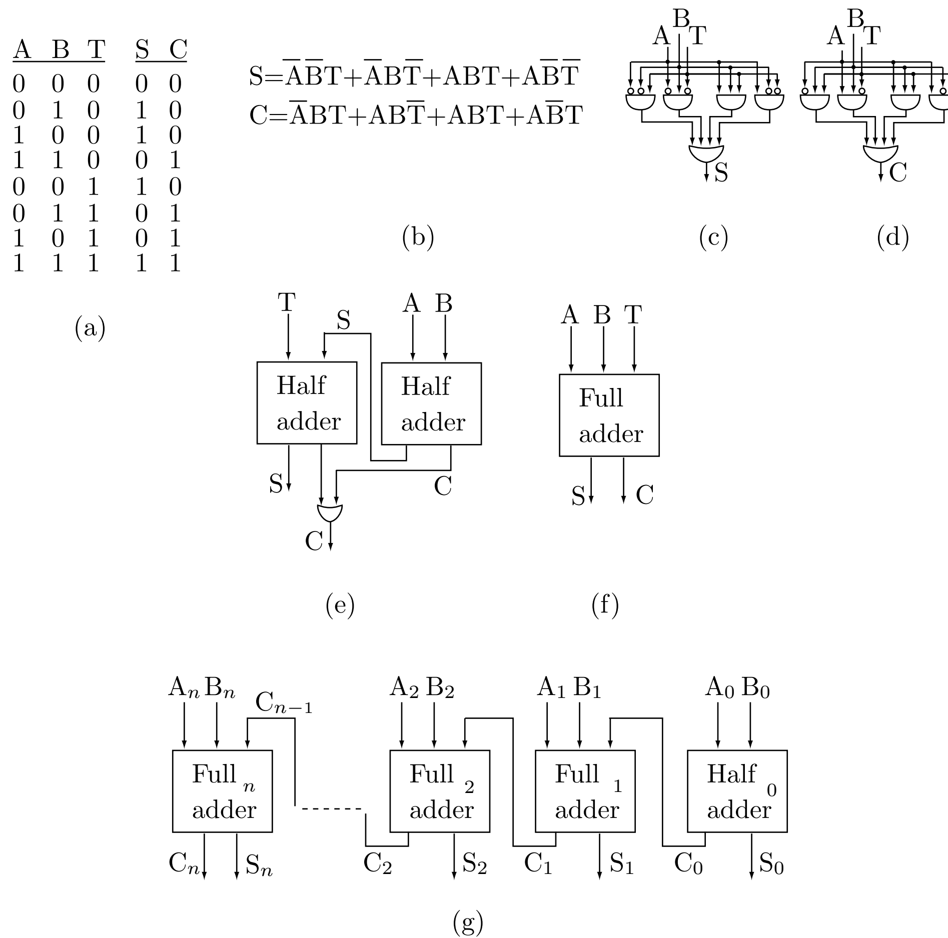


| A | B | S | C |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |



**Figure 1.23:** A half adder

To propagate the carry we need a circuit that can add two bits and a carry. Unlike the half-adder, such a circuit has *three* inputs and two outputs, it is called a *full adder*, and its truth table and logical design are shown in figure 1.24a–d. Figure 1.24e,f shows how a full-adder can be built by combining two half-adders.



| A | B | T | S | C |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 | 1 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 |



**Figure 1.24:** A full adder

The full adder can serve as the basis for an adding device that can add $n$-bit numbers. Figure 1.24g shows the so-called *parallel adder*. This device is a simple combination of one half-adder and $n-1$ full-adders, where the carry that is output by adder $i$ is propagated to become the third input of full-adder $i+1$. In a truly parallel adder, the execution time is fixed, but the execution time of our parallel adder is proportional to $n$, so this device is not truly parallel. To understand how it works, let's concentrate on the two rightmost

circuits, the half-adder 0 and the full-adder 1. They receive their inputs simultaneously and their outputs are therefore ready at the same time. However, the outputs of the the full-adder 1 are initially wrong since this device hasn't received the correct input $C_0$. Once the right $C_0$ is generated by half-adder 0 and is propagated to full-adder 1, the outputs of that full-adder become correct, and its $C_1$ output propagates to full-adder 2, that has, up until now, produced bad outputs.

Abstraction is our only mental tool to master complexity.
—Edsger Dijkstra

# 2
# Machine Instructions

An understanding of machine instructions is a prerequisite to an understanding of computer architecture. This is because the set of machine instructions reflects the architecture of the computer. This chapter discusses the important features and attributes of machine instructions, such as formats, sizes, opcodes, and addressing modes. The last few sections discuss data types—such as integer, floating-point, fixed-point, and BCD—and their properties.

## 2.1 Instruction Formats

One of the first lessons a beginner learns about computers is that in order to use a computer, a program must be written, translated into machine language, and loaded into memory; that such a program consists of machine instructions, and that a machine instruction is a number.

In this chapter we look at the details of machine instructions, and in particular the fact that a machine instruction is rarely a single number. It normally consists of several numbers that are assembled (placed together) by the assembler, to become the fields of the complete instruction.

As a result, any machine instruction has a certain format. The instruction is divided into fields, each a number. A machine instruction must contain at least one field, namely the operation code (opcode), that tells the control unit what the instruction is supposed to do. Most instructions contain other fields specifying registers, memory addresses, immediate quantities, addressing modes, and other parameters.

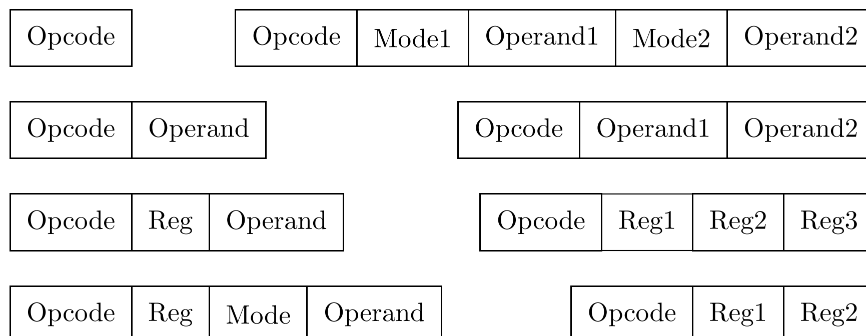



**Figure 2.1:** Instruction formats

Figure 2.1 illustrates typical instruction formats. They range from the simplest to the very complex and they demonstrate the following properties of machine instructions:

▪ Instructions can have different sizes. The size depends on the number and nature of the individual fields.

▪ The opcode can have different sizes. This property is discussed in Section 2.2. The opcode can also be broken up into a number of separate fields.

▪ A field containing an address is much larger than a register field. This is because a register number is small, typically 3–4 bits, while an address is large (typically 25–35 bits).

▪ Fields containing immediate operands (numbers used as data) can have different sizes. Experience shows that most data items used by programs are small. Thus, a well-designed instruction set should allow for both small and large data items, resulting in short instructions whenever possible.

The last point, about short instructions, is important and leads us to the next topic of discussion, namely the properties of a good instruction set. When a new computer is designed, one of the first features that has to be determined and fully specified is the instruction set. First, the instruction set has to be designed as a whole, then the designers decide what instructions should be included in the set. The general form of the instruction set depends on features such as the intended applications of the computer, the word size, address size, register set, memory organization, and bus organization. Instruction sets vary greatly, but a good design for an instruction set is based on a small number of principles that are independent of the particular computer in question. These principles demand that an instruction set should satisfy the following requirements:

1. Short instructions.
2. An instruction size that is both compatible with the word size, and variable.

**1. Short instructions:** Why should machine instructions be short? Not because short instructions occupy less memory. Memory isn't expensive and modern computers support large memories. Also, not because short instructions execute faster. The execution time of an instruction has nothing to do with its size. A "register divide" instruction, to divide the contents of two floating-point registers, is short, since it only has to specify the registers involved. It takes, however, a long time to execute.

▸ **Exercise 2.1:** Find an example of a long instruction with fast execution.

The reason why short instructions are preferable is that it takes less time to *fetch* them from memory. An instruction that is longer than one memory word has to be stored in at least two words. It therefore takes two memory cycles to fetch it from memory. Cutting the size of an instruction such that it fits in one word, cuts down the fetch time. Even though the fetch time is short (it is measured in nanoseconds), many instructions are located inside loops and have to be executed (and therefore fetched) many times. Also, since memory is slower than the processor, speeding up the instruction fetch can speed up the entire computer.

**2. Instruction size:** The instruction size should also be compatible with the computer's word size. The best design results in instruction sizes of $N$, $2N$ and $3N$ where $N$ is the word size. Instruction sizes such as $1.2N$ or $0.7N$ do not make any sense, since each memory read brings in exactly one word. In a computer with long words, several instructions can be packed in one word and, as a result, instruction sizes of $N/2$, $N/3$, and $N/4$ also make sense.

The two requirements above are satisfied by the use of variable-size opcodes and addressing modes. These two important concepts are discussed next.

## 2.2 The Opcode Size

If the instruction is to be short, individual fields in the instruction should be short. In this section we look at the opcode field. Obviously, the opcode cannot be too short or there would not be enough codes for all the instructions. An opcode size of 6–8 bits, allowing for 64–256 instructions, is common. Most modern computers, however, use variable size opcodes, for two good reasons. One reason has to do with the instruction size in relation to the word size, and the other has to do with future extensions of the instruction set.

The first reason is easy to understand. We want our instructions to be short, but some instructions have to contain more information than others and are naturally longer. If the opcode size varies, longer instructions can be assigned short opcodes. Other instructions, with short operands, can be assigned longer opcodes. This way the instruction size can be fine-tuned to fit in precisely $N$ or $2N$ bits.

The second advantage of variable-size opcodes has to do with extensions to an original instruction set. When a computer becomes successful and sells well, the manufacturer may decide to come up with a new, extended, and upward compatible version of the original computer. The 68000, 80x86, and Pentium families are familiar examples.

Upward compatibility means that any program running on the original computer should also run on the new one. The instruction set of the new computer must therefore be an extension of the original instruction set. With variable-size opcodes, such an extension is easy.

When an instruction is fetched, the hardware does not know what its opcode is. It has to decode the instruction first. In a computer with variable-size opcodes, when an instruction is fetched, the control unit does not even know how long the opcode is. It has to start by identifying the opcode size, following which it can decode the instruction. Thus, with variable-size opcodes, the control unit has to work harder.

Three methods to implement variable-size opcodes are described here.

**Prefix codes.** We start by simply choosing several opcodes with the sizes that we need. For example, if we need a 3-bit opcode in order to adjust the size of instruction $A$ to one word, a 6-bit opcode to adjust the size of instruction $B$ to one word, and other opcodes with sizes 2, 4, and 7 bits, we can select the set of opcodes 000, 101010, 11, 0001, and 1110001. A little thinking, however, shows that our opcodes cannot be decoded uniquely. Imagine an instruction with opcode 0001 being fetched in Step 1 of the fetch-execute cycle and decoded in Step 3. The control unit examines the first few bits of the instruction and finds them to be 0001.... It does not know whether the opcode is the four bits 0001 or the three bits 000 (with the 1 being the start of the next field). The problem exists because we selected the bit pattern 000 as both a code *and* the start of another code. The solution is to avoid such a situation. We have to design our opcodes following the simple rule:

Once a bit pattern has been assigned as an opcode, no other opcode should start with that pattern

The five opcodes above can now be redesigned as, for example, 000, 101010, 11, 100, and 0100100. We say that these opcodes satisfy the *prefix property*, or that they are *prefix codes*. These codes are discussed in more detail in Section 3.11. It is easy to see, by checking each of the five codes, that they can be uniquely decoded. When the control unit finds, for example, the pattern 000 it knows that this must be the first code, since no other codes start with 000. When it finds 10..., it knows that this must be either code 2 or code 4. It checks the next bit to find out which code it is.

**Fixed prefix.** The first 2–4 bits of the instruction tell what the opcode size is (and even what the instruction format is). Figure 2.2 shows four instruction formats, denoted by 0–3 and identified by the first two opcode bits. When the control unit fetches an instruction that starts with 00, it knows that the next three bits are the opcode, so there can be eight instructions with a 5-bit opcode (including the identification bits). Similarly, when the control unit fetches an instruction that starts with 01, it knows that the opcode consists of the next two bits plus three more bits elsewhere in the instruction, so there can be 32 instructions with a 7-bit opcode (including the two identification bits).

▸ **Exercise 2.2:** How many instructions can there be with the other two opcode sizes?

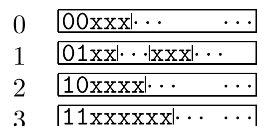



**Figure 2.2:** Opcodes with two identification bits

The average size of an opcode in this instruction set is therefore:

$$\frac{8\times 5+32\times 7+16\times 6+64\times 8}{8+32+16+64}=\frac{872}{120}\approx 7.26 \text{ bits.}$$

With fixed-size opcodes, the opcode size for a set of 120 instructions would be seven bits. The advantage of this method is that several opcode sizes are available and the opcode size can easily be determined. The disadvantages are: (1) the instruction set cannot be extended; there is no room for new opcodes and (2) because of the prefix, the average opcode is longer than in the case of fixed-size opcodes.

**Reserved patterns.** We can start with short, 2-bit opcodes. There are four of them, but if we assign all four, there would be no way to distinguish between say, the 2-bit opcode '01' and a longer opcode of the form '01xxx' (review the prefix rule above). Therefore, we reserve one of the four 2-bit opcodes to be the prefix of all the longer opcodes. If we reserve '00', then all the long opcodes have to start with '00'.

Suppose, as an example, that we need opcodes of sizes 2, 5, 8, and 13 bits. We reserve the 2-bit pattern 00 as the prefix of longer opcodes. This leaves the three 2-bit opcodes 01, 10, and 11. The 5-bit opcodes must, therefore, have the form $00xxx$, and one of them, say 00000, has to be reserved, so we end up with seven such opcodes. Similarly, the 8-bit opcodes must have the form $00000xxx$ with one pattern reserved, leaving seven 8-bit opcodes. Finally, the 13-bit opcodes must all start with eight zeros, so only five of the 13 bits vary. This allows for 32 opcodes, of which one should be reserved for future extensions.

| 2-bit | 5-bit | 8-bit | 13-bit |
|---|---|---|---|
| 00 | 00 000 | 00000 000 | 00000000 00000 |
| 01 | 00 001 | 00000 001 | 00000000 00001 |
| 10 | 00 010 | 00000 010 | 00000000 00010 |
| 11 | 00 011 | 00000 011 | 00000000 00011 |
| | ⋮ | ⋮ | 00000000 00100 |
| | 00 111 | 00000 111 | ⋮ |
| | | | ⋮ |
| | | | 00000000 11111 |

We thus end up with 48 opcodes whose average size is:

$$\frac{2\times 3+5\times 7+8\times 7+13\times 31}{3+7+7+31}=\frac{500}{48}\approx 10.42 \text{ bits.}$$

This is much longer than the six bits needed for a similar set of 48 fixed-size opcodes.

The advantages of this method are (1) several opcode sizes are available and (2) the instruction set can be extended, since more (longer) opcodes can be created and assigned to the new instructions. The only disadvantage is the large average size of the opcodes.

## 2.3 Addressing Modes

This is an important feature of computers. We start with the known fact that many instructions have to include addresses; the instructions should be short, but addresses tend to be long. Addressing modes are a solution to this problem. Using addressing modes, a short instruction may specify a long address.

The idea is that the instruction no longer includes the full address (also called the *effective address*, or EA), but instead contains a number, normally called a *displacement*, that's closely related to the address. Another field, an *addressing mode*, is also added to the instruction. The addressing mode is a code that tells the control unit how to obtain the effective address from the displacement. It is more accurate to say that the addressing mode is a *rule of calculation*, or a function, that uses the displacement as its main argument, and other hardware components—such as the PC, the registers, and memory locations—as secondary arguments, and produces the EA as a result.

The notation $\text{EA}=f_{\text{M}}(\text{Disp},\text{PC},\text{Regs},\text{Mem})$, where the subscript M indicates the mode is convenient. For different modes there are different functions, but the idea is the same.

Before any individual modes are discussed, it may be useful to look at some of the numbers involved when computers, memories, and instructions are designed.

Up until the mid 1970s, memories were expensive, and computers supported small memories. A typical memory size in a second generation computer was 16K–32K words (24–48 bits/word), and in an early third generation computer, 32K–64K words (48–60 bits/word). Today, with much lower hardware prices, modern computers can access much larger memories. Most of the early microprocessors could address 64K bytes, and most of today's microprocessors can address between 32M words and a few tera words (usually 8-bit words, bytes). As a result, those computers must handle long addresses. Since 1M (1 mega) is defined as $1024\,\text{K}=1024\times 2^{10}=2^{20}$, an address in a 1M memory is 20 bits long. In a 32M memory, an address is

25 bits long. The Motorola 68000 microprocessor [Kane 81] and the Intel 80386 [Intel 85] generate 32-bit addresses, and can thus physically address 4G (giga) bytes. (their virtual address space, though, is 64 tera bytes, $= 2^{46}$.) Computers on the drawing boards right now could require much longer addresses.

Let's therefore assume a range of 25–35 bits for a typical address size, which results in the following representative instruction formats

| Opcode | Reg | Operand |
|---|---|---|
| 6–8 | 3–6 | 25–35 |

| Opcode | Reg1 | Reg2 | Operand |
|---|---|---|---|
| 6–8 | 3–6 | 3–6 | 25–35 |

The operand field in those formats makes up about 63–73% of the total instruction size, and is therefore the main contributor to long instructions. A good instruction design should result in much shorter operands, and this is achieved by the use of addressing modes.

When an addressing mode is used, the instruction contains a displacement and a mode instead of an EA. The mode field is 3–4 bits long, and the result is the following typical format:

| Opcode | Reg | Mode | Displacement |
|---|---|---|---|
| 6–8 | 3–6 | 3–4 | 8–12 |

The operand (the mode and displacement fields) is now 11–16 bits long. It is still the largest field in the instruction, making up 50–55% of the instruction size, but is considerably shorter than before. Many instructions do not need an address operand and therefore do not have an addressing mode. They do not have a mode field and are therefore short.

There is, of course, a tradeoff. If an instruction uses a mode, the EA has to be calculated, in Step 4 of the fetch-execute cycle, before the instruction can be executed (in Step 5), which takes time. However, this calculation is done by the control unit, so it is fast.

Before looking at specific modes, another important property of addressing modes should be mentioned. They serve to make the machine instructions more powerful. We will see examples (such as those in Section 2.10.9) where a single instruction, using a powerful mode, does the work of two instructions.

The five main addressing modes, found on all modern computers and on many old ones, are direct, relative, immediate, index, and indirect. They are discussed here in detail.

## 2.4 The Direct Mode

This is the case where the displacement field is large enough to contain the EA. There is no need for any calculations, and the EA is simply the displacement. The definition of this simple mode is EA←Displacement, but this is not a very useful mode because it does not result in a short instruction (it is like not having an addressing mode at all). Nevertheless, if the assembler cannot assemble an instruction using any other mode, it has to use the direct mode, .

## 2.5 The Relative Mode

In this mode, the displacement field is set by the assembler to the distance between the instruction and its operand. This is a useful mode that is often selected by the assembler as a first choice when the user does not explicitly specify any other mode. It also results in an instruction that does not require relocation by the loader.

Using the concept of a function mentioned earlier, the definition of this mode is EA = Disp + PC. This means that, at run time, before the instruction can be executed, the control unit has to add the displacement and the PC to obtain the EA. The control unit can easily do that, and the only problem is to figure out the displacement in the first place. This is done by the compiler (or assembler) in a simple way. The compiler maintains a variable called the *location counter* (LC) that points to the instruction currently being compiled. Also, at run time, the PC always contains the address of the next instruction. Therefore the expression above can be written as Disp = EA − PC, which implies

$$\text{Disp} = \text{EA} - (\text{LC} + \text{size of current instr.}) = \text{EA} - \text{LC} - \text{size of current instr.}$$

Example: The simple JUMP instruction "`JMP B`". We assume that the `JMP` instruction is assembled and loaded into memory location 57, and that the value of symbol `B` is address 19. The expression above implies the displacement should be

$$\text{Disp} = 19 - 57 - 1 = -39.$$

Notice that the displacement is negative. A little thinking shows that the displacement is negative whenever the operand precedes the instruction. Thus, in this mode, the displacement should be a signed number. Since the sign uses one of the displacement bits, the maximum value of a signed displacement is only half that of an unsigned one. An 8-bit displacement, for instance, is in the range $[0, 255]$ if it is unsigned, but in the shifted interval $[-128, +127]$ if it is signed. The ranges have the same size, but the maximum values are different.

This example also illustrates the absolute nature of this mode. The displacement is essentially the distance between the instruction and its operand, and this distance does not depend on the start address of the program. We say that this mode generates *position independent code* and an instruction using it does not have to be relocated.

## 2.6 The Immediate Mode

This mode is used when the instruction requires the value of a constant, not the contents of a memory location. An `ADD` instruction, for example, is often written "`ADD R3,XY`" and it adds the contents of location `XY` to register 3. If, however, the programmer wants to add the number 67 to register 3, the same instruction can be used in the immediate mode, where it is typically written "`ADD R3,#67`". The number sign (or pound sign) "#" is usually used to indicate the immediate mode. Assuming that the code of this mode is 2, the instruction above will be assembled into

| opcode | 3 | 2 | 67 |
|---|---|---|---|

, where 3 is the register number, 2 is the mode, and 67 is the immediate operand.

The immediate quantity (the number to be used by the instruction) should be small enough to fit in the displacement field. This mode always generates an absolute instruction and it is different from all the other modes because it does not involve any EA. We say that in this mode, the operand is located in the instruction, not in memory.

## 2.7 The Index Mode

This mode is especially useful in loops, when the same operation is performed on the elements of an array. It uses an *index register*, which can normally be any general-purpose register (although on some computers only certain registers can be used as index registers). The EA function in this mode is EA = Disp + Index. This mode should be explicitly specified by the instruction; it is not automatically selected by the assembler. For example: "`LOD R1,0(R5)`". The displacement is 0, and the index register is `R5`. Before the loop starts, `R5` should be set to some value, most likely the beginning address of the array. `R5` should be incremented in each iteration, so that it points to the next array element. The loop becomes:

```
      LOD   R5,M          Load the start address of the array
LUP   LOD   R1,0(R5)      Load the next array element into R1
       .
       .
      INC   R5            Update the index register
      CMP   R5,#LEN       Compare it with the length of the array.
      BLT   LUP           Branch on Less Than
       .
       .
LEN   EQU   25            LEN is the array size
ARY   RES   LEN           ARY is the array itself
M     DAT   ARY           Location M contains the value of symbol ARY
```

▸ **Exercise 2.3:** What instruction can be used instead of "`LOD R5,M`" above, to load the value of `ARY`?

The index mode can also be used in a different way, as in "`LOD R1,ARY(R5)`" where `ARY` is the start address of the array and `R5` is initialized to 0. In this case, the index register really contains the index of the current array element used. This form can be used if the start address of array `ARY` fits in the displacement field.

## 2.8 The Indirect Mode

This is a more complex mode, requiring the control unit to work harder in order to calculate the EA. The rule of calculation is EA=Mem[disp] meaning, the control unit should fetch the content of the memory location whose address is disp., and that content is the EA. Obviously, this mode is slower than the ones discussed so far, since it involves a memory read. The EA has to be read from memory (in Step 4 of the fetch-execute cycle) before execution of the instruction can start. A simple example is

```
LC                         Obj.  Code
-----------------------------------------
            .              Opc m disp
 24        JMP   @TO       xxx 3  124
            .
            .
124   TO   DC    11387
            .
```

The at-sign "@" instructs the assembler to use the indirect mode (other symbols, such as a dollar sign, may be used by different assemblers). The value of TO (address 124) is the *indirect address* and, in the simplest version of the indirect mode, it has to be small enough to fit in the displacement field. This is one reason why several versions of this mode exist (see below). The EA in our example is 11387, the content of memory location 124.

What's the use of this mode? As this is a discussion of computer organization, and not of assembly language programming, a complete answer cannot be provided here. However, we can mention a common case where this mode is used, namely a return from a procedure. When a procedure is called, the return address has to be saved. Most computers save the return address in the stack, but some old computers save it in the first word of the procedure (in which case the first executable instruction of the procedure should be stored in the second word, see page 16). If the latter method is used, a return from the procedure is done by a jump to the memory location whose address is contained in the first word of the procedure. This is therefore a classical, simple application of the indirect mode. If the procedure name is P, then an instruction such as "JMP @P" (where "@" specifies the indirect mode) can easily accomplish the job

Incidentally, if the return address is saved in the stack, a special instruction, such as RET, is necessary to return from the procedure. Such an instruction should jump to the memory location whose address is contained at the top of the stack, and also remove that address from the stack. Thus, a RET instruction uses a combination of the indirect and stack modes.

Common extensions of the indirect mode combine it with either the relative or the index modes. The JMP instruction above could be assembled into "xxx 3 99", since the value of TO relative to the JMP instruction is $124 - 24 - 1 = 99$. We assume that the size of the JMP instruction is one word and that mode 3 is the combination indirect-relative. A combination indirect-index can also be used and, in fact, the (now obsolete) 6502 microprocessor used two such combinations, a pre-indexed indirect (that can only be used with index register X), and a post-indexed one (that can only be used with index register Y). Their definitions are

$$\text{Pre-indexed indirect: EA} = \text{Mem[disp} + \text{X]},$$
$$\text{Post-indexed indirect: EA} = \text{Mem[disp]} + \text{Y}.$$

In the former version, the indexing (disp + X) is done first, followed by the indirect operation (the memory read). In the latter version, the indirect is done first and the indexing ($\cdots$+Y) follows. [Leventhal 79] illustrates the use of those modes. The two instructions "LDA ($80,X)" and "LDA ($80),Y" are typical examples. The dollar sign "$" stands for hexadecimal and the parentheses imply the indirect mode. It is interesting to note that in order to keep the instructions short, there is no mode field in the 6502 instructions, and the mode is implied in the opcode. Thus, an instruction may have several opcodes, one for each valid mode. The two instructions above have opcodes of A1 and B1 (hexadecimal), respectively.

## 2.9 Multilevel or Cascaded Indirect

This is a rare version of the basic indirect mode. It is suitable for a computer with 16-bit words and 15-bit addresses (or, in general, $N$-bit words and $(N-1)$-bit addresses). The extra bit in such a word serves as a flag. A flag of 1 indicates that another level of indirect exists. The original instruction contains an indirect address `IA`, and the control unit examines the contents of `Mem[IA]`. If the flag (the leftmost bit of `Mem[IA]`) is 1, the control unit interprets the remaining bits as another indirect address. This process continues until the control unit gets to a memory location where the flag is zero (Figure 2.3). The remaining bits of that location constitute the EA, and the instruction is then executed. The old HP1000 and HP2100 minicomputers [Hewlett-Packard 72] are examples of this rare addressing mode.

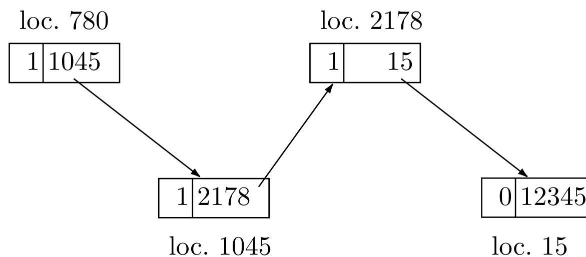



**Figure 2.3:** Cascaded indirect

▸ **Exercise 2.4:** How can the programmer generate both an address and a flag in a memory word?

## 2.10 Other Addressing Modes

Computers support many other modes, some simpler and others more powerful than the basic modes described so far. This section describes a few other modes, not trying to provide a complete list but only to illustrate what can be found on different computers.

### 2.10.1 Zero Page Mode

This is a version of the direct mode. If the displacement field is $N$ bits long, the (unsigned) displacement can have $2^N$ values. Memory may be divided into pages, each $2^N$ words long, and page zero—the first memory page—is special in the sense that any instruction accessing it uses a small address that fits in the displacement field. Hence the name *zero page mode*. Good examples are the 6502 microprocessor [Leventhal 79] and the DEC PDP-8 [Digital Equipment 68].

### 2.10.2 Current Page Direct Mode

The DEC PDP-8 was a 12-bit minicomputer. It had 12-bit addresses and thus a 4K-word memory. Memory is divided into pages, each 128 words long. The first memory page, addresses 0–127, was called *base page*. Many instructions have the format:

| opcode | m | disp | |
|---|---|---|---|
| 4 | 1 | 7 | = 12 bits |

If m=0, the direct mode is used and the EA is the displacement; this is zero page addressing. However, if m=1, the EA becomes the logical OR of the five most significant bits of the PC and the seven bits of the displacement, *pppppddddddd*. The displacement in this case is the address in the current page, and the mode is called current page direct mode.

### 2.10.3 Implicit or Implied Mode

This is the simple case where the instruction has no operands and is not using any mode. Instructions such as `HLT` or `EI` (enable all interrupts) are good examples. Those instructions do not use any modes, but the manufacturer's literature refers to them often as using the implicit or implied mode.

### 2.10.4 Accumulator Mode

In a computer with a single working register, the accumulator, many instructions implicitly use the accumulator. They don't have a mode field and should perhaps be considered "modeless." However, some textbooks and manufacturers' manuals refer to them as using the accumulator mode.

### 2.10.5 Stack Mode

In a computer with a stack, some instructions use the stack and should update the stack pointer. Such instructions are said to use the *stack mode*, even though they need not have a separate mode field. The use of the stack is implied in the opcode. Examples are `POP` and `PUSH`. The former pops a data item from the stack, then updates the stack pointer to point to the new top-of-stack. The latter updates the stack pointer to point to the next available stack location, then pushes the new data item into the stack.

▸ **Exercise 2.5:** : A stack is a LIFO data structure, which implies using the top element. What could be a reason for accessing a stack element other than the top?

### 2.10.6 Stack Relative Mode

This is a combination of the relative mode and the stack mode. In this mode, EA = disp + SP where SP is the stack pointer, a register that always points to the top of the stack. This mode allows access to any stack element, not just the one at the top.

### 2.10.7 Register Mode

This is the case where the instruction specifies a register that contains the operand. This mode does not use an EA and there is no memory access.

### 2.10.8 Register Indirect Mode

In this mode, the register contains the EA; it points to the operand in memory.

### 2.10.9 Auto Increment/Decrement Mode

This is a powerful version of the index mode. The control unit uses the index register to calculate the EA, then increments or decrements the register. This is an example of a powerful addressing mode, because an instruction using this mode can do the work of two instructions. The PDP-11 instruction "`CLR (R5)+`" is executed by (1) use `R5` as an index (it contains the EA, so it points to the operand in memory), (2) execute the instruction (clear the operand, a memory word), and (3) finally, *increment* the index register so that it points to the next word in memory. This is clearly a powerful mode, since without it, another instruction would be needed, to increment the register. Similarly, the instruction "`INC -(R5)`" starts by decrementing the index register, then using it as an index, pointing to the memory word that is to be INCremented.

In the DEC PDP-11, memory is physically divided into bytes, and a word consists of two consecutive bytes. The instructions above operate on a word and, therefore, the index is updated by 2, not by 1, so it points to the next/previous word. In an instruction such as `CLRB` (clear a byte), the register is updated by 1.

In the Nova minicomputer [Data General 69] [Data General 70], memory locations 16–31 are special. Locations 16–23 are auto increment and locations 24–31 are auto decrement. When any of those locations is accessed by an indirect instruction, the computer first increments/decrements that location, then uses it to calculate the effective address.

Figure 2.4 is a graphic representation of 11 of the modes discussed here.

## 2.11 Instruction Types

Modern computers support many instructions, sometimes about 300 different instruction, with features such as variable-size opcodes and complex addressing modes. At the same time, the instructions in a typical instruction set can be classified into a small number of classes. The instruction types discussed here are general and are not those of any particular computer. Some—such as data movement instructions—can be found on every computer while others—such as three-address instructions—are rare.

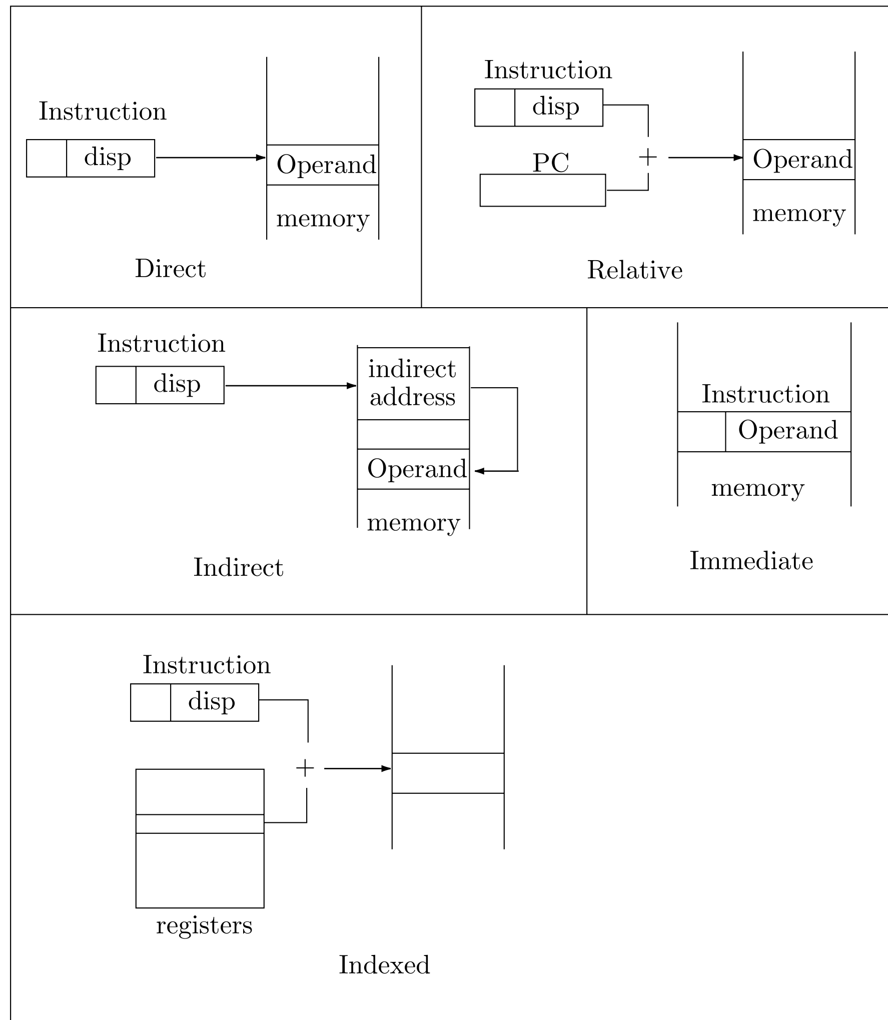



**Figure 2.4:** Addressing modes (part 1 of 2)

## 2.12 Data Movement Instructions

These are simple instructions that just move (or transfer) data from place to place inside the computer. The fact that they are important and are found in every computer illustrates how the processor works. Fetching and executing instructions by the control unit involves many data transfers in which the data items are moved but no operations are performed on them.

Data can be moved in one of four ways, as shown in Figure 2.5

1. A movement from memory to one of the register. This is traditionally called a `LOAD`.
2. A movement in the opposite direction is called a `STORE`.
3. Moving data inside memory, from one location to another, is normally called a `MOVE`.
4. Moving data between registers is sometimes called a `TRAnsfer`.

Traditions are not always adhered to, and in many computers the data movement instructions have different names. Some computers have only one such instruction, called `MOV`, that is used for all data movements. Also, some computers have instructions that perform simple operations on data while moving it. The IBM/360 had instructions such as "`LPR R1,R2`", an instruction that moves the absolute value of register 2 to register 1 ($R1 \leftarrow |R2|$).

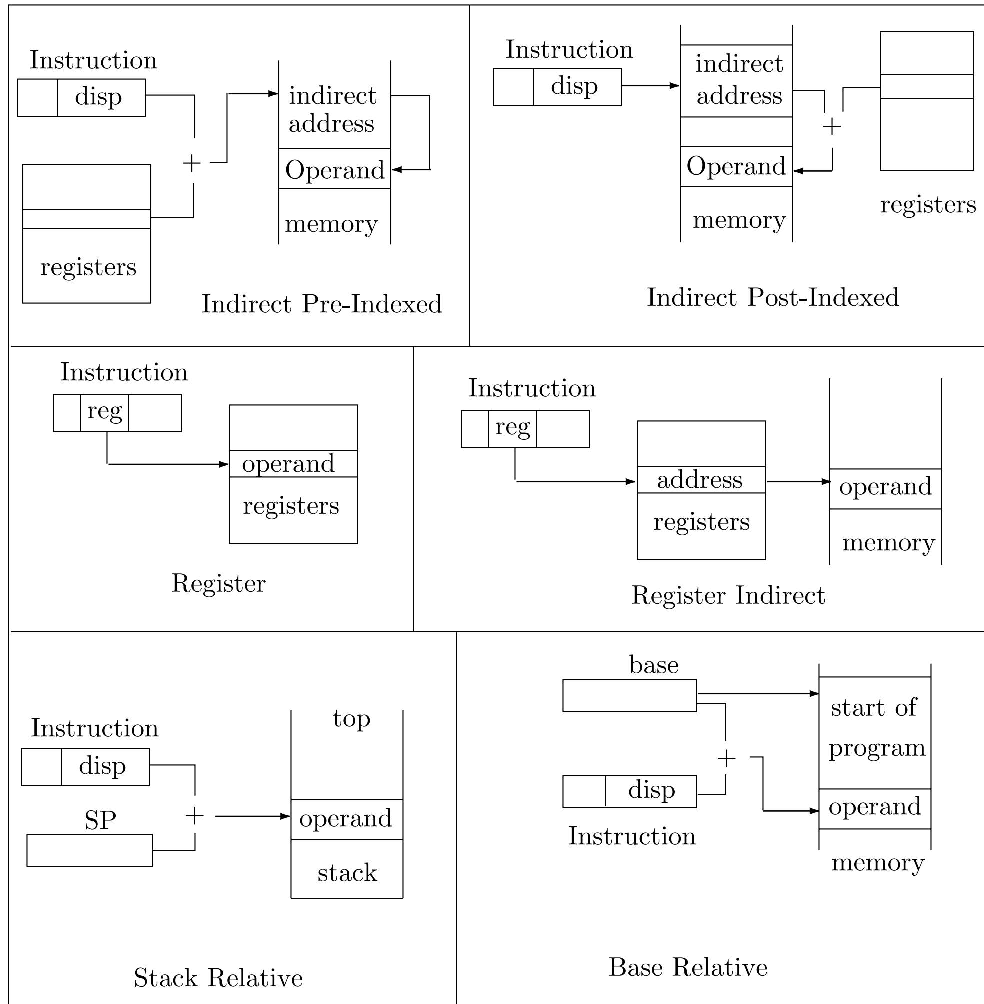



**Figure 2.4:** Addressing modes (part 2 of 2)

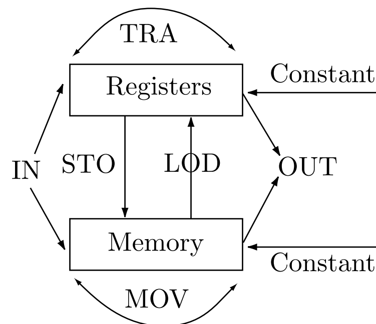



**Figure 2.5:** Data movements in a computer

## 2.13 Operations

These instructions specify operations on data items. They are all executed in a similar way. The control unit fetches the two operands (or the single operand), sends them to the ALU, sends a function code to the ALU, and waits for the ALU to perform the operation. The control unit then picks up the result at the ALU

and transfers it to its destination.

There are three types of operations that can be performed on data items (numeric and non numeric), namely arithmetic operations, logical operations, and shifts.

### 2.13.1 Arithmetic Operations

These include the four arithmetic operations on binary integers. However, most modern computers have circuits in the ALU to operate on floating-point numbers (Section 2.21), and sometimes also on BCD numbers (Section 2.23), in addition to integers. Such a computer may have 12 circuits for arithmetic operations, four for each numeric data type. Supercomputers, designed for fast number crunching, can pipeline certain ALU operations, a feature known as *vector processing*. A few older computers had ALU circuits for calculating square roots.

### 2.13.2 Logic Operations

These operations treat a data item as a collection of bits, not as a number or a character. Most logic operations accept two inputs and generate an output by going over the inputs and examining pairs of bits. Such an operation may proceed from right to left, starting with the rightmost bits of the two inputs. The operation generates one output bit according to a predetermined rule (the truth table of the operation), and proceeds to the next pair of input bits.

A common example is the logical AND operation. It generates an output bit of 1 if, and only if, both input bits are 1's (Table 2.6). As an example, the logical AND of the pair 0011, 0101 is 0001.

| | | | | | |
|---|---|---|---|---|---|
| | A | 1 | 1 | 0 | 0 |
| | B | 1 | 0 | 1 | 0 |
| negation | $\neg$ A | 0 | | 1 | |
| conjunction (AND) | A$\wedge$B | 1 | 0 | 0 | 0 |
| disjunction (OR) | A$\vee$B | 1 | 1 | 1 | 0 |
| exclusive OR (XOR) | A$\oplus$B | 0 | 1 | 1 | 0 |
| equivalence | A$\equiv$B | 1 | 0 | 0 | 1 |

**Table 2.6:** Truth tables of some logic operations

A look at the truth tables in table 2.6 shows two important facts about the logic operations:

1. The negation operation has just one operand. The truth table is fully defined by two bits, and there can therefore be only four logic operations with one input.

▸ **Exercise 2.6:** Show a simple example of another logic operation with one input.

2. Each of the other tables has two inputs and is fully defined by specifying four bits. This implies that there can only be $2^4 = 16$ such tables. This is not a limitation, since only four or five of the 16 operations are useful in practice and are implemented as ALU circuits. Most ALUs have circuits to perform the AND, OR, and NOT operations. The XOR operation can also be performed by the ALUs of many computers, but if any of the other operations is needed, it has to be implemented in software.

It should also be noted that the NOT operation can be achieved by XORing a number with all 1's. Thus, $1010 \oplus 1111 = 0101$.

The importance of arithmetic operations in a computer is obvious, but what is the use of the logical operations? Perhaps their most important use is in bit manipulations. They make it easy to isolate or to change an individual bit or a field of bits in a word. The operation $xxxyxxxx$ AND 00100000 results in $000y0000$ which, when compared to zero, reveals the value of bit $y$. Similarly, the operation $xxxxx000$ OR $00000yyy$ results in $xxxxxyyy$, making it easy to combine individual pieces of data into larger segments.

The XOR operation is especially interesting. Its main properties are:

1. It is commutative $a \oplus b = b \oplus a$.
2. It is associative $(a \oplus b) \oplus c = a \oplus (b \oplus c)$.
3. It is reversible. If $a = b \oplus c$, then $b = a \oplus c$ and $c = a \oplus b$.

One important application of the XOR is the stream ciphers of Section 3.19.4. Another, less familiar application is hiding an $n$-bit file $F$ in $k$ cover files of the same size. Given $k$ random files $C_0$, $C_1$,...,$C_{k-1}$, we can hide a file $F$ in them in the following way:

A $k$-bit password $P$ is chosen. It is scanned, and for each bit $P_j = 1$ in $P$, cover file $C_j$ is selected. All the selected $C_j$ files are XORed, the result is XORed with file $F$, and that result, in turn, is XORed with any of the $C_j$'s, say $C_a$. The process is summarized by

$$C_a = \Big(\bigoplus_{P_j=1} C_j\Big) \oplus F \oplus C_a.$$

Only one of the cover files, $C_a$, is modified in the process and there is no need to memorize which one it is. Once $F$ is hidden this way, it can be retrieved by

$$F = \bigoplus_{P_j=1} C_j.$$

▸ **Exercise 2.7:** We choose $n = 4$, $k = 5$, the password $P = 01110$, and the five cover files $C_0 = 1010$, $C_1 = 0011$, $C_2 = 1110$, $C_3 = 0111$, and $C_4 = 0100$. Show how file $F = 1001$ is hidden using $P$.

This clever application of the XOR hides a file $F$ in such a way that anyone with the password can easily retrieve it, but someone who does not have the password cannot even verify that the file exists, even if they search every bit of the entire file system. See Section 3.20 for a discussion of other data hiding techniques.

### 2.13.3 Shifts

These are simple, important operations, that are used on both numeric and nonnumeric data, and come in several varieties.

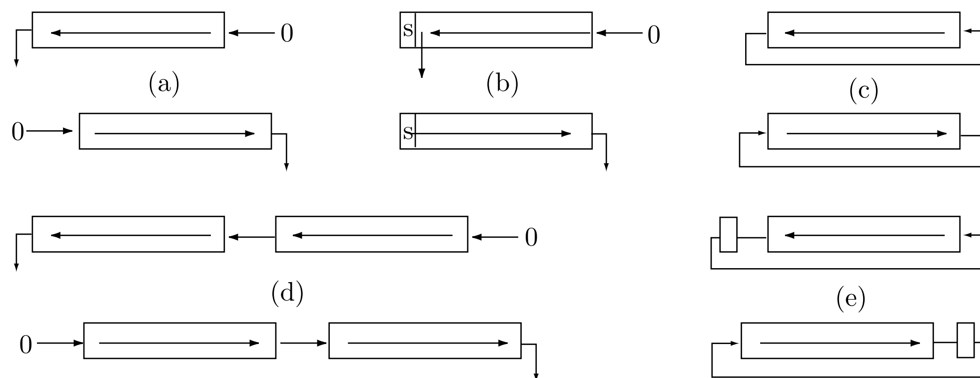



**Figure 2.7:** Types of shift

The simplest type of shift is the logical shift (Figure 2.7a). In a left logical shift, the leftmost bit is shifted out and is lost, and a zero is entered on the right. In a right logical shift, this operation is reversed.

The arithmetic shift (Figure 2.7b) is designed for multiplying and dividing numbers by powers of 2. This is easy and fast with a shift, and the only problem is to preserve the sign. The ALU circuit for arithmetic shift knows that the leftmost bit contains the sign, and is therefore special and should be preserved. An arithmetic left shift shifts bits out of the second bit position from the left, entering a zero on the right. The leftmost bit (the sign) is not affected. An arithmetic right shift shifts out the rightmost bit, entering copies of the sign bit into the second position from the left. This always results in the original number being either multiplied or divided by powers of 2, with the original sign preserved.

Also common is the circular shift (or rotation, Figure 2.7c). The number is rotated left by shifting out the leftmost bit and entering it from the right. Similarly for a circular right shift. A typical use of such a shift is the case where every bit of a certain number has to be examined. The number is repeatedly shifted,

in a loop, and each bit is examined when it gets to the leftmost position. In that position, the bit temporarily becomes the sign of the number, making it easy to examine.

A double shift is depicted in Figure 2.7d. A pair of registers is shifted as one unit. This is useful in a computer with a short word size, where double precision operations are common. The double shift itself can be of any of the varieties mentioned here.

Another variation is a shift through the carry. The carry status flag (C) can participate in a shift, making it easy to isolate and test individual bits. In a right shift through the carry, the rightmost bit is moved to the C flag and can later be examined by testing that flag. If such a shift is also circular (Figure 2.7e), it is equivalent to shifting $N+1$ bits.

### 2.13.4 Comparisons

In a typical comparison operation, two numbers are compared and a result is generated. The result, however, is not a number, which is why comparisons are not considered operations on data, but are instead classified as a separate class of instructions. If the two numbers compared are $a$ and $b$, then the result of the comparison is one of the three relations:

$$a < b, \qquad a = b, \qquad a > b.$$

The result is not a number and should therefore be stored in a special place, not in a register or in memory. The best place for the result is the Z and S status flags (Section 1.4).

By definition, the result of a comparison is the status of the difference $a-b$. The computer does not have to actually subtract the numbers, but it sets the flags according to the status of the difference. Thus, if $a=b$, the difference is zero and the Z flag is set to 1. If $a<b$, the difference is negative and the S flag is set (the Z flag is reset, since the difference is not zero). If $a>b$, both flags are cleared.

One way to compare numbers is to subtract them and check the difference. The subtract circuit in the ALU can be used, which automatically updates the Z and S flags. This method is simple but may result in a carry or overflow (see discussion of carry and overflow in Section 2.26).

Another technique to compare numbers is to compare pairs of bits from left (the most significant pair) to right (least significant one). As long as the pairs are equal, the comparison continues. When the first unequal pair is encountered, the result is immediately known and the process terminates. For example, given the numbers $a=00010110$ and $b=00100110$, $b$ is clearly the greater. A left to right comparison will stop at the third pair, where $b$ has a 1 while $a$ has a 0. Note that the next (fourth) pair has the opposite values but that cannot change the relative sizes of the two numbers.

The *test* instruction `TST`, found on some computers, has one operand, and it is compared to zero. Thus, the Z and S flags are set according to the status of the number being tested.

### 2.13.5 Branches, Conditional and Unconditional

An unconditional branch is a simple, common instruction, executed by resetting the PC to the branch address. This instruction is also called a jump, a transfer, or a goto. The importance of the conditional branches, however, is perhaps less obvious to the new programmer. The conditional branches turn out to be very important because they are the only way to perform decisions and loops in a program.

The programmer who is familiar with higher-level constructs such as

`if... then... else,  do... while,  repeat... until,`

is not always aware that the compiler uses the conditional branches as the only way to compile such constructs. Similarly, the programmer who writes a complete program without using any **goto**s is not always aware that the compiler generates many branches (conditional and unconditional) in the compiled code.

A conditional branch instruction works by examining one or more of the status flags. It causes a branch (by resetting the PC) if the flag has a certain value (0 or 1). For example, the `BCC` instruction (Branch Carry Clear) branches if the C flag is 0. The `BZE` instruction (Branch on ZEro) branches if the Z flag is 1.

▸ **Exercise 2.8:** What does Z=1 mean?

Some conditional branches examine two flags. An instruction such as `BPL` (Branch PLus) branches if the most recent result recorded in the flags is positive. The instruction tests the Z and S flags and branches if Z=0 (nonzero) and S=0 (nonnegative).

A computer may have two types of conditional branches, jumps and branches. The only difference between them is the addressing mode. A jump uses the direct mode, and can jump anywhere in memory. A branch uses the relative mode, and can only jump over short distances (Sections 2.4 and 2.5).

A computer may also have a `SKIP` instruction. This is a conditional branch that increments the PC instead of resetting it. For example, the `SKPN` (skip on Negative) instruction increments the PC by 2 if S = 1. This effectively skips the next instruction. In the sequence

```
SKPN
INCR
ADD
```

the `INCR` instruction is skipped if S=1 (negative).

The various procedure call instructions (page 16) are also included in this type. The difference between a branch and a procedure call is that the latter saves the return address before branching. Note that a procedure call instruction can also be conditional.

### 2.13.6 I/O Instructions

These instructions are divided into two groups, instructions that move data, and control instructions. The first group includes just two instructions, namely `IN` and `OUT`. Their general format is "`IN R,DEV`" where `R` is any register and `DEV` is a device select number (Section 1.8). Such an instruction moves one byte of data between the I/O device and the register.

The second group consists of a number of instructions that send commands to the I/O devices and also receive device status. On older computers, these instructions are very specific. A control instruction may only send commands to a disk unit, or may only receive status from a keyboard. This is mostly true for On modern computers, an I/O control instruction can be used to control several devices. An instruction such as "`BRDY DEV,ADDR`", for example, reads status from device `DEV` and branches to location `ADDR` when a "ready" status is read.

The single-bus computer, mentioned in Chapter 3, do not support any I/O instructions. On such a machine the I/O is memory mapped and data is moved between the I/O devices and the processor by means of memory reference instructions such as `LOD` and `STO`. When an I/O device is interfaced with such a computer, it is assigned several select numbers, one (or two) for the data and others for reading status and sending commands. A printer, for instance, may be assigned two select numbers, one to send data to be printed, and the other for reading printer status (ready, busy, out of paper, jammed) and also for sending printer commands (backspace, carriage return, line feed, form feed, ring bell, switch color).

An advantage of single-bus computer organization is that many different devices can be added to the computer without the need for special instructions.

### 2.13.7 Miscellaneous Instructions

Various instructions may be included in this category, ranging from the very simple (instructions with no operands) to very complex (instructions that are as powerful as statements found in a higher-level language). Following are a few examples.

1. `HLT`. An obvious instruction. Sometimes privileged (Section 1.7.1).

2. `NOP`. Does nothing. A very useful instruction, used to achieve precise delays between consecutive operations.

3. PSW updating. The PSW (program status word, sometimes referred to as PS) contains the various status flags and also hardware components such as the state of interrupts, processor mode, and memory protection information. Some of the PSW bits are updated only by the hardware, some can be updated only by privileged instructions, others can easily be updated by the user, using nonprivileged instructions. As a result, the computer has at least one instruction to access the PSW or parts of it.

4. Interrupt control. On a computer with more than one interrupt request line, instructions are needed to enable and disable individual interrupts.

5. `ED`. The EDit instruction is a good example of a high-level machine instruction found in older mainframes. It has two operands, a pattern and an array in memory. It scans the array and updates its contents according to the pattern. Thus, for example, all the "(" characters in an array can be replaced by

"[", all the periods can be replaced by commas, and all the spaces changed to an "x", all in one instruction. This is an example of a complex instruction, common in CISC (complex instruction set computers).

## 2.14 $N$-Operand Instructions

Many of the instructions discussed here come in several varieties that differ by the number of their operands. The `ADD` instruction, for example, is supposed to add two numbers, so it seems that it should have two operands. On many computers it is indeed a two–operand instruction, written as "`ADD R,ADDR`" (add a memory location to a register) or "`ADDR R1,R2`" (add two registers). On some computers, however, the same instruction has only one operand and is therefore written "`ADD ADDR`". Such an instruction adds its operand (a register or a memory location) to the accumulator (the main functional register).

It is also possible to have a 3-operand `ADD` instruction. "`ADD A,B,C`" can perform either A←B+C or A+B→C. Such an instruction is long (especially when A, B, C are memory addresses) and therefore rarely used.

Extending the same idea in the other direction, is it possible to design an `ADD` instruction with zero operands? The interesting answer is yes. On such a computer, the two numbers being added are the two top operands of the stack. They are also removed and replaced by the sum.

It is possible to design many other instructions as 0, 1, 2, or 3-operand instructions, which introduces the concept of an $N$-operand computer. In an $N$-operand computer many instructions (including most of the operations) have $N$ operands. Of course, not every instruction can have $N$ operands. The `LOD` instruction, for instance, can have either one operand (`LOD ADDR`) or two operands (`LOD R,ADDR`). A `LOD` with zero or with three operands cannot be designed in a reasonable way. An instruction that moves a block of words in memory is a natural candidate for two operands. Thus "`MOV SIZE,SOURCE,DESTIN`" (that moves `SIZE` words from `SOURCE` to `DESTIN`) cannot be reasonably designed as a 1-operand or a 3-operand. Note that the `SIZE` operand is not really a third operand, since it is a short number and not an address. Note also that it is possible to design this instruction as "`MOV SIZE,R1,R2`" where the two registers point to the two memory areas.

## 2.15 Actual Instruction Sets

The instruction sets summarized here are those of the Intel 8080 microprocessor, The VAX, and the Berkeley RISC I computer. The first set is representative of the early 8-bit microprocessors; it is simple, neatly organized, and not large. The second set is typical of large, old mainframes. It is very large, contains many different types and sizes of instructions, and uses many addressing modes. It is an example of a complex instruction set. The third set is an example of the opposite approach, that of a reduced instruction set. It is very small and very simple. It uses single-size instructions and very few addressing modes. The concept of RISC (reduced instruction set computers) is discussed in many texts on computer architecture.

The Intel 8080 was one of the first 8-bit microprocessors and was extensively used in the late 1970s, until better microprocessors were developed. Its instruction set is simple and reflects its architecture. It has a relatively small number of instructions, three addressing modes, three different instruction sizes, and seven types of instructions.

The 8080 has eight 8-bit registers and two 16-bit registers. Their organization is summarized in Figure 2.8. The PSW (program status word) contains the status flags. The A register (accumulator) is the main functional register. Registers B–L are the general-purpose registers. The PC (program counter) contains the address of the next instruction, and the SP (stack pointer) always points to the top item in the stack. Certain instructions use the BC, DE, and HL pairs of registers.

<table>
<tr><td>PSW</td><td>A</td></tr>
<tr><td>B</td><td>C</td></tr>
<tr><td>D</td><td>E</td></tr>
<tr><td>H</td><td>L</td></tr>
<tr><td colspan="2">SP</td></tr>
<tr><td colspan="2">PC</td></tr>
</table>

**Figure 2.8:** The registers of the 8080 microprocessor

The addressing modes are direct, implied and register indirect. In the direct mode, the 16-bit EA is stored in two bytes in the instruction and the instruction itself is 3-bytes long. In the implied mode, the EA is stored in the HL register pair, and the instruction is either one or two bytes long. In the register indirect mode, the 2-byte EA is stored in memory, in the location pointed to by one of the register pairs.

Most instructions are one byte long. The move-immediate, arithmetic-immediate, logical-immediate, and I/O instructions are two bytes long, with the second byte containing either the immediate quantity or an I/O device number. Branches and some memory-reference instructions are three bytes long (they use the direct mode mentioned earlier).

The first byte of an instruction contains the opcode. This allows for 256 instructions but only 244 opcodes are used. However, many of those instructions are very similar and can be considered identical. For example the "`MOV A,B`" and "`MOV C,B`" instructions have different opcodes (the opcodes contain the register codes) but are really the same instruction or, at least, the same type of instruction. As a result, the total number of different instructions is around 50–60, the precise number depends on what one considers to be different instructions.

Table 2.9 summarizes the seven instruction formats and eight register codes. It is followed by examples of instructions. The notation *p* in this table refers to an opcode bit, *rrr, ddd*, and *sss* are registers, Y is either an immediate number or an I/O device number and EA, an effective address.

| **Instruction Formats** | **Reg. Codes** |
|---|---|
| 0. | B |
| 1. *pppppr rr* | C |
| 2. 8 bit opcode, 8 bit Y. | D |
| 3. 8 bit opcode, 16 bit EA. | E |
| 4. *pprrrppp* | H |
| 5. *ppdddsss* | L |
| 6. *ppdddppp*, 8 bit Y. | M* |
| 7. 8 bit opcode | A |

* The memory loc. pointed to by the HL pair.

**Table 2.9:** 8080 instruction formats and register codes

Examples:

| | Source Instruction | Type | Binary | Hex | Description |
|---|---|---|---|---|---|
| 1. | `MOV A,B` | 5 | 01 111 000 | `78` | A←B |
| 2. | `ADD M` | 1 | 10000 110 | `86` | A←A+Mem[HL] |
| 3. | `IN 17` | 2 | | `DB 11` | A←Dev[17] |
| 4. | `MVI E,25` | 6 | 00 011 110 | `1E 19` | E←25 |
| 5. | `JMP 1030` | 3 | | `C3 0406` | PC←1030 |
| 6. | `STAX D` | 1 | | `12` | Mem[DE]←A |

Table 2.10 is a very compact summary of the entire instruction set. Each asterisk "*" in the table represents an additional byte. The row numbers on the left (00–37) are in octal and represent the leftmost five bits of the opcode. The column numbers on the top row (0–7) represent the rightmost three bits. Thus, for example, the `CNZ` instruction has an opcode of $304_8 = 11000100_2$.

The table is divided into four parts, each containing certain groups of instructions. The first part contains the increment, decrement, index, and move immediate instructions. The second part contains just the move instructions. The third part, the arithmetic operations, and the fourth part, the jump, call, arithmetic immediate, and a few special instructions. The table demonstrates the neat organization of this instruction set.

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 00 | NOP | LXI B** | STAX B | INX B | INR B | DCR B | MVI B* | RLC |
| 01 | | DAD B | LDAX B | DCX B | INR C | DCR C | MVI C* | RRC |
| 02 | | LXI D** | STAX D | INX D | INR D | DCR D | MVI D* | RAL |
| 03 | | DAD D | LDAX D | DCX D | INR E | DCR E | MVI E* | RAR |
| 04 | | LXI H** | SHLD | INX H | INR H | DCR H | MVI H* | DAA |
| 05 | | DAD H | LHLD H | DCX H | INR L | DCR L | MVI L* | CMA |
| 06 | | LXI SP** | STA | INX SP | INR M | DCR M | MVI M* | STC |
| 07 | | DAD SP | LDA | DCX SP | INR A | DCR A | MVI A* | CMC |
| 10 | MOV B,B | MOV B,C | MOV B,D | MOV B,E | MOV B,H | MOV B,L | MOV B,M | MOV B,A |
| 11 | MOV C,B | MOV C,C | MOV C,D | MOV C,E | MOV C,H | MOV C,L | MOV C,M | MOV C,A |
| 12 | MOV D,B | MOV D,C | MOV D,D | MOV D,E | MOV D,H | MOV D,L | MOV D,M | MOV D,A |
| 13 | MOV E,B | MOV E,C | MOV E,D | MOV E,E | MOV E,H | MOV E,L | MOV E,M | MOV E,A |
| 14 | MOV H,B | MOV H,C | MOV H,D | MOV H,E | MOV H,H | MOV H,L | MOV H,M | MOV H,A |
| 15 | MOV L,B | MOV L,C | MOV L,D | MOV L,E | MOV L,H | MOV L,L | MOV L,M | MOV L,A |
| 16 | MOV M,B | MOV M,C | MOV M,D | MOV M,E | MOV M,H | MOV M,L | HLT | MOV M,A |
| 17 | MOV A,B | MOV A,C | MOV A,D | MOV A,E | MOV A,H | MOV A,L | MOV A,M | MOV A,A |
| 20 | ADD B | ADD C | ADD D | ADD E | ADD H | ADD L | ADD M | ADD A |
| 21 | ADC B | ADC C | ADC D | ADC E | ADC H | ADC L | ADC M | ADC A |
| 22 | SUB B | SUB C | SUB D | SUB E | SUB H | SUB L | SUB M | SUB A |
| 23 | SBB B | SBB C | SBB D | SBB E | SBB H | SBB L | SBB M | SBB A |
| 24 | ANA B | ANA C | ANA D | ANA E | ANA H | ANA L | ANA M | ANA A |
| 25 | XRA B | XRA C | XRA D | XRA E | XRA H | XRA L | XRA M | XRA A |
| 26 | ORA B | ORA C | ORA D | ORA E | ORA H | ORA L | ORA M | ORA A |
| 27 | CMP B | CMP C | CMP D | CMP E | CMP H | CMP L | CMP M | CMP A |
| 30 | RNZ | POP B | JNZ** | JMP** | CNZ** | PUSH B | ADI * | RST 0 |
| 31 | RZ | RET B | JZ ** | | CZ** | CALL** | ACI* | RST 1 |
| 32 | RNC | POP D | JNC** | OUT* | CNC** | PUSH D | SUI* | RST 2 |
| 33 | RC | | JC ** | IN * | CC** | | SBI* | RST 3 |
| 34 | RPO | POP H | JPO** | XHTL | CPO** | PUSH H | SNI* | RST 4 |
| 35 | RPE | PCHL | JPE** | XCHG | CPE** | | XRI* | RST 5 |
| 36 | RP | POP PSW | JP** | DI | CP** | PUSH PSW | ORI* | RST 6 |
| 37 | RM | SPHL | JM ** | EI | CM** | | CPI* | RST 7 |

**Table 2.10:** The 8080 instruction set

## 2.16 The VAX Instruction Set

The VAX was a family of 32-bit mainframes made by Digital Equipment Corporation (DEC). The VAX family was the successor of the very successful PDP family of minicomputers. The name VAX is an acronym that stands for *Virtual Address eXtension.* The VAX was, in fact, an extension of the PDP (Programmed Data Processor) computers, an extension that supports virtual memory. A programmer on a VAX can assume an address space of 4 Gb ($2^{32}$ or over 4.3 billion bytes), even though any specific VAX may not have such a large memory. The program is then divided into pages, each 512 bytes long, which reside on a disk and are brought into memory as they are needed. A complete discussion of the virtual address concept and the address mapping process is beyond the scope of this text and can be found in [Digital Equipment 81].

The instruction set is large and complex, consisting of more than 260 instructions, of many types and sizes. A total of 24 addressing modes is supported by the hardware and some of the instructions are quite powerful.

The VAX has sixteen 32-bit registers, 10 of which are truly general-purpose, and the rest have special designations.

All those features combine with a memory access time of 290 nsec to produce a powerful and easy to

use computer, one of the successful mainframes in use today.

All instructions have an 8-bit opcode, which seems to limit the instruction set to 256 instructions. However, certain instructions operate on different data types (there are 9 different data types) and may therefore be considered different instructions. Instructions can have from zero up to five operands, which results in several possible instruction sizes.

Integers can be represented (and operated on) in 5 sizes; 8 bits (byte), 16 bits (word), 32 bits (longword), 64 bits (quadword) and even 128 bits (octaword. The basic MOV instruction is a good example. It has 5 variations, for the 5 types of integers.

B. `MOVB` moves just one byte. `MOVB #-5,A` moves the constant −5 into the byte at address `A`.

W. `MOVW` moves a word. `MOVW G,A` moves the 2 bytes at address `G` to the 2 bytes at address `A`.

L. `MOVL` moves a long word. `MOVL R5,A` moves register 5 (32 bits) to the 4 bytes that start at address `A`.

Q. `MOVQ` moves a quadword. `MOVQ R5,A` moves registers 5 and 6 to the 8 bytes that start at address `A`.

O. `MOVO` moves an octaword.

Table 2.11 contains representative examples of instructions with different operands, modes, and sizes.

| | Source Instruction | No. of Operands | Size of code | Object code | Note |
|---|---|---|---|---|---|
| 1. | `NOP` | 0 | 1 | `01` | |
| 6. | `BVS DES` | 1 | 2 | `22 1D` | |
| 2. | `MOVL #2,R4` | 2 | 3 | `54 02 D0` | 1 |
| 9. | `BRW LAB1` | 1 | 3 | `001B 31` | |
| 3. | `BLEQ 2` | 1 | 5 | `00000002 15` | |
| 4. | `MOVL -4(R7),200(R5)` | 2 | 6 | `00C8 C5 FC A7 D0` | 2 |
| 5. | `JMP 251` | 1 | 7 | `00000019 EF 17` | 3 |
| 8. | `ADDL3 N1,N2,N3` | 3 | 9 | `FF70 CF 8F CF FF8E CF C1` | 4 |
| 7. | `MOVL 8,10` | 2 | 11 | `00000010 EF 00000008 EF D0` | |

**Table 2.11:** Some VAX instructions

Notes:

- The opcode is `D0`, the `02` is the immediate number (the second operand) and the 54 signifies register 4 in mode 5 (register direct)

- The opcode is `D0`, the `FC` is the immediate quantity −4 (the second operand), the `A7` signifies register 7 in mode `A` (byte displacement), the `00C8` is the constant 200, and the `C5` signifies register 5 in mode `C` (word displacement).

- The opcode is `17`, and `EF` means that register 15 (which is also the PC) is used here in mode 14 (longword relative).

- Assuming that `N1`, `N2`, and `N3` have been declared as longwords. `C1` is the opcode, `CF` means register `F` in mode `C` (word relative), and `FF70` is assumed to be the relative address of operand `N1`.

The addressing mode is a 4-bit field in a byte which also includes a 4-bit register number. This is the *operand specifier byte*, which is a part of most instructions. Since register 15 is also the PC, it is different from the other registers. This is why certain mode bits, when used with register 15, mean something different than usual.

| Mode | Register |
|---|---|

| Mode | 1 1 1 1 |
|---|---|

Mode bits 1010(=`A`), for example, mean *byte displacement* when used with any of the registers 0–14; when used with register 15, the same bits mean *byte relative deferred* mode. Table 2.12 summarizes all the 24 different addressing modes.

| | | Registers 0–14 | Register 15 |
|---|---|---|---|
| Hex | Dec | Mode | Mode |
| 0–3 | 0–3 | literal | |
| 4 | 4 | indexed | |
| 5 | 5 | register | |
| 6 | 6 | register deferred | |
| 7 | 7 | autodecrement | |
| 8 | 8 | autoincrement | immediate |
| 9 | 9 | autoincr. deferred | absolute |
| A | 10 | byte displacement | byte relative |
| B | 11 | byte displ deferred | byte relative deferred |
| C | 12 | word displacement | word relative |
| D | 13 | word displ deferred | word rel deferred |
| E | 14 | longword displacement | longword relative |
| F | 15 | longword displ deferred | longword rel deferred |

**Table 2.12:** VAX addressing modes

An example of a complex VAX instruction is the `MOVTC` *Move Translated Characters*. It accepts a string of characters, scans it, and replaces each character in the string with another one, taken from a special 256-byte table of replacement codes. As an alternative, it can leave the input string intact, and generate an output string. This instruction is considered complex because it can only be replaced by a loop, not just by a few instructions.

The `MOVTC` instruction thus requires the following operands: Source string length, source string address, fill character*, address of table, destination string length, destination string address.

More information on the VAX and its instruction set can be found in [Digital Equipment 81] as well as in many books on the VAX assembler language.

## 2.17 The RISC I Instruction Set

The RISC I [Patterson 85] is an experimental computer, designed and built at The University of California, Berkeley in 1979–1981. The background and design principles of RISC are covered in many texts on computer architecture. Here, we briefly describe the instruction set, which constitutes a good example of a reduced set of instructions (see also Section 6.3.1).

The RISC I instruction set contains 39 instructions, each 32-bits long. All the instructions have a 7-bit opcode, although only 39 out of the 128 possible opcodes are used. Figures 2.13 and 6.1 show the instruction formats.

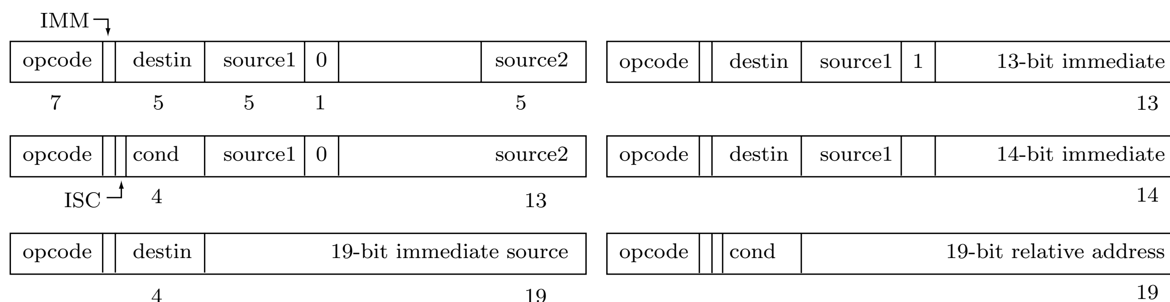



**Figure 2.13:** RISC I instruction formats

* If the destination string is longer than the source string, the fill character is used to fill the rest of the destination string

The 1-bit SCC (set condition code) field specifies whether an instruction should update the status flags. The source1 and destination fields select two register operands (out of 32 registers). The IMM field defines the meaning of the source2 field. If IMM is zero, the five low-order bits of source2 specify a register number. If IMM is one, then source2 specifies either an immediate number, or a 13-bit relative address, depending on the opcode. Some instructions use the COND field (four bits) instead of source1. Finally, the branch relative instructions `JMPR` and `CALLR` use the PC-relative mode, where the source1, IMM, and source2 fields are combined to form a 19-bit relative address. Some examples are:

| Name | Mnemonic | Operands | Description |
|---|---|---|---|
| Integer add | ADD | Rs,S2,Rd | Rd←Rs+M[S2] |
| Load long | LDL | (Rx)S2,Rd | Rd←M[Rx+S2] |
| Jump relative | JMPR | Y | PC←PC+Y |
| Load PSW | GETPSW | Rd | Rd←PSW |

In those examples, Rs and Rd stand for the source1 and destination registers, S2 is the source2 field (a source address), Rx is the source1 register, used as an index register, and Y is the 19-bit relative address mentioned earlier. M[S2] is the contents of location S2 in memory, M[Rx+S2] specifies indexing (an EA is obtained by adding Rx+S2 followed by a memory access M[EA]).

This is a simple instruction set that tries to show, in the spirit of RISC, that a complex instruction set is not a necessary condition for a fast, efficient computer.

## 2.18 Non-Numeric Data Types

The rest of this chapter is devoted to the different types of data supported by current computers. The following important types are described, character, boolean, signed integer, floating-point, fixed-point, and BCD. Also discussed are the concepts of carry and overflow.

The nonnumeric data types dealt with in this section are character and boolean. First, however, let's try to understand the meaning of the term "data type." A data type is something like "integer", "real", or "boolean." Those words are used all the time in higher level programming and most users have a vague, intuitive understanding of them. It is possible, however, to assign them a precise meaning. Recall, for example, that a boolean quantity can take one of two values, namely "true" or "false." We say that the set of all boolean values is of size 2, which leads to the definition

A data type is a set of values.

### 2.18.1 Characters

Characters constitute a simple data type, since there are relatively few of them, and since relatively few operations can be performed on them. Characters are important, because they are used in word processing, data bases, the compilation of programs, and other important applications. In fact, there are many computer users who rarely, if ever, use numbers in their programs. The fact that computers are useful for nonnumeric applications was recognized early in their history, when programmers realized that the computer could be used to compile its own programs.

Character sets typically contain 100 to 200 characters, and the most important character codes are described in Chapter 3. This section concentrates on the few operations that can be performed on characters, namely input, comparisons, and output. When dealing with strings of characters, two more operations namely, concatenation and substring, are useful.

Most of the time, characters are input, stored in memory, and eventually output as text. Comparing characters is a simple, important operation that involves the character codes. It is easy, of course, to compare characters for equality and nonequality. It is also possible to compare them for the other four relationals $<$, $>$, $\leq$, and $\geq$. The comparison $a > b$ is done by comparing the numeric codes of the two characters. The character with the smaller code would thus be the "smaller" character for the purpose of comparisons. The code, in fact, defines an order on the characters, an order called the *collating sequence* of the computer.

The term *concatenating* means to place two objects side by side. The substring operation extracts a substring from a given string. These are two important operations when dealing with strings of text.

### 2.18.2 Boolean

Boolean is a word derived from the name of George Boole, an Irish mathematician and philosopher who founded the field of modern symbolic logic. He was interested in operations on numbers that result in nonnumeric quantities—such as $a > b$, which results in either true or false—and came up with a formalism of such operations, which we now call *boolean algebra*. His work has been widely used in the design of computers, since all the arithmetic operations in the computer are performed in terms of logical operations. His main work is *An Investigation of the Laws of Thought* (1854).

The two boolean values are represented in the computer using either the sign bit

$$0x \ldots x \text{ for true, and } 1x \ldots x \text{ for false (or vice versa).}$$

(where $x$ stands for 'don't care'), or using all the bits in a word

$$00 \ldots 0 \text{ for true, and } 11 \ldots 1 \text{ for false (or vice versa).}$$

## 2.19 Numeric Data Types

Most computers include two data types in this category, namely integer and real. However, this, section shows that it is possible to define other numeric data types that can be useful in special situations.

## 2.20 Signed Integers

The binary number system was discovered by the great mathematician and philosopher Gottfried Wilhelm von Leibniz on March 15, 1679. This representation of the integers is familiar and is the most "natural" number representation on computers in the sense that it simply uses the binary values of the integers being represented. The only feature that needs to be discussed, concerning integers, is the way *signed integers* are represented internally. Unsigned numbers are certainly not enough for practical calculations, and any number representation should allow for both positive and negative numbers. Three methods have been used, throughout the history of computing, to represent signed integers. They are sign-magnitude, one's complement, and two's complement. All three methods reserve the leftmost bit for the sign of the number, and all three use the same sign convention, namely 1 represents a negative sign and 0 represents a positive sign.

### 2.20.1 Sign-Magnitude

In this method the sign of a number is changed by simply complementing the sign bit. The magnitude bits are not changed. To illustrate this method (and also the following ones) we use 4-bit words, where one bit is reserved for the sign, leaving three magnitude bits. Thus $0|111$ is the representation of $+7$ and $1|111$, that of $-7$. The number $0|010$ is $+2$ and $1|010$ is $-2$. In general, $1xxx$ is the negative value of $0xxx$. The largest number in our example is $+7$ and the (algebraically) smallest one is $-7$. In general, given $N$-bit words, where the leftmost bit is reserved for the sign, leaving $N-1$ magnitude bits, the largest possible number is $2^{N-1}-1$ and the smallest one is $-(2^{N-1}-1)$.

This representation has the advantage that the negative numbers are easy to read but, since we rarely have to read binary numbers, this is not really an advantage. The disadvantage of this representation is that the rules for the arithmetic operations are not easy to implement in hardware. Before the ALU can add or subtract such numbers, it has to compare them, in order to decide what the sign of the result should be. When the result is obtained, the ALU has to append the right sign to it explicitly.

The sign-magnitude method was used on some old, first-generation computers, but is no longer being used.

### 2.20.2 One's Complement

This method is based on the simple concept of *complement*. It is more suitable than the sign-magnitude method for hardware implementation. The idea is to represent a negative number by complementing the bits of the original, positive number. This way, we hope to eliminate the need for a separate subtraction circuit in the ALU and to subtract numbers by adding the complement. Perhaps the best way to understand this method is to consider the complement of a decimal number.

Example: Instead of subtracting the decimal numbers $12845 - 3806$, we try to add 12845 to the decimal complement of 3806. The first step in this process is to append sign digits to both numbers, which results in $0|12845$ and $0|03806$. The second step is to complement the second number. The most natural way to

### Number Bases

Decimal numbers use base 10. The number $2037_{10}$, e.g., is worth $2\times10^3+0\times10^2+3\times10^1+7\times10^0$. We can say that 2037 is the sum of the digits 2, 0, 3 and 7, each weighted by a power of 10. Fractions are represented in the same way, using negative powers of 10. Thus $0.82=8\times10^{-1}+2\times10^{-2}$ and $300.7=3\times10^2+7\times10^{-1}$.

Binary numbers use base 2. Such a number is represented as a sum of its digits, each weighted by a power of 2. Thus $101.11_2=1\times2^2+0\times2^1+1\times2^0+1\times2^{-1}+1\times2^{-2}$.

Since there is nothing special about 10 or 2*, it should be easy to convince yourself that any positive integer $n>1$ can serve as the basis for representing numbers. Such a representation requires $n$ 'digits' (if $n>10$ we use the ten digits and the letters '`A`', '`B`', '`C`'...) and represents the number $d_3d_2d_1d_0.d_{-1}$ as the sum of the 'digits' $d_i$, each multiplied by a power of $n$, thus $d_3n^3+d_2n^2+d_1n^1+d_0n^0+d_{-1}n^{-1}$. The base for a number system does not have to consist of powers of an integer, but can be any *superadditive* sequence that starts with 1.

Definition: A superadditive sequence $a_0$, $a_1$, $a_2$,... is one where any element $a_i$ is greater than the sum of all its predecessors. An example is 1, 2, 4, 8, 16, 32, 64,... where each element equals one plus the sum of all its predecessors. This sequence consists of the familiar powers of 2, so we know that any integer can be expressed by it using just the digits 0 and 1 (the two bits). Another example is 1, 3, 6, 12, 24, 50,... where each element equals 2 plus the sum of all its predecessors. It is easy to see that any integer can be expressed by it using just the digits 0, 1 and 2 (the 3 trits).

Given a positive integer $k$, the sequence 1, $1+k$, $2+2k$, $4+4k$,...,$2^i(1+k)$ is superadditive, since each element equals the sum of all its predecessors plus $k$. Any non-negative integer can be *uniquely* represented in such a system as a number $x\ldots xxy$, where $x$ are bits and $y$ is in the range $[0,k]$.

In contrast, a general superadditive sequence, such as 1, 8, 50, 3102 can be used to represent integers, but not uniquely. The number 50, e.g., equals $8\times6+1+1$, so it can be represented as $0062=0\times3102+0\times50+6\times8+2\times1$, but also as $0100=0\times3102+1\times50+0\times8+0\times1$.

It can be shown that $1+r+r^2+\cdots+r^k<r^{k+1}$ for any real number $r>1$, which implies that the powers of any real number $r>1$ can serve as the base of a number system using the digits 0, 1, 2,..., $d$ for some $d$.

The number $\phi=\frac{1}{2}(1+\sqrt{5})\approx1.618$ is the well known golden ratio. It can serve as the base of a number system using the two binary digits. Thus, e.g., $100.1_\phi=\phi^2+\phi^{-1}\approx3.23_{10}$.

Some real bases have special properties. For example, any positive integer $R$ can be expressed as $R=b_1F_1+b_2F_2+b_3F_3+b_4F_5+\cdots$ (that's $b_4F_5$, not $b_4F_4$) where $b_i$ are either 0 or 1, and the $F_i$ are the Fibonacci numbers $1,2,3,5,8,13,\ldots$ This representation has the interesting property that the string $b_1b_2\ldots$ does not contain any adjacent 1's (this property is used by certain data compression methods). As an example, the integer 33 equals the sum $1+3+8+21$, so it is expressed in the Fibonacci base as the 7-bit number 1010101.

A non-negative integer can be represented as a finite sum of binomials

$$n=\binom{a}{1}+\binom{b}{2}+\binom{c}{3}+\binom{d}{4}+\cdots;\quad \text{Where } 0\le a<b<c<d\ldots$$

are integers and $\binom{i}{n}$ is the binomial $\frac{i!}{n!(i-n)!}$. This is the *binomial number system.*

*Actually, there is. Two is the smallest integer that can be a base for a number system. Ten is the number of our fingers.

complement a decimal number is to complement each digit with respect to 9 (the largest decimal digit). Thus the complement of 0|03806 would be 9|96193 (this suggests that 9 represents a negative sign). The third step is to add the two numbers, which yields the sum 1009038. This is truncated to six digits (one sign digit plus five magnitude digits), to produce 0|09038. The correct result, however, is +9039. We end up with a result that differs from the correct one by 1. It is, of course, easy to artificially add a 1 to our result to get the correct sum, but before doing so, the reader should try some more decimal examples to convince himself that this result happens often, but not always, when numbers with different signs are added.

The binary case is similar. The one's complement of the number 0|101 is 1|010 but, when adding numbers with different signs, a 1 has sometimes to be added in order to obtain the correct result. A complete discussion of one's complement arithmetic is outside the scope of this text and can be found in texts on computer arithmetic. However, the general rule is easy to state.

> **Rule:** When adding numbers in one's complement, a 1 has to added to the result if there is a carry out of the addition.

In our decimal example, the result was 1009038, which can be divided into carry = 1, sign = 0, and sum = 09038. There was a carry, which is why the result had to be incremented by 1.

The one's complement method was used on many second generation computers, but has been rejected in favor of the newer two's complement method.

Another disadvantage of this method is the *negative zero* anomaly. The one's complement of 0|000 is 1|111 which may be called a negative zero. However, in mathematics there is no difference between zero and negative zero (i.e., there is no such number as negative zero), and a good number representation should not have a special bit pattern that represents $-0$. The $1|111\ldots1$ bit combination is therefore superfluous.

| Decimal | Binary | | |
|---|---|---|---|
| | Sign-Magnitude | 1's Complement | 2's Complement |
| $2^{N-1}-1$ | $01\ldots111$ | | |
| ⋮ | ⋮ | | |
| 7 | $00\ldots111$ | Same as | Same as |
| ⋮ | ⋮ | sign-magnitude | sign-magnitude |
| 2 | $00\ldots010$ | | |
| 1 | $00\ldots001$ | | |
| 0 | $00\ldots000$ | | |
| $-0$ | $10\ldots000$ | $11\ldots111$ | $NA$ |
| $-1$ | $10\ldots001$ | $11\ldots110$ | $11\ldots111$ |
| $-2$ | $10\ldots010$ | $11\ldots101$ | $11\ldots110$ |
| $-3$ | $10\ldots011$ | $11\ldots100$ | $11\ldots101$ |
| ⋮ | ⋮ | ⋮ | ⋮ |
| $-7$ | $10\ldots111$ | $11\ldots000$ | $11\ldots001$ |
| ⋮ | ⋮ | ⋮ | ⋮ |
| $-2^{N-1}-1$ | $11\ldots111$ | $10\ldots000$ | $10\ldots001$ |
| $-2^{N-1}$ | NA | NA | $10\ldots000$ |

**Table 2.14:** Three methods for representing signed integers

### 2.20.3 Two's Complement

Two's complement is the method used to represent signed integers on all modern computers. It is based on the idea that, since we sometimes have to artificially add a 1 to our one's complement results, why not try to add this 1 when the signed number is generated in the first place. The principle of this method is to complement a number in two steps: (1) every bit is complemented (we get the one's complement) and (2) a 1 is added. Thus, the two's complement of $+5 = 0|101$ is obtained by

$$0|101 \to 1|010 + 1 = 1|011.$$

This is a general rule. It can be used to obtain the two's complement of a positive number as well as that of a negative one. The reader can easily verify this by applying the rule to the $-5$ above.

The downside of the two's complement method is easy to see; the negative numbers are hard to read. However, as pointed out before, this is not an serious disadvantage. The main advantage of this method is not immediately apparent but is important. The two's complement method simplifies the arithmetic rules for adding and subtracting signed numbers. In fact, there are only two simple rules:

1. To subtract $a - b$, first obtain the two's complement $\bar{b}^2$, of $b$, then add $a + \bar{b}^2$.
2. To add $a + b$, add corresponding pairs of bits, from right to left, including the pair of sign bits, while propagating the carry in the usual way.

A simple example is $6 + (-3) = 0|110 + 1|101$. A direct addition from right to left gives $10|011$ which should be read: carry = 1, sign = 0, magnitude = 011 = 3. The carry should be ignored, since we can only deal with fixed-size, 4-bit numbers. Note how adding the third pair $(1 + 1)$ generates an intermediate carry that is propagated to the fourth pair. The fourth pair is added as $(0 + 1) + \text{carry} = 1 + \text{carry} =$ (the right sign).

The interested reader is referred any other text on computer arithmetic for a more thorough discussion of the properties of the two's complement method. Perhaps the only point worth mentioning here is the absence of negative zero. The two's complement of $0|000$ is $1|111 + 1 = 10|000 =$ (after ignoring the carry) $0|000$. This has an interesting side effect. With 4 bits we can have 16 combinations. If one combination is used for zero, 15 combinations are left. Since 15 is an odd number, we cannot have the same amounts of positive and negative numbers. It turns out (Table 2.14) that with four bits, we can represent the positive numbers $1, 2, \ldots, 7$ and the negative numbers $-1, -2, \ldots, -8$. In the two's complement method, the range of negative numbers is always greater, by one number, than the range of positive numbers.

▸ **Exercise 2.9:** What are the two ranges for $n$-bit numbers.

## 2.21 Floating-Point Numbers

Since integers are not sufficient for all calculations, computer designers have developed other representations where nonintegers can be represented and operated on. The most common of which is the *floating point* representation (f.p. for short), normally called *real*. In addition to representing nonintegers, the floating point method can also represent very large and very small (fractions very close to zero) numbers. The method is based on the common scientific notation of numbers (for example $56 \times 10^9$) and represents the real number $x$ in terms of two signed integers $a$ (the mantissa) and $b$ (the exponent) such that $x = a \times 2^b$ (Figure 2.15).

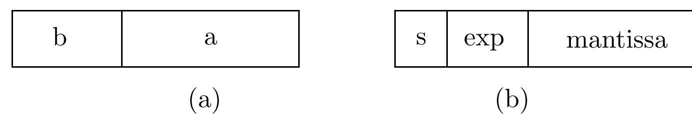



**Figure 2.15:** Floating-point numbers, (a) as two integers, (b) with a sign bit

The word mantissa is a Latin term of Etruscan origin, meaning a makeweight, a small weight added to a scale to calibrate it [Smith 23].

Table 2.16 shows some simple examples of real numbers that consist of two integers. Note that none of the integers is very large or very small, yet some of the floating point numbers obtained are extremely large or are very small fractions. Also some floating point numbers are integers, although in general they are not.

| | a | b | $a \times 2^b$ | value |
|---|---|---|---|---|
| 1 . | 1 | 1 | $1 \times 2^1$ | 2 |
| 2 . | 1 | 10 | $1 \times 2^{10}$ | 1024 |
| 3 . | 1 | 20 | $1 \times 2^{20}$ | $\approx 10^6$ |
| 4 . | 1 | 21 | $1 \times 2^{21}$ | $\approx 2 \times 10^6$ |
| 5 . | 15 | -3 | $15 \times 2^{-3}$ | $15/8 = 1.875$ |
| 6 . | 1 | -10 | $1 \times 2^{-10}$ | $\approx 0.001$ |
| 7 . | 10 | -20 | $10 \times 2^{-20}$ | $\approx 10 \times 10^{-6} = 10^{-5}$ |
| 8 . | 100 | -1 | $100 \times 2^{-1}$ | 50 |
| 9 . | 1234 | -100 | $1234 \times 2^{-100}$ | $\approx 1234 \times 10^{-30}$ |
| 10. | 1235 | -100 | $1235 \times 2^{-100}$ | $\approx 1235 \times 10^{-30}$ |

**Table 2.16:** Examples of floating-point numbers

Two important properties of floating point numbers are clearly illustrated in the table. One is that the magnitude of the number is sensitive to the size of the exponent $b$; the other is that the sign of $b$ is an indication of whether the number is large or small. Values of $b$ in the range of 20 result in floating point numbers in the range of a million, and a small change in $b$ (as, for example, from 20 to 21) doubles the value of the floating point number. Also, floating point numbers with positive $b$ tend to be large, while those with negative $b$ tend to be small (usually less than 1). This is not always true, as example 8 shows, but that example is atypical, as is illustrated later.

The function of the mantissa $a$ is not immediately clear from the table, but is not hard to see. The mantissa contributes the *significant digits* to the floating point number. The numbers in examples 9 and 10 are both very small fractions; they are not much different (the difference between them is $10^{-26}$), but they are not the same.

More insight into the nature of the mantissa is gained when we consider how floating point numbers are multiplied. Given the two floating point numbers $x = a{\times}2^b$ and $y = c{\times}2^d$, their product is $x{\times}y = (a{\times}c){\times}2^{b+d}$. Thus, to multiply two floating point numbers, their exponents have to be added, and their mantissas should be multiplied. This is easy since the mantissas and exponents are integers, but it involves two problems:

1. The sum $b + d$ may overflow. This happens when both exponents are very large. The floating point product is, in such a case, too big to fit in one word (or one register), and the multiplication results in overflow. In such a case, the ALU should set the V flag, and the program should test the flag after the multiplication, before it uses the result.

2. The product $a{\times}c$ is too big. This may happen often because the product of two $n$-bit integers can be up to $2n$ bits long. When this happens, the least-significant bits of the product $a{\times}c$ should be cut off, resulting in an approximate floating point product $x{\times}y$. Such truncation, however, is not easy to perform when the mantissas are integers, as the next paragraph illustrates.

Suppose that the computer has 32-bit words, and that a floating-point number consists of an 8-bit signed exponent and a 24-bit mantissa (we ignore the sign of the mantissa). Multiplying two 24-bit integer mantissas produces a result that's up to 48 bits long. The result is stored in a 48-bit temporary register, and its most-significant 24 bits are extracted, to become the mantissa of the product $x{\times}y$. Figure 2.17a,b,c illustrates how those 24 bits can be located anywhere in the 48-bit register, thereby complicating the task of extracting them.

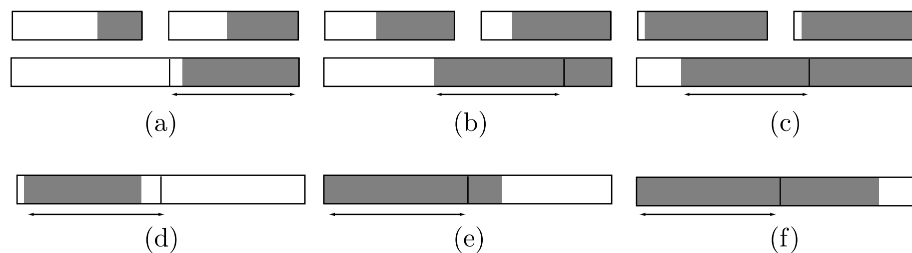



**Figure 2.17:** Multiplying two mantissas

Computer designers solve this problem by considering the mantissa of a floating-point number a fraction, rather than an integer. Thus, a mantissa of $10110\ldots0$ equals $0.1011_2 = 2^{-1} + 2^{-3} + 2^{-4} = 11/16$. The mantissa is stored in the floating point number as $10110\ldots0$, and the (binary) point is assumed to be to the left of the mantissa. This explains the term *mantissa* and also implies that the examples in Table 2.16 are wrong. In practice, the mantissa $a$ is always less than 1, it is a fraction. The examples of Table 2.18 have such mantissas and are therefore more realistic. (They are still wrong because they are not normalized. Normalization of floating-point numbers is introduced below.) The values of the exponents and the mantissas are shown in Table 2.18 both in binary and in decimal. It is clear that the smallest $m$-bit mantissa is $00\ldots01 = 0.00\ldots01_2 = 2^{-m}$ (a small fraction), and the largest one is $11\ldots1 = 0.11\ldots1_2 \approx 0.99\ldots9_{10}$ (very close to 1).

**From the Dictionary**

The decimal part of a logarithm is called mantissa. This word comes from the Latin *mantisa*, meaning addition or make weight, because the decimal part comes in addition to the integral part (the characteristic) of the logarithm.

| | a (binary) | a (decimal) | $b$ (binary) | $b$ (decimal) | $a \times 2^b$ | value |
|---|---|---|---|---|---|---|
| 1 . | 1000 | $2^{-1} = .5$ | 1000 | 0 | $1/2 \times 2^0$ | $1/2$ |
| 2 . | 0100 | $2^{-2} = .25$ | 1001 | 1 | $1/4 \times 2^1$ | $2/4 = 1/2$ |
| 3 . | 0010 | $2^{-3} = 1/8$ | 1010 | 2 | $1/8 \times 2^2$ | $4/8 = 1/2$ |
| 4 . | 0001 | $2^{-4} = 1/16$ | 1011 | 3 | $1/16 \times 2^3$ | $8/16 = 1/2$ |
| 5 . | 1010 | $2^{-1} + 2^{-3} = 5/8$ | 0001 | $-7$ | $5/8 \times 2^{-7}$ | .0048828125 |
| 6 . | 1111 | 15/16 | 0110 | $-2$ | $15/16 \times 2^{-2}$ | .234375 |

**Table 2.18:** More examples of floating-point numbers

Another feature of floating-point numbers is apparent from Table 2.18. Examples 1–4, even though different, all represent the same number, namely 0.5. It turns out that there are normally several (sometimes many) ways of representing a given number as a floating-point number, as illustrated here

| mantissa | exp. | mantissa | exp. | mantissa | exp. | mantissa | exp. |
|---|---|---|---|---|---|---|---|
| $1000_2$ | 4 | $0100_2$ | 5 | $0010_2$ | 6 | $0001_2$ | 7 |

These four numbers have the same value (what is it?), since each was obtained from its predecessor by two steps. First the exponent was incremented by 1—which resulted in multiplying the number by 2—then the mantissa was shifted to the right—which has divided by number by 2—effectively cancelling the change in the exponent!

Which of those different representations is the best? The answer is, the first one, since this is where the mantissa is shifted to the left as much as possible, allowing for the maximum number of significant digits. This representation is called the *normalized representation* of the floating point number, and every floating point number, except zero, has a unique normalized representation. Zero, by the way, can be represented as a floating-point number, with a mantissa of 0 and any exponent. However, the best representation of floating-point zero is a number of all zeros, since this makes floating-point zero identical to integer zero, simplifying a comparison between them.

We therefore conclude that the smallest normalized mantissa is $10\ldots0 = 0.1_2 = 0.5$, regardless of the size of the mantissa.

The ALU circuits are therefore designed to normalize every floating point result when it is generated, and also to test the mantissa of every new floating-point result. If the mantissa is zero, the entire number is cleared to make it a true floating-point zero.

Since the mantissa is a fraction, multiplying two mantissas results in a fraction. If the mantissas are large (i.e., close to 1), their product will be close to 1. If they are small (i.e., 0.5 or a little larger), their product will be $0.25 = 0.01_2$ or a little larger. The point is that it is now easy to identify the region containing

the 24 most-significant bits of the product. As Figure 2.17d,e,f shows, this is normally the left half of the 48-bit register, except when the product is less than 0.5, in which case there is one zero on the left, and the product should be left-shifted one position in the 48-bit register before the leftmost 24 bits are extracted.

▸ **Exercise 2.10:** Can there be more than one zero on the left of the product of two normalized mantissas?

It is important to realize that even though mantissas are fractions, they can be multiplied as integers. The ALU does not need a special circuit to multiply them.

The rules for multiplying the two floating point numbers $x = a\times 2^b$ and $y = c\times 2^d$ are thus (we assume 8-bit exponents and 24-bit mantissas):

1. Add $b + d$. This becomes the exponent $E$ of the product.

2. Multiply $a\times c$ as integers and store the result on the left side of a temporary, 48-bit register. If the leftmost bit of the product is zero, shift the product one position to the left, and compensate by decrementing the exponent $E$ by 1. If $E$ is too big to fit in eight bits, set the V flag and terminate the multiplication.

3. Extract the leftmost 24 bits of the 48-bit register and make them the mantissa $M$ of the product. This mantissa is already normalized.

4. Pack the new exponent $E$ and mantissa $M$ to form the floating-point product of $x$ and $y$.

As can be seen from Table 2.18, the binary representation of the exponent is special in that it uses a sign convention that's the opposite of the normal. A sign bit of 0 indicates a negative exponent, so an exponent of $1001_2$ has a value of +1 and the bit pattern 0110 means an exponent of −2. Another way of looking at the exponent is to visualize it as *biased*. We can imagine the exponent as if a certain number, a bias, is always added to it and—in order to get the true value of the exponent—the bias should be subtracted. In our example, with a 4-bit exponent, the bias is $2^{4-1} = 8$, and we say that the exponent is represented in excess-8. All the 4-bit exponents are summarized in Table 2.19. A biased exponent is sometimes called an *exrad*, meaning, it is an exponent expressed in a special radix.

| Binary | Decimal | After bias is subtracted | Binary | Decimal | After bias is subtracted |
|---|---|---|---|---|---|
| 0000 | 0 | −8 | 1000 | 8 | 0 |
| 0001 | 1 | −7 | 1001 | 9 | 1 |
| 0010 | 2 | −6 | 1010 | 10 | 2 |
| 0011 | 3 | −5 | 1011 | 11 | 3 |
| 0100 | 4 | −4 | 1100 | 12 | 4 |
| 0101 | 5 | −3 | 1101 | 13 | 5 |
| 0110 | 6 | −2 | 1110 | 14 | 6 |
| 0111 | 7 | −1 | 1111 | 15 | 7 |

**Table 2.19:** Biased and unbiased values of a 4-bit exponent

An important example that's missing from Tables 2.16, 2.18 and 2.19 is that of a negative floating point number. Such a number can be represented by adding another bit, a sign bit, that can be considered either the sign bit of the mantissa or the sign bit of the entire number. Thus, a floating-point number has the format as shown in Figure 2.15b. It has three fields, the sign, the mantissa, and the exponent. Notice that the sign bit, even though it is the sign of the mantissa, is the leftmost bit, and is therefore separate from the mantissa. This convention makes it easy to compare integer and floating-point numbers.

How large should the mantissa be? That, of course, depends on how much precision is necessary for a particular calculation. It is worth noting, however, that numbers with many significant digits occur very often even in simple calculations. The familiar number $1/3 = 0.333\ldots$ is a repeating fraction with an infinite number of significant digits. Any number that's a repeating fraction in one number base is also a repeating fraction in any other base. Thus, the binary value of 1/3 is $0.010101\ldots \approx 0.1010\ldots 10\times 2^{-1}$ and, as a floating point number, it has the representation $0\ \underbrace{01\ldots 1}_{e}\ \underbrace{1010\ldots 10}_{m}$. This is an approximate representation of 1/3, accurate to $2^{-m}$ (or to one part in $2^m$), since it has $m$ significant bits.

We now turn to the problem of adding floating-point numbers. Surprisingly, adding floating-point numbers is more difficult than multiplying them. The problem is that the exponents are generally different. In the special case where the exponents are equal, the two numbers can be added by adding their mantissas,

normalizing the sum and packing it with the common exponent. In the general case, where the exponents are different, the first step in adding the numbers is to equate the exponents. This can be done either by decreasing the larger of the two (and compensating by shifting its mantissa to the left), or by increasing the smaller of the two and shifting its mantissa to the right. The ALU uses the latter method.

A simple example is the sum $x + y = 0.1100_2 \times 2^3 + 0.1010_2 \times 2^{-3}$. The first step is to increase the smaller exponent from $-3$ to 3 (6 units) and compensate by shifting the corresponding mantissa 6 positions to the right, changing it from $0.1010_2$ to $.0000001010_2$. The two numbers can now be easily added by adding the mantissas (as if they were integers) and packing the sum with the common exponent. The result in our example is $.1100001010 \times 2^3$, and it serves to illustrate an important point in floating point arithmetic. The result has a 9-bit mantissa, compared to the 2-bit and 3-bit mantissas of the original numbers $x$ and $y$. If we are limited to, say, 4-bit mantissas, we have to truncate the result by cutting off the least significant bits. This, however, leaves us with $0.1100 \times 2^3$, which is the original value of $x$. Adding $x + y$ has resulted in $x$ because of the limited capacity of our computer. We say that this operation has resulted in a complete loss of significance, and many ALUs generate an interrupt in such a case, to let the user know that an arithmetic operation has lost its meaning. This is just one example of the differences between computer arithmetic and mathematics.

Adding two floating-point numbers is done in the following steps:

1. Compare the exponents. If they are equal, go to Step 3.
2. Select the number with the smaller exponent and shift its mantissa to the right by the difference of the exponents.
3. Add the two mantissas.
4. Pack the sum with the larger of the two exponents.
5. Normalize the result if necessary.

Normalization is simple. Each mantissa is in the range $[0.5, 1)$. If the exponents are equal, no mantissa is shifted, and the sum of two mantissas is in the range $[1, 2)$. If one mantissa is shifted to the right, the sum may be less than 1, but is always greater than or equal 0.5. Normalization is therefore needed only if the sum is greater than 1. Since it is always less than 2, the sum of the two mantissas can be normalized by shifting it to the right one position.

▸ **Exercise 2.11:** Add the two floating-point numbers $0|0111|10\ldots0$ and $0|0111|11\ldots0$. Notice that the size of the mantissa is unspecified, but the exponent consists of one sign bit and three bits of magnitude.

Even though floating-point addition is a complex process involving several steps, it can be speeded up by using the idea of pipelining. Several pairs of numbers may simultaneously be in different stages of being added. Pipelining numbers in an ALU circuit is called *vector processing* (Section 5.12).

[Juffa 00] is a detailed bibliography of about 1600 references for floating-point operations.

How many floating-point numbers can be represented in a given computer? In our simple example, the mantissa and exponent occupy four bits each, and there is also the sign bit of the entire number. Each of our floating point numbers thus occupies nine bits which, consequently, allows for up to $2^9 = 512$ numbers to be represented. In general, if the word size is $N$, then $2^N$ floating-point numbers can be represented. Since the sign is a single bit, about half of those numbers are positive and half are negative. Also, since the exponent's sign requires one bit, about half the $2^N$ numbers are less than 1 (in absolute value) and half, greater than 1. The result is (Figure 2.20) that half the floating-point numbers are concentrated in the range $(-1, 1)$ and the other half are in the ranges $(-\infty, -1)$ and $(1, \infty)$.

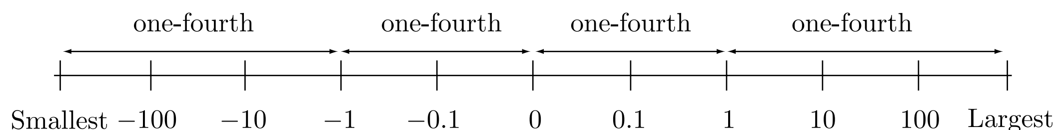



**Figure 2.20:** Ranges of floating-point numbers

More insight into the behavior of floating-point numbers is gained by considering the distance between consecutive floating-point numbers. High-school mathematics teaches that every integer $I$ has a successor,

denoted by $I+1$, but that a real number $R$ has no immediate successor. This is proved by arguing that if $R$ has a successor $S$, then we could always find another number, namely $(R+S)/2$, that's located between $R$ and $S$. The real numbers are therefore said to form a *continuum*. In the computer, in contrast, everything is finite and discrete. Floating-point is normally used as the representation of the real numbers, but there is only a finite quantity of floating-point numbers available ($2^N$, where $N$ is the word size), so they do not form a continuum. As a result, each floating-point number has a successor.

To find the successor of a floating-point number we have to increment its mantissa by the smallest amount possible (we already know that incrementing the exponent, even by 1, doubles the size of the number). The smallest increment is obtained by considering the mantissa an integer and incrementing that integer by one. As an example consider a hypothetical computer with 32 bits per word. If we assume that the exponent field is eight bits wide (one bit for the sign and seven for the magnitude), then the mantissa gets the remaining 24 bits. Again, one bit is reserved for the sign and 23 bits are left for the magnitude. Since the mantissa is a fraction, its 23 bits have values ranging from $2^{-1}$ (for the leftmost bit) to $2^{-23}$ (for the rightmost bit). Incrementing the mantissa by 1 therefore increases its value by $2^{-23}$ and increases the value of the floating-point number by $2^{-23}2^e = 2^{-23+e}$ where $e$ is the exponent of the number.

Since the smallest 8-bit signed exponent is $-128$, we conclude that the smallest possible increment of a floating-point number on our example computer is $2^{-23-128} = 2^{-151} \approx 3.5 \cdot 10^{-46}$. Adding this increment to the smallest floating-point number results in the successor of that number. The quantity $2^{-151}$ is called the *resolution* of the floating-point numbers on the particular computer.

On the other hand, the largest exponent is 127, so the distance between the largest floating-point number and its predecessor is $2^{-23+127} = 2^{104} \approx 2 \cdot 10^{31}$. The conclusion is that as we move, on this particular computer, from the smallest floating-point number to the largest one, the distance between consecutive numbers varies from the very small quantity $3.5 \cdot 10^{-46}$ to the very large quantity $2 \cdot 10^{31}$; a wide range.

The range of floating-point numbers is not hard to estimate. It depends mainly on the size of the exponent and is surprisingly sensitive to that size. Assuming a floating-point representation with an $e$-bit exponent and an $m$-bit mantissa, the smallest positive floating-point value is obtained when the exponent has the largest negative value and the mantissa, the smallest (normalized) value. The format of this number is therefore:

$$0\,\underbrace{00\ldots0}_{e}\underbrace{10\ldots0}_{m} = 1/2 \times 2^{-2^{e-1}} = 2^{-2^{e-1}-1}.$$

Since the mantissa is normalized, the smallest mantissa has a value of 1/2.

The largest floating-point number is obtained when both the exponent and the mantissa have their largest values:

$$0\,\underbrace{11\ldots1}_{e}\underbrace{11\ldots1}_{m} = 0.11\ldots1_2 \times 2^{2^{e-1}-1} = (1-2^{-m}) \times 2^{2^{e-1}-1} \approx 2^{2^{e-1}-1}.$$

Assuming that $2^{-m}$ is a very small fraction, it can be ignored and $(1-2^{-m})$ can be replaced by 1. This is another way of saying the the mantissa only affects the precision and not the size, of the floating-point number. Both extreme sizes depend only on $e$. On the negative side, the largest and smallest negative floating-point numbers are about the same size as their positive counterparts. Table 2.21 summarizes the properties of the floating-point numbers found on some historically-important computers.

| Computer | Exponent size | Mantissa size | Total size | Largest value |
|---|---|---|---|---|
| PDP-11 | 8 | 23 | 32 | $10^{38}$ |
| IBM/370 | 7 | 24 | 32 | $10^{19}$ |
| Cyber 70 | 11 | 48 | 60 | $10^{307}$ |
| VAX | 8 | 24 | 32 | $10^{38}$ |
| CRAY 1 | 15 | 48 | 64 | $10^{2500}$ |
| Intel 8087 | | The IEEE Standard | | |

**Table 2.21:** Floating-point numbers in various computers

Since floating-point numbers are so popular, attempts have been made to standardize their representation. The most important of these is the IEEE floating-point standard [Stevenson 81], The acronym IEEE stands for "Institute of Electrical and Electronics Engineers." Among its many activities, this organization also develops standards. The full title of the IEEE floating-point standard is "ANSI/IEEE Standard 754-1985, Standard for Binary Floating Point Arithmetic." It defines three representations:

1. A Short Real. This has one sign bit, eight exponent bits (the exponent is biased by 127), and 23 significand (mantissa) bits, for a total of 32 bits. The smallest value is

$$2^{-2^{e-1}-1} = 2^{-128-1} = 2^{-129} \approx 10^{-38}.$$

And the largest value is

$$2^{2^{e-1}-1} = 2^{128-1} = 2^{127} \approx 10^{38}.$$

2. A Long Real. The 11-bit exponent (biased by 1023) and 52-bit significand combine with the sign bit to form a 64-bit floating point number. The smallest value is

$$2^{-2^{e-1}-1} = 2^{-1024-1} = 2^{-1025} \approx 10^{-308}.$$

And the largest value is

$$2^{2^{e-1}-1} = 2^{1024-1} = 2^{1023} \approx 10^{307}.$$

3. A Temporary Real. This has an 18-bit exponent (biased by 131071) and a 64-bit significand. This is typically used for intermediate results, which may be very large, to avoid accumulating round-up errors. The smallest value is

$$2^{-2^{e-1}-1} = 2^{-131072-1} \approx 10^{-39360}.$$

And the largest value is

$$2^{2^{e-1}-1} = 2^{131072-1} \approx 10^{39360}.$$

This standard defines a floating-point zero as a number where both the exponent and the mantissa are zeros (the sign, however, may be either 0 or 1). The standard defines any floating-point number with a mantissa of zero and with the largest exponent as *infinity* (again, there are two infinities, with sign bits of 0 and 1). Similarly, any floating-point number with a nonzero mantissa and with the largest exponent is defined as *not a number* (NaN). Such a bit pattern is generated by the ALU whenever a floating-point operation results in a division by zero.

The reason for the name *floating point* should now be obvious. Incrementing the exponent by 1 doubles the size of the floating-point number, and is therefore equivalent to moving the (binary) point one position to the right. Similarly, decrementing the exponent by 1 is equivalent to moving the (binary) point one position to the left.

## 2.22 Fixed-Point Numbers

A floating-point number is constructed from two integers (more precisely, from an integer exponent and a fractional mantissa). Varying the exponent amounts to moving the binary point in the number. We can therefore consider the exponent an indicator of the position of the binary point in the mantissa. The principle of a fixed-point number is to have a constant exponent. If the exponent is always the same, there is no need to store it with the number. The fixed-point number is simply stored as an integer, and we assume that there is an implied binary point at a certain position in the integer. For example, if we assume a binary point to the left of the rightmost bit, then the binary number $0\ldots0111$, that is normally the integer 7, is interpreted as the value $0\ldots011.1_2 = 3.5$. We say that the *fixed-point resolution* in this case is $2^1 = 2$. (Notice that the resolution is a power of 2. Some authors define the resolution as $1/2^1 = 0.5$.) An $N$-bit, fixed-point representation with resolution 2 makes it possible to represent the $2^N$ real numbers

$$-2^{N-1}/2, \ldots, -2, -1.5, -1, -0.5, 0, 0.5, 1, 1.5, 2, 2.5, \ldots, (2^{N-1}-1)/2.$$

Operations on fixed-point numbers represent their own problems. Adding is simple. The real sum $1.5+3 = 4.5$ is performed by adding $3+6 = 9$, and the result is correctly interpreted as 4.5. However, trying to multiply $1.5 \cdot 3 = 4.5$ by multiplying $3 \cdot 6 = 18$, yields a wrong result. The product, 18, has to be divided by the resolution 2 before it can be interpreted as the correct product 4.5.

▸ **Exercise 2.12:** What is the rule for dividing fixed-point numbers?

In the general case, we represent the real number $R$ by the integer $\lfloor 2^p R \rfloor$ where $2^p$ is the fixed-point resolution ($p$ is normally positive). As an example consider the case $p = 3$ on a 16-bit computer. The largest positive integer on such a computer is $2^{15} - 1 = 32{,}767$ (since only 15 of the 16 bits are used for the magnitude). Table 2.22 shows some examples of real numbers and their fixed-point representations.

| $R$ | $8R$ | $\lfloor 8R \rfloor$ | $R$ | $8R$ | $\lfloor 8R \rfloor$ |
|---:|---:|---:|---:|---:|---:|
| .100 | .800 | 0 | −.100 | −.800 | 0 |
| .120 | .960 | 0 | −.120 | −.960 | 0 |
| .125 | 1.000 | 1 | −.125 | −1.000 | −1 |
| .130 | 1.040 | 1 | −.130 | −1.040 | −1 |
| .250 | 2.000 | 2 | −.250 | −2.000 | −2 |
| .260 | 2.080 | 2 | −.260 | −2.080 | −2 |
| 100.000 | 800.000 | 800 | −100.000 | −800.000 | −800 |
| 100.100 | 800.800 | 800 | −100.100 | −800.800 | −800 |
| 100.125 | 801.000 | 801 | −100.125 | −801.000 | −801 |
| 4095.875 | 32767.000 | 32767 | −4095.875 | −32767.000 | −32767 |
| | | | −4096.000 | −32768.000 | −32768 |
| | | | −4096.124 | −32768.992 | −32768 |
| | | | −4096.125 | −32769.000 | −32769 |

**Table 2.22**: Some 16-bit fixed-point numbers with resolution $2^p = 8$

The table shows that the largest fixed-point number that can be represented in this way is 4095.875 (it becomes the integer 32,767) and the smallest fixed-point numbers are any real numbers close to and less than −4096.125 (they are represented by the smallest 16-bit 2's-complement integer, $-32,768$). The real numbers in the interval $(-4096.125, 4095.875]$ are thus represented by $2^{16} = 65{,}536$ different fixed-point numbers. Dividing 65,536 by the length 8,192 of the interval yields the resolution $8 = 2^3 = 2^p$.

Different applications require different fixed-point resolutions, which is why the fixed-point representation is implemented by software. Another point to consider is that floating-point operations are different from integer operations and require hardware implementation for fast execution. In contrast, fixed-point operations are operations on integers and can be performed by the same ALU circuits that perform integer arithmetic.

## 2.23 Decimal (BCD) Numbers

We are used to decimal numbers; computers find it easy to deal with binary numbers. Conversions are therefore necessary, and can easily be done by the computer. However, sometimes it is preferable to avoid number conversions. This is true in an application where many numbers have to be input, stored in memory, and output, with very little processing done between the input and the output. Perhaps a good example is inventory control.

Imagine a large warehouse where thousands or even tens of thousands of items are stored. In a typical inventory control application, a record has to be input from a master file for each item, stored in memory, updated, perhaps printed, and finally written on a new master file. The updating usually involves a few simple operations such as incrementing or decrementing the number of units on hand. In such a case it may be better not to convert all the input to binary, which also saves conversion of the output from binary.

Such a situation is typical in data processing applications, which is why computers designed specifically for such applications support BCD numbers in hardware. The idea in a BCD number is to store the decimal digits of a number in memory, rather than converting the entire number to binary (integer or floating point). Since memory can only contain bits, each decimal digit has to be converted to bits, but this is a very simple process. As an example, consider the decimal number −8190. Converting this number to binary integer is time consuming (try it!) and the result is 13 bits long (14, including the sign bit). On the other hand, converting each decimal digit is quick and easy. It yields 1000 0001 1001 0000. Each decimal digit is represented as a group of four bits, since the largest digit (=9) requires four bits.

The principle of this number representation is to code each decimal digit in binary, hence the name *binary coded decimal* (BCD). It turns out that the sign of such a number can also be represented as a group of four bits, and any of the six unused groups $1010, 1011, \ldots, 1111$ (representing the numbers 10–15) can be used. Assuming that 1101 represents a negative sign, our number can be represented as 1101 1000 0001 1001 0000. This is very different from the integer representation of $-8190 = 1|0000000000010_2$ and requires 20 bits instead of 14. It also requires special ALU circuits for the arithmetic operations. The only advantage of BCD numbers is the easy conversion. In many data processing applications, this advantage translates into considerable increase in speed, justifying hardware support of this representation.

---

**Computer arithmetic as distinct from arithmetic in math**

Computers, of course, are commonly used for numerical calculations. However, computer arithmetic is different from the arithmetic that we study in school, in two ways: (1) computers have finite capacity and (2) they operate at a finite (albeit high) speed. The finite capacity means that certain numbers (very large numbers, very small ones, and numbers with many significant digits) cannot be stored in the computer. If such a number is generated during calculations, only an approximate value can be stored. If the number is a final result, this may not be a serious problem. If, however, it is an intermediate result, needed for later calculations, its approximate value may affect the final result seriously and we may end up with a wrong result. This is especially noticeable in floating point calculations, where the hardware may generate approximate results, store them in memory, and use them in later calculations, without any error message or even a warning.

It is well known that any numerical algorithm must be stable. If a numerical method is not stable, small arithmetic errors, resulting from the use of approximate numbers, may accumulate and cause a wrong final result.

The other difference, the finite speed of the computer, is both a blessing and a curse. Computers are, of course, much faster than humans and make possible many calculations that, in the past, were impossible because of the amount of work involved. On the other hand, there are calculations that, even on the most powerful computers, may take many years to accomplish. Two familiar examples are the search for large prime numbers and code breaking.

Even though prime numbers don't have many practical uses, mathematicians keep searching for larger and larger ones. Since there are infinitely many prime numbers, there is a limit even to the power of the fastest computers to find large primes.

Secret codes (Section 3.16) are certainly very practical. Cryptogtraphers are engaged in developing secure codes but, at the same time, cryptanalysts are kept busy trying to break those very codes. It is a common belief today that there is no absolutely secure code that will also be practical (although impractical codes may be fully secured). Any code can be broken if it is used enough times, given enough expertise and enough computer time. In practice, however, a code that takes a year of computer time to break can be considered very secure. Such codes exist and are an example of the limitations put on computer arithmetic because of the limited speed of computers.

---

## 2.24 Other BCD Codes

A BCD code is any binary code assigned to the ten decimal digits. Many such codes are possible, and some have special properties that make them useful for certain applications. Since there are 10 decimal digits, any BCD code must be at least four bits long, but only 10 of the 16 4-bit combinations are used.

▸ **Exercise 2.13:** How many BCD codes are possible?

**The Excess-3 Code**. This code (also referred to as XS3) is derived from the familiar BCD code by adding 3 to each of the 10 codes (Table 2.23). Thus, 0 is represented in XS3 by 0011 (3), 1 is represented by 0100 (4), and so on up to 9, which is represented by 1100 (12). This code simplifies the addition of BCD digits. Adding two such digits results in a sum that has an excess of 6. Suppose that we add $a + b = c$. If $a + b \geq 10$, then $c \geq 16$ and therefore has the form $1xxxx$. The correct result is obtained if we consider the 1 on the left the tens digit of the sum, delete it, and add 3 to the remaining four bits of $c$. An example is $4 + 8 = 12$. In excess-3 this produces $0111 + 1100 = 10010 = 18$. Removing the tens digit leaves 0010, and adding 3 produces 0101. The final result is 12, where the most-significant digit is 1 and the least-significant

digit is 0101 (2 in XS3). On the other hand, if $a+b<10$, then $c<16$ and the correct result is obtained if 3 is subtracted from $c$. As an example, consider $2+6=0101+1001=1110=14$. When 3 is subtracted, the result is $14-3=11$, which is 8 in XS3.

Many BCD codes are *weighted*. Each bit $b_i$ in such a code is assigned a weight $w_i$, and the decimal digit represented by bits $b_3b_2b_1b_0$ is the weighted sum $b_3w_3+b_2w_2+b_1w_1+b_0w_0$. The familiar BCD code is the weighted 8421 code. Table 2.23 lists the $631\bar{1}$ and 2421 weighted codes.

| | XS3 | $631\bar{1}$ | 2421 | III | II | I |
|---|---|---|---|---|---|---|
| 0 | 0011 | 0011 | 0000 | 1001 | 1000 | 0000 |
| 1 | 0100 | 0010 | 0001 | 1000 | 0011 | 0001 |
| 2 | 0101 | 0101 | 0010 | 0101 | 0010 | 0010 |
| 3 | 0110 | 0111 | 0011 | 0000 | 1011 | 0011 |
| 4 | 0111 | 0110 | 0100 | 1100 | 0000 | 0100 |
| 5 | 1000 | 1001 | 1011 | 0011 | 0100 | 1100 |
| 6 | 1001 | 1000 | 1100 | 1111 | 1111 | 1011 |
| 7 | 1010 | 1010 | 1101 | 1010 | 0110 | 1010 |
| 8 | 1011 | 1101 | 1110 | 0111 | 0111 | 1001 |
| 9 | 1100 | 1100 | 1111 | 0110 | 1100 | 1000 |
| | | self complementing | | | reflected | |

**Table 2.23:** Six BCD codes

▸ **Exercise 2.14:** Show the 5421, 5311, and 7421 BCD codes.

The weights must be selected such that they can produce the ten decimal digits. The set 8765, for example, cannot serve as weights since the digits 1–4 cannot be expressed as a weighted sum of 8, 7, 6, and 5. Weighted BCD codes with more than four bits are sometimes useful when increased reliability is needed. For example, a parity bit can be added to the 8421 code, to produce the $p8421$ code. A 2-out-of-5 code has five bits, two of which are 1's. This can be used for error detection. The number of ways to select two objects out of five is

$$\frac{5!}{2!(5-2)!}=10,$$

and these ten codes are assigned to the ten digits such that this code looks like the weighted code 74210, with the exception of the code for 0.

| 00011 | 00101 | 00110 | 01001 | 01010 | 01100 | 10001 | 10010 | 10100 | 11000 |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0. |

The biquinary code (2-out-of-7) is a 7-bit code where exactly two bits are 1's. This is almost twice the number of bits required to code the ten decimal digits, and the justification for such a long code is that it offers a simple error checking and simple arithmetic rules. The seven code bits are divided into two groups of two bits and five bits, with a single 1 bit in each group, as shown in Table 2.24.

| | 65 43210 |
|---|---|
| 0 | 01 00001 |
| 1 | 01 00010 |
| 2 | 01 00100 |
| 3 | 01 01000 |
| 4 | 01 10000 |
| 5 | 10 00001 |
| 6 | 10 00010 |
| 7 | 10 00100 |
| 8 | 10 01000 |
| 9 | 10 10000 |

**Table 2.24:** The 2-out-of-7 BCD code

The first four codes of Table 2.23 are self-complementing. The 1's complement of the code of digit $d$ is also the 9's complement of $d$. Thus, the XS3 code of 3 is 0110 and the 1's complement 1001 is the XS3 code of 6. Notice that the last of these codes, code III, is unweighted.

The last two codes of Table 2.23 are reflected BCD codes. In such a code, the 9's complement of a digit $d$ is obtained by complementing just one bit. In code II, this bit is the leftmost one and in code I it is the second bit from the left.

**Unit distance codes**: The property of unit distance codes is that the codes of two consecutive symbols $x_i$ and $x_{i+1}$ differ by exactly one bit. The most common unit distance code is the Gray code (sometimes called *reflected Gray code* or RGC), developed by Frank Gray in the 1950s. This code is easy to generate with the following recursive construction:

Start with the two 1-bit codes $(0, 1)$. Construct two sets of 2-bit codes by duplicating $(0, 1)$ and appending, either on the left or on the right, first a zero, then a one, to the original set. The result is $(00, 01)$ and $(10, 11)$. We now reverse (reflect) the second set, and concatenate the two. The result is the 2-bit RGC $(00, 01, 11, 10)$; a binary code of the integers 0 through 3 where consecutive codes differ by exactly one bit. Applying the rule again produces the two sets $(000, 001, 011, 010)$ and $(110, 111, 101, 100)$, which are concatenated to form the 3-bit RGC. Note that the first and last codes of any RGC also differ by one bit. Here are the first three steps for computing the 4-bit RGC:

$$\text{Add a zero } (0000, 0001, 0011, 0010, 0110, 0111, 0101, 0100),$$
$$\text{Add a one } (1000, 1001, 1011, 1010, 1110, 1111, 1101, 1100),$$
$$\text{reflect } (1100, 1101, 1111, 1110, 1010, 1011, 1001, 1000).$$

▸ **Exercise 2.15:** Write software to calculate the RGC and use it to compute and list the 32 5-bit RGC codes for the integers 0–31.

The conversion from the Gray code of the integer $i$ back to its binary code is $b_i = g_i \oplus b_{i+1}$, for $i = n, n-1, \ldots, 1, 0$, where $g_n \ldots g_2 g_1 g_0$ is the RGC of $i$ and $0b_n \ldots b_2 b_1 b_0$ is the binary code of $i$ (bit $b_{n+1}$ is zero).

It is also possible to generate the reflected Gray code of an integer $n$ with the following nonrecursive rule: Exclusive-OR $n$ with a copy of itself that's logically shifted one position to the right. In the C programming language this is denoted by `n^(n>>1)`.

**History of Gray Codes**

Gray codes are named after Frank Gray, who patented their use for shaft encoders in 1953. However, the work was performed much earlier, the patent being applied for in 1947. Gray was a researcher at Bell Telephone Laboratories. During the 1930s and 1940s he was awarded numerous patents for work related to television. According to [Heath 72] the code was first, in fact, used by J. M. E. Baudot for telegraphy in the 1870s (Section 3.7.2), though it is only since the advent of computers that the code has become widely known.

The Baudot code uses five bits per symbol. It can represent $32 \times 2 - 2 = 62$ characters (each code can have two meanings, the meaning being indicated by the `LS` and `FS` codes). It became popular and, by 1950, was designated the International Telegraph Code No. 1. It was used by many first- and second generation computers.

The August 1972 issue of *Scientific American* contains two articles of interest, one on the origin of binary codes [Heath 72], and another [Gardner 72] on some entertaining aspects of the Gray codes.

The binary Gray code is fun,
For in it strange things can be done.
Fifteen, as you know,
Is one, oh, oh, oh,
And ten is one, one, one, one.

—Anonymous

## 2.25 Rational Numbers

Rational numbers provide an alternative to floating-point and fixed-point numbers. A number $x$ is represented as a pair of integers $a$ (signed), and $b$ (unsigned), usually packed in one $N$-bit word, representing the value $x = a/b$. If $b$ occupies $M$ of the $N$ bits, then $a$ is $N - M$ bits long. Figure 2.25 depicts two possible ways of using such numbers, namely fixed-slash and floating-slash.

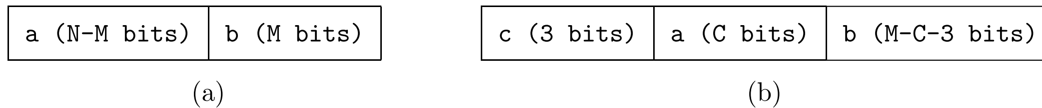



**Figure 2.25:** Rational numbers, a. fixed-slash, b. floating-slash

In a fixed-slash rational, the value of $M$ is fixed. All such numbers allocate $M$ of the $N$ bits to $b$, and $N - M$ bits to $a$. In a floating-slash rational, the relative sizes of $a$ and $b$ may vary and the $N$ bits of a word are divided into three fields. A count field $C$ (of fixed size, usually 3–4 bits) specifying the size of $a$, followed by the $C$ bits of the $a$ field, followed by the $N - C - 3$ bits of the $b$ field.

The advantages of rational numbers are as follows:

- Many numbers that can be only approximately represented as floating point numbers, can be represented as rational numbers exactly.

- Certain mathematical operations produce very large intermediate results (although the final results may be reasonably sized). Using floating point arithmetic, every time a very large result is generated, precision (i.e., significance) may be lost. With rational arithmetic, it is easier to represent large results in double precision (using a pair of words for each number) without losing accuracy. The final results may be represented in single precision.

- It is easy to obtain the inverse of a rational number, and the inverse is always exact. The inverse of [a|b] is, of course [b|a]. The inverse is easy to generate and store as a rational number. This is not always true for floating point numbers.

The main questions that have to be answered before implementing such a representation on a computer are:

- How complex are the arithmetic operations on such numbers? If it is very difficult to add or to multiply such numbers, the rational representation may not be practical.

- How accurate can those numbers be? Given an $N$-bit word, what are the smallest and largest numbers that can fit in the word.

- Compatibility. Performing the same calculations on computers with different word size may not yield the same results. How does the accuracy of the result depend on the word size?

A good reference for fixed-slash and floating-slash representations is [Matula and Kornerup 78].

## 2.26 Carry and Overflow

These two concepts are associated with arithmetic operations. Certain operations may result in a carry, in an overflow, in none, or in both. There is much confusion and misunderstanding among computer users concerning carry and overflow, and this section attempts to clear up this confusion by defining and illustrating these concepts.

Definitions: Given two $N$-bit numbers used in an addition or subtraction, we would like, of course, to end up with a result that has the same size, and fits in the same registers or memory words where the original operands came from. We consequently say that the operation results in a *carry* if the sum (or difference) has $N + 1$ bits. Similarly, the operation results in an *overflow* if the result is too large to fit in a register or a computer word.

These definitions look suspiciously similar, and the next example only reinforces this suspicion. Imagine a computer with 3-bit words (so the largest number is 7) and we first try to add 5+7. In binary, the operation

is: $101 + 111 = 1100$. The addition is done in the usual way, moving from right to left and propagating the carry. The result, naturally, is 12.

Did we get a carry? Yes, since we started with 3-bit numbers and ended up with a 4-bit result. Did we get an overflow? Yes, since the maximum number that can be represented on our computer is 7, and the result is 12, a number too large to fit in a 3-bit register.

It seems that although our definitions of carry and overflow look different, they are actually two aspects of the same phenomenon, and always occur together, which should make them identical.

The explanation is that yes, in the above example, carry and overflow always go together, but in a real computer, they are different. This is because real computers use signed numbers, while our example uses unsigned numbers. When signed numbers are used, with the leftmost bit reserved for the sign, carry and overflow are indeed different.

To illustrate this, we extend our computer's capacity to four bits per word, a sign bit plus three magnitude bits. Working with 2's complement numbers, the range of numbers that can be represented is from $-8$ to $+7$. We use the two numbers $5 = 0\,101_2$ (with $-5 = 1\,011_2$) and $6 = 0\,110_2$ (with $-6 = 1\,010_2$) for the examples in Table 2.26.

The table summarizes four different operations on these numbers. It is clear that carry and overflow do not always occur together; they are different aspects of the arithmetic operations. The carries in the table are generated when the results are 5 bits long. The overflows occur when the results do not fit in a register, i.e., they are less than $-8$ or greater than 7.

| decimal operands | 5<br>6 | −5<br>−6 | 6<br>−5 | −6<br>5 |
|---|---|---|---|---|
| binary operands | 0101<br>0110 | 1011<br>1010 | 0110<br>1011 | 1010<br>0101 |
| result: | 1011 | 10101 | 10001 | 1111 |
| carry: | no | yes | yes | no |
| overflow: | yes | yes | no | no |

**Table 2.26:** Carry and overflow

Note that a carry does not necessarily indicate a bad result. The two examples with carry have correct results and the carries can simply be ignored. The reason the ALU still has to detect a carry has to do with double-precision arithmetic and is explained below. Overflow, on the other hand, is always bad and should be detected by the program by means of a conditional branch (most computers can optionally generate an interrupt when overflow is detected).

An interesting question is, how does the ALU detect overflow? Carry is easy for the ALU to detect, since it involves an additional bit. Overflow, however, causes a wrong result and is easy for us to detect, since we know in advance what the result should be. The ALU, however, does not know the result in advance, and a simple test is needed to let the ALU detect overflow.

This test involves the sign bits. A look at Table 2.26 verifies that every time overflow occurs, a bit has overflowed into the sign position, corrupting the sign. Overflowed results always have the wrong sign. Since the sign is easy to determine in advance, the ALU uses it as an indication of overflow. The test uses the simple fact that, when adding numbers of the same sign, the result should have that sign. Thus, when two positive numbers are added, the result should be positive. The ALU therefore performs the following test. If the two numbers being added have the same sign and the result has the opposite sign, the overflow flag should be set.

▸ **Exercise 2.16:** What is the overflow test when adding numbers with different signs?

The last point discussed here is the use of the carry. Normally, the presence of a carry does not indicate a bad result, and carries are ignored. However, the carry is important in double-precision arithmetic. On a computer with a short word size, the largest number that can be stored in a single word may sometimes be too small and, in order to handle larger numbers, double precision is necessary. In double-precision arithmetic, a number is stored in two consecutive words (Figure 2.27), where only one word has a sign bit.

**The 9/19/89 Syndrome.**

How can a date, such as 11/12/71, be represented inside a computer? One way to do this is to store the number of days since January 1, 1900 in an integer variable. If the variable is 16 bits long, it will overflow after $2^{15} = 32K = 32,768$ days, which is September 19, 1989. This is precisely what happened on that day in several computers (see the Jan., 1991 issue of the *Communications of the ACM*). Notice that doubling the size of such a variable to 32 bits would have delayed the problem until after $2^{31} =$ 2giga days have passed, which would occur sometimes in the fall of year 5,885,416.

Adding such numbers involves two steps (1) the least significant parts are added (producing, perhaps, a carry and/or overflow), (2) the most significant parts are added, together with any carry from the first step.

**Figure 2.27:** Double precision numbers

Example: Add 25 and 46 on a computer with four-bit words. Each number is represented in two parts 25 = 0001 1001 and 46 = 0010 1110. Adding the least significant parts 1001 + 1110 produces 0111 plus both carry and overflow. The overflow simply indicates that the leftmost bit of the sum (=0) is different from the leftmost bits of the original operands (=1). Since we know that those bits are magnitude, and not sign bits, the overflow indication can be ignored. The carry, however, should not be ignored; it should be carried over and added to the most significant parts. Adding these parts 0001 + 0010 produces 0011, and adding the carry yields the final result 0100 which, combined with 0111, produces the correct sum, 0100 0111 = 72.

Sir Thomas would fall back from his usual modes
of expression when he was a little excited.

—Anthony Trollope, 1881, *Ayala's Angel*

The modes and sources of this kind of error are well
typified in the contemplation of the heavenly bodies

—Edgar Allan Poe, *The Murders in the Rue Morgue*

# 3
# Input/Output

The discussion so far has concentrated on the computer's important internal operations, such as instruction fetch and execute, and the handling of interrupts. The computer, however, has to communicate with the outside world, a task that is done by means of input/output. This is why an understanding of computers and their operations is incomplete without an understanding of input/output (or I/O for short).

This chapter consists of three main parts, the I/O processor, I/O codes, and computer communications. The first part discusses important techniques for input/output, such as register I/O and DMA. The second part is a survey of codes used for computer data, such as the ASCII code, error-correcting codes, codes used to compress data, and secure codes. The third part is concerned with the way binary data is sent between computers. It shows how bits can be sent serially, one by one, on a single communications line and how computers can be networked.

The input/output devices themselves are not described; this chapter concentrates on how data is moved in and out of the computer.

▸ **Exercise 3.1:** Is it possible to have a meaningful program without input and output?

## 3.1 The I/O Processor

The I/O processor is the third component of the mainframe—the CPU and the memory being the first two components. The I/O processor is responsible for interfacing the computer to the outside world, i.e., to the I/O devices. The control unit uses the I/O processor to execute I/O instructions in much the same way that it uses the ALU to execute instructions that operate on numbers.

The I/O processor is usually interfaced to the CPU and to the memory by means of the computer buses (Figure 3.1), and the first example will show why an understanding of computer buses is essential to an understanding of the I/O processor.

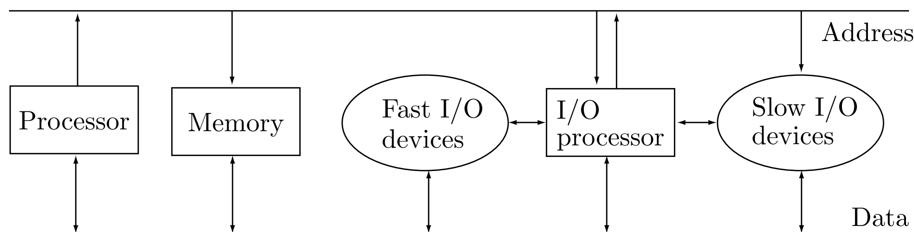



**Figure 3.1:** Organization of a mainframe around the address and data buses

The example has to do with the way the control unit executes two different but similar instructions, a `LOD` and an `IN`. The execution of the instruction `LOD R1,17` has already been discussed in Section 1.4; it involves the steps

1. Address bus←$IR_c$
2. 'read'
3. wait
4. R1←Data bus

A typical input instruction is `IN R1,17`. It instructs the computer to input one piece of data (normally one byte) from input device 17 and store it in register1. This instruction is executed by the following, similar steps:

1. Address bus←$IR_c$
2. 'input'
3. wait
4. R1←Data bus

In either case the number 17 is sent, from $IR_c$, to the address lines. In the former case it should go to memory, to select location 17. In the latter case it should go to the I/O devices, to select device 17. How is the number 17 sent to the right place? The answer has to do with the *bus organization* of the computer.

There are two main ways to organize the buses of a computer. They are called a *single-bus organization* and a *double-bus organization*. There are also combinations and variations of these two methods.

A double bus organization is shown in Figure 3.2a. The principle is to have two buses, each with its own address, data, and control lines. One bus connects the CPU to the memory and the other connects it to the I/O devices.

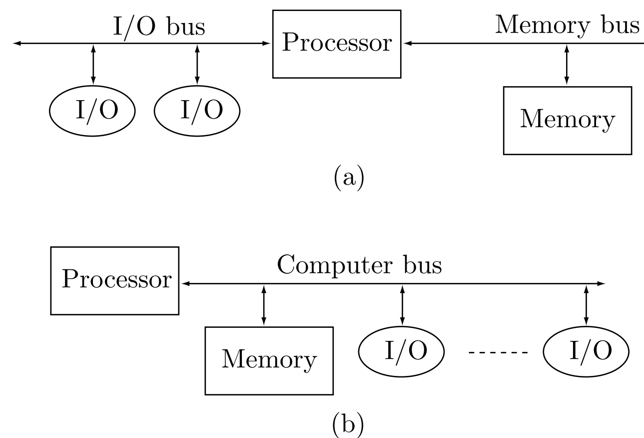



**Figure 3.2:** Two bus organizations of a computer

The control unit of such a computer knows about the two buses, and always uses the right one. When it executes a `LOD` or any other memory reference instruction, it sends and receives address and data signals on the memory bus. Similarly, the I/O bus is used to route signals when an I/O instruction (such as `IN` or `OUT`) is executed. In such a computer, address 17 can be sent to either bus. When sent on the memory bus it identifies memory location 17. When sent on the I/O bus, the same number selects device 17. We say that a double-bus computer has two *address spaces*. In a typical double-bus computer, the memory bus may have 25 address lines and the I/O bus may have 12 address (or rather, *device select*) lines. The memory address space in this case has $2^{25} = 32$M addresses, while the I/O address space has $2^{12} = 4$K select numbers.

▸ **Exercise 3.2:** Does that mean that the computer can have 4K I/O devices?

Just having two separate buses does not mean that they are used simultaneously. The control unit executes only one operation at a time, and therefore only one bus is used at any given time. However, when the computer has a DMA device or an I/O channel (Section 3.4), the two buses may be used at the same time.

Figure 3.2b shows a single bus organization of a computer. There is only one bus, with one set of address lines. These lines carry signals to the memory *and* to the I/O devices. The problem with this organization is that if an instruction sends the number 17 on the address lines, then memory and I/O devices have to have a way to figure out whether this 17 is a memory address or is a device select number. There are three solutions to this problem.

1. The single-bus computer has only one address space, i.e., each address appears only once, and is *either* the address of a memory location or the select number of an I/O device. If the single bus contains 16 address lines—for an address space of 64K—then the address space may be divided, for example, into a 62K partition and a 2K partition. The lowest partition contains 62K memory addresses, while the upper partition has 2K I/O device numbers. The 16 address lines allow for only 62K (instead of 64K) memory words, since part of the address space must be reserved for I/O device numbers. Often, it is easy to modify the way the address space is partitioned. A user with modest memory requirements and many I/O devices may want a division of 60K + 4K or a different partition. As a result, a single-bus organization offers more flexibility than a double bus organization.

One feature of the single-bus organization is that the control unit executes the `LOD` and `IN` instructions in an identical way. If we compare the sequences for the two instructions `LOD R1,17` and `IN R1,63017` we find that in either case the address is sent from the $IR_c$ to the same address lines and the only difference between the two instructions is the value of the address.

As a result, a single-bus computer has no need for separate I/O instructions, and the input/output is done by means of memory reference instructions. This is called *memory-mapped I/O*.

2. The single bus gets one more line added to it, to distinguish between memory addresses and I/O device select numbers. When a memory-reference instruction, such as `LOAD` or `STORE` is executed, the control unit sends a zero on this line. When an I/O instruction, such as `IN` or `OUT` is executed, the control unit sends a 1 on the same line. The memory unit examines this line every time it senses something new on the address lines. If it sees a zero, it knows that the address lines carry a valid address. The I/O devices do likewise.

3. Not every computer is a pure single-bus or double-bus machine. It is possible to design computers whose bus organization is a combination of the two. The Intel 8080, one of the first successful 8-bit microprocessors, uses a single bus with 16 address lines. It is possible to use the 8080 as a single-bus machine with memory-mapped I/O and, in such a case, some of the 64K addresses in the address space must be devoted to I/O device select numbers.

It is, however, also possible to use the 8080 as a double-bus computer. It has two instructions, `IN` and `OUT`, that when executed, send a device select number on 8 of the 16 address lines. The particular organization that's being used at any time, single bus or double bus, is determined by the state of an additional control line, the M/I line. When the control unit executes an `IN` or an `OUT` instruction, it sends the device number on the lower 8 bits of the address bus and sets the M/I line to 0, to indicate an I/O operation. When a memory reference instruction is executed, the control unit sends a 16-bit address on the address lines and sets the M/I line to 1, to indicate a memory operation. When the 8080 is used as the CPU of a computer, the computer should be designed in such a way that memory would be disabled when the M/I line is low.

The description above is a simplification of the actual operation of the 8080. For more information on how the 8080 handles memory and I/O operations, see [Intel 75].

Following this introduction, the I/O processor can be described. Five types of I/O processors are discussed in Sections 3.2 through 3.5 and are summarized in Table 3.3.

| | name | location | features |
|---|---|---|---|
| 1. | Register (polled) I/O | part of the | low volume & |
| 2. | Interrupt I/O | control unit | low speed |
| 3. | DMA | an independent | High volume & |
| 4. | I/O Channel | part of the | fast. |
| 5. | Peripheral Processor | mainframe | Intelligent control |

**Table 3.3:** Five types of I/O processors

## 3.2 Polled I/O

This method, also called *register I/O* (Section 1.8), is used for low-volume transfer of information between the CPU and slow I/O devices. The data is transferred, one byte at a time, between the I/O device and one of the registers in the CPU. A typical instruction is `IN R1,17` which attempts to read one byte of information from device 17 and move it to register 1 in the CPU. The execution of such an instruction is similar to a memory read. The device number (=17) is sent on the device select lines, followed by an 'input' signal to indicate the direction of data transfer. The control unit then waits for the I/O device to prepare the data, then moves the data lines into register 1.

The main difference between this sequence of steps and a memory read is in the wait period. In the case of a memory read, the wait is necessary, because memory is slow. In the case of an input, the wait is necessary not just because the I/O device is slow but because there may not be any data in the device to send to the computer. Consider, for instance, the case of a keyboard. The keyboard is certainly an input device. However, it can send data only after the user has pressed a key. The user, however, is human and is therefore limited to a rate of at most 5–10 characters/s. This is very fast for a human typist but very slow for a computer, which can easily execute millions of instructions per second. At a rate of 10 char/s., the computer can execute 100,000 or more instructions between successive key strokes. This means that, when the `IN R1,17` is executed, there is a very good chance that the keyboard will have no input to send.

As a result, before the program attempts to execute the "`IN`" instruction, it should make sure that the input device really has input to send. We say that the "`IN`" instruction should only be executed when the device is *ready*. A program using register I/O should therefore include test instructions before each I/O instruction. An I/O device should have at least two select numbers, one for the data and the other, for its status. Assuming an input device with the two select numbers 16 (for status) and 17 (for data), a typical input sequence is:

```
LP: IN  R0,16
    BMI LP
    IN  R1,17
```

The first `IN` inputs the status from select number 16 (the status number of the input device) into register 0. We assume that the status is one bit with 0 meaning "ready" and 1 meaning "not ready." We also assume that the status bit is moved into the leftmost bit of R0. The `BMI` (Branch on MInus) is a conditional branch that always tests the sign of the most recent result. In our example, that result is the input into R0. If R0 is negative (status = 1), the `BMI` branches back, to input the status again. The first two instructions thus constitute a tight loop that iterates as many times as necessary until the device status changes to "ready". At that point, the `IN R1,17` is executed, which actually inputs the data from device select 17 (the data number of the input device).

Clearly, a lot of computer time can be spent (wasted?) on such tight test loops. If the computer can execute 100,000 instructions between successive keystrokes, often all those instructions will be iterations of the test above. This is why register I/O is useful only for small quantities of I/O, and can be used only in a single-user computer.

### 3.2.1 Memory-Mapped I/O

In a single-bus computer with one address space, some addresses are reserved for I/O device select numbers, and input/output is done by memory-reference instructions. If address 65000 is reserved as a device select

number, then the instruction `LOD R1,65000` reads data from an input device instead of loading data from memory. When register I/O is performed on such a computer, it is called *memory-mapped I/O*.

## 3.3 Interrupt I/O

The concept of interrupts (Section 1.7) is general and is used extensively in computers, not just for I/O. To use interrupts for I/O (see also Section 1.8), the I/O device must be able to interrupt the computer. The device generates an interrupt whenever it switches its state to "ready." For an input device, a "ready" state means that the device has input to send to the computer. For an output device it means that the device is ready to accept the next byte of output from the computer.

The concept of interrupt driven I/O is best illustrated by considering a slow output device, such as a dot-matrix printer (yes, some of them are still in use). Imagine a slow printer, with select numbers of 14 (for the status) and 15 (for the data), printing at a rate of 100 char/s (very fast for a mechanical device). In the time it takes the printer to print one character, the computer can execute 10,000 or more instructions. The computer first prepares the data in a buffer in memory starting at, say, address `B`. It then turns on the printer interrupts, loads the first character to be printed into a register (register 2 in our example), and goes into a tight test loop where it checks status and prints that character.

```
WAIT: IN  R0,14
      BMI WAIT
      OUT R2,15
```

After the first character has been printed, the main program can forget about the printer output for a while and can continue its main task. After about 10,000 instructions, the printer will finish with the first character and will interrupt the computer. The interrupt handling routine is invoked and should execute the following:

Test buffer `B` to see whether more characters remain to be printed.
If yes, execute:
```
 LOD R2,from current position in buffer B
 OUT R2,15
```
 Update current position in buffer.
Else (buffer is empty) set a flag in memory.
Return.

This arrangement frees the program from having to test device status and wait for a "ready" status. Once the program verifies that the printer is ready and will take the first character, it can switch to other tasks. The entire output is performed by the interrupt handling routine. When there is no more output to be sent to the printer, the handling routine sets a special flag in memory. The main program should test this flag when it wants to use the printer again. If the flag is set, the printer is ready for the next stream of output. Otherwise, the flag should be tested a while later.

Interrupt I/O can also be used for input. However, a program often needs the complete input in order to proceed and, in such cases, using interrupt I/O does not save any time. Consider, for example, a command-driven program. The next command is entered, character by character, from the keyboard, the last character being a carriage return (*cr*). While the command is being entered, the individual characters can be displayed. However, the program cannot execute the command until it is fully entered (i.e., until the *cr* is sent from the keyboard). In such a case, the program cannot do anything useful while the command is being entered, and it therefore does not matter if the input is executed by register I/O (which occupies the main program) or by interrupt I/O (which only occupies the interrupt handling routine).

Interrupt I/O saves computer time, especially if the I/O device is slow. This method is used for slow devices and for low- to medium volumes of information.

## 3.4 DMA

The acronym DMA stands for *direct memory access*. In this method, the I/O processor, called a DMA device, handles the entire input or output process without help from the CPU. All that the CPU has to do is initialize the DMA device by means of a few instructions, following which the DMA device starts, and it performs the entire input or output operation independently of the CPU. After initializing the DMA device, the CPU continues its fetch-execute cycle and executes the program without having to worry about the details of the I/O. This makes DMA suitable for large quantities of I/O, and also makes it fast. In fact, using DMA is like having two processes running simultaneously in the computer. One is the processor, executing programs, and the other is the DMA device, controlling the I/O process.

The DMA device causes data to be moved between the I/O device and memory *directly*. The word "directly" is important; it means that the data moves between the two components in the shortest possible way, without going through the processor or through the DMA device.

To start a DMA process, the program (in a higher-level language) executes a statement (or a command) such as `read(vol, filename,buf,10000)`. This command is compiled into a machine instruction (normally called `BRK` or `SVC`) that creates an artificial interrupt (or a software interrupt, Section 1.7) that tells the operating system that the program is asking for service (in this case, a DMA transfer), and provides the operating system with the following pieces of information:

1. The direction of the data. It can be either input or output.
2. The volume name (the volume is normally a disk or a CD-ROM) and the filename.
3. The start address of the memory buffer reserved for the file.
4. The size of the buffer.

The operating system uses the volume's name to select one of possibly several DMA devices. It uses the volume's directory to find the volume address of the file (and its size, if known), and prepares the following items:

1. The direction of the DMA transfer (one bit).
2. The volume address (on a disk, this is the track and sector numbers) of the file.
3. The start address of the memory buffer.
4. The size of the buffer.
5. The file size (if known).
6. Other information, such as a start/stop bit, and a bit which indicates whether the DMA should issue an interrupt when it is done.

The DMA device has several internal registers where this information is stored. One register stores the buffer start address, another stores the buffer size, a third one saves the disk address, and a fourth one contains all the other items. The operating system starts a DMA operation by sending the five items above to the registers. This information is called the *DMA command*. The last part of the command contains a bit that actually starts the DMA device. Figure 3.4 shows a typical arrangement of the registers.

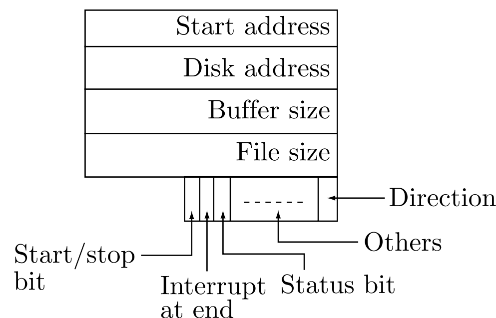



**Figure 3.4. Typical DMA registers**

How is the command sent to the DMA device? This is done by the operating system but without the need for any special instructions. Each of the DMA registers is assigned a device select (or device id)

number, and the operating system uses register I/O or memory-mapped I/O to send the individual parts of the command to the DMA registers. Thus, assuming that the individual parts of a DMA command have already been generated by the program and stored by it in the four registers R1–R4, and also assuming that the DMA registers have id numbers 6–9, the operating system should execute the following instructions:

```
OUT 6,R1
OUT 7,R2
OUT 8,R3
OUT 9,R4
```

The last part of the DMA command also contains the bit that starts the DMA device. When the device starts, it enters a loop where, in each iteration, it causes the transfer of one byte between the I/O device (the volume) and memory. Note that the start/stop bit acts as an on/off switch. In order to start the DMA device, the operating system should set the bit. When the DMA device finishes, it clears the bit automatically. Also the operating system can clear the start/stop bit at any time if it decides to stop the DMA prematurely.

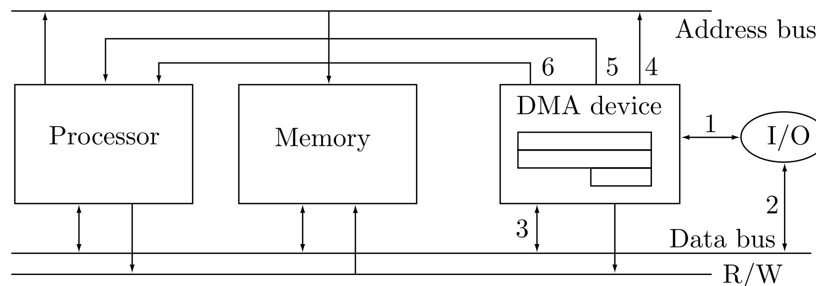



**Figure 3.5:** A DMA device

A complete process of DMA input is shown here as an example. Figure 3.6 shows the details. Data bytes should be read from the I/O device and written in memory, in consecutive locations. Each time through the loop, the following steps are executed:

1. The DMA sends the disk address of the current byte to the disk, followed by a "read" signal. It then waits for the disk to read the byte. The disk may take a relatively long time (measured in milliseconds) until the byte moves under the disk's read/write head, so it can be read. When the disk drive has read the byte, it sends a status of "ready" to DMA, ending the wait. In practice, the disk drive reads an entire block (a few hundred to a few thousand bytes) and keeps it in a buffer. In the following iteration, the next byte is simply read from the buffer.

2. DMA checks the status. If the status is "ready", the DMA stops the processor temporarily by sending a "hold" signal on line 5 (Figure 3.5). It then proceeds to Step 3. However, if the status is "eof" (the end of the file has just been sensed), then the entire file has been transferred, the DMA clears its start/stop bit and interrupts the processor by sending a signal on line 6.

3. DMA starts the memory write operation by sending the current buffer address to the address bus. Note that the DMA device acts as a memory requestor.

4. The DMA sends a "write" signal to memory and another signal to the I/O device, to place the byte on the data bus. The DMA enters another wait period (much shorter than the previous one) waiting for memory to complete the write.

5. Memory now has all the information that it needs. It performs its internal operations and, when done, sends a feedback signal that is detected by DMA, ending the wait.

6. DMA drops the "hold" signal. It increments the current buffer address and the current disk address. If the memory buffer has not overflowed, DMA goes to Step 1 and repeats the loop. If the buffer is full, an overflow situation has been reached. DMA clears its start/stop bit and interrupts the processor by sending a signal on line 6.

Note that one of the bits in one of the registers (interrupt on end) can be set to instruct the DMA not to interrupt. In such a case the processor is not told when the DMA has finished. To find out the status of the DMA process, the processor should poll the start/stop bit of the DMA often. Also, the processor should check the status bit of the DMA from time to time to discover any unusual status.

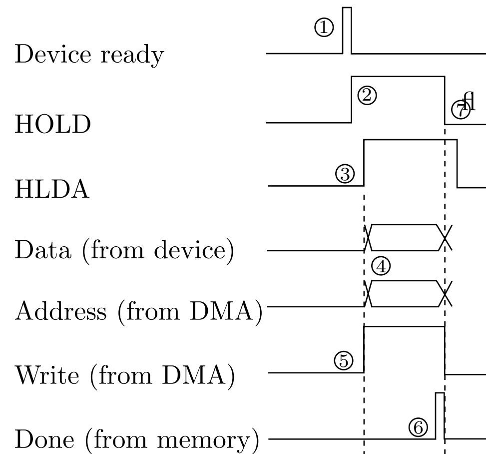



**Figure 3.6:** Writing one byte in memory

## 3.5 I/O Channels

An I/O channel is a generalization of the DMA device. The channel can do everything that a DMA device can do and more. The channel works like a DMA, transferring bytes one by one between the I/O device and memory, sending and receiving the same signals, and placing the processor on hold for each byte transferred.

The main difference between the two is that a DMA device has to be programmed for each I/O transfer, whereas an I/O channel can fetch its commands from memory. The operating system) prepares channel commands (each one very similar to a DMA command) in response to service requests from user programs, and stores the commands in memory. The channel reads the next command from memory, stores the individual fields in its registers, and executes the command by performing an entire I/O transfer. It then reads the next command from memory, executes it, and continues until no more commands are left. While the channel executes a command, the operating system can prepare more commands and store them in memory.

The main advantage of this type of operation is that no interrupts are necessary. When there are no more commands in memory, the channel turns itself off (it may optionally interrupt the CPU to let the operating system know that it is off). When the operating system prepares a new channel command in memory, it has to check channel status to make sure the channel is still on. If the channel is off, the operating system should start it by means of a special I/O instruction. This instruction (Start Channel) has one operand, the address of the current channel command in memory.

I/O channels are especially suitable for multiuser computers, where several user programs are stored in memory, waiting for execution. The processor can execute only one program at a time. When the current program needs I/O, it places a request to the operating system by interrupting itself. The proper channel command is prepared by the operating system and is stored in memory, and the processor is switched by the operating system to another user program. The current program is placed off time slices. As a result, the channel commands pending in memory may come from different user programs, and the operating system has to (1) prepare the channel commands, (2) start the channel (or the channels), and (3) switch the processor between user programs. The operating system of a multiuser computer is therefore much more complex that that of a single-user computer.

Channel commands have the format shown in Figure 3.7.

The "last command" field is normally 0 and is set to 1 in the last channel command stored in memory. Each time the operating system stores a command in memory, it sets this field to 1 and resets this field in the preceding command.

| last command | done | interrupt on end | goto | start address | block size | direction In or Out |
|---|---|---|---|---|---|---|

**Figure 3.7:** A typical channel command

The "done" field is usually 0 and is set to 1 by the channel after it has finished executing the command. At that point, the channel goes to memory, sets the "done" field of the command just executed, and fetches the next one. This way, the operating system can test the channel commands in memory at any time, find out which ones have been executed, and restart the user programs that have originally requested those I/O transfers.

The "interrupt on end" field tells the channel whether it is supposed to interrupt the processor at the end of the current channel command.

The 'goto' field is used to indicate a jump in the channel program. If there is no more room in the current buffer for more channel commands, the last command should have just the "goto" field set, and an address in the "start address" field. Such a command does not indicate any I/O transfer, just a jump in the channel program to an area where more commands can be found.

The remaining fields have the same meaning as in a DMA command.

**Types of channels**.

There are two main types of I/O channels, a *selector* and a *multiplexor*. A selector channel is connected to one I/O device and works as explained earlier. A multiplexor channel is connected to several devices and works by looping over them. It may transfer a byte between device $A$ and memory, then another byte between device $B$ and memory, etc.

A selector channel is suitable for a high-speed I/O device such as a disk drive. The high speed of the device may require the entire attention of the channel. A multiplexor channel, on the other hand, is connected to several slow devices, where it should be fast enough to match the total speed of all the devices.

## 3.6 I/O Codes

One of the main features of computers, a feature that makes computers extremely useful, is their ability to deal with nonnumeric quantities. Compiling a program is an example of such a task. The compiler reads a source program which is made up of strings of text, analyzes each string and translates it. Another example is word processing. A word processor deals mostly with text and performs relatively few calculations.

Since the computer can only handle binary quantities, any nonnumeric items have first to be coded into binary numbers. This is why I/O codes are so important. Any symbol that we want the computer to input, process, and output, should have a binary code.

There are many different codes. Some are suitable for general use, others have been developed for special applications. Some are very common, while others are rarely used. Since standards are useful in any field, there have been several attempts to standardize I/O codes. Today, most computers use the ASCII code (Section 3.7), more and more new computers use the new Unicode (Section 3.7), and some old IBM computers still use the EBCDIC code. Older, obsolete second and third generation computers used other codes.

An interesting question is: How long should the code of a character be (how many bits per character)? The answer depends, of course, on the number of characters to be coded (the size of the character set). In the past, computer printers were limited to just digits, upper case letters, and a few punctuation marks. Since there are 10 digits and 26 letters (blank space is considered a punctuation mark), a character set of size 64 was considered sufficient. Such a set can include 28 punctuation marks, in addition to the letters and digits, and the code size is therefore six bits per character.

In the last three decades, however, printers have become much more sophisticated. In addition to the traditional character set, common laser and inkjet printers can print lower case letters, any symbols, and artwork. Also, advances in communications have created a need for a special group of characters, called *control characters*. As a result, modern computers can handle a larger set of characters. Increasing the code size to seven bits/character doubles the size of the character set, from 64 to 128. This is a good size, providing codes for the 10 decimal digits, the upper- and lowercase letters, and up to 66 more characters

including punctuation marks, special graphics symbols, and control characters.

A character set of size 128 is usually considered adequate. Section 3.7 shows that there are larger character sets. The next two logical steps in designing character sets allocate 8 and 16 bits/character-code, allowing for up to 256 and 64K characters, respectively.

### 3.7 ASCII and Other Codes

The discussion above suggests that 7 bits/character is a good size for a character code. This, however, ignores one important aspect of character codes, namely reliability.

Character codes are stored in memory, moved inside the computer, sent along computer buses to I/O devices, and even transmitted over long distances between computers. As a result, errors can creep in, so a well-designed character set should pay attention to code reliability. A very simple way of increasing the reliability of a code is to add one more bit, called a *parity bit*, to each code.

The parity bit is chosen such that the total number of 1's in the code is always even or always odd. The code 1101001, that has four 1's to begin with, is assigned a parity bit of 0, while the code 1101011, with five 1's, gets a parity bit of 1, to complete the number of ones to an even number.

If a single bit gets corrupted, because of electrical or some other interference, a check of the parity bit will reveal it. When a character is received by the computer as input, parity is checked. Often, when a character is loaded from memory, the memory hardware also checks parity. Obviously, a single parity bit cannot detect every possible error. The case where two bits get bad is the simplest example of an error that cannot be detected by parity.

The result of adding a parity bit is that the ideal code size is now 8 (or rather 7+1) bits/character. This is one reason why most computers have a word size that's a multiple of 8.

The term ASCII (pronounced ASS–KEY) stands for *American Standard Code for Information Interchange*. This code was developed in the early 1960s by the American National Standards Institute (ANSI Standard No. X3.4) and is based on older telegraph codes. The ASCII code is shown in Table 3.8, and the control characters are listed and explained in Table 3.9. Notice that the codes in Table 3.8 are shown in octal and in hexadecimal, but not in decimal. Octal numbers are set in italics and are preceded by a quote, while hexadecimal numbers are set in typewriter type and are preceded by a double-quote. Thus, the code of 'A' is '$101_8$ or "$\texttt{49}_{16}$ but it takes some work to convert it to decimal.

▸ **Exercise 3.3:** Why is it unimportant to know the decimal value of a character code?

The following should be noted about the ASCII codes:

1. The first 32 codes are control characters. These are commands used in input/output and communications, and have no corresponding graphics, i.e., they cannot be printed out. Note that codes "$\texttt{20}_{16}$ and "$\texttt{7F}_{16}$ are also control characters.

2. The particular codes are arbitrary. The code of `A` is "$\texttt{41}_{16}$, but there was no special reason for assigning that particular value, and almost any other value would have served as well. About the only rule for assigning codes is that the code of `B` should follow, numerically, the code of `A`. Thus. `B` has the code "$\texttt{42}_{16}$, `C` has "$\texttt{43}_{16}$, etc. The same is true for the lowercase letters and for the 10 digits.

There is also a simple relationship between the codes of the uppercase and lowercase letters. The code of `a` is obtained from the code of `A` by setting the most significant (7th) bit to 1.

3. The parity bit in Table 3.8 is always 0. The ASCII code does not specify the value of the parity bit, and any value can be used. Different computers may therefore use the ASCII code with even parity, odd parity, or no parity.

4. The code of the control character `DEL` is all ones (except the parity which is, as usual, unspecified). This is a tradition from the old days of computing (and also from telegraphy), when paper tape was an important medium for input/output. When punching information on a paper tape, whenever the user noticed an error, they would delete the bad character by pressing the `DEL` key on the keyboard. This worked by backspacing the tape and punching a frame of all 1's on top of the bad character. When reading the tape, the reader would simply skip any frame of all 1's.

A related code is the EBCDIC (Extended BCD Information Code), shown in Table 3.10. This code was used on IBM computers and may still be used (as an optional alternative to ASCII) by some old IBM personal computers. EBCDIC is an 8-bit code, with room for up to 256 characters. However, it assigns codes

| | ′0 | ′1 | ′2 | ′3 | ′4 | ′5 | ′6 | ′7 | |
|---|---|---|---|---|---|---|---|---|---|
| ′00x | NUL | SOH | STX | ETX | EOT | ENQ | ACK | BEL | ″0x |
| ′01x | BS | HT | LF | VT | FF | CR | SO | SI | |
| ′02x | DLE | DC1 | DC2 | DC3 | DC4 | NAK | SYN | ETB | ″1x |
| ′03x | CAN | EM | SUB | ESC | FS | GS | RS | US | |
| ′04x | SP | ! | " | # | $ | % | & | ' | ″2x |
| ′05x | ( | ) | * | + | , | - | . | / | |
| ′06x | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | ″3x |
| ′07x | 8 | 9 | : | ; | < | = | > | ? | |
| ′10x | @ | A | B | C | D | E | F | G | ″4x |
| ′11x | H | I | J | K | L | M | N | O | |
| ′12x | P | Q | R | S | T | U | V | W | ″5x |
| ′13x | X | Y | Z | [ | \ | ] | ^ | _ | |
| ′14x | ` | a | b | c | d | e | f | g | ″6x |
| ′15x | h | i | j | k | l | m | n | o | |
| ′16x | p | q | r | s | t | u | v | w | ″7x |
| ′17x | x | y | z | { | \| | } | ~ | DEL | |
| | ″8 | ″9 | ″A | ″B | ″C | ″D | ″E | ″F | |

**Table 3.8:** The ASCII code

to only 107 characters, and there are quite a few unassigned codes. The term BCD (binary coded decimal) refers to the binary codes of the 10 decimal digits.

EBCDIC was developed by IBM, in the late 1950s, for its 360 computers. However, to increase compatibility, the 360 later received hardware that enabled it to also use the ASCII code. Because of the influence of IBM, some of the computers designed in the 1960s and 70s also use the EBCDIC code. Today, however, the ASCII code is a de facto standard (with unicode catching on).

A quick look at the EBCDIC control characters shows that they were developed to support punched card equipment and simple line printers. Missing are all the ASCII control characters used for telecommunications and for driving high-speed disk drives.

Another important design flaw is the gaps in the middle of the letters (between `i` and `j`, `r` and `s`) and the small number of punctuation marks.

**Unicode**

A new international standard code, the Unicode, has been proposed, and is being developed by an international Unicode organization (`www.unicode.org`). Unicode uses 16-bit codes for its characters, so it provides for $2^{16} = 64\text{K} = 65{,}536$ codes. (Doubling the size of a code much more than doubles the number of possible codes.) Unicode includes all the ASCII codes plus codes for characters in foreign languages (including complete sets of Korean, Japanese and Chinese characters) and many mathematical and other symbols. Currently about 39,000 out of the 65,536 possible codes have been assigned, so there is room for adding more symbols in the future.

The Microsoft Windows NT operating system has adopted Unicode, as have also AT&T Plan 9 and Lucent Inferno. See Appendix D for more information.

### 3.7.1 The CDC Display Code

One of the older codes is the 6-bit code used on the CDC computers (Table 3.11). Because of the code size, these computers had a word size that was always a multiple of 6. This code is called the CDC display code,

**NUL** (Null): No character, Used for filling in space in an I/O device when there are no characters.

**SOH** (Start of heading): Indicates the start of a heading on an I/O device. The heading may include information pertaining to the entire record that follows it.

**STX** (Start of text): Indicates the start of the text block in serial I/O.

**ETX** (End of text): Indicates the end of a block in serial I/O. Matches a STX.

**EOT** (End of transmission): Indicates the end of the entire transmission in serial I/O.

**ENQ** (Enquiry): An enquiry signal typically sent from a computer to an I/O device before the start of an I/O transfer, to verify that the device is there and is ready to accept or to send data.

**ACK** (Acknowledge): An affirmative response to an ENQ.

**BEL** (Bell): Causes the I/O device to ring a bell or to sound a buzzer or an alarm in order to call the operator's attention.

**BS** (Backspace): A command to the I/O device to backspace one character. Not every I/O device can respond to BS. A keyboard is a simple example of an input device that cannot go back to the previous character. Once a new key is pressed, the keyboard loses the previous one.

**HT** (Horizontal tab): Sent to an output device to indicate a horizontal movement to the next tab stop.

**LF** (Line feed): An important control code. Indicates to the output device to move vertically, to the beginning of the next line.

**VT** (Vertical tab): Commands an output device to move vertically to the next vertical tab stop.

**FF** (Form feed): Commands the output device to move the output medium vertically to the start of the next page. some output devices, such as a tape or a plotter, do not have any pages and for them the FF character is meaningless.

**CR** (Carriage return): Commands an output device to move horizontally, to the start of the line.

**SO** (Shift out): Indicates that the character codes that follow (until an SI is sensed), are not in the standard character set.

**SI** (Shift in): Terminates a non-standard string of text.

**DLE** (Data link escape): Changes the meaning of the character immediately following it.

**DC1–DC4** (Device controls): Special characters for sending commands to I/O devices. Their meaning is not predefined.

**NAK** (Negative acknowledge): A negative response to an enquiry.

**SYN** (Synchronous idle): Sent by a synchronous serial transmitter when there is no data to send.

**ETB** (End transmission block): Indicates the end of a block of data in serial transmission. Is used to divide the data into blocks.

**CAN** (Cancel): Tells the receiving device to cancel (disregard) the previously received block because of a transmission error.

**EM** (End of medium): Sent by an I/O device when it has sensed the end of its medium. The medium can be a tape, paper, card, or anything else used to record and store information.

**SUB** (Substitute): This character is substituted by the receiving device, under certain conditions, for a character that has been received incorrectly (had a bad parity bit).

**ESC** (Escape): Alters the meaning of the immediately following character. This is used to extend the character set. Thus ESC followed by an 'X' may mean something special to a certain program.

**FS** (File separator):
**GS** (Group separator):
**RS** (Record separator):
**US** (Unit separator):

The 4 separators on the left have no predefined meaning in ASCII, except that FS is the most general separator (separates large groups) and US, the least general.

**SP** (Space): This is the familiar blank or space between words. It is non-printing and is therefore considered a control character rather than a punctuation mark.

**DEL** (Delete): This is sent immediately after a bad character has been sent. DEL Indicates deleting the preceding character (see note 4).

**Table 3.9:** The ASCII control characters

and was last used on the Cyber mainframes (that became obsolete in the 1980s) which had a word size of 60 bits.

### 3.7.2 The Baudot Code

Another old code worth mentioning is the Baudot code (Table 3.12). this is a 5-bit code developed by Emile Baudot around 1880 for telegraph communication. It became popular and, by 1950, was designated the International Telegraph Code No. 1. It was used by many first- and second generation computers. The code uses 5 bits per character, but encodes more than 32 characters. Each 5-bit code can be the code of two characters, a letter and a figure. The "letter shift" (LS) and "figure shift" (FS) codes are used to shift between letters and figures.

Using this technique, the Baudot code can represent $32 \times 2 - 2 = 62$ characters (each code can have two meanings except the `LS` and `FS` codes). The actual number of characters is, however, less than that since five of the codes have one meaning each, and some codes are not assigned.

The code does not employ any parity bits and is therefore unreliable. A bad bit can transform a character into another. In particular, a corrupted bit in a shift character causes a wrong interpretation of

| Bits 3210 | Bit positions 7654: 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | null | | | | sp | & | - | | | | | | | | | 0 |
| 1 | | | | | | | / | | a | j | | | A | J | | 1 |
| 2 | | | | | | | | | b | k | s | | B | K | S | 2 |
| 3 | | | | | | | | | c | l | t | | C | L | T | 3 |
| 4 | pf | res | byp | pn | | | | | d | m | u | | D | M | U | 4 |
| 5 | ht | nl | lf | rs | | | | | e | n | v | | E | N | V | 5 |
| 6 | lc | bs | eob | uc | | | | | f | o | w | | F | O | W | 6 |
| 7 | del | il | pre | eot | | | | | g | p | x | | G | P | X | 7 |
| 8 | | | | | | | | | h | q | y | | H | Q | Y | 8 |
| 9 | | | | | | | | | i | r | z | | I | R | Z | 9 |
| A | | | sm | | ¢ | ! | | : | | | | | | | | |
| B | | | | | . | $ | , | # | | | | | | | | |
| C | | | | | < | * | % | @ | | | | | | | | |
| D | | | | | ( | ) | - | ' | | | | | | | | |
| E | | | | | + | ; | > | = | | | | | | | | |
| F | | | | | $\vert$ | ¬ | ? | " | | | | | | | | |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| **null** | Null/Idle | **sm** | Set Mode | **sp** | Space | | |
| **pf** | Punch Off | **res** | Restore | **byp** | Bypass | **pn** | Punch On |
| **ht** | Horizontal Tab | **nl** | New Line | **lf** | Line Feed | **rs** | Reader Stop |
| **lc** | Lower Case | **bs** | Backspace | **eob** | End of Block | **uc** | Upper Case |
| **del** | Delete | **il** | Idle | **pre** | Prefix | **eot** | End of Transmission |

**Table 3.10:** The EBCDIC code and its control characters

| Bits 543 | Bit positions 210: 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 0 | | A | B | C | D | E | F | G |
| 1 | H | I | J | K | L | M | N | O |
| 2 | P | Q | R | S | T | U | V | W |
| 3 | X | Y | Z | 0 | 1 | 2 | 3 | 4 |
| 4 | 5 | 6 | 7 | 8 | 9 | + | - | * |
| 5 | / | ( | ) | $ | = | sp | , | . |
| 6 | ≡ | [ | ] | : | ≠ | _ | ∨ | ∧ |
| 7 | ↑ | ↓ | < | > | ≤ | ≥ | ¬ | ; |

**Table 3.11:** The CDC display code

all the characters following, up to the next shift character.

The term "baud" used for the speed (in bits per second) of serial communications is derived from Baudot's name.

## 3.8 Information Theory and Algebraic Coding

Information Theory is the mathematical field dealing with the transfer of information from one location to another, or with the storage of information for later retrieval and use. Both tasks can be regarded as passing information or data through a channel from a source to a receiver. Consequently, information theory deals with the following three aspects:

Reliability—detection and removal of errors caused by noise in the channel (channel coding or error

| Letters | Code | Figures | Letters | Code | Figures |
|---|---|---|---|---|---|
| A | 10000 | 1 | Q | 10111 | / |
| B | 00110 | 8 | R | 00111 | - |
| C | 10110 | 9 | S | 00101 | SP |
| D | 11110 | 0 | T | 10101 | na |
| E | 01000 | 2 | U | 10100 | 4 |
| F | 01110 | na | V | 11101 | ' |
| G | 01010 | 7 | W | 01101 | ? |
| H | 11010 | + | X | 01001 | , |
| I | 01100 | na | Y | 00100 | 3 |
| J | 10010 | 6 | Z | 11001 | : |
| K | 10011 | ( | LS | 00001 | LS |
| L | 11011 | = | FS | 00010 | FS |
| M | 01011 | ) | CR | 11000 | CR |
| N | 01111 | na | LF | 10001 | LF |
| O | 11100 | 5 | ER | 00011 | ER |
| P | 11111 | % | na | 00000 | na |

**LS** = Letter Shift ; **FS** = Figure Shift **CR** = Carriage Return ; **LF** = Line Feed;
**ER** = Error ; **na** = Not Assigned **SP** = Space ;

**Table 3.12:** The Baudot Code

control coding).

Efficiency—efficient encoding of the information (source coding or data compression).

Security—protection against eavesdropping, intrusion, or tampering (cryptography).

The main principles and algorithms of these aspects are dealt with in the sections that follow.

Figure 3.13 shows the stages that a piece of computer data may go through when it is created, transmitted, received, and used at the receiving end.

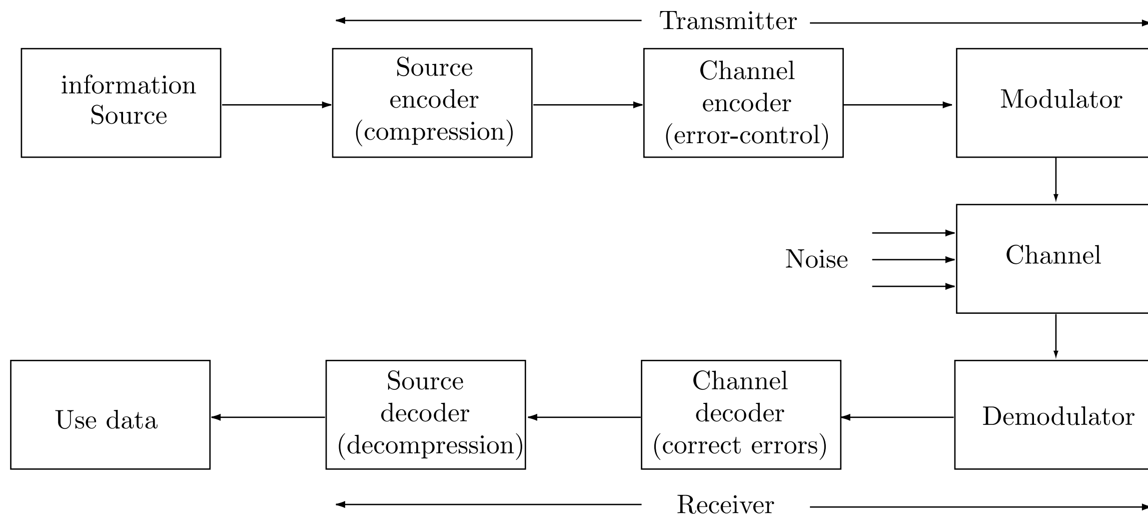



**Figure 3.13:** A Communication System

The information source provides the original data to be transmitted. If this is in analog form, it has to be digitized before it proceeds to the next step. The source encoder translates the data to an efficient form by compressing it. The channel encoder adds an error-control (i.e., detection or correction) code to the data, to make it more robust before it is sent on the noisy channel. The modulator (Section translates the

digital data to a form that can be sent on the channel (usually an electromagnetic wave). The channel itself may be a wire, a microwave beam, a satellite link, or any other type of hardware that can transmit signals. After demodulation, channel decoding (error check and correction) and source decoding (decompression), the received data finally arrives at the user on the receiving end. The source encoder, channel encoder, and modulator are sometimes called the transmitter.

We denote by $p$ the probability that a 1 bit will change to a 0 during transmission. The probability that a 1 will remain uncorrupted is, of course, $1-p$. If the probability that a 0 bit will be degraded during transmission and recognized as 1 is the same $p$, the channel is called *binary symmetric*. In such a channel, errors occur randomly. In cases where errors occur in large groups (bursts), the channel is a *burst-error channel*.

## 3.9 Error-Detecting and Error-Correcting Codes

Every time information is transmitted, on any channel, it may get corrupted by noise. In fact, even when information is stored in a storage device, it may become bad, because no piece of hardware is absolutely reliable. This also applies to non-computer information. Speech sent on the air gets corrupted by noise, wind, variations in temperature, etc. Speech, in fact, is a good starting point for understanding the principles of *channel coding* (error-detecting and error-correcting codes). Imagine a noisy cocktail party where everybody talks simultaneously, on top of blaring music. We know that even in such a situation, it is possible to carry on a conversation, except that more attention than usual is needed.

What makes our language so robust, so immune to errors? There are two properties, *redundancy* and *context*.

- Our language is redundant because only a very small fraction of all possible words are valid. A huge number of words can be constructed with the 26 letters of the Latin alphabet. Just the number of 7-letter words, e.g., is $26^7 \approx 8.031$ billion. Yet only about 50,000 words are commonly used, and even the Oxford English Dictionary lists "only" about 500,000 words. When we hear a garbled word, our brain searches through many similar words for the "closest" valid word. Computers are very good at such searches, which is why redundancy is the basis of error-detecting and error-correcting codes.

- Our brain works by associations. This is why we humans excel at using the context of a message to repair errors in the message. In receiving a sentence with a garbled word or a word that doesn't belong, such as "pass the thustard please", we first use our memory to find words that are associated with "thustard," then we use our accumulated life experience to select, among many possible candidates, the word that best fits in the present context. If we are driving on the highway, we pass the bastard in front of us; if we are at dinner, we pass the mustard (or custard). Another example is the (corrupted) written sentence "a∗l n∗tu∗al l∗∗gua∗es a∗e red∗∗∗ant", which we can easily complete. Computers don't have much life experience and are notoriously bad at such tasks, which is why context is not used in computer codes. In extreme cases, where much of the sentence is bad, even we may not be able to correct it, and we have to ask for a retransmission "say it again, Sam."

The idea of using redundancy to add reliability to information is due to Claude Shannon, the founder of information theory. It is not an obvious idea, since we are conditioned against it. Most of the time, we try to *eliminate* redundancy in computer information, in order to save space. In fact, all the data-compression methods do just that.

We discuss two approaches to reliable codes. The first one is to *duplicate* the code, an approach that leads to the idea of *voting codes*; the second approach uses *check bits* and is based on the concept of *Hamming distance*.

### 3.9.1 Voting Codes

Perhaps the first idea that comes to mind, when thinking about redundancy, is to duplicate every bit of the message. Thus, if the data 1101 has to be transmitted, the bits 11|11|00|11 are sent instead. A little thinking shows that this results in error detection, but not in error correction. If the receiver receives a pair of different bits, it cannot tell which bit is correct. This is an example of a *receiver failure*. A little thinking may convince the reader that sending each bit in triplicate can lead to error-correction (although not absolute). We can transmit 111|111|000|111 and tell the receiver to compare the three bits of each triplet. If all three are identical, the receiver assumes that they are correct. Moreover, if only two are identical and

the third one is different, the receiver assumes that the two identical bits are correct. This is the principle of *voting codes*. The receiver (decoder) makes the correct decision when either (1) two of the three bits are identical and the third one is different *and* the two identical bits are correct or (2) all three bits are identical and correct. Similarly, the decoder makes the wrong decision when (1) two of the three bits are identical and the third one is different *and* the two identical bits are bad or (2) all three bits are identical and are bad. Before deciding to use such a code, it is important to compute the probabilities of these cases and try to estimate their values in practical applications.

If we duplicate each bit an odd number of times, the receiver may sometimes make the wrong decision, but it can *always* make a decision. If each bit is duplicated an even number of times, the receiver will fail (i.e., will not be able to make any decision) in cases where half the copies are 0s and the other half are 1s.

> What I tell you three times is true.
>
> —Lewis Carroll, *The Hunting of the Snark*

In practice, errors may occur in bursts, so it is preferable to seperate the copies of each bit. Instead of transmitting 111|111|000|111, it is better to transmit 1101 . . . 1101 . . . 1101 . . . . The receiver has first to identify the three bits of each triplet, then compare them.

A voting code where each bit is duplicated $n$ times is called an $(n, 1)$ voting code.

It is intuitively clear that the reliability of the voting code depends on $n$ and on the quality of the transmission channel. The latter can be estimated by measuring the probability $p$ that any individual bit will be corrupted. This can be done by transmitting large quantities of known data through the channel and counting the number of bad bits received. Such an experiment should be carried out with many millions of bits and over long time periods, to account for differences in channel reliability between day and night, summer and winter, intense heat, high humidity, lightnings, and so on. Actual values of $p$ for typical channels used today are in the range $10^{-7}$ to $10^{-9}$, meaning that on average one bit in ten million to one bit in a billion bits transmitted gets corrupted.

Once $p$ is known, we compute the probability that $j$ bits out of the $n$ bits transmitted will go bad. Given any $j$ bits of the $n$ bits, the probability of those $j$ bits going bad is $p^j(1-p)^{n-j}$, because we have to take into account the probability that the remaining $n-j$ bits did not go bad. However, it is possible to select $j$ bits out of $n$ bits in several ways, and this also has to be taken into account in computing the probability. The number of ways to select $j$ objects out of any $n$ objects without selecting the same object more than once is denoted by ${}^nC_j$ and is

$$ {}^nC_j = \binom{n}{j} = \frac{n!}{j!(n-j)!}. $$

We therefore conclude that the probability of any group of $j$ bits out of the $n$ bits getting bad is $P_j = {}^nC_j p^j(1-p)^{n-j}$. Based on this, we can analyze the behavior of voting codes by computing three basic probabilities. (1) The probability $p_c$ that the receiver will make the correct decision (i.e., will find and correct an error), (2) the probability $p_e$ that the receiver will make the wrong decision (i.e., will "detect" and correct a nonexistent error), and (3) the probability $p_f$ that the receiver will fail (i.e., will not be able to make any decision). We start with the simple case $n = 3$ where each bit is transmitted three times.

When $n = 3$, the receiver will make a correct decision when either all three bits remain good or when one bit got bad. Thus, $p_c$ is the sum $P_0 + P_1$ which equals $\left[{}^3C_0p^0(1-p)^{3-0}\right] + \left[{}^3C_1p^1(1-p)^{3-1}\right] = (1-p)^3 + 3p(1-p)^2$. Similarly, the receiver will make the wrong decision when either two of the three bits get bad or when all three have been corrupted. The probability $p_e$ is therefore the sum $P_2 + P_3$ which equals $\left[{}^3C_2p^2(1-p)^{3-2}\right] + \left[{}^3C_3p^3(1-p)^{3-3}\right] = 3p^2(1-p) + 3p^3$. Since $n$ is odd, the receiver will always be able to make a decision, implying that $p_f = 0$. Any code where $p_f = 0$ is a *complete decoding* code. Notice that the sum $p_c + p_e + p_f$ is 1.

As a simple example, we compute $p_c$ and $p_e$ for the $(3, 1)$ voting code for $p = 0.001$ and for $p = 0.001$. The former yields $(1-0.01)^3 + 3\cdot 0.01(1-0.01)^2 = 0.999702$ and $p_c = 3\cdot 0.01^2(1-0.01) + 3\cdot 0.01^3 = 0.000298$. The latter yields $(1-0.001)^3 + 3\cdot 0.001(1-0.001)^2 = 0.999997$ and $p_c = 3\cdot 0.001^2(1-0.001) + 3\cdot 0.001^3 = 0.000003$. This shows that the simple $(3, 1)$ voting code features excellent behavior even for large bit failure rates.

The $(3, 1)$ voting code can correct up to one bad bit. Similarly, the $(5, 1)$ voting code can correct up to two bad bits. In general, the $(n, 1)$ voting code can correct $\lfloor(n-1)/2\rfloor$ bits (in future, the "$\lfloor$" and "$\rfloor$" will be omitted). Such a code is simple to generate and to decode, and it provides high reliability, but at a price; an $(n, 1)$ code is very long and can be used only in applications where the length of the code (and consequently, the transmission time) is unimportant. It is easy to compute the probabilities of correct and incorrect decisions made by the general $(n, 1)$ voting code. The code will make a correct decision if the number of bad bits is 0, 1, 2,. . . ,$(n-1)/2$. The probability $p_c$ is therefore

$$p_c = P_0 + P_1 + P_2 + \cdots + P_{(n-1)/2} = \sum_{j=0}^{(n-1)/2} {}^nC_j p_j (1-p)^{n-j}.$$

Similarly, the $(n, 1)$ voting code makes the wrong decision when the number of bad bits is greater than $(n+1)/2$. Thus, the value of $p_e$ is

$$p_e = P_{(n+1)/2} + P_{(n+3)/2} + P_{(n+5)/2} + \cdots + P_n = \sum_{(n+1)/2}^{n} {}^nC_j p_j (1-p)^{n-j}.$$

Notice that $p_f = 0$ for an odd $n$ because the only case where the receiver cannot make a decision is when half the $n$ bits are bad, and this requires an even $n$. Table lists the values of $p_e$ for five odd values of $n$. Values of the redundancy (or the *code rate*) $R$ are also included. This measure is the ratio $k/n$, i.e., the number of data bits in the $(n, 1)$ code (i.e., 1) divided by the total number $n$ of bits.

| $n$ | $p_e$ | $R$ |
|---|---|---|
| 3 | $3.0 \times 10^{-4}$ | 0.33 |
| 5 | $9.9 \times 10^{-6}$ | 0.20 |
| 7 | $3.4 \times 10^{-7}$ | 0.14 |
| 9 | $1.2 \times 10^{-8}$ | 0.11 |
| 11 | $4.4 \times 10^{-10}$ | 0.09 |

**Table 3.14:** Probabilities of Wrong Decisions for Voting Codes.

▸ **Exercise 3.4:** Compute $p_e$ for the $(7, 1)$ voting code assuming a bit failure rate of $p = 0.01$.

▸ **Exercise 3.5:** Calculate the probability $p_f$ of failure for the $(6, 1)$ voting code assuming that $p = 0.01$.

### 3.9.2 Check Bits

In practice, error detection and correction is done by means of *check bits* that are added to the original *information bits* of each word of the message. In general, $k$ check bits are appended to the original $m$ information bits, to produce a *codeword* of $n = m+k$ bits. Such a code is referred to as an $(n, m)$ code. The codeword is then transmitted to the receiver. Only certain combinations of the information bits and check bits are valid, in analogy with a natural language. The receiver knows what the valid codewords are. If a nonvalid codeword is received, the receiver considers it an error. Section 3.9.7 shows that by adding more check bits, the receiver can also *correct* certain errors, not just detect them. The principle of error correction is that, on receiving a bad codeword, the receiver selects the valid codeword that is the "closest" to it.

Example: A set of 128 symbols needs to be coded. This implies $m = 7$. If we select $k = 4$, we end up with 128 valid codewords, each 11 bits long. This is an $(11, 7)$ code. The valid codewords are selected from a total of $2^{11} = 2048$ possible codewords, so there remain $2048 - 128 = 1920$ nonvalid codewords. The big difference between the number of valid (128) and nonvalid (1920) codewords means that, if a codeword gets corrupted, chances are that it will change to a nonvalid one.

It may, of course, happen that a valid codeword gets changed, during transmission, to another valid codeword. Thus, our codes are not completely reliable, but can be made more and more reliable by adding more check bits and by selecting the valid codewords carefully. One of the basic theorems of information theory says that codes can be made as reliable as desired by adding check bits, as long as $n$ (the size of a codeword) does not exceed the channel's capacity.

It is important to understand the meaning of the word "error" in data transmission. When an $n$-bit codeword is sent and received, the receiver always receives $n$ bits, but some of them may be bad. A bad bit does not disappear, nor does it change into something other than a bit. A bad bit simply changes its value, either from 0 to 1, or from 1 to 0. This makes it relatively easy to correct the bit. The code should tell the receiver which bits are bad, and the receiver can then easily correct the bits by inverting them.

In practice, bits may be sent on a wire as voltages. A binary 0 may, e.g., be represented by any voltage in the range 3–25 volts. A binary 1 may similarly be represented by the voltage range of $-25$v to $-3$v. Such voltages tend to drop over long lines, and have to be amplified periodically. In the telephone network there is an amplifier (a *repeater*) every 20 miles or so. It looks at every bit received, decides if it is a 0 or a 1 by measuring the voltage, and sends it to the next repeater as a clean, fresh pulse. If the voltage has deteriorated enough in passage, the repeater may make a wrong decision when sensing it, which introduces an error into the transmission. At present, typical transmission lines have error rates of about one in a billion but, under extreme conditions—such as in a lightning storm, or when the electric power suddenly fluctuates—the error rate may suddenly increase, creating a burst of errors.

### 3.9.3 Parity Bits

A parity bit can be added to a group of $m$ information bits to complete the total number of 1 bits to an odd number. Thus the (odd) parity of the group 10110 is 0, since the original group plus the parity bit has an odd number (3) of 1's. It is also possible to use even parity, and the only difference between odd and even parity is that, in the case of even parity, a group of all zeros is valid, whereas, with odd parity, any group of bits with a parity bit added, cannot be all zeros.

Parity bits can be used to design simple, but not very efficient, error-correcting codes. To correct 1-bit errors, the message can be organized as a *rectangle* of dimensions $(r-1)\times(s-1)$. A parity bit is added to each row of $s-1$ bits, and to each column of $r-1$ bits. The total size of the message (Table 3.14) is now $s\times r$.

| | | | | |
|---|---|---|---|---|
| 0 | 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 | 1 |
| 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 | 1 |

**Table 3.15:**

| | | | | |
|---|---|---|---|---|
| 0 | 1 | 0 | 0 | **1** |
| 1 | 0 | 1 | **0** | |
| 0 | 1 | **0** | | |
| 0 | **0** | | | |
| **1** | | | | |

**Table 3.16:**

If only one bit becomes bad, a check of all $s-1+r-1$ parity bits will discover it, since only one of the $s-1$ parities and only one of the $r-1$ ones will be bad.

The overhead of a code is defined as the number of parity bits divided by the number of information bits. The overhead of the rectangular code is, therefore,

$$\frac{(s-1+r-1)}{(s-1)(r-1)}\approx\frac{s+r}{s\times r-(s+r)}.$$

A similar, slightly more efficient, code is a triangular configuration, where the information bits are arranged in a triangle, with the parity bits placed on the diagonal (Table 3.15). Each parity bit is the parity of all the bits in its row *and* column. If the top row contains $r$ information bits, the entire triangle has $r(r+1)/2$ information bits and $r$ parity bits. The overhead is thus

$$\frac{r}{r(r+1)/2}=\frac{2}{r+1}.$$

It is also possible to arrange the information bits in a number of two-dimensional planes, to obtain a three-dimensional cube, three of whose six outer surfaces consist of parity bits.

It is not obvious how to generalize these methods to more than 1-bit error correction.

| Symbol | $code_1$ | $code_2$ | $code_3$ | $code_4$ | $code_5$ | $code_6$ | $code_7$ |
|---|---|---|---|---|---|---|---|
| $A$ | 0000 | 0000 | 001 | 001001 | 01011 | 110100 | 110 |
| $B$ | 1111 | 1111 | 010 | 010010 | 10010 | 010011 | 0 |
| $C$ | 0110 | 0110 | 100 | 100100 | 01100 | 001101 | 10 |
| $D$ | 0111 | 1001 | 111 | 111111 | 10101 | 101010 | 111 |
| $k$: | 2 | 2 | 1 | 4 | 3 | 4 | |

**Table 3.17:** $(m = 2)$

### 3.9.4 Hamming Distance and Error Detecting

Richard Hamming developed the concept of distance, in the 1950s, as a general way to use check bits for error detection and correction.

To illustrate this concept, we start with a simple example involving just the four symbols $A$, $B$, $C$, and $D$. Only two information bits are required, but the codes of Table 3.16 add some check bits, for a total of 3–6 bits per symbol. The first of these codes, $code_1$, is simple. Its four codewords were selected from the 16 possible 4-bit numbers, and are not the best possible ones. When the receiver receives one of them, say, 0111, it assumes that there is no error and the symbol received is $D$. When a nonvalid codeword is received, the receiver signals an error. Since $code_1$ is not the best possible, not every error is detected. Even if we limit ourselves to single-bit errors, this code is not very good. There are 16 possible single-bit errors in our 4-bit codewords and, of those, the following 4 cannot be detected: A 0110 changed during transmission to 0111, a 0111 changed to 0110, a 1111 corrupted to 0111, and a 0111 changed to 1111. The error detection rate is thus 12 out of 16, or 75%. In comparison, $code_2$ does a much better job. It can detect every single-bit error since, when only a single bit is changed in any of the codewords, the result is not any of the other codewords. We say that the four codewords of $code_2$ are sufficiently *distant* from each other. The concept of distance of codewords is, fortunately, easy to define.

**Definitions.** 1. Two codewords are a Hamming distance $d$ apart if they differ in exactly $d$ of their $n$ bits; 2. A code has a Hamming distance of $d$ if every pair of codewords in the code is at least a Hamming distance $d$ apart.

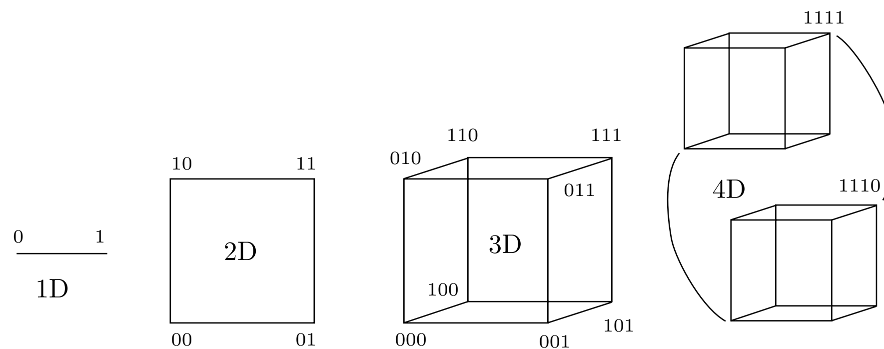



**Figure 3.18:** Cubes of various dimensions and corner numbering

These definitions have a simple geometric interpretation (for the mathematical readers). Imagine a hypercube in $n$-dimensional space. Each of its $2^n$ corners can be numbered by an $n$-bit number (Figure 3.17), such that each of the $n$ bits corresponds to one of the $n$ dimensions. In such a cube, points that are directly connected (near neighbors) have a Hamming distance of 1, points with a common neighbor have a Hamming distance of 2, etc. If a code with a Hamming distance of 2 is desired, only points that are not directly connected should be selected as valid codewords.

The reason $code_2$ can detect all single-bit errors is that it has a Hamming distance of 2. The distance between valid codewords is 2, so a one-bit error always changes a valid codeword into a nonvalid one. When

two bits go bad, a valid codeword is moved to another codeword at distance 2. If we want that other codeword to be nonvalid, the code must have at least distance 3.

In general, a code with a Hamming distance of $d+1$ can detect all $d$-bit errors. In comparison, $\text{code}_3$ has a Hamming distance of 2 and can therefore detect all 1-bit errors even though it is short ($n = 3$). Similarly, $\text{code}_4$ has a Hamming distance of 4, which is more than enough to detect all 2-bit errors. It is obvious now that we can increase the reliability of our transmissions, but this feature does not come free. As always, there is a tradeoff, or a price to pay, in the form of the overhead. Our codes are much longer than $m$ bits per symbol because of the added check bits. A measure of the price is $n/m = \frac{m+k}{m} = 1 + k/m$, where the quantity $k/m$ is called the *overhead* of the code. In the case of $\text{code}_1$ the overhead is 2, and in the case of $\text{code}_3$ it is 3/2.

**Example:** A code with a single check bit, that is a parity bit (even or odd). Any single-bit error can easily be detected since it creates a nonvalid codeword. Such a code therefore has a Hamming distance of 2. Notice that $\text{code}_3$ uses a single, odd, parity bit.

**Example:** A 2-bit error-detecting code for the same 4 symbols. It must have a Hamming distance of at least 3, and one way of generating it is to duplicate $\text{code}_3$ (this creates $\text{code}_4$ with a distance of 4).

### 3.9.5 Error-Correcting Codes

The principle of error-correcting codes is to separate the codes even farther by increasing the code's redundancy (i.e., adding more check bits). When an invalid codeword is received, the receiver corrects the error by selecting the valid codeword that is closest to the one received. An example is $\text{code}_5$, that has a Hamming distance of 3. When one bit is changed in any of its four codewords, that codeword is one bit distant from the original one, but is still two bits distant from any of the other codewords. Thus, if there is only one error, the receiver can always correct it.

In general, when $d$ bits go bad in a codeword $C_1$, it turns into an invalid codeword $C_2$ at a distance $d$ from $C_1$. If the distance between $C_2$ and the other valid codewords is at least $d+1$, then $C_2$ is closer to $C_1$ than it is to any other valid codeword. This is why a code with a Hamming distance of $d+(d+1) = 2d+1$ is able to correct all $d$-bit errors.

How are the codewords selected? The problem is to select a good set of $2^m$ codewords out of the $2^n$ possible ones. The simplest approach is to use brute force. It is easy to write a computer program that will examine all the possible sets of $2^m$ codewords, and will stop at the first one that has the right distance. The problems with this approach are (1) the time and storage required at the receiving end to verify and correct the codes received, and (2) the amount of time it takes to examine all the possibilities.

Problem 1. The receiver must have a list of all the $2^n$ possible codewords. For each codeword it must have a flag indicating if it is valid and, if not, which valid codeword is the closest to it. Every codeword received has to be located in this list in order to verify it.

Problem 2. In the case of four symbols, only four codewords need be selected. For $\text{code}_1$ and $\text{code}_2$, these four codewords had to be selected from among 16 possible numbers, which can be done in $\binom{16}{4} = 7280$ ways. It is possible to write a simple program that will systematically select sets of four codewords until it finds a set with the required distance. In the case of $\text{code}_4$, however, the four codewords had to selected from a set of 64 numbers, which can be done in $\binom{64}{4} = 635{,}376$. It is still possible to write a program that will systematically explore all the possible codeword selections. In practical cases, however, where sets of hundreds of symbols are involved, the number of possibilities of selecting codewords is too large even for the fastest computers to handle in reasonable time.

### 3.9.6 Hamming Codes

Clearly, a clever algorithm is needed, to select the best codewords, and to verify them on the fly, as they are being received. The transmitter should use the algorithm to generate the codewords right before they are sent, and the receiver should use it to check them when they are received. The second approach discussed here is due to Richard Hamming [Hamming 80]. In Hamming's codes, the $n$ bits of a codeword are indexed from 1 (least significant) to $n$ (most significant). The check bits are those with indexes that are powers of 2. Thus, bits $b_1, b_2, b_4, b_8, \ldots$ are check bits, and $b_3, b_5, b_6, b_7, b_9, \ldots$ are information bits. The index of each information bit can be written as the sum of the indexes of certain check bits. Thus, $b_7$ can be written as

$b_{1+2+4}$ and is, therefore, used in determining check bits $b_1$, $b_2$, and $b_4$. *The check bits are simply parity bits.* The value of $b_2$, for example, is the parity (odd or even) of $b_3$, $b_6$, $b_7$, $b_{10}, \ldots$ .

**Example:** A 1-bit error-correcting code for the set of symbols $A$, $B$, $C$, $D$. It must have a Hamming distance of 3. Two information bits are needed to code the four symbols, so they must be $b_3$ and $b_5$. The parity bits are therefore $b_1$, $b_2$, and $b_4$. Since $3 = 1+2$ and $5 = 1+4$, the 3 parity bits are defined as: $b_1$ is the parity of bits $b_3$, $b_5$; $b_2$ is the parity of $b_3$; and $b_4$ is the parity of $b_5$. This is how code$_5$ was constructed.

**Example:** A 1-bit error-correcting code for a set of 256 symbols. It must have a Hamming distance of 3. Eight information bits are required to code the 256 symbols, so they must be $b_3$, $b_5$, $b_6$, $b_7$, $b_9$, $b_{10}$, $b_{11}$, and $b_{12}$. The parity bits are, therefore, $b_1$, $b_2$, $b_4$, and $b_8$. The total size of the code is 12 bits. The following relations define the four parity bits:

$3 = 1+2$, $5 = 1+4$, $6 = 2+4$, $7 = 1+2+4$, $9 = 1+8$, $10 = 2+8$, $11 = 1+2+8$, and $12 = 4+8$.

This implies that $b_1$ is the parity of $b_3$, $b_5$, $b_7$, $b_9$, and $b_{11}$. The definitions of the other parity bits are left as an exercise.

▸ **Exercise 3.6:** Construct a 1-bit error-correcting Hamming code for 16-bit codes ($m = 16$).

What is the size of a general Hamming code? The case of a 1-bit error-correcting code is easy to analyze. Given a set of $2^m$ symbols, $2^m$ valid codewords are needed, each $n$ bits long. The $2^m$ valid codewords should, therefore, be selected from a total of $2^n$ numbers. Each codeword consists of $m$ information bits and $k$ check bits, where the value of $m$ is given, and we want to know the minimum value of $k$.

Since we want any single-bit error in a codeword to be corrected, such an error should not take us too far from the original codeword. A single-bit error takes us to a codeword at distance 1 from the original one. As a result, all codewords at distance 1 from the original codeword should be nonvalid. Each of the original $2^m$ codewords should thus have the $n$ codewords at distance 1 from it, declared nonvalid. This means that the total number of codewords (valid plus nonvalid) is $2^m + n2^m = (1+n)2^m$. This number has to be selected from the $2^n$ available numbers, so we end up with the relation $(1+n)2^m \le 2^n$. Since $2^n = 2^{m+k}$, we get $1+n \le 2^k$ or $k \ge \log_2(1+n)$. The following table illustrates the meaning of this relation for certain values of $m$.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| $n$ | : | 4 | 7 | 12 | 21 | 38 | 71 |
| $k$ | : | 2 | 3 | 4 | 5 | 6 | 7 |
| $m = n - k$ | : | 2 | 4 | 8 | 16 | 32 | 64 |
| $k/m$ | : | 1 | .75 | .5 | .31 | .19 | .11 |

The geometric interpretation provides another way of obtaining the same result. We imagine $2^m$ spheres of radius one tightly packed in our $n$-dimensional cube. Each sphere is centered around one of the corners, and encompasses all its immediate neighbors. The *volume* of a sphere is defined as the number of corners it includes, which is $1+n$. The spheres are tightly packed but they don't overlap, so their total volume is $(1+n)2^m$, and this should not exceed the total volume of the cube, which is $2^n$.

The case of a 2-bit error-correcting code is similarly analysed. Each valid codeword should define a set including itself, the $n$ codewords at distance 1 from it, and the set of $\binom{n}{2}$ codewords at distance 2 from it, a total of $\binom{n}{0} + \binom{n}{1} + \binom{n}{2} = 1 + n + n(n-1)/2$. Those sets should be non-overlapping, which implies the relation

$$\bigl(1+n+n(n-1)/2\bigr)2^m \le 2^n \Rightarrow 1+n+n(n-1)/2 \le 2^k \Rightarrow k \ge \log_2\bigl(1+n+n(n-1)/2\bigr).$$

In the geometric interpretation we again imagine $2^m$ spheres of radius 2 each. Each sphere centered around a corner and containing the corner, its $n$ immediate neighbors, and its $\binom{n}{2}$ second place neighbors (corners differing from the center corner by 2 bits).

### 3.9.7 The SEC-DED Code

However, even though we can estimate the length of a 2-bit error-correcting Hamming code, we don't know how to construct it! The best that can be done today with Hamming codes is single error correction combined with double error detection. An example of such a SEC-DED code is code$_6$. It was created by simply adding a parity bit to code$_5$.

The receiver checks this code in two steps. In step 1, the single parity bit is checked. If it is bad, the receiver assumes that a 1-bit error occurred, and it uses the other parity bits, in step 2, to correct the error.

It may happen, of course, that three or even five bits are bad, but the simple SEC-DED code cannot detect such errors.

If the single parity is good, then there are either no errors, or two bits are wrong. The receiver proceeds to step 2, where it uses the other parity bits to distinguish between these two cases. Again, there could be four or six bad bits, but this code cannot handle them.

The SEC-DED code has a Hamming distance of 4. In general, a code for $c$-bit error correction and $d$-bit error detection, should have a distance of $c+d+1$.

The following discussion provides better insight into the bahavior of the Hamming code. For short codewords, the number of parity bits is large compared to the size of the codeword. When a 3-bit code, such as $xyz$, is converted to SEC-DED, it becomes the 6-bit codeword $xyb_4zb_2b_1$, which doubles its size. Doubling the size of a code seems a high price to pay in order to gain only single-error correction and double-error detection. On the other hand, when a 500-bit code is converted to SEC-DED, only the 9 parity bits $b_{256}$, $b_{128}$, $b_{64}$, $b_{32}$, $b_{16}$, $b_8$, $b_4$, $b_2$, and $b_1$, are added, resulting in a 509-bit codeword. The size of code is increased by 1.8% only. Such a small increase in size seems a good price to pay for even a small amount of increased reliability. With even larger codes, the benefits of SEC-DED can be obtained with even a smaller increase in the code size. There is clearly something wrong here, since we cannot expect to get the same benefits by paying less and less. There must be some tradeoff. Something must be lost when a long code is converted to SEC-DED.

What is lost is reliability. The SEC-DED code can detect 2-bit errors and correct 1-bit errors in the short, 6-bit codeword and also in the long, 509-bit codeword. However, there may be more errors in 509 bits than in 6 bits. The difference in the corrective power of SEC-DED is easier to understand if we divide the 500-bit code into 100 5-bit segments and convert each individually into a SEC-DED code by adding four parity bits. The result is a string of $100(5+4)=900$ bits. Now assume that 50 bits become bad. In the case of a single, 509-bit long string, the SEC-DED code may be able to detect an error but will not be able to correct any bits. In the case of 100 9-bit strings, the 50 bad bits constitute 5.6% of the string size. This implies that many 9-bit strings will not suffer any damage, many will have just 1 or 2 bad bits (cases which the SEC-DED code can handle), and only a few will have more than 2 bad bits.

The conclusion is that the SEC-DED code provides limited reliability and should be applied only to resonably short strings.

### 3.9.8 Periodic Codes

No single error-correcting code provides absolute reliability in every situation. This is why each application should ideally have a special reliable code designed for it, based on what is known about possible data corruption. One common example of errors is a *burst*. Imagine a reliable cable through which data is constantly transmitted with high reliability. Suddenly a storm approaches and lightning strikes nearby. This may cause a burst of errors that will corrupt hundreds or thousands of bits in a small region of the data. Another example is a CD. Data is recorded on a CD in a spiral path that starts on the inside and spirals outside. When the CD is cleaned carelessly, it may get scratched. If the cleaning is done in a circular motion, parallel to the spiral, even a small scratch may cover (and therefore corrupt) many consecutive bits (which is why a CD should be cleaned with radial motions, in a straight line from the inside toward the rim).

The SEC-DED, or any similar code, cannot deal with a burst of errors, so different codes must be designed for applications where such bursts may occur. One way to design such a code is to embed parity bits among the data bits, such that each parity bit is the parity of several, nonadjacent data bits. Such a code is called *periodic*.

As a simple example of a periodic code, imagine a string of bits divided into 5-bit segments. Each bit is identified by the segment number and by its number (1 through 5) within the segment. Each segment is followed by one parity bit. Thus, the data with the parity bits becomes the string

$$b_{1,1},\ b_{1,2},\ b_{1,3},\ b_{1,4},\ b_{1,5},\ p_1,\ \ ,b_{2,1},\ b_{2,2},\ b_{2,3},\ b_{2,4},\ b_{2,5},\ p_2,\ldots,b_{i,1},\ b_{i,2},\ b_{i,3},\ b_{i,4},\ b_{i,5},\ p_i,\ldots.$$

We now define each parity bit $p_i$ as the (odd) parity of the five bits $b_{i,1}$, $b_{i-1,2}$, $b_{i-2,3}$, $b_{i-3,4}$, and $b_{i-4,5}$. Thus, each parity bit protects five bits in five different segments, so any error affects five parity bits, one in its segment and four in the four segments that follow. If all the data bits of segment $j$ become corrupted, it

will be reflected in the five parity bits $p_j$, $p_{j+1}$, $p_{j+2}$, $p_{j+3}$, and $p_{j+4}$. In general, errors will be detected if every two bursts are at least five segments apart.

This simple code cannot correct any errors, but the principle of periodic codes can be extended to allow for as much error correction as necessary.

### 3.9.9 A Different Approach

This section describes error correction in a CD (see Appendix C).

It is obvious that reading a CD-ROM must be error free, but error correction is also important in an audio CD, because one bad bit can cause a big difference in the note played. Consider the two 16-bit numbers 0000000000000000 and 1000000000000000. The first represents silence and the second, a loud sound. Yet they differ by one bit only! The size of a typical dust particle is 40$\mu$m, enough to cover more than 20 laps of the track, and cause several bursts of errors (Figure C.1b). Without extensive error correction, the music would sound as one long scratch.

Any error correction method used in a CD must be very sophisticated, since the errors may come in bursts, or may be individual. The use of parity bits makes it possible to correct individual errors, but not a burst of consecutive errors. This is why *interleaving* is used, in addition to parity bits. The principle of interleaving is to rearrange the samples before recording them on the CD, and to reconstruct them after they have been read. This way, a burst of errors during the read is translated to individual errors (Figure C.1a), that can then be corrected by their parity bits.

The actual code used in CDs is called the Cross Interleaved Reed-Solomon Code (CIRC). It was developed by Irving S. Reed and Gustave Solomon at Bell labs in 1960, and is a powerful code. One version of this code can correct up to 4000 consecutive bit errors, which means that even a scratch as long as three millimeters can be tolerated on a CD. The principle of CIRC is to use a geometric pattern that is so familiar that it can be reconstructed even if large parts of it are missing. It's like being able to recognize the shape of a rectangular chunk of cheese after a mouse has nibbled away large parts of it.

Suppose that the data consists of the two numbers 3.6 and 5.9. We consider them the $y$ coordinates of two-dimensional points and we assign them $x$ coordinates of 1 and 2, respectively. We thus end up with the points $(1, 3.6)$ and $(2, 5.9)$. We consider those points the endpoints of a line and we calculate four more points on this line, with $x$ coordinates of 3, 4, 5, and 6. They are $(3, 8.2)$, $(4, 10.5)$, $(5, 12.8)$, and $(6, 15.1)$. Since the $x$ coordinates are so regular, we only need to store the $y$ coordinates of these points. We thus store (or write on the CD) the six numbers 3.6, 5.9, 8.2, 10.5, 12.8, and 15.1.

Now suppose that two errors occur among those six numbers. When the new sequence of six numbers is checked for the straight line property, the remaining four numbers can be identified as being collinear and can still be used to reconstruct the line. Once this is done, the two bad numbers can be corrected, since their $x$ coordinates are known. Even three bad numbers out of those six can be corrected since the remaining three numbers would still be enough to identify the original straight line.

It is even more reliable to start with three numbers $a$, $b$, and $c$, to convert them to the points $(1, a)$, $(2, b)$, and $(3, c)$, and to calculate the (unique) parabola that passes through these points. Four more points, with $x$ coordinates of 4, 5, 6, and 7, can then be calculated on this parabola. Once the seven points are known, they provide a strong pattern. Even if three of the seven get corrupted, the remaining four can be used to reconstruct the parabola and correct the three bad ones. However, if four of the seven get corrupted, then no four numbers will be on a parabola (and any group of three will define a different parabola). Such a code can correct three errors in a group of seven numbers, but it requires high redundancy (seven numbers instead of four).

### 3.9.10 Generating Polynomials

There are many approaches to the problem of developing codes for more than 1-bit error correction. They are, however, more complicated than Hamming's method, and require a background in group theory and Galois (finite) fields. One such approach, using the concept of a *generating polynomial*, is briefly sketched here.

We consider the case $m = 4$. Sixteen codewords are needed, that can be used to code any set of 16 symbols. We already know that three parity bits are needed for 1-bit correction, thereby bringing the total size of the code to $n = 7$. Here is an example of such a code:

| 0000000 | 0001011 | 0010110 | 0011101 | 0100111 | 0101100 | 0110001 | 0111010 |
|---|---|---|---|---|---|---|---|
| 1000101 | 1001110 | 1010011 | 1011000 | 1100010 | 1101001 | 1110100 | 1111111 |

Note that it has the following properties:

- The sum (modulo 2) of any two codewords equals another codeword. This implies that the sum of any number of codewords is a codeword. Thus, the 16 codewords above form a *group* under this operation. (Addition and subtraction modulo-2 is done by $0+0=1+1=0$, $0+1=1+0=1$, $0-1=1$. The definition of a group should be reviewed in any text on algebra.)
- Any circular shift of a codeword is another codeword. Thus, this code is *cyclic*.
- It has a Hamming distance of 3, as required for 1-bit correction.

Interesting properties! The sixteen codewords were selected from the 128 possible ones by means of a *generator polynomial*. The idea is to look at each codeword as a polynomial, where the bits are the coefficients. Here are some 7-bit codewords associated with polynomials of degree 6.

| 1 | 0 | 0 | 1 | 1 | 1 | 1 |
|---|---|---|---|---|---|---|
| $x^6$ | | | $+x^3$ | $+x^2$ | $+x$ | $+1$ |

| 0 | 1 | 1 | 0 | 0 | 1 | 0 |
|---|---|---|---|---|---|---|
| | $x^5$ | $+x^4$ | | | $+x$ | |

| 0 | 1 | 0 | 0 | 1 | 1 | 1 |
|---|---|---|---|---|---|---|
| | $x^5$ | | | $+x^2$ | $+x$ | $+1$ |

The 16 codewords in the table above were selected by finding the degree-6 polynomials that are evenly divisible (modulo 2) by the generating polynomial $x^3+x+1$. For example, the third codeword '0100111' in the table corresponds to the polynomial $x^5+x^2+x+1$, which is divisible by $x^3+x+1$, because $x^5+x^2+x+1=(x^3+x+1)(x^2+1)$.

To understand how such polynomials can be calculated, let's consider similar operations on numbers. Suppose that we want to know the largest multiple of 7 that's less than or equal 30. We divide 30 by 7, obtaining a remainder of 2, and then subtract the 2 from the 30, getting 28. Polynomials are divided in a similar way. Let's start with the four information bits 0010, and calculate the remaining three parity bits. We write 0010*ppp* which gives us the polynomial $x^4$. We divide $x^4$ by the generating polynomial, obtaining a remainder of $x^2+x$. Subtracting that remainder from $x^4$ gives us something that will be evenly divisible by the generating polynomial. The result of the subtraction is $x^4+x^2+x$, so the complete codeword is 0010110.

Any generating polynomial can get us the first two properties. To get the third property (the necessary Hamming distance), the right generating polynomial should be used, and it can be selected by examining its roots (see [Lin 70]).

A common example of a generating polynomial is $\mathrm{CRC}(x)=x^{16}+x^{12}+x^5+1$. When dividing a large polynomial by $\mathrm{CRC}(x)$, the result is a polynomial of degree 15, which corresponds to a 16-bit codeword. There are standard polynomials used to calculate the CRC (cyclic redundancy codes) at the end of blocks of transmitted data (see page 140).

Other generating polynomials are $\mathrm{CRC}_{12}(x)=x^{12}+x^3+x+1$ and $\mathrm{CRC}_{16}(x)=x^{16}+x^{15}+x^2+1$. They generate the common CRC-12 and CRC-16 codes, which have lengths of 12 and 16 bits, respectively.

### 3.10 Data Compression

Data compression (officially called *source coding*) is a relatively new field, but it is an active one, with many researchers engaged in developing new approaches, testing new techniques, and implementing software. Quite a few compression methods currently exist. Some are suitable for the compression of text, while others have been developed for compressing images, video, or audio data. The main aim of this section is to introduce the reader to the main techniques and principles used to compress data. It turns out that the many compression methods in existence are based on just a small number of principles, the most important of which are variable-size codes, dictionaries, quantization, and transforms. These approaches are illustrated in the following sections, together with a few examples of specific compression methods.

Before we can look at the details of specific compression methods, we should answer the basic question, "How can data be compressed? How can we take a piece of data that's represented by $n$ bits and represent it by fewer than $n$ bits?" The answer is: Because data that's of interest to us contains redundancies. Such

data is not random and contains various patterns. By identifying the patterns and removing them, we can hope to reduce the number of bits required to represent the data. Any data compression algorithm must therefore examine the data to find the patterns (redundancies) in it and eliminate them. Data that does not have any redundancies is random and cannot be compressed.

The following simple argument illustrates the essence of the statement "Data compression is achieved by reducing or removing redundancy in the data." The argument shows that most data files cannot be compressed, no matter what compression method is used. This seems strange at first because we compress our data files all the time. The point is that most files cannot be compressed because they are random or close to random and therefore have no redundancy. The (relatively) few files that can be compressed are the ones that we *want* to compress; they are the files we use all the time. They have redundancy, are nonrandom and therefore useful and interesting.

Given two different files $A$ and $B$ that are compressed to files $C$ and $D$, respectively, it is clear that $C$ and $D$ must be different. If they were identical, there would be no way to decompress them and get back file $A$ or file $B$.

Suppose that a file of size $n$ bits is given and we want to compress it efficiently. Any compression method that can compress this file to, say, 10 bits would be welcome. Even compressing it to 11 bits or 12 bits would be great. We therefore (somewhat arbitrarily) assume that compressing such a file to half its size or better is considered good compression. There are $2^n$ $n$-bit files and they would have to be compressed into $2^n$ different files of sizes less than or equal $n/2$. However, the total number of these files is

$$N = 1 + 2 + 4 + \cdots + 2^{n/2} = 2^{1+n/2} - 1 \approx 2^{1+n/2},$$

so only $N$ of the $2^n$ original files have a chance of being compressed efficiently. The problem is that $N$ is much smaller than $2^n$. Here are two examples of the ratio between these two numbers.

For $n = 100$ (files with just 100 bits), the total number of files is $2^{100}$ and the number of files that can be compressed efficiently is $2^{51}$. The ratio of these numbers is the ridiculously small fraction $2^{-49} \approx 1.78 \cdot 10^{-15}$.

For $n = 1000$ (files with just 1000 bits, about 125 bytes), the total number of files is $2^{1000}$ and the number of files that can be compressed efficiently is $2^{501}$. The ratio of these numbers is the incredibly small fraction $2^{-499} \approx 9.82 \cdot 10^{-91}$.

▸ **Exercise 3.7:** Assuming that compressing a file down to 90% of its size or less is still considered good compression, compute the fraction of $n$-bit files that can be compressed well for $n = 100$ and $n = 1000$.

Most files of interest are at least some thousands of bytes long. For such files, the percentage of files that can be efficiently compressed is so small that it cannot be computed with floating-point numbers even on a supercomputer (the result is zero).

It is therefore clear that no compression method can hope to compress all files or even a significant percentage of them. In order to compress a data file, the compression algorithm has to examine the data, find redundancies in it, and try to remove them. Since the redundancies in data depend on the type of data (text, images, sound, etc.), any compression method has to be developed for a specific type of data and works best on this type. There is no such thing as a universal, efficient data compression algorithm.

## 3.11 Variable-Size Codes

So far we have discussed reliable codes. The principle behind those codes is increased redundancy, and this always results in codes that are longer than strictly necessary. In this section we are interested in short codes. We start with the following simple question: Given a set of symbols to be coded, what is the shortest possible set of codes for these symbols? In the case of four symbols, it seems that the shortest code should have a size of $m = 2$ bits per symbol. However, $\text{code}_7$ of Table 3.16 is different. It illustrates the principle of *variable-size codes*.

Codes of different sizes have been assigned, in $\text{code}_7$, to the four symbols, so now we are concerned with the *average size* of the code. The average size of this code is clearly between one and three bits per symbol. This kind of code makes sense if its average size turns out to be less than 2. To calculate the average size, we need to examine a typical message that consists of these symbols, such as

`BBBCABBCDCCBBABBCBDC.`

These 20 symbols are easy to code, and the result is the 35-bit string:

0|0|0|10|110|0|0|10|111|10|10|0|0|110|0|0|10|0|111|10.

The average size of the code, in this case, is $35/20 = 1.75$ bits per symbol. This is better than 2 bits/symbol, and it also illustrates the first principle of variable-size codes. The most-frequently occurring symbols should be assigned the shorter codes. These codes, therefore, make sense mostly in cases where we know in advance the frequency of occurrence of each symbol coded.

A common example is plain English text. It is well known that in "typical" English text (such as the works of Shakespeare), the letter "E" occurs most often, and the letters "Z" and "Q" are the least common, (actually, spaces between words occur more often than "E" and, since a space should also be assigned a code, it should be the shortest code).

A compression method is useful only if it creates code that can be decompressed correctly. When trying to decode the message above, we find out that it can be done easily and unambiguously. The first three zeros clearly indicate three "B"s. The following "1" indicates either an "A", a "C", or a "D". The receiver has to receive and examine the next bit (sometimes the next two bits) to find out what the next symbol is.

The reason our message can be decoded in just one way, has to do with the way the codes were assigned. Since "B" is the most common symbol, it was assigned the shortest code. The decision to assign "0" as the code of "B" was arbitrary; we could as well have assigned a "1". Once "0" was assigned as the code of "B", it is clear that none of the other codes can start with a "0"; they must, therefore, start with a "1", so their form is $1xxx$. Since "C" was the second most common symbol, it was assigned a short, 2-bit, code. It was "10" (but could have been "11"). It is now clear that the remaining symbols must be assigned codes starting with "11". Thus, the symbols "A" and "D" were assigned codes "110" and "111".

This process can be summarized by saying that, once a certain bit pattern has been assigned as the code of a symbol, that pattern cannot be used as the prefix of any other code. Our codes must have the *prefix property* (page 35) and are therefore called prefix codes. It is also clear that our codes are not unique. We could have assigned "C" the code 11, "A" the code 101, and "D", the code 100.

So far, our variable-size codes are not especially reliable. It is easy to see that one error during transmission can cause the receiver to get out of sync, and thus destroy a large part of the message. It is easy, however, to make those codes more reliable by adding parity bits. We can break up the coded message into chunks of reasonable size, and add a parity bit to each chunk. We can even add more than one parity bit to each chunk, making it extremely reliable. However, the reader should keep in mind that reliability and compression are opposites. The more reliable a code is made, the longer it gets.

## 3.12 Huffman Codes

An important question relating to variable-size codes is, given a set of symbols with known frequencies of occurrence, what code has the shortest average size for the set? There is, fortunately, a simple algorithm, due to David Huffman [Huffman 52], that produces such a code. It is also clear, from the discussion above, that such a code is not unique.

In the general case, where we have a set of symbols to be coded, and we are not limited to any particular sample of text, the average length depends only on the frequencies of occurrence of the different characters. Imagine a set of five symbols with frequencies: 10%, 15%, 30%, 20% and 25%. We can easily guess the set of codes 1101, 1100, 0, 111, and 10. The average size would be:

$$0.10 \times 4 + 0.15 \times 4 + 0.30 \times 1 + 0.20 \times 3 + 0.25 \times 2 = 2.65.$$

The average size is 2.65 bit/char., and again we may ask, is this the shortest possible size for the given frequencies?

The Huffman method is illustrated in Figure 3.18 using the five characters $A$, $B$, $C$, $D$, and $E$. We assume that they occur with frequencies of 10%, 15%, 30%, 20% and 25%, respectively.

The two lowest-frequency characters are $A$ and $B$ (10% and 15%). Their codes should therefore be the longest, and we start by tentatively assigning them the codes $0\ldots$ and $1\ldots$ where the "..." stands for more bits to be assigned later. In our character set, we temporarily replace the two characters $A$ and $B$ with the single character $x$, and assign $x$ a frequency that's the sum (25%) of the frequencies of $A$ and $B$.

Our character set now consists of the four symbols $D$ (20%), $E$ (25%), $x$ (25%), and $C$ (30%), and we repeat the previous step. We select any two characters with the lowest frequencies, say, $D$ and $x$, assign them the codes $0\ldots$ and $1\ldots$, and replace both temporarily with a new character $y$ whose frequency is the

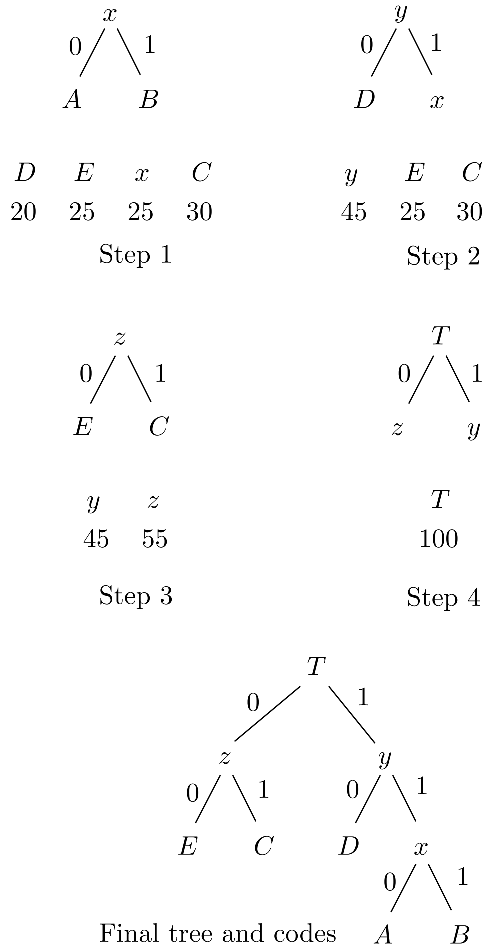



**Figure 3.19:** The Huffman method

sum $20+25=45\%$. Note that we could have selected $D$ and $E$ instead of $D$ and $x$. The Huffman code, like any other variable-length code, is not unique.

In the third step we select $E$ and $C$ and replace them with $z$ whose frequency is $25+30=55\%$. In the fourth and final step, we select the only remaining symbols, namely $z$ and $y$, assign them the codes 0 and 1, respectively, and replace them with the symbol $T$, whose frequency is $55+45=100\%$. Symbol $T$ stands for the entire character set.

In the general case of a set of $N$ symbols, this loop is repeated $N-1$ times. Following the loop, we construct a binary tree with $T$ as the root, $z$ and $y$ as the two sons of $T$ , $E$ as the left son of $z$, and so on. Comparing our tentative code assignments to the final tree shows that each left branch corresponds to an assignment of a 0, and each right branch, to a 1. The final codes are therefore:

$$A=110,\quad B=111,\quad C=01,\quad D=10,\quad \text{and}\quad E=00,$$

and the average code length is $0.10\times 3+0.15\times 3+0.30\times 2+0.20\times 2+0.25\times 2=2.25$ bits/char.

The Huffman algorithm is simple to implement, and it can be shown that it produces a set of codes with the minimum average size.

The most obvious disadvantage of variable-length codes is their vulnerability to errors. To achieve minimum size we have omitted parity bits and, even worse, we use the prefix property to decode those codes. As a result, an error in a single bit can cause the receiver to lose synchronization and be unable to decode the rest of the message. In the worst case, the receiver may even read, decode, and interpret the rest of the transmission wrong, without realizing that a problem has occurred.

Example: Using the code above, the string $CEDBCE$ is coded into: “01 00 10 111 01 00” (without the spaces). Assuming the following error: “01 00 10 111 00 00” the receiver will not notice any problem, but the fifth character decoded will be $E$ instead of $C$.

▸ **Exercise 3.8:** What will happen in the case ‘01 00 11 111 01 00’?

A simple way of adding reliability to variable length codes is to break a transmission into groups of 7 bits and add a parity bit to each group. This way the receiver will at least be able to detect a problem and ask for a retransmission.

### 3.12.1 Adaptive Huffman Coding

Up until now we have assumed that the compressor (encoder) knows the frequencies of occurrence of all the symbols in the message being compressed. In practice, this rarely happens, and practical Huffman coders can operate in one of three ways as follows:

1. A fixed set of Huffman codes. Both encoder and decoder use a fixed, built-in set of Huffman codes. This set is prepared once by selecting a set of representative texts and counting the frequencies of all the symbols in this set. We say that this set is used to “train” the algorithm. For English text, this set of “training” documents can be the complete works of Shakespeare.

2. A two-pass compression job. The encoder reads the input data twice. In the first pass it counts symbol frequencies and in the second pass it does the actual compression. In between the passes, the encoder computes a set of Huffman codes based on the frequencies counted in pass 1. This approach is conceptually simple and produces good compression, but it is too slow to be practical, because the input data has to be read from a disk, which is much slower than the processor. An encoder using this method has to write the set of Huffman codes at the start of the compressed file, since otherwise the decoder would not be able to decode the data.

3. An adaptive algorithm. The encoder reads the input data once and in this single pass it counts symbol frequencies *and* compresses the data. While data is being input and compressed, new Huffman codes are computed and assigned all the time, based on the symbol frequencies counted so far. An adaptive algorithm must be designed such that the decoder would be able to mimic the operations of the encoder at any point.

We present here a short description of an adaptive Huffman algorithm. The main idea is for the compressor and the decompressor to start with an empty Huffman tree, and to modify it as more and more characters are being read and processed (in the case of the compressor the word “processed” means compressed. In the case of the decompressor it means decompressed). The compressor and decompressor should modify the tree in the same way, so at any point in the process they should use the same codes, although those codes may change from step to step.

Initially, the compressor starts with an empty Huffman tree. No characters have any codes assigned. The first character read is simply written on the compressed file in its uncompressed form. The character is then added to the tree, and a code assigned to it. The next time this character is encountered, its current code is written on the file, and its frequency incremented by one. Since the tree has been modified, it is checked to see if it still a Huffman tree (best codes). If not, it is rearranged, thereby changing the codes.

The decompressor follows the same steps. When it reads the uncompressed form of a character, it adds it to the tree and assigns it a code. When it reads a compressed (variable-size) code, it uses the current tree to determine what character it is, and it updates the tree in the same way as the compressor.

The only subtle point is that the decompressor needs to know whether it is reading an uncompressed character (normally an 8-bit ASCII code) or a variable-size code. To remove any ambiguity, each uncompressed character is preceded by a special variable-size code. When the decompressor reads this code, it knows that the next 8 bits are the ASCII code of a character which appears in the file for the first time.

The trouble is that the special code should not be any of the variable-size codes used for the characters. Since these codes are being changed all the time, the special code should also be changed. A natural way to do this is to add an empty leaf to the tree, a leaf with a zero frequency of occurrence, which is always assigned to the 0-branch of the tree. Since the leaf is in the tree it gets a variable-size code assigned. This code is the special code preceding every uncompressed character. As the tree is being modified, the position of the empty leaf (and also its code) change, but the code is always used to identify uncompressed characters in the compressed file.

This method is used to compress/decompress data in the V.32 protocol for 14400 baud modems (Section 3.22.7).

## 3.13 Facsimile Compression

So far, we have assumed that the data to be compressed consists of a set of $N$ symbols, which is true for text. This section shows how Huffman codes can be used to compress simple images, images sent between facsimile (fax) machines. Those machines are made by many manufacturers, so a standard compression method was needed when they became popular. Several such methods were developed and proposed by the ITU-T.

The ITU-T is one of four permanent parts of the International Telecommunications Union (ITU), based in Geneva, Switzerland (`http://www.itu.ch/`). It issues recommendations for standards applying to modems, packet switched interfaces, V.24 connectors, etc. Although it has no power of enforcement, the standards it recommends are generally accepted and adopted by industry. Until March 1993, the ITU-T was known as the Consultative Committee for International Telephone and Telegraph (CCITT).

The first data compression standards developed by the ITU-T were T2 (also known as Group 1) and T3 (group 2). They are now obsolete and have been replaced by T4 (group 3) and T6 (group 4). Group 3 is currently used by all fax machines designed to operate with the Public Switched Telephone Network (PSTN). These are the machines we have at home, and they operate at maximum speeds of 9600 baud. Group 4 is used by fax machines designed to operate on a digital network, such as ISDN. They have typical speeds of 64K baud. Both methods can produce compression ratios of 10:1 or better, reducing the transmission time of a typical page to about a minute with the former, and a few seconds with the latter.

A fax machine scans a document line by line, converting each line to small black and white dots called *pels* (from Picture ELement). The horizontal resolution is always 8.05 pels per millimeter (about 205 pels per inch). An 8.5-inch wide scan line is thus converted to 1728 pels. The T4 standard, though, recommends scanning only about 8.2 inches, thus producing 1664 pels per scan line (these numbers, as well as the ones in the next paragraph, are all to within $\pm1\%$ accuracy).

The vertical resolution is either 3.85 scan lines per millimeter (standard mode) or 7.7 lines/mm (fine mode). Many fax machines have also a very-fine mode, where they scan 15.4 lines/mm. Table 3.19 assumes a 10-inch high page (254 mm) and shows the total number of pels per page, and typical transmission times for the three modes without compression. The times are long, which shows how important data compression is in fax transmissions.

Ten inches equal 254mm. The number of pels is in the millions and the transmission times, at 9600 baud without compression, are between 3 and 11 minutes, depending on the mode. However, if the page is shorter than 10 inches, or if most of it is white, the compression ratio can be 10:1 or better, resulting in transmission times of between 17 and 68 seconds.

| scan lines | pels per line | pels per page | time (sec) | time (min) |
|---|---|---|---|---|
| 978 | 1664 | 1.670M | 170 | 2.82 |
| 1956 | 1664 | 3.255M | 339 | 5.65 |
| 3912 | 1664 | 6.510M | 678 | 11.3 |

**Table 3.20:** Fax transmission times

The group 3 compression method is based on the simple concept of *run lengths*. When scanning a row of pels in an image, we can expect to find runs of pels of the same color. In general, if we point to a pel at random and find that it is white, chances are that its immediate neighbors will also be white, and the same is true for black pels. The group 3 method starts by counting the lengths of runs of identical pels in each scan line of the image. An image with long runs yields a small number of run lengths, so writing them on a file produces compression of the original image. The problem is that a run length can be any number between 1 and the length of a scan line (1664 pels), so the algorithm should specify a compact way to write numbers of various sizes on a file.

The group 3 method solves this problem by assigning Huffman codes to the various run lengths. The codes are written on the compressed file, and they should be selected such that the shortest codes should be assigned to the most common run lengths.

To develop the group 3 code, the ITU-T selected a set of eight representative "training" documents and analysed the run lengths of white and black pels on these documents. The Huffman algorithm was then used

to assign a variable-size code to each run length. The most common run lengths were found to be 2, 3, and 4 black pixels, so they were assigned the shortest codes (Table 3.20). Next came run lengths of 2 to 7 white pixels. They were assigned slightly longer codes. Most run lengths were rare, and were assigned long, 12-bit codes.

| run length | White code-word | Black code-word |
|---|---|---|
| 0 | 00110101 | 0000110111 |
| 1 | 000111 | 010 |
| 2 | 0111 | 11 |
| 3 | 1000 | 10 |
| 4 | 1011 | 011 |
| 5 | 1100 | 0011 |
| 6 | 1110 | 0010 |
| 7 | 1111 | 00011 |
| 8 | 10011 | 000101 |
| 9 | 10100 | 000100 |
| 10 | 00111 | 0000100 |
| ⋮ | | |
| 34 | 00010011 | 000011010010 |
| 35 | 00010100 | 000011010011 |
| 36 | 00010101 | 000011010100 |
| 37 | 00010110 | 000011010101 |
| 38 | 00010111 | 000011010110 |
| 39 | 00101000 | 000011010111 |
| 40 | 00101001 | 000001101100 |
| 41 | 00101010 | 000001101101 |
| 42 | 00101011 | 000011011010 |
| ⋮ | | |
| 63 | 00110100 | 000001100111 |

(a)

| run length | White code-word | Black code-word |
|---|---|---|
| 64 | 11011 | 0000001111 |
| 128 | 10010 | 000011001000 |
| 192 | 010111 | 000011001001 |
| 256 | 0110111 | 000001011011 |
| 320 | 00110110 | 000000110011 |
| 384 | 00110111 | 000000110100 |
| 448 | 01100100 | 000000110101 |
| 512 | 01100101 | 0000001101100 |
| 576 | 01101000 | 0000001101101 |
| 640 | 01100111 | 0000001001010 |
| 704 | 011001100 | 0000001001011 |
| ⋮ | | |
| 1600 | 010011010 | 0000001011011 |
| 1664 | 011000 | 0000001100100 |
| 1728 | 010011011 | 0000001100101 |
| 1792 | 00000001000 | same as |
| 1856 | 00000001100 | white |
| 1920 | 00000001101 | from this |
| 1984 | 000000010010 | point |
| 2048 | 000000010011 | |
| 2112 | 000000010100 | |
| ⋮ | | |
| 2560 | 000000011111 | |

(b)

**Table 3.21:** Some Group 3 and 4 fax codes, (a) termination codes, (b) make-up codes

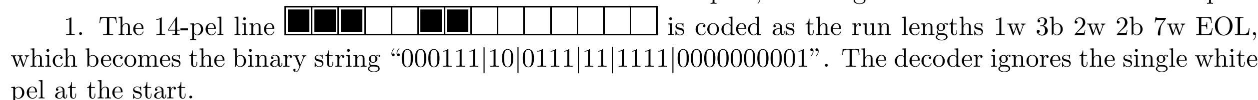
▸ **Exercise 3.9:** A run length of 1664 white pels was assigned the short code 011000. Why is this length so common?

Since run lengths can be long, the Huffman algorithm was modified. Codes were assigned to run lengths of 1 to 63 pels (they are the termination codes of Table 3.20a) and to run lengths that are multiples of 64 pels (the make-up codes of Table 3.20b). Group 3 is thus a *modified Huffman code* (also called MH). The code of a run length is either a single termination code (if the run length is short) or one or more make-up codes, followed by one termination code (if the run length is long). Here are some examples:

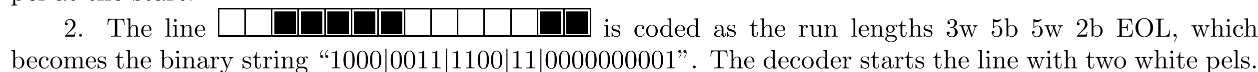
1. A run length of 12 white pels is coded as 001000.
2. A run length of 76 white pels (=64+12) is coded as 11011|001000 (without the vertical bar).
3. A run length of 140 white pels (=128+12) is coded as 10010|001000.
4. A run length of 64 black pels (=64+0) is coded as 0000001111|0000110111.
5. A run length of 2561 black pels (2560+1) is coded as 000000011111|010.

▸ **Exercise 3.10:** An 8.5-inch-wide scan line results in 1728 pels, so how can there be a run of 2561 consecutive pels?

Each scan line is coded separately, and its code is terminated with the special EOL code 00000000001. Each line also gets one white pel appended to it on the left when it is scanned. This is done to remove any ambiguity when the line is decoded on the receiving side. After reading the EOL for the previous line, the receiver assumes that the new line starts with a run of white pels, and it ignores the first of them. Examples:



1. The 14-pel line is coded as the run lengths 1w 3b 2w 2b 7w EOL, which becomes the binary string "000111|10|0111|11|1111|0000000001". The decoder ignores the single white pel at the start.



2. The line is coded as the run lengths 3w 5b 5w 2b EOL, which becomes the binary string "1000|0011|1100|11|0000000001". The decoder starts the line with two white pels.

▸ **Exercise 3.11:** The group 3 code for a run length of five black pels (0011) is also the prefix of the codes for run lengths of 61, 62, and 63 white pels. Explain this.

The group 3 code has no error correction, but many errors can be detected. Because of the nature of the Huffman code, even one bad bit in the transmission can cause the receiver to get out of synchronization, and to produce a string of wrong pels. This is why each scan line is encoded separately. If the receiver detects an error, it skips bits, looking for an EOL. This way, at most one scan line can be received incorrectly. If the receiver does not see an EOL after a certain number of lines, it assumes a high error rate, and it aborts the process, notifying the transmitter. Since the codes are between 2 and 12 bits long, the receiver can detect an error if it cannot decode a valid code after reading 12 bits.

Each page of the coded document is preceded by one EOL and is followed by six EOL codes. Because each line is coded separately, this method is a *one-dimensional coding* scheme. The compression ratio depends on the image. Images with large contiguous black or white areas (text or black and white diagrams) will highly compress. Images with many short runs can sometimes produce negative compression. This is especially true in the case of images with shades of gray (mostly photographs). Such shades are produced by halftoning, which covers areas with alternating black and white pels (runs of length 1).

▸ **Exercise 3.12:** What is the compression ratio for runs of length one (many alternating pels)?

The T4 standard also allows for *fill bits* to be inserted between the data bits and the EOL. This is done in cases where a pause is necessary, or where the total number of bits transmitted for a scan line must be a multiple of 8. The fill bits are zeros.

Example: The binary string "000111|10|0111|11|1111|0000000001" becomes "000111|10|0111|11|1111|0000|0000000001" after four zeros are added as fill bits, bringing the total length of the string to 32 bits ($= 8 \times 4$). The decoder sees the four zeros of the fill, followed by the nine zeros of the EOL, followed by the single 1, so it knows that it has encountered a fill followed by an EOL.

The group 3 method is described in `http://www.cis.ohio-state.edu/htbin/rfc/rfc804.html`. At the time of writing, the T.4 and T.6 recommendations can also be found as files `7_3_01.ps.gz` and `7_3_02.ps.gz` at `src.doc.ic.ac.uk/computing/ccitt/ccitt-standards/1988/`.

The group 3 method is called one-dimensional because it encodes each scan line of the image separately. For images with gray areas, this type of coding does not produce good compression, which is why the group 4 method was developed. The group 4 method is a two-dimensional compression method. It compresses a scan line by recording the differences between it and its predecessor. Obviously, the first scan line has to compressed as with group 3. The method is efficient since consecutive scan lines are normally very similar (remember that there are about 200 of them per inch). However, a transmission error in one scan line will cause that line and all its successors to be bad. Therefore, two-dimensional compression reverts to one-dimensional compression from time to time, compresses one scan line independently of its predecessor, then switches back to two-dimensional coding.

## 3.14 Dictionary-Based Methods

Such a method reads the data to be compressed and tries to match any new data to data fragments that have already been seen. The algorithm has to keep as much data as possible in memory in a data structure called a *dictionary*. When a string of new data matches a string in the dictionary, the algorithm outputs a pointer $P$ to the dictionary. The length $L$ of the matched string is also output. If the total size of $P$ and $L$ is less than that of the matched string, compression is achieved.

Following is a short description of the oldest and simplest of the dictionary-based methods, the LZ77 sliding window method. The main idea behind this method [Ziv 77] is to use part of the previously-seen input data as the dictionary. The encoder maintains a window to the input data, and shifts the input in that window to the left as strings of symbols are being encoded. Thus, the method is based on a *sliding window*. The window below is divided into two parts. The part on the left is called the *search buffer*. This is the current dictionary, and it always includes symbols that have been input and encoded recently. The part on the right is the *look-ahead buffer*, containing text to be encoded. In practical implementations, the search buffer is some thousands of bytes long, and the look-ahead buffer is only tens of bytes long. The vertical bar between the '`t`' and the '`e`' below represents the current dividing line between the two buffers.

← encoded text. . . `sir␣sid␣eastman␣easily␣t|eases␣sea␣sick␣seals` . . . ← text yet to be input

The encoder scans the search buffer backwards (from right to left) for a match to the first symbol '`e`' in the look-ahead buffer. It finds it at the '`e`' of the word '`easily`'. This '`e`' is at a distance (offset) of 8 from the end of the search buffer. The encoder then matches as many symbols following the two '`e`'s as possible. Three symbols '`eas`' match in this case, so the length of the match is 3. The encoder than continues the backward scan, trying to find longer matches. In our case, there is one more match, at the word '`eastman`', with offset 16, and it has the same length. The encoder selects the longest match or, if they are all the same length, the last one found, and prepares the token (16, 3, '`e`').

Selecting the last match, rather than the first one, simplifies the program, since it only has to keep track of the last match found. It is interesting to note that selecting the first match, while making the program somewhat more complex, also has an advantage. It selects the smallest offset. It would seem that this is not an advantage, since a token should have room for the largest possible offset. However, it is possible to follow LZ77 with Huffman coding of the tokens, where small offsets are assigned shorter codes. This method is called LZH. Many small offsets implies a smaller output file in LZH.

In general, a token has three parts, offset, length and next symbol in the look-ahead buffer (which, in our case, is the **second** '`e`' of the word '`teases`'). This token is written on the output file, and the window is shifted to the right (or, alternatively, the input data is moved to the left) four positions, 3 positions for the matched string and one position for the next symbol.

`...sir |sid␣eastman␣easily␣tease|s␣sea␣sick␣seals...|...`

If the backward search yields no match, a token with zero offset and length, and with the unmatched symbol is written. This is also the reason why a token has to have a third component. Such tokens are common at the beginning of the compression job, when the search buffer is empty. The first five steps in encoding our example are

| Search buffer | Look-ahead buffer | | Token |
|---|---|---|---|
| | `sir␣sid␣eastman␣` | $\Rightarrow$ | (0,0,'`s`') |
| `s` | `ir␣sid␣eastman␣e` | $\Rightarrow$ | (0,0,'`i`') |
| `si` | `r␣sid␣eastman␣ea` | $\Rightarrow$ | (0,0,'`r`') |
| `sir` | `␣sid␣eastman␣eas` | $\Rightarrow$ | (0,0,'`␣`') |
| `sir␣` | `sid␣eastman␣easi` | $\Rightarrow$ | (4,2,'`d`') |

▸ **Exercise 3.13:** What are the next two steps?

Clearly, a token of the form (0,0,...), which encodes a single symbol, does not provide good compression. It is easy to estimate its length. The size of the offset is $\lceil \log_2 S \rceil$ where $S$ is the length of the search buffer. In practice, the search buffer may be a few thousand bytes long, so the offset size is typically 10–12 bits. The size of the 'length' field is, similarly $\lceil \log_2(L-1) \rceil$ where $L$ is the length of the look-ahead buffer (see below for the $-1$). In practice the look-ahead buffer is only a few tens of bytes long, so the size of the 'length' field is just a few bits. The size of the 'symbol' field is typically 8 bits but, in general it is $\lceil \log_2 A \rceil$ where $A$ is the alphabet size. The total size of the 1-symbol token (0,0,...) may typically be $11 + 5 + 8 = 24$ bits, much longer than the raw 8-bit size of the (single) symbol it encodes.

Here is an example showing why the 'length' field may be longer than the size of the look-ahead buffer.

`...Mr.␣|alf␣eastman␣easily␣grows␣alf|alfa␣in␣his␣|garden...`

The first letter '`a`' in the look-ahead buffer matches the five '`a`'s in the search buffer. It seems that the two extreme '`a`'s match with a length of 3 and the encoder should select the last (leftmost) of them and create the token (28 ,3, '`a`'). In fact it creates the token (3, 4, '`␣`'). The 4-letter string '`alfa`' in the look-ahead buffer is matched to the last three letters '`alf`' in the search buffer **and** the first letter '`a`' in the look-ahead buffer. The reason for this is that the decoder can handle such a token naturally, without any modifications. It starts at position 3 of its search buffer and copies the next four letters, one by one, extending its buffer to the right. The first three letters are copies of the old buffer contents, and the fourth one is a copy of the first of those three. The next example is even more convincing (and only somewhat contrived)

`···|alf␣eastman␣easily␣yells␣A|AAAAAAAAAA|AAAAAH....`

The encoder creates the token (1, 9, '`A`'), matching the first nine copies of '`A`' in the look-ahead buffer and including the tenth '`A`'. This is why, in principle, the length of a match could be up to the size of the look-ahead buffer minus one.

The decoder is much simpler than the encoder. It has to maintain a buffer, equal in size to the encoder's window. The decoder inputs a token, finds the match in its buffer, writes the match and the third token field

on its output file, and shifts the matched string and the third field into the buffer. This implies that LZ77, or any variants, are useful in cases where a file is compressed once (or a few times) and is decompressed often. A rarely-used archive of compressed files is a good example.

At first it seems that this method does not make any assumptions about the input data. Specifically, it does not pay attention to any symbol frequencies. A little thinking, however, shows that, because of the nature of the sliding window, the LZ77 method always compares the look-ahead buffer to the recently input text in the search buffer and never to text that was input long ago. Thus, the method implicitly assumes that patterns in the input data occur close together. Data that satisfies this assumption will compress well.

The basic LZ77 method has been improved in several ways by researchers and programmers during the 1980s and 1990s. One way to improve it is to use variable-size 'offset' and 'length' fields in the tokens. Another way is to increase the sizes of both buffers. Increasing the size of the search buffer makes it possible to find better matches, but the tradeoff is an increased search time. A large search buffer thus requires a more sophisticated data structure that allows for fast search. A third improvement has to do with sliding the window. The simplest approach is to move all the text in the window to the left after each match. A faster method is to replace the linear window with a *circular queue*, where sliding the window is done by resetting two pointers. Yet another improvement is adding an extra bit (a flag) to each token, thereby eliminating the third field.

## 3.15 Approaches to Image Compression

A digital image is a rectangular set of dots, or picture elements, arranged in $m$ rows and $n$ columns. The pair $m \times n$ is called the *resolution* of the image, and the dots are called *pixels* (except in the case of a fax image, where they are referred to as *pels*). For the purpose of image compression it is useful to distinguish the following types of images:

1. A *bi-level* (or monochromatic) image. This is an image where the pixels can have one of two values, normally referred to as black and white. Each pixel can thus be represented by one bit. This is the simplest type of image.

2. A *grayscale* image. A pixel in such an image can have one of $n$ values, indicating one of $2^n$ shades of gray (or shades of some other color). The value of $n$ is normally compatible with a byte size, i.e., it is 4, 8, 12, 16, 24, or some other convenient multiple of 4 or of 8.

3. A *continuous-tone* image. This type of image can have many similar colors (or grayscales). When adjacent pixels differ by just one unit, it is hard or even impossible for the eye to distinguish between their colors. As a result, such an image may contain areas with colors that seem to vary continuously as the eye moves along the area. A pixel in such an image is represented by either a single large number (in the case of many grayscales) or by three components (in the case of many colors). This type of image can easily be obtained by taking a photograph with a digital camera, or by scanning a photograph or a painting.

4. A *cartoon-like* image. This is a color image which consists of uniform areas. Each area has a uniform color but adjacent areas may have very different colors. This feature may be exploited to obtain better compression.

This section discusses general approaches to image compression, not specific methods. These approaches are different, but they remove redundancy from an image by using the following principle: If we select a pixel in the image at random, there is a good chance that its neighbors will have the same color or very similar colors. Thus, image compression is based on the fact that neighboring pixels are *highly correlated.*

### 3.15.1 Differencing

It is posible to get fairly good compression by simply subtracting each pixel in a row of an image from its predecessor and writing the differences (properly encoded by variable-size, prefix codes) on the compressed file. Differencing produces compression because the differences are small numbers and are decorrelated. The following sequence of values are the intensities of 24 adjacent pixels in a row of a continuous-tone, grayscale image

$$12, 17, 14, 19, 21, 26, 23, 29, 41, 38, 31, 44, 46, 57, 53, 50, 60, 58, 55, 54, 52, 51, 56, 60.$$

Only two of the 24 pixels are identical. Their average value is 40.3. Subtracting pairs of adjacent pixels results in the sequence

$$12, 5, -3, 5, 2, 4, -3, 6, 11, -3, -7, 13, 4, 11, -4, -3, 10, -2, -3, 1, -2, -1, 5, 4.$$

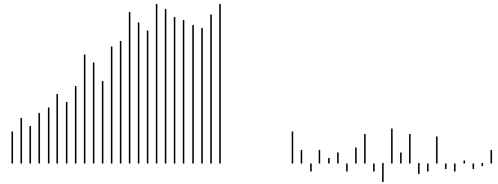


**Figure 3.22:** Values and Differences of 24 Adjacent Pixels.

The two sequences are illustrated in Figure 3.21.

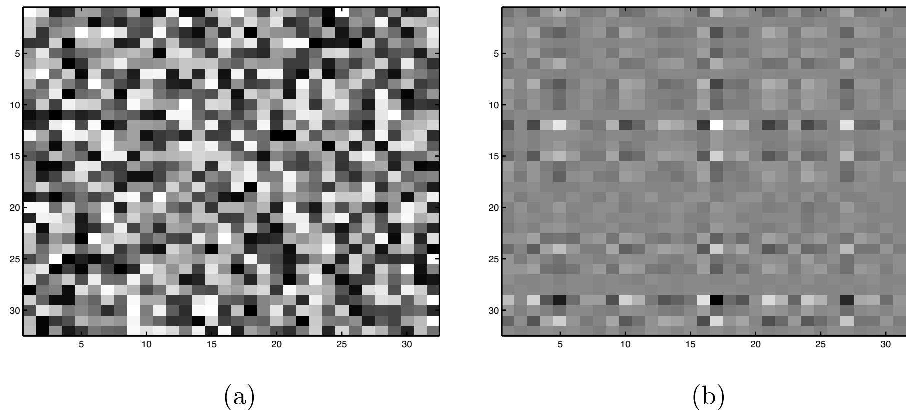
The sequence of difference values has three properties that illustrate its compression potential: (1) The difference values are smaller than the original pixel values. Their average is 2.58. (2) They repeat. There are just 15 distinct difference values, so in principle they can be coded by four bits each. (3) They are *decorrelated*; adjacent difference values tend to be different. This can be seen by subtracting them, which results in the sequence of 24 second differences

$$12,\ -7,\ -8,\ 8,\ -3,\ 2,\ -7,\ 9,\ 5,\ -14,\ -4,\ 20,\ -11,\ 7,\ -15,\ 1,\ 13,\ -12,\ -1,\ 4,\ -3,\ 1,\ 6,\ 1.$$

They are larger than the differences themselves.

Figure 3.22 provides another illustration of the meaning of the words "correlated quantities." A 32×32 matrix $A$ is constructed of random numbers, and its elements are displayed in part (a) as shaded squares. The random nature of the elements is obvious. The matrix is then inverted and stored in $B$, which is shown in part (b). This time, there seems to be more structure to the 32×32 squares. A direct calculation using Equation (3.1) shows that the cross-correlation between the top two rows of $A$ is 0.0412, whereas the cross-correlation between the top two rows of $B$ is −0.9831. The elements of $B$ are correlated since each depends on *all* the elements of $A$

$$R = \frac{n\sum x_i y_i - \sum x_i \sum y_i}{\sqrt{[n\sum x_i^2 - (\sum x_i)^2][n\sum y_i^2 - (\sum y_i)^2]}}. \tag{3.1}$$



(a) (b)

**Figure 3.23:** Maps of a Random Matrix (a) and Its Inverse (b).

```
n=32; a=rand(n); imagesc(a); colormap(gray)
b=inv(a); imagesc(b)
```

Code for Figure 3.22.

▶ **Exercise 3.14:** Use mathematical software to illustrate the covariance matrices of (1) a matrix with correlated values and (2) a matrix with decorrelated values.

### 3.15.2 Image Transforms

The mathematical concept of a transform is a powerful tool in many areas and can also serve as an approach to image compression. An image can be compressed by transforming its pixels (which are correlated) to a representation where they are *decorrelated*. Compression is achieved if the new values are smaller, on average, than the original ones. Lossy compression can be achieved by quantizing the transformed values. The decoder inputs the transformed values from the compressed stream and reconstructs the (precise or approximate) original data by applying the opposite transform. The transforms discussed in this section are *orthogonal*. There is also a large family of *subband transforms*. They are generated by wavelet methods and will not be discussed here.

The term *decorrelated* means that the transformed values are independent of one another. As a result, they can be encoded independently, which makes it simpler to construct a statistical model. An image can be compressed if its representation has redundancy. The redundancy in images stems from pixel correlation. If we transform the image to a representation where the pixels are decorrelated, we have eliminated the redundancy and the image has been fully compressed.

We start with a simple example, where we scan an image in raster order and group pairs of adjacent pixels. Because the pixels are correlated, the two pixels $(x, y)$ of a pair normally have similar values. We now consider the pairs of pixels as points in two-dimensional space, and plot them. We know that all the points of the form $(x, x)$ are located on the 45° line $y = x$, so we expect our points to be concentrated around this line. Figure 3.23a shows the results of plotting the pixels of a typical image—where a pixel has values in the interval $[0, 255]$—in such a way. Most points form a cloud around this line, and only a few points are located away from it. We now transform the image by rotating all the points 45° clockwise about the origin, such that the 45° line now coincides with the $x$-axis (Figure 3.23b). This is done by the simple transformation

$$(x^*, y^*) = (x, y)\begin{pmatrix}\cos 45^\circ & -\sin 45^\circ\\ \sin 45^\circ & \cos 45^\circ\end{pmatrix} = (x, y)\frac{1}{\sqrt{2}}\begin{pmatrix}1 & -1\\ 1 & 1\end{pmatrix} = (x, y)\mathbf{R}, \tag{3.2}$$

where the rotation matrix $\mathbf{R}$ is orthonormal (i.e., the dot product of a row with itself is 1, the dot product of different rows is 0, and the same is true for columns). The inverse transformation is

$$(x, y) = (x^*, y^*)\mathbf{R}^{-1} = (x^*, y^*)\mathbf{R}^T = (x^*, y^*)\frac{1}{\sqrt{2}}\begin{pmatrix}1 & 1\\ -1 & 1\end{pmatrix}. \tag{3.3}$$

(The inverse of an orthonormal matrix is its transpose.)

It is obvious that most points end up having $y$-coordinates that are zero or close to zero, while the $x$-coordinates don't change much. Figure 3.24a,b shows that the distributions of the $x$ and $y$ coordinates (i.e., the odd-numbered and even-numbered pixels of an image) before the rotation don't differ substantially. Figure 3.24c,d shows that the distribution of the $x$ coordinates stays almost the same but the $y$ coordinates are concentrated around zero.

Once the coordinates of points are known before and after the rotation, it is easy to measure the reduction in correlation. A simple measure is the sum $\sum_i x_i y_i$, also called the *cross-correlation* of points $(x_i, y_i)$.

▸ **Exercise 3.15:** Given the five points $(5, 5)$, $(6, 7)$, $(12.1, 13.2)$, $(23, 25)$, and $(32, 29)$ rotate them 45° clockwise and calculate their cross-correlations before and after the rotation.

We can now compress the image by simply writing the transformed pixels on the compressed stream. If lossy compression is acceptable, then all the pixels can be quantized, resulting in even smaller numbers. We can also write all the odd-numbered pixels (those that make up the $x$ coordinates of the pairs) on the compressed stream, followed by all the even-numbered pixels. These two sequences are called the *coefficient vectors* of the transform. The latter sequence consists of small numbers and may, after quantization, have runs of zeros, resulting in even better compression.

It can be shown that the total variance of the pixels does not change by the rotation, since a rotation matrix is orthonormal. However, since the variance of the new $y$ coordinates is small, most of the variance is now concentrated in the $x$ coordinates. The variance is sometimes called the *energy* of the distribution of pixels, so we can say that the rotation has concentrated (or compacted) the energy in the $x$ coordinate and has created compression this way.

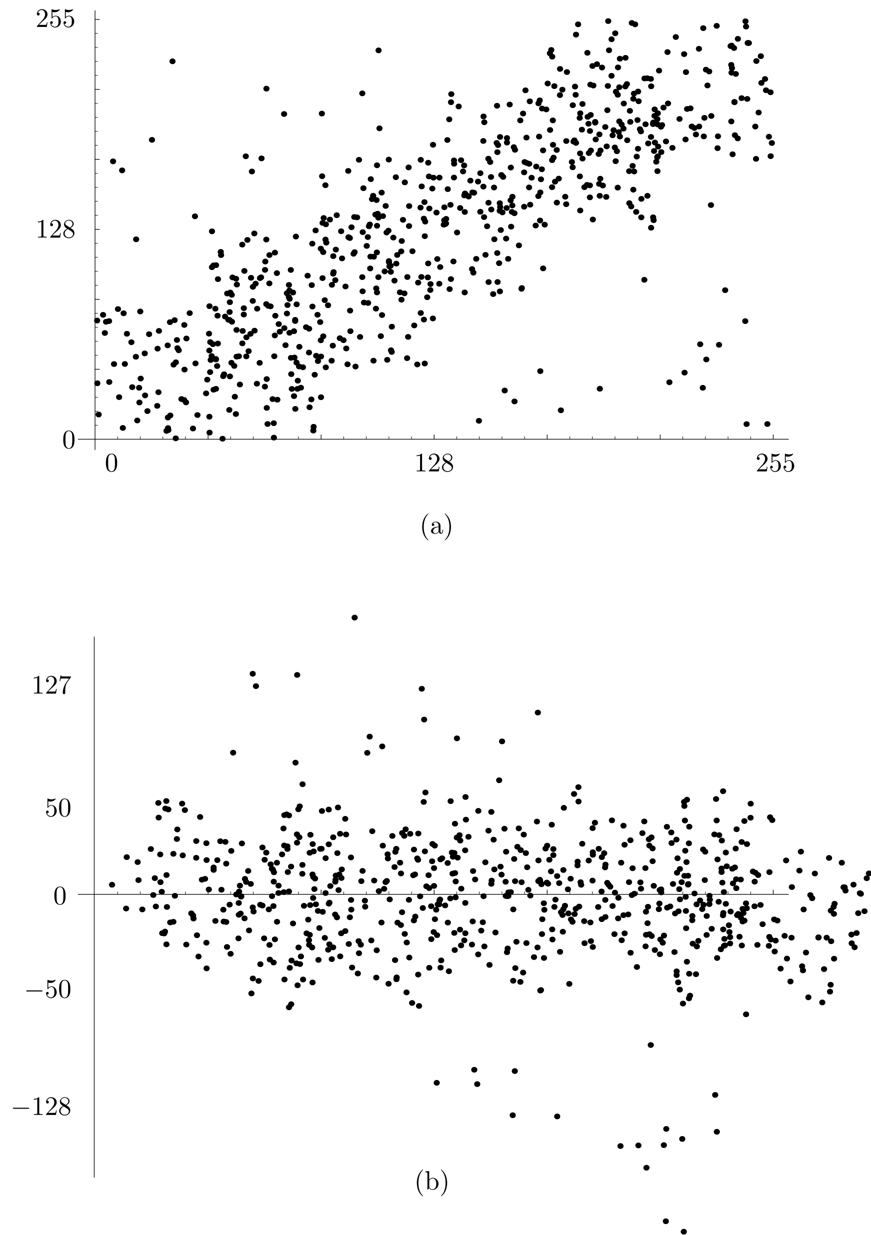



**Figure 3.24:** Rotating A Cloud of Points.

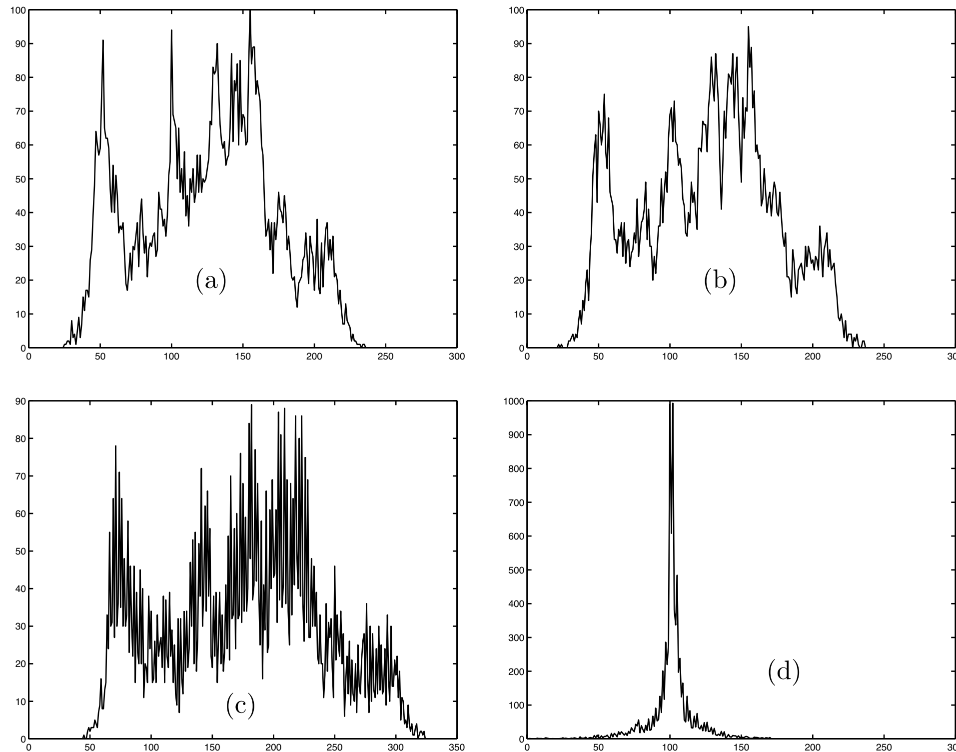



**Figure 3.25:** Distribution of Image Pixels Before and After Rotation.

Concentrating the energy in the $x$ coordinate has another advantage. We know that this coordinate is important, so we quantize the $y$ coordinate (which is unimportant) coarsely. This increases compression while losing only unimportant image data. The $x$ coordinate should be quantized only lightly (fine quantization).

The following simple example illustrates the power of this basic transform. We start with the point $(4,5)$, whose two coordinates are similar. Using Equation (3.2) the point is transformed to $(4,5)\mathbf{R} = (9,1)/\sqrt{2} \approx (6.36396, 0.7071)$. The energies of the point and its transform are $4^2+5^2 = 41 = (9^2+1^2)/2$. If we delete the smaller coordinate (4) of the point, we end up with an error of $4^2/41 = 0.39$. If, on the other hand, we delete the smaller of the two transform coefficients (0.7071), the resulting error is just $0.7071^2/41 = 0.012$. Another way to obtain the same error is to consider the reconstructed point. Passing $\frac{1}{\sqrt{2}}(9,1)$ through the inverse transform [Equation (3.3)] results in the original point $(4,5)$. Doing the same with $\frac{1}{\sqrt{2}}(9,0)$ results in the approximate reconstructed point $(4.5, 4.5)$. The energy difference between the original and reconstructed points is the same small quantity

$$\frac{\left[(4^2+5^2)-(4.5^2+4.5^2)\right]}{4^2+5^2} = \frac{41-40.5}{41} = 0.0012.$$

This simple transform can easily be extended to any number of dimensions, with the only difference that we cannot visualize spaces of more than three dimensions. However, the mathematics can easily be extended. Instead of selecting pairs of adjacent pixels we can select triplets. Each triplet becomes a point in three-dimensional space, and these points form a cloud concentrated around the line that forms equal (although not 45°) angles with the three coordinate axes. When this line is rotated such that it coincides with the $x$ axis, the $y$ and $z$ coordinates of the transformed points become small numbers. The transformation is done by multiplying each point by a 3×3 rotation matrix, and such a matrix is, of course, orthonormal. The transformed points are then separated into three coefficient vectors, of which the last two consist of small numbers. For maximum compression each coefficient vector should be quantized separately.

Here is a very practical approach to lossy or lossless image compression that uses rotations in 8-dimensional space.

1. Divide the image into blocks of $8 \times 8$ pixels each. One such block, with 64 pixels denoted by L (large) is shown in Figure 3.25a.

2. Rotate each of the eight rows of a block. The result of rotating a row in eight dimensions is eight numbers, of which the last 7 are small (denoted by S in Figure 3.25b).

3. Rotate each of the eight columns of a block. The result of rotating all the columns is a block of $8 \times 8$ *transform coefficients*, of which the coefficient at the top-left corner is large (it is called the DC coefficient) and the remaining 63 coefficients (called the AC coefficients) are small (Figure 3.25c). Thus, this double rotation concentrates the energy in the first of 64 dimensions. This type of transform is called the two-dimensional discrete cosine transform or DCT.

4. If lossy compression is desired, the 63 AC coefficients can be coarsely quantized. This brings most of them to zero, which helps in subsequent steps, while losing only unimportant image data.

5. Experience shows that the set of 64 transform coefficients get smaller in value as we move from the top-left to the bottom-right corner of the $8 \times 8$ block. We therefore scan the block in zigzag pattern (Figure 3.25d), resulting in a linear sequence of 64 coefficients that get smaller and smaller.

6. This sequence normally has many zeros. The run lengths of those zeros are determined and the entire sequence is encoded by replacing each nonzero coefficient and each run length of zeros with a Huffman code.

This method is the basis of the well-known JPEG image compression standard.

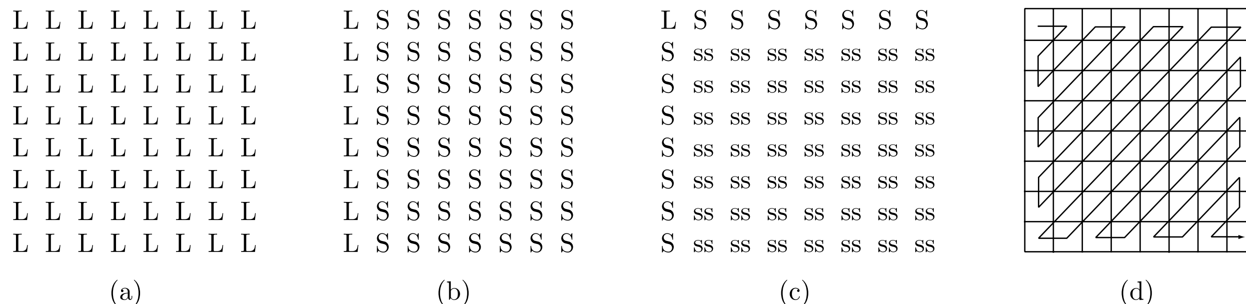



**Figure 3.26:** A two-dimensional DCT and a zigzag scan pattern

## 3.16 Secure Codes

Secure codes have always been important to kings, tyrants, and generals; and equally important to their opponents. Messages sent by a government to various parts of the country have to be encoded in case they fall into the wrong hands, and the same is true of orders issued by generals. However, the "wrong hands" consider themselves the right side, not the wrong one, and always try to break secure codes. As a result, the development of secure codes has been a constant race between cryptographers (code makers) and cryptanalysts (code breakers). New codes have been developed throughout history and were broken, only for newer, more sophisticated codes to be developed. This race has accelerated in the 20th century, because of (1) the two World Wars, (2) advances in mathematics, and (3) the development of computers. A general online reference for secure codes is [Savard 01].

Secure codes are becoming prevalent in modern life because of the fast development of telecommunications. Our telephone conversations are sometimes transmitted through communications satellites and our email messages may pass through many computers, thereby making our private conversations vulnerable to interception. We sometimes wish we had a secure code to use for our private communications. Businesses and commercial enterprises rely heavily on sending and receiving messages and also feel the need for secure communications. On the other hand, widespread use of secure codes worries law enforcement agencies, since criminals (organized or otherwise) and terrorists may also use secure codes if and when they become widely available.

The field of secure codes is divided into cryptography and steganography, and cryptography is further divided into codes and ciphers (Figure 3.26). Steganography (Section 3.20) deals with information hiding, while cryptography—derived from the Greek word *kryptos*, meaning "hidden"—tries to encode information, so it is impossible (or at least very hard) to decipher. The term "code" refers to codes for words, phrases, or for entire messages, while `cipher` is a code for each individual letter. For example, army units may agree

on the codeword `green` to mean `attack at dawn` and on `red` to imply `retreat immediately`. The words `green` and `red` are codes. A code may be impossible to break, but the use of codes is limited, because codes have to be agreed upon for every possible eventuality. A cipher, on the other hand, is a rule that tells how to encode each letter in a message. Thus, for example, if we agree to replace each letter with the one two places ahead of it in the alphabet, then the message `attack at dawn` will be encoded as `crrcem cr fcyp` or, even more securely, as `crrcemcrfcyp`. A cipher is general, but can be broken. In practice, however, we use the terms *code* and *codebreaker* instead of the more accurate *cipher* and *cipherbreaker*.

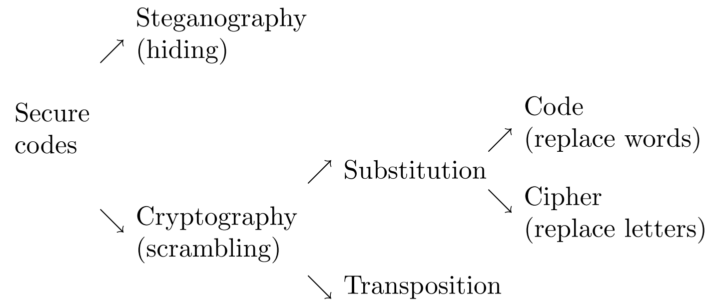



**Figure 3.27:** The Terminology of Secure Codes.

A combination of code and cipher, called *nomenclator*, is also possible. Parts of a message may be encrypted by codes, and the rest of the message, the parts for which codes do not exist, is encrypted by a cipher.

It is a popular saying that the First World War was the chemists' war, because of the large-scale use of poison gas for the first time. The Second World War was the physicists' war because of the use of the atom bomb. Similarly, the Third World War (that we hope can be avoided) may turn out to be the mathematicians' war, because winning that war, if at all possible, may depend on the use of and the breaking of, secure codes.

Some of the development as well as the breaking of codes is done by researchers at universities and research institutes all over the world (see [Flannery 00] for an unusual example of this). It is generally agreed, however, that most of the work in this field is done in secret by government agencies. Two well-known examples of such agencies are the national security agency (NSA) in the United States and the government communications headquarters (GCHQ) in the United Kingdom.

Encrypting a message involves two ingredients, an algorithm and a key. There are many known encryption algorithms, but the details of each depend on the choice of a key. Perhaps the simplest example of an encryption algorithm is letter shifting. The algorithm is: A message is encrypted by replacing each letter with the letter located $n$ positions ahead of it (cyclically) in the alphabet. The key is the value of $n$. Here is an example for $n = 3$ (note how `Y` is replaced by `A`).

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ␣
DEFGHIJKLMNOPQRSTUVWXYZ␣ABC
```

The top line is the *plain alphabet* and the bottom line is the *cipher alphabet*.

Such simple shifting is called the Caesar cipher, because it is first described in Julius Caesar's book *Gallic Wars*. It is an example of a *substitution* algorithm, in which each letter is substituted by a different letter (or sometimes by itself). Most encryption algorithms are based on some type of substitution. A simple example of a substitution algorithm that can also be made very secure is the *book cipher*. A book is chosen and a page number is selected at random. The words on the page are $^{1}$numbered $^{2}$and $^{3}$a $^{4}$table $^{5}$is $^{6}$prepared, $^{7}$with $^{8}$the $^{9}$first $^{10}$letter $^{11}$of $^{12}$each $^{13}$word $^{14}$and $^{15}$the $^{16}$word's $^{17}$number. $^{18}$This $^{19}$code $^{20}$table $^{21}$is $^{22}$later $^{23}$used $^{24}$to $^{25}$encrypt $^{26}$messages $^{27}$by $^{28}$replacing $^{29}$each $^{30}$letter $^{31}$of $^{32}$the $^{33}$message $^{34}$with $^{35}$a $^{36}$number $^{37}$from $^{38}$the $^{39}$table. For example, the message `NOT NOW` may be encoded as 36|31|20|17|11|13 (but may also be encoded differently). If the messages are short, and if a different page is used for each message, then this simple code is very secure (but the various page numbers have to be agreed upon in advance, which makes this method impractical in most situations).

Before we discuss substitution algorithms in detail, it is important to mention another large family of encryption algorithms, namely *transposition* algorithms. In such an algorithm, the original message is replaced by a permutation of itself. If the message is long, then each sentence is replaced by a permutation of itself. The number of permutations of $n$ objects is $n!$, a number that grows much faster than $n$. However, the permutation being used has to be chosen such that the receiver will be able to decipher the message. A simple example of such a method is to break a message up into two strings, one with the odd-numbered letters of the message and the other with the even-numbered letters, then concatenate the two strings. For example:

```
WAIT␣FOR␣ME␣AT␣MIDNIGHT
W I ␣ O ␣ E A ␣ I N G T
 A T F R M ␣ T M D I H
WI␣O␣EA␣INGTATFRM␣TMDIH
```

This method can be extended by selecting a key $n$, breaking the message up into $n$ strings (where the first string contains letters 1, $n+1$, $2n+1,\ldots$), and concatenating them.

### 3.16.1 Kerckhoffs' principle

Back to substitution algorithms. An important principle in cryptography, due to the Dutch linguist Auguste Kerckhoffs von Nieuwenhoff, states that the security of an encrypted message must depend on keeping the key secret. It must not depend on keeping the encrypting algorithm secret. This principle is widely accepted and it implies that there must be many possible keys to an algorithm. The Caesar algorithm, for example, is very insecure because it is limited to the 26 keys 1 to 26. Shifting the 27-symbol sequence (26 letters and a space) 27 positions returns it to its original position, and shifting it $27+n$ positions is identical to shifting it just $n$ positions.

> Kerckhoffs' principle
>
> One should assume that the method used to encipher data is known to the opponent, and that security must lie in the choice of key. This does not necessarily imply that the method should be public, just that it is considered public during its creation.
>
> —Auguste Kerckhoffs

### 3.16.2 Monoalphabetic Substitution Ciphers

A code where each symbol is replaced by another symbol, where the replacement does not vary, is called a *monoalphabetic substitution* code.

A more secure monoalphabetic substitution algorithm matches the plain alphabet with a permutation of itself, a permutation determined by a key. (Notice that this is a substitution algorithm, because it replaces the plain alphabet, not the encrypted message, with a permutation.) Here is a simple algorithm of this type. Start with a key that's a string from the alphabet, say, `LAND␣AND␣TREE`. Remove duplicate letters to obtain the key phrase `LAND␣TRE`. Use this as the start of the cipher alphabet, and append the rest of the plain alphabet in its original order, starting where the key phrase ends (i.e., with the letter `F`). The result is

| | |
|---|---|
| Plain alphabet | `ABCDEFGHIJKLMNOPQRSTUVWXYZ␣` |
| Cipher alphabet | `LAND␣TREFGHIJKMOPQSUVWXYZBC` |

An alternative way to apply the key is to count its length (8 in our example), to place it under the plain alphabet starting at position 8 (i.e., starting under the `I`) and to follow with the rest of the alphabet starting from `A` (in our example, starting from `B`, since `A` is part of the key).

| | |
|---|---|
| Plain alphabet | `ABCDEFGHIJKLMNOPQRSTUVWXYZ␣` |
| Cipher alphabet | `QSUVWXYZLAND␣TREBCFGHIJKMOP` |

A relatively short key is easy to memorize and can produce good substitution (i.e., the chance that a letter would replace itself is negligible). Monoalphabetic substitution codes were extensively used during the first millennium A.D. but fell victims to the development of statistics and concepts of probability. Breaking such a code is easy and does not require checking many alternatives. It is based on the fact that in any

language, the various letters appear in texts with different probabilities (we say that the *distribution* of letters is nonuniform). Some letters are common while others are rare. The most common letters in English, for example, are `E`, `T`, and `A` (if a blank space is also considered a letter, then it is the most common), and the least common are `Z` and `Q`. If a monoalphabetic substitution code replaces `E` with, say , `D`, then `D` should be the most common letter in the ciphertext.

The letter distribution in a language is computed by selecting documents that are considered typical in the language and counting the number of times each letter appears in those documents. For reliable results, the total number of letters in all the documents should be in the hundreds of thousands. Table 3.27 lists the letter distribution computed from the (approximately) 708,000 letters constituting a previous book by this author. Any single document may have letter distribution very different from the standard. In a mathematical text, the words **`quarter`**, **`quadratic`**, and **`quadrature`** may be common, increasing the frequencies of `Q` and `U`, while a text on the effects of ozone on zebras' laziness in Zaire may have unusually many occurrences of `Z`. Also, any short text may have letter distribution sufficiently different from the standard to defy simple analysis. The short story *The Gold Bug* by Edgar Allan Poe is an early example of breaking a monoalphabetic substitution code.

**The Gold Bug**

Here, then, we have, in the very beginning, the groundwork for something more than a mere guess. The general use which may be made of the table is obvious—but, in this particular cipher, we shall only very partially require its aid. As our predominant character is 8, we will commence by assuming it as the "e" of the natural alphabet. To verify the supposition, let us observe if the 8 be seen often in couples—for "e" is doubled with great frequency in English—in such words, for example, as "meet," "fleet," "speed," "seen," "been," "agree," etc. In the present instance we see it doubled no less than five times, although the cryptograph is brief.

—Edgar Allan Poe.

Another statistical property of letters in a language is the relation between pairs (and even triplets) of letters (digrams and trigrams). In English, a `T` is very often followed by `H` (as in **`this`**, **`that`**, **`then`**, and **`them`**) and a `Q` is almost always followed by a `U`. Thus, *digram frequencies* can also be used in deciphering a stubborn monoalphabetic substitution code.

Not everything that counts can be counted, and not everything that can be counted counts.

—Albert Einstein

### 3.16.3 Polyalphabetic Substitution Ciphers

Perhaps the simplest way to extend the basic monoalphabetic substitution codes is to define two cipher alphabets and use them alternately. In the example below, the first letter `S` of the plain message `SURRENDER` is replaced by `D` from cipher alphabet 1, but the second letter `U` is replaced by `D` from cipher alphabet 2. The cipher letter `D` now stands for two plaintext letters, thereby making the encrypted message much harder to break. This cipher was developed by the Renaissance figure Leon Battista Alberti.

```
Plain alphabet       ABCDEFGHIJKLMNOPQRSTUVWXYZ␣
Cipher alphabet 1    QWERTYUIOPLKJH␣GFSDAZXCVBNM
Cipher alphabet 2    LASKM␣JFHGQPWOEIRUTYDZXNCBV
```

A man can do all things if he but wills them.

—Leon Battista Alberti

| Letter | Freq. | Prob. | Letter | Freq. | Prob. |
|---|---|---|---|---|---|
| A | 51060 | 0.0721 | E | 86744 | 0.1224 |
| B | 17023 | 0.0240 | T | 64364 | 0.0908 |
| C | 27937 | 0.0394 | I | 55187 | 0.0779 |
| D | 26336 | 0.0372 | S | 51576 | 0.0728 |
| E | 86744 | 0.1224 | A | 51060 | 0.0721 |
| F | 19302 | 0.0272 | O | 48277 | 0.0681 |
| G | 12640 | 0.0178 | N | 45212 | 0.0638 |
| H | 31853 | 0.0449 | R | 45204 | 0.0638 |
| I | 55187 | 0.0779 | H | 31853 | 0.0449 |
| J | 923 | 0.0013 | L | 30201 | 0.0426 |
| K | 3812 | 0.0054 | C | 27937 | 0.0394 |
| L | 30201 | 0.0426 | D | 26336 | 0.0372 |
| M | 20002 | 0.0282 | P | 20572 | 0.0290 |
| N | 45212 | 0.0638 | M | 20002 | 0.0282 |
| O | 48277 | 0.0681 | F | 19302 | 0.0272 |
| P | 20572 | 0.0290 | B | 17023 | 0.0240 |
| Q | 1611 | 0.0023 | U | 16687 | 0.0235 |
| R | 45204 | 0.0638 | G | 12640 | 0.0178 |
| S | 51576 | 0.0728 | W | 9244 | 0.0130 |
| T | 64364 | 0.0908 | Y | 8953 | 0.0126 |
| U | 16687 | 0.0235 | V | 6640 | 0.0094 |
| V | 6640 | 0.0094 | X | 5465 | 0.0077 |
| W | 9244 | 0.0130 | K | 3812 | 0.0054 |
| X | 5465 | 0.0077 | Z | 1847 | 0.0026 |
| Y | 8953 | 0.0126 | Q | 1611 | 0.0023 |
| Z | 1847 | 0.0026 | J | 923 | 0.0013 |

Frequencies and probabilities of the 26 letters in a prepublication version of another book by this author, containing 708,672 letters (upper- and lowercase) comprising approximately 145,000 words.

Most, but not all, experts agree that the most common letters in English, in order, are `ETAOINSHRDLU` (normally written as two separate words `ETAOIN SHRDLU`). However, see [Fang 66] for a different viewpoint. The most common digrams (2-letter combinations) are TH, HE, AN, IN, HA, OR, ND, RE, ER, ET, EA, and OU. The most frequently appearing letters *beginning* words are S, P, C, and the most frequent final letters are E, Y, S.

**Table 3.28:** Probabilities of English Letters.

Alberti's cipher algorithm was extended by several people, culminating in the work of Blaise de Vigenère, who in 1586 published the *polyalphabetic substitution* cipher named after him. The Vigenère system uses 26 cipher alphabets (including just letters, not a space), each shifted one position relative to its predecessor. Figure 3.28 shows the entire Vigenère square, generally regarded as the most perfect of the simpler polyalphabetical substitution ciphers.

| Plain | ABCDEFGHIJKLMNOPQRSTUVWXYZ |
|---|---|
| 1 | BCDEFGHIJKLMNOPQRSTUVWXYZA |
| 2 | CDEFGHIJKLMNOPQRSTUVWXYZAB |
| 3 | DEFGHIJKLMNOPQRSTUVWXYZABC |
| 4 | EFGHIJKLMNOPQRSTUVWXYZABCD |
| 5 | FGHIJKLMNOPQRSTUVWXYZABCDE |
| 6 | GHIJKLMNOPQRSTUVWXYZABCDEF |
| 7 | HIJKLMNOPQRSTUVWXYZABCDEFG |
| 8 | IJKLMNOPQRSTUVWXYZABCDEFGH |
| 9 | JKLMNOPQRSTUVWXYZABCDEFGHI |
| 10 | KLMNOPQRSTUVWXYZABCDEFGHIJ |
| 11 | LMNOPQRSTUVWXYZABCDEFGHIJK |
| 12 | MNOPQRSTUVWXYZABCDEFGHIJKL |
| 13 | NOPQRSTUVWXYZABCDEFGHIJKLM |
| 14 | OPQRSTUVWXYZABCDEFGHIJKLMN |
| 15 | PQRSTUVWXYZABCDEFGHIJKLMNO |
| 16 | QRSTUVWXYZABCDEFGHIJKLMNOP |
| 17 | RSTUVWXYZABCDEFGHIJKLMNOPQ |
| 18 | STUVWXYZABCDEFGHIJKLMNOPQR |
| 19 | TUVWXYZABCDEFGHIJKLMNOPQRS |
| 20 | UVWXYZABCDEFGHIJKLMNOPQRST |
| 21 | VWXYZABCDEFGHIJKLMNOPQRSTU |
| 22 | WXYZABCDEFGHIJKLMNOPQRSTUV |
| 23 | XYZABCDEFGHIJKLMNOPQRSTUVW |
| 24 | YZABCDEFGHIJKLMNOPQRSTUVWX |
| 25 | ZABCDEFGHIJKLMNOPQRSTUVWXY |
| 26 | ABCDEFGHIJKLMNOPQRSTUVWXYZ |

**Figure 3.29:** The Vigenère Cipher System.

A message is encrypted by replacing each of its letters using a different row, selecting rows by means of a key. The key (a string of letters with no spaces) is written above the Vigenère square and is repeated as many times as necessary to produce a row of 26 letters. For example, the key `LAND AND TREE` produces the following

| | |
|---|---|
| Key | LANDANDTREELANDANDTREELAND |
| Plain alphabet | ABCDEFGHIJKLMNOPQRSTUVWXYZ |
| Cipher alphabet 1 | BCDEFGHIJKLMNOPQRSTUVWXYZA |
| Cipher alphabet 2 | CDEFGHIJKLMNOPQRSTUVWXYZAB |
| 24 more lines | ... ... ... |

Each letter of the original message is replaced using the row indicated by the next letter of the key. Thus, the first letter of the message is replaced using the row that starts with `L` (row 11). If the first letter is, say, `E`, it is replaced by `P`. The second letter is replaced by a letter from the row indicated by the `A` of the key (row 26), and so on.

It is obvious that the strength of this simple algorithm lies in the key. A sufficiently long key, with many different letters uses most of the rows in the Vigenère square and results in a secure encrypted string. The number of possible keys is enormous, since a key can be between 1 and 26 letters long. The knowledge

that a message has been encrypted with this method helps little in breaking it. Simple frequency analysis does not work, since a letter in the encrypted message, such as S, is replaced by different letters each time it occurs in the original text.

The Vigenère cipher, as all other polyalphabetic substitution methods, is much more secure than any monoalphabetic substitution method, and was not broken until the 19th century, first in the 1850s by Charles Babbage who never published this achievement (Babbage is well-known for his attempts to construct mechanical computers, see Appendix A), then, in the 1860s, independently by Friedrich W. Kasiski, a retired officer in the Prussian army.

The key to breaking a polyalphabetic substitution cipher is the key (no pun intended) used in the encryption. The keys commonly used are meaningful words and phrases and they tend to be short. It has been proved that the use of a random key can lead to absolute security, provided that the key is used just once. Thus, polyalphabetic substitution ciphers can be the ultimate secret codes provided that (1) random keys are used and (2) a key is never used more than once. Such a *one-time pad* cipher is, of course, not suitable for general use. It is impractical and unsafe to distribute pads of random keys to many army units or to many corporate offices. However, for certain special applications, a one-time pad cipher is practical and absolutely secure. An example is private communication between world leaders.

In spite of its obvious advantage, the Vigenère method was not commonly used after its publication because of its complexity. Encrypting a message with this method (as with any other polyalphabetic substitution method) by hand is slow, tedious, and error prone. It should be done by machine, and such machines have been developed since the 1500s. A historically notable example is a device with 36 rotating cylinders, designed and built in the 1790s by Thomas Jefferson, third President of the United States.

A better-known code machine is the German Enigma, developed in the 1910s by the German inventor Arthur Scherbius. The Enigma was an electric machine with a 26-key keyboard to enter the plaintext and 26 lamps to indicate the ciphertext. The operator would press a key and a lamp would light up, indicating a letter. The Morse code for that letter would then be sent by the operator on the military telegraph or radio. Between the keyboard and the lamps, the Enigma had 3 scrambling rotors (with a 4th one added later) that changed the substitution rule each time a key was pressed. The Enigma was used extensively by the German army during World War II, but its code was broken by British code breakers working at Bletchley Park. A detailed description of the Enigma and the story of breaking its code can be found in [Singh 99].

In the absence of encrypting machines, past cryptographers that needed a cipher more secure than simple monoalphabetic substitution started looking for methods simpler than (but also weaker than) polyalphabetic substitutions.

### 3.16.4 Homophonic Substitution Ciphers

A satisfactory compromise was found in the family of *homophonic substitution* ciphers. A typical algorithm in this family replaces each letter by a number. Table 3.27 shows that the probability of the letter B in English is about 2% (i.e., B accounts for roughly 2% of all letters in English texts), so two numbers are assigned to encode B. Similarly, the probability of D is close to 3%, so three numbers are assigned to encode D. The result is that 100 numbers are assigned, each counting for roughly 1% of the encrypted text. A possible assignment of 100 numbers to the 26 letters is shown in Table 3.29. (The reason for the term *homophonic*, meaning "same sound," is that we can consider each of the numbers assigned to D the sound of that letter.) The encryption method is simple. Each time the letter D is found in the plaintext, one of the three numbers is selected at random and is appended to the ciphertext. Thus, the message TENNESSEE may be encrypted to 66|01|26|77|63|21|99|32|19.

The advantage of the homophonic substitution cipher is that the distribution of the 100 numbers in the ciphertext of a long message is uniform (i.e. each of the 100 numbers occurs roughly the same number of times). It seems that a homophonic substitution cipher cannot be broken by frequency analysis, but it is still vulnerable to digram analysis. Breaking a ciphertext, once it is known (or suspected) that it has been encrypted with a homophonic substitution code, can start, for example, with the letter Q. Since this letter is rare, there is a good chance that only one number was assigned to it. We know that Q is almost always followed by U, and that U appears 2–3% of the time, so it has been assigned 2 or 3 numbers. Thus, we can look for a number that's always followed by one of the same 2 or 3 numbers.

There are other ways to construct codes that are easy to use and are more secure than monoalphabetic

| a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 90 | 70 | 62 | 18 | 76 | 89 | 61 | 83 | 71 | 44 | 91 | 94 | 52 | 81 | 67 | 43 | 31 | 46 | 13 | 72 | 60 | 87 | 07 | 28 | 79 | 51 |
| 03 | 84 | 82 | 14 | 32 | 53 | 64 | 37 | 27 | | | 36 | 96 | 77 | 00 | 22 | | 98 | 93 | 17 | 65 | | 45 | | 38 | |
| 23 | | 75 | 54 | 01 | | | 20 | 12 | | | 69 | | 08 | 97 | | | 57 | 10 | 80 | 73 | | | | | |
| 09 | | | 95 | 63 | | | 74 | 56 | | | 58 | | 26 | 29 | | | 92 | 21 | 66 | | | | | | |
| 55 | | | | 33 | | | 25 | 05 | | | | | 48 | 11 | | | 02 | 39 | 59 | | | | | | |
| 40 | | | | 68 | | | 15 | 41 | | | | | 34 | 85 | | | 78 | 99 | 86 | | | | | | |
| 49 | | | | 47 | | | | | | | | | | 16 | | | | | 04 | | | | | | |
| 88 | | | | 06 | | | | | | | | | | | | | | | 35 | | | | | | |
| | | | | 19 | | | | | | | | | | | | | | | 50 | | | | | | |
| | | | | 24 | | | | | | | | | | | | | | | | | | | | | |
| | | | | 30 | | | | | | | | | | | | | | | | | | | | | |
| | | | | 42 | | | | | | | | | | | | | | | | | | | | | |

**Table 3.30:** A Homophonic Substitution Cipher.

substitution codes. Numbers can be assigned to the individual letters and also to many common syllables. When a common syllable is encountered in the plaintext, it is replaced by its code number. When a syllable is found that has no code assigned, its letters are encrypted individually by replacing each letter with its code. Certain codes may be assigned as traps, to throw any would-be cryptanalyst off the track. Such a special code may indicate, for example, that the following (or the preceding) number is spurious and should be ignored.

## 3.17 Transposition Ciphers

In a substitution cipher, each letter in the plaintext is replaced by another letter. In a transposition cipher, the entire plaintext is replaced by a permutation of itself. If the plaintext is long, then each sentence is replaced by a permutation of itself. The number of permutations of $n$ objects is $n!$, a number that grows much faster than $n$. However, the permutation being used has to be chosen such that the receiver will be able to decipher the message. [Barker 92] describes a single-column transposition cipher.

It is important to understand that in a transposition cipher the plainletters are not replaced. An `E` in the plaintext does not become another letter in the ciphertext, it is just moved from its original position to another place. Frequency analysis of the ciphertext will reveal the normal letter frequencies, thus serving as a hint to the cryptanalyst that a transposition cipher was used. A digram frequency analysis should then be performed, to draw detailed conclusions about the cipher. Common digrams, such as `th` and `an` would be torn apart by a substitution cipher, reinforcing the suspicion that this type of cipher was used.

We start with a few simple transposition ciphers. These are easy to break but they indicate what can be done with this type of cipher.

1. Break the plaintext into fixed-size blocks and apply the same permutation to each block. As an example, the plaintext `ABCDE`... can be broken into blocks of four letters each and one of the $4! = 24$ permutations applied to each block. The ciphertext can be generated by either collecting the four letters of each block, or by collecting the first letter of each block, then the second letter, and so on. Figure 3.30 shows a graphic example of some 4-letter blocks. The resulting ciphertext can be either `ACBD EGFH IKJL MONP`... or `AEIM`... `CGKO`... `BFJN`... and `DHLP`....

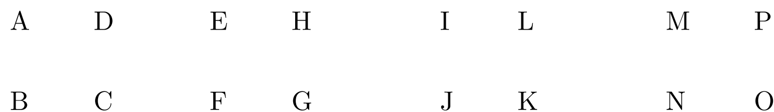



**Figure 3.31:** 2×2 Blocks For a Transposition Cipher.

2. A square or a rectangle with $n$ cells is filled with the integers from 1 to $n$ in some order. The first $n$ plainletters are placed in the cells according to these integers and the ciphertext is generated by scanning the rectangle in some order (possibly rowwise or columnwise) and collecting letters. An alternative is to place

the plainletters in the rectangle in rowwise or columnwise order, and generate the ciphertext by scanning the rectangle according to the integers, collecting letters. If the plaintext is longer than $n$ letters, the process is repeated. Some null letters, such as `X` or `Q`, may have to be appended to the last block. Figure 3.31a–c shows three $8\times 8$ squares. The first with integers arranged in a knight's tour, the second a magic square, and the third a self-complementary magic square (where if the integer $i$ is replaced by $65-i$, the result is the same square, rotated).

| 47 | 10 | 23 | 64 | 49 | 2 | 59 | 6 |
|---|---|---|---|---|---|---|---|
| 22 | 63 | 48 | 9 | 60 | 5 | 50 | 3 |
| 11 | 46 | 61 | 24 | 1 | 52 | 7 | 58 |
| 62 | 21 | 12 | 45 | 8 | 57 | 4 | 51 |
| 19 | 36 | 25 | 40 | 13 | 44 | 53 | 30 |
| 26 | 39 | 20 | 33 | 56 | 29 | 14 | 43 |
| 35 | 18 | 37 | 28 | 41 | 16 | 31 | 54 |
| 38 | 27 | 34 | 17 | 32 | 55 | 42 | 15 |

(a)

| 52 | 61 | 4 | 3 | 20 | 29 | 36 | 45 |
|---|---|---|---|---|---|---|---|
| 14 | 3 | 62 | 51 | 46 | 35 | 30 | 19 |
| 53 | 60 | 5 | 12 | 21 | 28 | 37 | 44 |
| 11 | 6 | 59 | 54 | 43 | 38 | 27 | 22 |
| 55 | 58 | 7 | 10 | 23 | 26 | 39 | 42 |
| 9 | 8 | 57 | 56 | 41 | 40 | 25 | 24 |
| 50 | 63 | 2 | 15 | 18 | 31 | 34 | 47 |
| 16 | 1 | 64 | 49 | 48 | 33 | 32 | 17 |

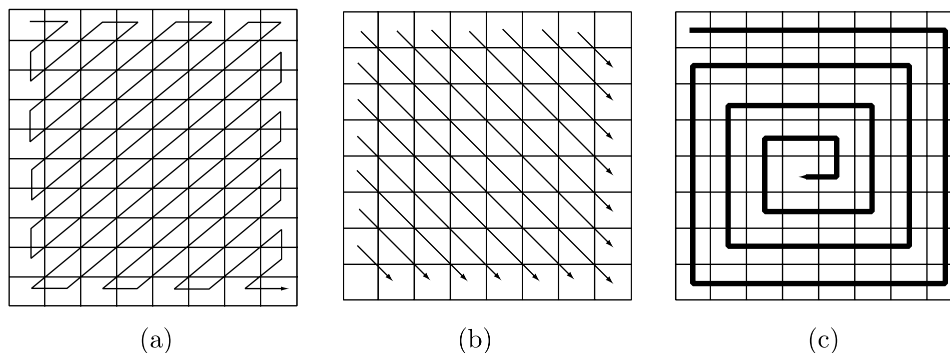
(b)

| 1 | 62 | 5 | 59 | 58 | 12 | 61 | 2 |
|---|---|---|---|---|---|---|---|
| 57 | 14 | 50 | 48 | 19 | 45 | 18 | 9 |
| 10 | 27 | 34 | 25 | 41 | 36 | 33 | 54 |
| 49 | 30 | 26 | 23 | 37 | 22 | 21 | 52 |
| 13 | 44 | 43 | 28 | 42 | 39 | 35 | 16 |
| 11 | 32 | 29 | 24 | 40 | 31 | 38 | 55 |
| 56 | 47 | 20 | 46 | 17 | 15 | 51 | 8 |
| 63 | 4 | 53 | 7 | 6 | 60 | 3 | 64 |

(c)

**Figure 3.32:** Three $8\times 8$ Squares For Transposition Ciphers.

In addition to row and column scanning, such a square can be scanned in zigzag, by diagonals, or in a spiral (Figure 3.32a–c, respectively).



(a) (b) (c)

**Figure 3.33:** Three Ways To Scan An $8\times 8$ Square.

3. The plaintext is arranged in a zigzagging rail fence whose height is the key of the cipher. Assuming the plaintext `ABCDEFGHIJK` and a key of 3, the rail fence is

```
A       E       I
  B   D   F   H   J
    C       G       K
```

The fence is scanned row by row to form the ciphertext `AEI.BDGHJ.CGK`, where the dots are special symbols needed for decryption. This is an example of an irregular transposition cipher.

4. The plaintext is divided into groups of $n\times n$ letters and each group is placed into an $n\times n$ square as shown in Figure 3.33, which illustrates the plaintext `PLAINTEXT IN A SQUARE`. The ciphertext is obtained by scanning the square in rows `PLNIR IATNE TXEAX AUQSX XXXXX`.

5. Start with a rectangle of $m$ rows and $n$ columns. In each column, select half the rows at random. This creates a template of $m\times n$ cells, half of which have been selected. All authorized users of the cipher must have this template. The plaintext is written in rows and is collected by columns to form the ciphertext. Figure 3.34 shows an example where the plaintext is the letters `A` through `R` and the ciphertext is `CFLGK MADPH NQEIR BJO`.

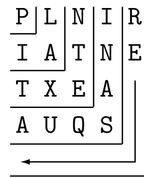



**Figure 3.34:** Transposition Cipher in a 5×5 Square.

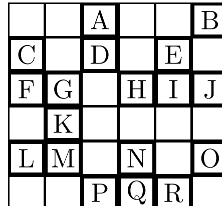



**Figure 3.35:** A 6×6 Template With Plaintext.

6. Select a key $k = k_1k_2\ldots k_n$. Write the plaintext in an $n$-row rectangle row by row. On row $i$, start the text at position $k_i$. The ciphertext is generated by collecting the letters in columns from left to right. Figure 3.35 is an example. It shows how the plaintext `ABCD...X` is enciphered into `NAOBG PCHQU DIRVE JLSWF KMTX` by means of the key 23614. This type of transposition cipher is irregular and thus somewhat more secure, but involves a long key.

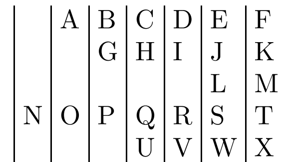



**Figure 3.36:** An Irregular Rectangle With Plaintext.

7. A simple example of a substitution cipher is to reverse every string of three letters, while keeping the positions of all blank spaces. For example, the message `ATTACK AT DAWN` becomes `TTAKCA DA TNWA`. Another elementary transposition is to break a message up into two strings, one with the odd-numbered letters of the message and the other with the even-numbered letters, then concatenate the two strings. For example:

```
WAIT␣FOR␣ME␣AT␣MIDNIGHT
W I ␣ O ␣ E A ␣ I N G T
 A T F R M ␣ T M D I H
WI␣O␣EA␣INGTATFRM␣TMDIH
```

This method can be extended by selecting a key $n$, breaking the message up into $n$ strings (where the first string contains letters 1, $n+1$, $2n+1,\ldots$), and concatenating them.

8. An anagram is a rearrangement of the letters of a word or a phrase. Thus, for example, `red code` is an anagram of `decoder` and `strict union` is an anagram of `instruction`. It is possible to encrypt a message by scrambling its letters to generate an anagram (meaningful or not). One simple way to create nonsense anagrams is to write the letters of a message in alphabetical order. Thus `I came, I saw, I conquered` becomes `AACCDEEEIIIMNOQRSUW`. This kind of cipher is practically impossible for a receiver (even an authorized one) to decipher, but has its applications. In past centuries, scientists sometimes wanted to keep a discovery secret and at the same time keep their priority claim. Writing a report of the discovery in anagrams was a way to achieve both aims. No one could decipher the code, yet when someone else claimed to have made the same discovery, the original discoverer could always produce the original report in plaintext and prove that this text generates the published anagram.

The fundamental problem of transposition ciphers is how to specify a permutation by means of a short, easy to remember key. A detailed desciption of a permutation can be as long as the message itself, or even longer. It should contain instructions such as "swap positions 54 and 32, move position 27 to position 98," and so on. The following sections discuss several ways to implement transposition ciphers.

> As he sat on the arm of a leather-covered arm chair, putting his face through all its permutations of loathing, the whole household seemed to spring into activity around him.
>
> —Kingsley Amis, *Lucky Jim*, 1953

### 3.18 Transposition by Turning Template

The number of permutations of 36 objects is $36! \approx 3.72 \cdot 10^{41}$; an immense number. Many, although not all, of these permutations can be specified by means of a small, $6\times 6$ template with some holes cut in it. This method was developed in 1881 by the Austrian Colonel Eduard Fleissner von Wostrowitz, and is easy to describe and to implement. A square piece of cardboard is prepared, whose size equals the width of six letters, and nine square holes are cut in it (Figure 3.36a). The first 36 letters of the plaintext are arranged in a $6\times 6$ square and the template placed over it (Figure 3.37a). The nine letters seen through the holes are collected by rows and become the first nine letters of the ciphertext. The template is then rotated 90° (clockwise or counterclockwise, but encoder and decoder have to agree on the direction and be consistent about it) and the nine letters now showing through the holes are appended to the ciphertext. The holes are arranged such that after four such readings, all 36 letters under the table have been seen through the holes and were included in the ciphertext. The process is repeated for segments of 36 plaintext letters and some random letters may have to be added to the last segment to complete its length to 36.

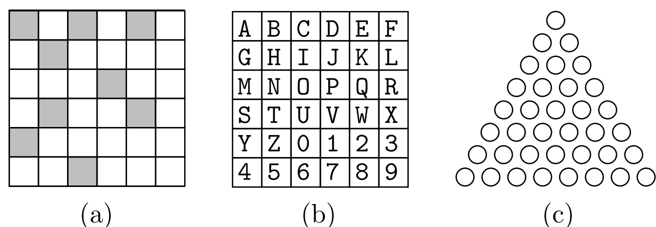



**Figure 3.37:** Two Turning Templates.

Naturally, the opposite process is also possible. The first nine letters of the plaintext are written through the template holes, the template is rotated, and the process repeated three more times, for a total of 36 letters arranged in a $6\times 6$ square. The ciphertext is prepared by reading the 36 letters row by row.

The template is normally a square where the number of cells is divisible by 4. However, the template can also be an equilateral triangle where the number of cells is divisible by 3 (Figure 3.36b) or a square with an odd number of cells such as $5\times 5$), in which case the central row and column can simply be ignored.

Decrypting is the reverse of encrypting, but breaking this code, even though not especially hard, is tricky and requires intuition. The length of the ciphertext provides a clue to the size of the template (in the case of a 6×6 template, the length of the ciphertext is a multiple of 36). Once the codebreaker knows (or guesses or suspects) the size of the template, further clues can be found from the individual letters of common digrams. If it is suspected that the template size is 6×6, the ciphertext is broken up into 36-letter segments, and each segent written as a 6×6 square. Any occurrences of the letters `T` and `H` in a square could be due to the digram `TH` or the trigram `THE` that were broken apart by the encryption. If the codebreaker discovers certain hole positions that would unite the `T` and `H` in one square, then other squares (i.e., other ciphertext segments) may be tried with these holes, to check whether their `T` and `H` also unite. This is a tedious process, but it can be automated to some degree. A computer program can try many hole combinations and display several squares of ciphertext partly decrypted with those holes. The user may reject some hole combinations and ask for further tests (perhaps with another common digram) with other combinations.

It is very easy to determine hole positions in such a square template. Figure 3.37a shows a 6×6 template divided into four 3×3 templates rotated with respect to each other. The 9 cells of each small template are numbered in the same way. The rule for selecting cells is: If cell 3 is selected for a hole in one of the 3×3 templates, then cell 3 should not be a hole in any of the other three templates. We start by selecting cell 1 in one of the four templates (there are four choices for this). In the next step, we select cell 2 in one of the four templates (again four choices). This is repeated nine times, so the total number of possible hole configurations is $4^9 = 262{,}144$. This, of course is much smaller than the total number 36! of permutations of 36 letters.

▸ **Exercise 3.16:** How many hole configurations are possible in an 8×8 template?

We illustrate this method with the plaintext `happy families are all alike every unhappy family is unhappy in its own way`. The first 36 letters are arranged in a 6×6 square

```
happyf
amilie
sareal
lalike
everyu
nhappy
```

Figure 3.37b shows how the same template is placed over this square four times while rotating it each time. The resulting cipher is `hpymeakea afiislieu peaalyhpy alrlevrnp`

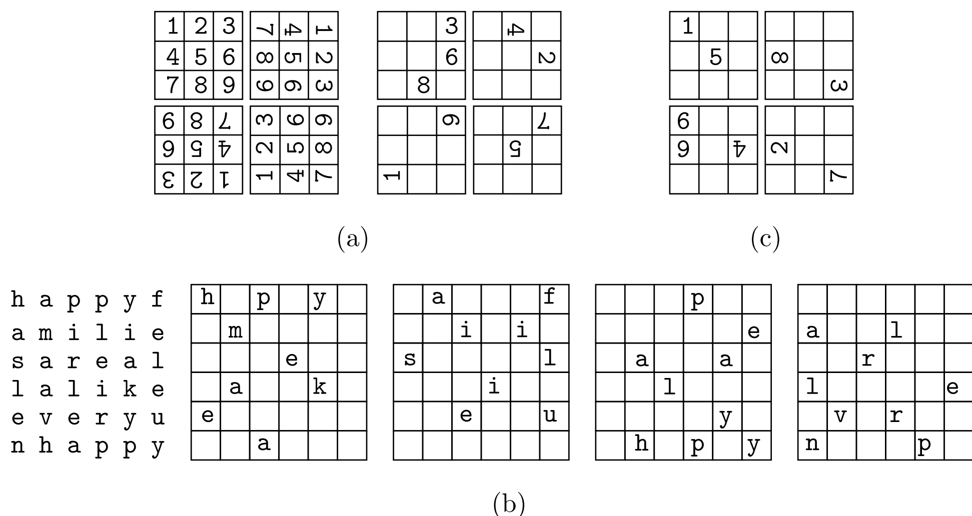



**Figure 3.38:** Turning Template Method.

It is also possible to use a key to determine the hole positions. The advantage is that the users don't have to keep the actual template (which may be hard to destroy in a sudden emergency). Consider, for example, the key `FAKE CIPHER`. The first key letter, `F` is the 6th letter of the alphabet, which implies that the square should be of size 6×6. Such a square has 36 cells, so $36/4 = 9$ hole positions are needed, where each position is determined by one of the key's letters. The key should therefore be nine letters long. If it is shorter, it is repeated as needed, and if it is longer, only the first nine letters are considered. The letters are numbered according to their alphabetic order. `A` is numbered 1. The next key letter in alphabetical order is `C`, so it is numbered 2. The next one is `E`, so its two occurrences are numbered 3 and 4. Thus, the key becomes the nine digits 518327964. These are divided into the four groups 51, 83, 27, and 964 for the four quadrants. The first group corresponds to holes 5 and 1 in the first quadrant. The second group corresponds to holes 8 and 3 in the second quadrant, and so on (Figure 3.37c).

This method can be made more secure by double encryption. Once the ciphertext is generated, it is encrypted, by means of a mirror image of the template, or even by a completely different template. However,

even with this enhancement, the turning template method is unsafe, and has the added problem of keeping the template from falling into the wrong hands.

## 3.19 Columnar Transposition Cipher

A template requires constant protection, so it is preferable to specify a permutation by means of a key. Here is a simple way of doing this. To specify a permutation of, say, 15 letters, we start with a key whose length is at least 15 letters (not counting spaces), eliminate the spaces, and leave the first 10 letters. Thus, if the key is the phrase `TRANSPOSITION CIPHERS`, we first generate the 15-letter string `TRANSPOSITIONCI`. The 15 letters are then numbered as follows: The `A` is numbered 1. There are no `B`'s, so the `C` is numbered 2. The next letter (alphabetically) is `I`. There are three `I`'s, so they are numbered 3, 4, and 5, and so on, to obtain

| T | R | A | N | S | P | O | S | I | T | I | O | N | C | I |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 14 | 11 | 1 | 6 | 12 | 10 | 8 | 13 | 3 | 15 | 4 | 9 | 7 | 2 | 5 |

The sequence 14, 11, 1, 6, 12, 10, 8, 13, 3, 15, 4, 9, 7, 2, and 5 specifies the permutation. After the permutation, the third plainletter should be in position 1, the 14th plainletter should be in position 2, and so on. The plaintext is now arranged as a rectangle with 15 columns, the columns switched according to the permutation, and the ciphertext collected column by column. Figure 3.38 shows an example.

| T | R | A | N | S | P | O | S | I | T | I | O | N | C | I |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 14 | 11 | 1 | 6 | 12 | 10 | 8 | 13 | 3 | 15 | 4 | 9 | 7 | 2 | 5 |
| h | a | p | p | y | f | a | m | i | l | i | e | s | a | r |
| e | a | l | l | a | l | i | k | e | e | v | e | r | y | u |
| n | h | a | p | p | y | f | a | m | i | l | y | i | s | u |
| n | h | a | p | p | y | i | n | i | t | s | o | w | n | w |
| a | y | e | v | e | r | y | t | h | i | n | g | w | a | s |
| i | n | c | o | n | f | u | s | i | o | n | i | n | t | h |
| e | o | b | l | o | n | s | k | y | s | h | o | u | s | e |
| t | h | e | w | i | f | e | h | a | d | d | i | s | c | o |
| v | e | r | e | d | t | h | a | t | t | h | e | | | |
| p | a | i | i | r | p | s | a | e | f | a | y | m | h | l |
| l | y | e | v | u | l | r | i | e | l | a | a | k | e | e |
| a | s | m | l | u | p | i | f | y | y | h | p | a | n | i |
| a | n | i | s | w | p | w | i | o | y | h | p | n | n | t |
| e | a | h | n | s | v | w | y | g | r | y | e | t | a | i |
| c | t | i | n | h | o | n | u | i | f | n | n | s | i | o |
| b | s | y | h | e | l | u | s | o | n | o | o | k | e | s |
| e | c | a | d | o | w | s | e | i | f | h | i | h | t | d |
| r | | t | h | | e | | h | e | t | e | d | a | v | t |

```
PLAAECBER AYSNATSC IEMIHIYAT IVLSNNHDH
RUUWSHEO PLPPVOLWE SRIWWNUS AIFIYUSEH
EEYOGIOIE FLYYRFNFT AAHHYNOHE YAPPENOID
MKANTSKHA HENNAIETV LEITIOSDT
```

**Figure 3.39:** Transposition Cipher With a Key.

This transposition cipher does not require a template, and the key is easy to memorize and replace, but the method isn't very secure. If the codebreaker can guess the number of columns (or try many guesses), then the permutation, or parts of it, can be guessed from the positions of the members of common digrams. Thus, if the codebreaker arranges the ciphertext in a rectangle of the right size, then columns can be moved around until all or most occurrences of `H` follow `T`'s. If the result starts making sense, more digrams can be tried.

As is true with many encryption methods, this cipher can be strengthened by performing double encryption. After the ciphertext is obtained by reading the 15 columns and collecting letters, it can be written in rows and reencrypted with the same key. An alternative is to reencrypt the ciphertext before it is read from the 15 columns. This is done using another key on the rows. There are 9 rows in our example and they can be swapped according to a secondary key.

Replacing a key or even two keys often is the key to security. This can easily be done once the authorized users agree (or are being told) how to do this. The users may agree on a book and on a formula. The formula converts a date to a page number in the book, and the key consists of the last letters of the first 15 lines on that page. Every day, all users apply the formula to the current date and construct a new key from the same page. A possible formula may start with a 6-digit date of the form $d_1d_2m_1m_2y_1y_2$ and compute

$$\big((50d_1 + 51d_2 + 52m_1 + 53m_2 + 54y_1 + 55y_2) \bmod k\big) + 1$$

where $k$ is the number of pages in the book. This formula (which is a weighted sum, see the similar formula for ISBN below) depends on all six digits of the date in such a way that changing any digit almost always changes the sum. The result of computing any integer modulo $k$ is a number in the range $[0, k-1]$, so adding 1 to it produces a valid page number.

The international standard book number (ISBN), is a unique number assigned to most book published. This number has four parts, a country code, a publisher code, a book number assigned by the publisher, and a check digit, for a total of 10 digits. For example, ISBN 0-387-95045-1 has country code 0, publisher code 387, book number 95045, and check digit 1. The check digit is computed by multiplying the leftmost digit by 10, the next digit by 9, and so on, up to ninth digit from the left, which is multiplied by 2. The products are then added, and the check digit is determined as the smallest integer that when added to the sum will make it a multiple of 11. The check digit is therefore in the range $[0, 10]$. If it happens to be 10, it is replaced by the Roman numeral X in order to make it a single symbol.

▸ **Exercise 3.17:** Assuming a 190-page book, compute the page number for today.

### 3.19.1 The Myszkowsky Cipher

This method, due to E. Myszkowsky, is a variation on the general columnar transposition cipher. The key is duplicated as many times as needed to bring it to the size of the plaintext. Each key letter is replaced by a number according to its alphabetic position, and the plainletters are collected by these numbers to form the ciphertext. The example shown here uses the key `QUALITY`, which is duplicated four times to cover the entire 28-letter plaintext `Come home immediately. All is lost`.

| Q | U | A | L | I | T | Y | Q | U | A | L | I | T | Y | Q | U | A | L | I | T | Y | Q | U | A | L | I | T | Y |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 13 | 21 | 1 | 9 | 5 | 17 | 25 | 14 | 22 | 2 | 10 | 6 | 18 | 26 | 15 | 23 | 3 | 11 | 7 | 19 | 27 | 16 | 24 | 4 | 12 | 8 | 20 | 28 |
| C | O | M | E | H | O | M | E | I | M | M | E | D | I | A | T | E | L | Y | A | L | L | I | S | L | O | S | T |

The ciphertext is `MMESH EYOEM LLCEA LODAS OITIM ILT`. Since the extended key contains duplicate letters, the original key may also contain duplicates, such as in the word `INSTINCT`.

### 3.19.2 The AMSCO Cipher

The AMSCO cipher, due to A. M. Scott, is a columnar transposition cipher where the plaintext is placed in the columns such that every other column receives a pair of plainletters. The example presented here uses the same key and plaintext as in Section 3.19.1.

| Q | U | A | L | I | T | Y |
|---|---|---|---|---|---|---|
| 4 | 6 | 1 | 3 | 2 | 5 | 7 |
| C | OM | E | HO | M | EI | M |
| ME | D | IA | T | EL | Y | AL |
| L | IS | L | OS | T | | |

The ciphertext is `EIALM ELTHO TOSCM ELEIY OMDIS MAL`.

### 3.19.3 Cryptography in the Computer Age

The discussion until now was based on letters and digits. With the development of the modern computer which uses binary numbers, secure codes had to be based on bits. If the plaintext is text, then it is represented internally in ASCII (or Unicode, see Appendix D), with 7 bits per character. Any transposition or substitution algorithms should be modified to operate on bits, which happens to be an easy task. The ASCII codes of the plaintext string `MASTER` are

1001101|1000001|1010011|1010100|1000101|1010010.

It is very easy to come up with simple permutations that will make a transposition cipher for such a string. Some possible permutations are: (1) Swap every pair of consecutive bits. (2) Reverse the seven bits of each code, then swap consecutive codes. (3) Perform a perfect shuffle as shown here

```
100110110000011010011|101010010001011010010
 1 0 0 1 1 0 1 1 0 0 0 0 0 1 1 0 1 0 0 1 1
  1 0 1 0 1 0 0 1 0 0 0 1 0 1 1 0 1 0 0 1 0

 110001101100101100000001001111001100001110
```

### 3.19.4 Stream Ciphers

The principle of a stream cipher is to create a string of bits, called the *keystream* and to exclusive-or (XOR) the bits of the plaintext with those of the keystream. The resulting bit string is the ciphertext. Decrypting is done by XORing the ciphertext with the same keystream. Thus, a stream cipher is based on the following property of the XOR operation (Table 2.6): If $B = A \oplus K$, then $A = B \oplus K$. As an example, the plaintext `MASTER` is enciphered with the keystream `KEY` (= 1001011|1000101|1011001).

| | |
|---|---|
| Key | 100101110001011011001\|100101110001011011001 |
| Text | 100110110000011010011101010010001011010010 |
| XOR | 000111000001000001010001111100000000001011 |

The resulting ciphertext can then easily be deciphered by performing an XOR between it and the binary key.

The main advantage of stream ciphers is their high speed for both encryption and decryption. A stream cipher is easily implemented in software, but special hardware can also be constructed to generate the keystream and perform the encryption and decryption operations. There are two main types of stream ciphers, *synchronous* and *self synchronizing*. The former uses a keystrem that's independent of the plaintext and cipher text; this is the common type. The latter uses a keystream that depends on the plaintext and may even depend on the cipher text that has been generated so far.

The simplest (and also most secure) type of stream cipher is the one-time pad, sometimes called the *Vernam cipher*. Given the plaintext (a string of $n$ bits), the keystream is selected as a random string of $n$ bits that's used just once. The resulting ciphertext is also a random bit string. If a codebreaker knows (or suspects) that this method was used to encrypt a message, they can try to break it by generating *every* random string of $n$ bits and using it as a keystream. The original plaintext would eventually be produced this way, but the cryptanalyst would have to read every plaintext generated; an impossible task given the immense number of random strings of $n$ bits, even for modest values of $n$. There is also the possibility that many meaningful plaintexts would be generated, making it even harder to decide which one is *the* real plaintext.

Even though it offers absolute security, the one-time pad is generally impractical, because the one-time pads have to be generated and distributed safely to everyone in the organization. This method can be used in only a limited number of situation, such as exchanging top-secret messages between a government and its ambassador abroad.

In practice, stream ciphers are used with keystreams that are *pseudo-random* bit strings. Such a bit string is generated by repeatedly using a recursive relation, so it is deterministic and therefore not random. Still, if a sequence of $n$ pseudo-random bits does not repeat itself, it can be used as the keystream for a stream cipher with relative safety.

One way to generate pseudo-random bits is a *shift register sequence*, most often used in hardware stream ciphers. Another is the mixed congruential pseudorandom number generator, usually used in software. Figure 3.39a shows a simple shift register with 10 stages, each a flip-flop (such as an SR latch, Section 7.2.1). Bits are shifted from left to right and the new bit entered on the left is the XOR of bits 6 and 10. Such a shift register is said to correspond to the polynomial $x^{10}+x^6+1$. The latches at stages 6 and 10 are said to be *tapped*. Before encryption starts, the register is initialized to a certain, agreed-upon 10-bit number (its initial state, the key). The register is then shited repeatedly, and the bits output from the rightmost stage are used to encode the plaintext.

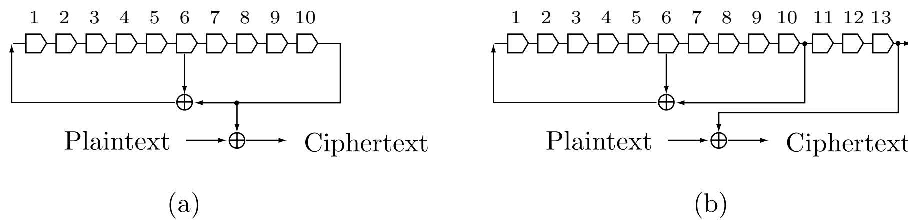



**Figure 3.40:** Shift Register Sequences.

The bit string output by the register depends, of course, on the initial state and on the corresponding polynomial. If the initial state of the register is reached after $s$ steps, there will be a cycle, and the same $s$ states will repeat over and over. Repetition, as is well known, is a major sin in cryptography and should be avoided. A shift register should therefore be designed to go through the maximum number of states before it cycles. In a 10-stage shift register, there can be $2^{10}=1024$ different 10-bit states. Of those, the special state of all zeros should be avoided because it degenerates the output to a stream of zeros. Thus, it is clear that the best that can be achieved in a 10-stage shift register is a sequence of $2^{10}-1=1023$ different states. It can be shown that this can be achieved if the polynomial associated with the register is *primitive*.

[A polynomial $P(x)$ of degree $n$ is primitive if it satisfies (1) $P(0)\neq 0$, (2) $P(x)$ is of order $2^n-1$, and (3) $P(x)$ is irreducible. The order of a polynomial is the smallest $e$ for which $P(x)$ divides $x^e+1$. For example, the polynomial $x^2+x+1$ has degree 2. It has order $e=3=2^2-1$ because $(x^2+x+1)(x+1)=x^3+1$. A polynomial is irreducible if it cannot be written as the product of lower-degree polynomials. For example, $x^2+x+1$ is irreducible, but $x^2-1$ is not because it equals the product $(x+1)(x-1)$. We therefore conclude that $x^2+x+1$ is primitive.]

Also, a shift register with the maximum number of states must have an even number of taps and the rightmost stage (the oldest bit) must be a tap. The latter property is illustrated in Figure 3.39b. It is easy to see that the output bit string generated by this 13-stage register is identical to that generated by the similar 10-stage register, except that it is delayed.

Selecting the right polynomial is important. We never use our initials, our address, or our phone number as a password. Similarly, certain polynomials have been adopted for use in various international standards, such as CRC. These polynomials should be avoided in a shift register. Some well-known examples (see also page 93) are $x^{16}+x^{12}+x^5+1$ which is used by the CCITT, $x^{12}+x^3+x+1$ and $x^{16}+x^{15}+x^2+1$ which generate the common CRC-12 and CRC-16 cyclic redundancy codes, and $x^{10}+x^3+1$ and $x^{10}+x^9+x^8+x^6+x^3+x^2+1$ which are used by the GPS satellites.

### 3.19.5 The Data Encryption Standard

During the 1950s and 1960s, relatively few people used computers and they had to write their own programs. Thus, the adoption of a standard computer-based cipher had to wait until the 1970s. This standard, known today as the data encryption standard or DES, was adopted by the US government in November 1976. It is based, with few modifications, on a cipher system known as Lucifer that was developed in the early 1970s by Horst Feistel for IBM. Lucifer is, in principle, a simple cipher employing both transposition and substitution. The plaintext is divided into blocks of 64 bits each and each block is enciphered separately. The 64 bits of a block are first perfectly shuffled and split into two 32-bit strings denoted by $L_0$ and $R_0$. String $R_0$ is first saved in $T$, then its bits $R_0$ are scrambled by a complex substitution cipher that depends on a key. Two

**From the Dictionary**

Tap (noun).
1. A cylindrical plug or stopper for closing an opening through which liquid is drawn, as in a cask; spigot.
2. A faucet or cock.
3. A connection made at an intermediate point on an electrical circuit or device.
4. An act or instance of wiretapping.

new strings $L_1$ and $R_1$ are constructed by $R_1 = L_0 + R_0$ and $L_1 = T$. This process is repeated 16 times. Deciphering is the reverse of enciphering.

This method is a perfect illustration of Kerckhoffs' principle. The details of the algorithm are known, so its security depends on the key. The longer the key, the more complex the cipher operations, and the harder it is to break this code. When the Lucifer cipher was considered for adoption as a standard, the NSA argued for limiting the size of the key to 56 bits, thereby limiting the total number of keys to $2^{56} \approx 1.13 \cdot 10^{15}$. The NSA felt that a 56-bit key would guarantee reasonable security (because no organization had at that time computers fast enough to break such a code in a reasonable amount of time), and at the same time will enable them to break any enciphered message because they had the biggest, fastest computers.

Many commercial entities have implemented the DES on their computers and started using it as soon as it became a standards. The DES has proved its worth as a secure code, but its users immediately ran into the same problem that has haunted cryptographers for many years; that of key distribution. How can a commercial organization, such as a stock broker, distribute keys to its many clients. The safest way to do this is to meet each client in person. Another safe solution is to hand deliver the key to each client with a trusted courier. The same problem faces a military organization, with the added complication that keys can be captured by the enemy, requiring an immediate replacement. The result was that the reliability of the DES, as of many ciphers before it, was being undermined by the problem of key distribution. Many experts on cryptography agreed that this was a problem without a solution. Then, completely unexpectedly, a revolutionary solution was discovered.

### 3.19.6 Diffie-Hellman-Merkle Keys

The following narrative illustrates the nature of the solution. Suppose that Alice wants to send Bob a secure message. She places the message in a strong box, locks it with a padlock, and mails it to Bob. Bob receives the box safely, but then realizes that he does not have the key to the padlock. This is a simplified version of the key distribution problem, and it has a simple, unexpected solution. Bob simply adds another padlock to the box and mails it back to Alice. Alice removes her padlock and mails the box to Bob, who removes his lock, opens the box, and reads the message.

The cryptographic equivalent is similar. Alice encrypts a message to Bob with her private key (a key unknown to Bob). Bob receives the encrypted message, encrypts it again, with his key, and sends the doubly-encrypted message back to Alice. Alice now decrypts the message with her key, but the message is still encrypted with Bob's key. Alice sends the message again to Bob, who decrypts it with his key and can read it.

The trouble with this simple approach is that most ciphers must obey the LIFO (last in first out) rule. The last cipher used to encrypt a doubly-encrypted message must be the first one used to decipher it. This is easy to see in the case of a monoalphabetic cipher. Suppose that Alice's key replaces `D` with `P` and `L` with `X` and Bob's key replaces `P` with `L`. After encrypting a message twice, first with Alice's key and then with Bob's key, any `D` in the message becomes an `L`. However, when Alice's key is used to decipher the `L`, it replaces it with `X`. When Bob's key is used to decipher the `X`, it replaces it with something different from the original `D`. The same happens with a polyalphabetic cipher.

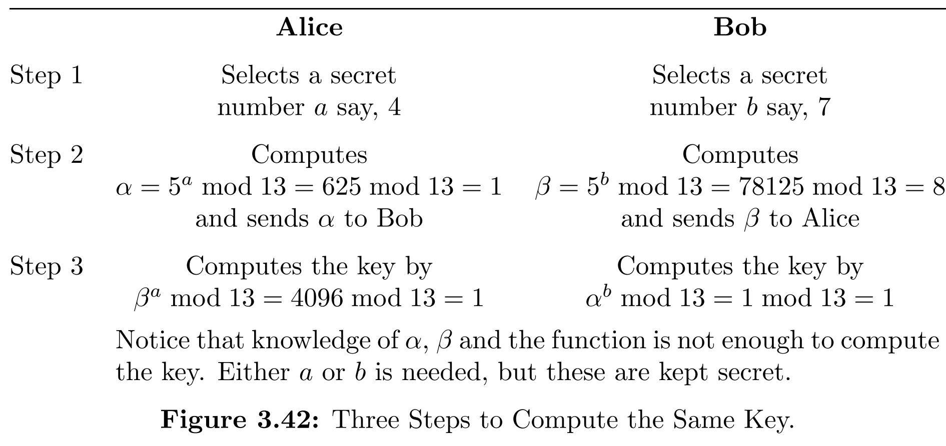
However, there is a way out, a discovery made in 1976 by Whitfield Diffie, Martin Hellman and Ralph Merkle. Their revolutionary Diffie-Hellman-Merkle key exchange method involves the concept of a *one-way function*, a function that either does not have an inverse or whose inverse is not unique. Most functions have simple inverses. The inverse of the exponential function $y = e^x$, for example, is the natural logarithm

$x = \log^e y$. However, modular arithmetic provides an example of a one-way function. The value of the modulo function $f(x) = x \bmod p$ is the remainder of the integer division $x \div p$ and is an integer in the range $[0, p-1]$. Table 3.40 illustrates the one-way nature of modular arithmetic by listing values of $3^x \bmod 7$ for 10 values of $x$. It is easy to see, for example, that the number 3 is the value of $3^x \bmod 7$ for $x = 1$ and $x = 7$. The point is that the same number is the value of this function for infinitely more values of $x$, effectively making it impossible to reverse this simple function.

| $x$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| $3^x$ | 3 | 9 | 27 | 81 | 243 | 729 | 2187 | 6561 | 19683 | 59049 |
| $3^x \bmod 7$ | 3 | 2 | 6 | 4 | 5 | 1 | 3 | 2 | 6 | 4 |

**Table 3.41:** Ten Values of $3^x \bmod 7$.

Based on this interesting property of modular arithmetic, the three researchers came up with an original and unusual scheme that's summarized in Figure 3.41. The process requires the modular function $L^x \bmod P$, whose two parameters $L$ and $P$ should satisfy $L < P$. The two parties have to select values for $L$ and $P$, but these values don't have to be secret.



| | **Alice** | **Bob** |
|---|---|---|
| Step 1 | Selects a secret number $a$ say, 4 | Selects a secret number $b$ say, 7 |
| Step 2 | Computes $\alpha = 5^a \bmod 13 = 625 \bmod 13 = 1$ and sends $\alpha$ to Bob | Computes $\beta = 5^b \bmod 13 = 78125 \bmod 13 = 8$ and sends $\beta$ to Alice |
| Step 3 | Computes the key by $\beta^a \bmod 13 = 4096 \bmod 13 = 1$ | Computes the key by $\alpha^b \bmod 13 = 1 \bmod 13 = 1$ |

Notice that knowledge of $\alpha$, $\beta$ and the function is not enough to compute the key. Either $a$ or $b$ is needed, but these are kept secret.

**Figure 3.42:** Three Steps to Compute the Same Key.

Careful study of Figure 3.41 shows that even if the messages exchanged between Alice and Bob are intercepted and even if the values $L = 5$ and $P = 13$ that they use are known, the key still cannot be derived since the values of either $a$ or $b$ are also needed but they are kept secret by the two parties.

This breakthrough has proved that cryptographic keys can be securely exchanged through unsecure channels, and users no longer have to meet personally to agree on keys or trust couriers to deliver them. However, the Diffie-Hellman-Merkle key exchange method described in Figure 3.41 is inefficient. In the ideal case, where both users are online at the same time, they can go through the process of Figure 3.41 (select the secret numbers $a$ and $b$, compute and exchange the values of $\alpha$ and $\beta$, and calculate the key) in just a few minutes. If they cannot be online at the same time (for example, if they live in very different time zones), then the process of determining the key may take a day or more.

The following analogy may explain why a one-way function is needed to solve the key distribution problem. Imagine that Bob and Alice want to agree on a certain paint color and keep it secret. Each starts with a container that has one liter of paint of a certain color, say, $R$. Each adds one liter of paint of a secret color. Bob may add a liter of paint of color $G$ and Alice may add a liter of color $B$. Neither knows what color was added by the other one. They then exchange the containers (which may be intercepted and examined). When each gets the other's container, each adds another liter of his or hers secret paint. Thus, each container ends up having one liter each of paints of colors $R$, $G$, and $B$. Each container contains paint of the same color. Intercepting and examining the containers on their ways is fruitless, because one cannot unmix paints. Mixing paints is a one-way operation.

### 3.19.7 Public Key Cryptography

In 1975, a little after the Diffie-Hellman-Merkle key exchange was published, Whitfield Diffie came up with the concept of an *asymmetric key*. Traditionally, ciphers use *symmetric keys*. The same key is used to encipher and decipher a message. Deciphering is the exact reverse of enciphering. Cryptography with asymmetric key requires two keys, one for enciphering and the other for deciphering. This seems a trivial concept, but is in fact revolutionary. In an asymmetric cipher, there is no need to distribute keys or to compute them by exchanging data as in the Diffie-Hellman-Merkle key exchange scheme. Alice could decide on two keys for her secret messages, make the encryption key public, and keep the decryption key secret (this is her private key). Bob could then use Alice's public key to encrypt messages and send them to Alice. Anyone intercepting such a message would not be able to decipher it because this requires the secret decryption key that only Alice knows.

> Whitfield Diffie took cryptography out of the hands of the spooks and made privacy possible in the digital age—by inventing the most revolutionary concept in encryption since the Renaissance.
>
> —*Wired*, November 1994

It was clear to Diffie that a cipher based on an asymmetric key would be the ideal solution to the troublesome problem of key distribution and would completely revolutionize cryptography. Unfortunately, he was unable to actually come up with such a cipher. The first (and so far, the only) simple, practical, and secure public key cipher, known today as RSA cryptography, was finally developed in 1977 by Ronald Rivest, Adi Shamir, and Leonard Adleman. RSA was a triumphal achievement, an achievement based on the properties of prime numbers.

A prime number, as most of us know, is a number with no divisors. More accurately, it is a positive integer $N$ whose only divisors are 1 and itself. (Nonprime integers are called composites.) For generations, prime numbers and their properties (the field of number theory) were of interest to mathematicians only, and had no practical applications whatsoever. RSA cryptography found an interesting, original, and very practical application for these numbers. This application relies on the most important property of prime numbers, the property that justifies the name *prime*. Any positive integer can be represented as the product of prime numbers (its prime factors) *in one way only*. In other words, any integer has a unique prime factorization. For example, the number 65,535 equals the product of the four primes 3, 5, 17, and 257.

The main idea behind RSA is to choose two large primes $p$ and $q$ that together constitute the private key. The public key $N$ is their product $N = p \times q$ (which, of course, is composite) The important (and surprising) point is that multiplying large integers is a one-way function! It is relatively easy to multiply integers, even very large ones, but it is practically impossible, or at least extremely time consuming, to find the prime factors of a large integer, with hundreds of digits. Today, after millennia of research (primes have been known to the ancients), no efficient method for factoring numbers has been discovered. All existing factoring algorithms are slow and may take years to factor an integer consisting of a few hundred decimal digits. The factoring challenge (with prizes) offered by RSA laboratories [RSA 01] testifies to the accuracy of this statement.

To summarize, we know that the public key $N$ has a unique prime factorization and that its prime factors are the private key. However, if $N$ is large enough, we will not be able to factor it, even with the fastest computers, which makes RSA a secure cipher. At the same time, no one has proved that a fast factorization method does not exist. It is not inconceivable that someone will develop such an algorithm, which would render RSA (impregnable for more than two decades) useless and would stimulate researchers to discover a different public key cipher.

(Recent declassifying of secret British documents suggests that a cipher very similar to RSA was developed by James Ellis and his colleagues starting in in 1969. They worked for the British government communications headquarters, GCHQ, and so had to keep their work secret. See [Singh 99].)

And now, to the details of RSA. They are deceptively simple, but the use of large numbers requires special arithmetic routines to be implemented and carefully debugged. We assume that Alice has selected

two large primes $p$ and $q$ as her private key. She has to compute and publish two more numbers as her public key. They are $N = p \cdot q$ and $e$. The latter can be any integer, but it should be relatively prime to $(p-1)\cdot(q-1)$. Notice that $N$ must be unique (if Joe has selected the same $N$ as his public key, then he knows the values of $p$ and $q$), but $e$ does not have to be. To encrypt a message $M$ (an integer) intended for Alice, Bob gets her public key ($N$ and $e$), computes $C = M^e \bmod N$, and sends $C$ to Alice through an open communications channel. To decrypt the message, Alice starts by computing the decryption key $d$ from $e\times d = 1 \bmod (p-1)\cdot(q-1)$, then uses $d$ to compute $M = C^d \bmod N$.

The security of the encrypted message depends on the one-way nature of the modulo function. Since the encrypted message $C$ is $M^e \bmod N$, and since both $N$ and $e$ are public, the message can be decrypted by inverting the modulo function. However, as mentioned earlier, this function is impossible to invert for large values of $N$. It is important to understand that polyalphabetic ciphers can be as secure as as RSA, are easier to implement and faster to execute, but they are symmetric and therefore suffer from the problem of key distribution.

The use of large numbers requires special routines for the arithmetic operations. Specifically the operation $M^e$ may be problematic, since $M$ may be a large number. One way to simplify this operation is to break the message $M$ up into small segments. Another option is to break up $e$ into a sum of terms and use each term separately. For example, if $e = 7 = 1 + 2 + 4$, then

$$M^e \bmod N = [(M^1 \bmod N)\times(M^2 \bmod N)\times(M^4 \bmod N)] \bmod N.$$

### 3.19.8 Pretty Good Privacy

The computations required for RSA encryption and decryption are straightforward, but require special software. Both accuracy and speed are important. Special routines, optimized for speed, must be implemented to perform the arithmetic operations on large numbers without any loss of accuracy. With the right software, anyone could access the power of RSA, and exchange messages confidentially. Without such software, RSA remains a largely unused resource. Such software was implemented in the late 1980s by Phil Zimmermann, who named it *pretty good privacy* or PGP for short. Certain versions of PGP are free, while others are commercial products. In the US, PGP software is available from `http://www.pgp.com/`. Outside the US it is available from `http://www.pgpi.org/`. The source code of PGP has been published in [Zimmermann 95] (currently out of print).

To overcome the speed problem, PGP was designed in a special way. The software uses a (symmetric) block cipher algorithm, similar to DES, known as IDEA (international data encryption algorithm, patented by Ascom but free for noncommercial use) to actually encrypt the message. The RSA method and the receiver's public key are used by PGP only to encrypt the IDEA key (i.e., the key used to encrypt the message).

PGP also has several features that make it especially user-friendly and easy to use. It can automatically generate private and public keys for the user. The user just wiggles the mouse randomly, and the software uses this movement to generate a unique key. PGP can also digitally authenticate encrypted messages. Authentication is needed in cases where the identity of the sender is important and is in doubt. Imagine a bank receiving an encrypted message signed "Bob" that asks them to transfer all of Bob's assets to a Swiss numbered bank account. Authentication would show whether the message really came from Bob. Bob can authenticate his message by including a short "signature message" in his main message. The main message is, of course, encrypted with the bank's public key, so only the bank can encipher it. The signature is simply a short, agreed-upon message encrypted with Bob's private key. Anyone in possession of Bob's public key (i.e., anyone at all) can decrypt this message and read it, but only Bob can encrypt it. The signature is not secure, but it verifies that it was Bob who sent the message.

In 1993, after PGP became widely used all over the world, the US government made Zimmermann the target of a three-year criminal investigation. The government claimed that US export restrictions for cryptographic software were violated when PGP spread all over the world. This investigation was dropped in early 1996, and today PGP is a widely-used, important encryption tool. For more details, see [Zimmermann 01].

> Computers are useless. They can only give you answers.
>
> —Pablo Picasso

## 3.20 Steganography

The word *steganography* is derived from the Greek *στεγανος γραφεν*, meaning "covered writing." The aim of steganography is to conceal data. This applies to data being stored or to a message being transmitted. We start with a few elementary and tentative methods most of which are used manually, although some can benefit from a computer. These are mostly of historical interest.

1. Write a message on a wooden tablet, cover it with a coat of wax, and write innocuous text on the wax. This method is related by Greek historians.

2. Select a messenger, shave his head, tattoo the message on his head, wait for the hair to grow, and send him to his destination, where his head is shaved again.

3. Use invisible ink, made of milk, vinegar, fruit juice, or even urine, and hide the message between the lines of a seemingly-innocent letter. When the paper is heated up, the text of the hidden message slowly appears.

4. The letters constituting the data are concealed in the second letter of every word of a specially-constructed *cover text*, or in the third letter of the first word of each sentence. An example is the data "coverblown" that can be hidden in the specially-contrived cover text "Accepted you over Neil Brown. About ill Bob Ewing, encountered difficulties." A built-in computer dictionary can help in selecting the words, but such specially-constructed text often looks contrived and may raise suspicion.

5. An ancient method to hide data uses a large piece of text where small dots are placed under the letters that are to be hidden. For example, this paragraph has dots placed under certain letters that together spell the message "i am hidden." A variation of this method slightly perturbs certain letters from their original positions to indicate the hidden data.

6. A series of lists of alternative phrases from which a paragraph can be built can be used, where each choice of a phrase from a list conceals one letter of a message. This method was published in 1518 in *Polygraphiae* by Johannes Trithemius, and was still used during World War II, as noted in [Marks 99].

7. Check digits. This idea is used mostly to verify the validity of important data, such as bank accounts and credit card numbers, but can also be considered a method to hide validation information in a number. A simple example is the check digit used in the well-known *international standard book number* (ISBN), assigned to every book published. This number has four parts, a country code, a publisher code, a book number assigned by the publisher, and a check digit, for a total of 10 digits. For example, ISBN 0-387-95045-1 has country code 0, publisher code 387, book number 95045, and check digit 1. The check digit is computed by multiplying the leftmost digit by 10, the next digit by 9, and so on, up to ninth digit from the left, which is multiplied by 2. The products are then added, and the check digit is determined as the smallest integer that when added to the sum will make it a multiple of 11. The check digit is therefore in the range $[0, 10]$. If it happens to be 10, it is replaced by the Roman numeral X in order ot make it a single symbol.

8. It is possible to assign 3-digit codes to the letters of the alphabet as shown in Table 3.42. Once this is done, each letter can be converted into three words of cover text according to its 3-digit code. A code digit of 1 is converted into a word with 1, 4, or 7 syllables, a code digit of 2 is converted into a word with 2 or 5 syllables, and a code digit of 3 is converted into a word with 3 or 6 syllables. This way, there is a large selection of words for each word in the cover text.

| | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| A | 111 | D | 121 | G | 131 | J | 211 | M | 221 | P | 231 | S | 311 | V | 321 | Y | 331 |
| B | 112 | E | 122 | H | 132 | K | 212 | N | 222 | Q | 232 | T | 312 | W | 322 | Z | 332 |
| C | 113 | F | 123 | I | 133 | L | 213 | O | 223 | R | 233 | U | 313 | X | 323 | | |

**Table 3.43:** Three-digit Codes for the Letters.

Modern steganography methods are more sophisticated and are based on the use of computers and on the binary nature of computer data. We start with a few simple methods, and follow with two important and commonly-used approaches to steganography, namely least-significant bit modification (Section 3.20.1), and BPCS steganography (Section 3.20.2).

A modern personal computer may have tens of thousands of files on one hard disk. Many files are part of the operating system, and are unfamiliar to the owner or even to an expert user. Each time a large

application is installed on the computer, it may install several system files, such as libraries, extensions, and preference files. A data file may therefore be hidden by making it look like a library or a system extension. Its name may be changed to something like `MsProLibMod.DLL` and its icon modified. When placed in a folder with hundreds of similar-looking files, it may be hard to identify as something special.

Camouflage is the name of a steganography method that hides a data file $D$ in a cover file $A$ by scrambling $D$, then appending it to $A$. The original file $A$ can be of any type. The camouflaged $A$ looks and behaves like a normal file, and can be stored or emailed without attracting attention. Camouflage is not very safe, since the large size of $A$ may raise suspicion.

When files are written on a disk (floppy, zip, or other types), the operating system modifies the disk directory, which also includes information about the free space on the disk. Special software can write a file on the disk, then reset the directory to its previous state. The file is now hidden in space that's declared free, and only special software can read it. This method is risky, because any data written on the disk may destroy the hidden file.

### 3.20.1 LSB Image Steganography

Data can be hidden in a color image (referred to as a *cover image*), since slight changes to the original colors of some pixels are many times invisible to the eye (after all, changing the original pixels is the basis of all lossy image compression methods, Section 3.15).

Given an image $A$ with 3 bytes per pixel, each of the three color components, typically red, green, and blue, is specified by one byte, so it can have one of 256 shades. The data to be hidden—which can be text, another image, video, or audio—is represented in the computer as a file, or a stream of bits. Each of those bits is hidden by storing it as the least-significant bit of the next byte of image $A$, replacing the original bit. The least-significant bit of the next byte of $A$ is, of course a 0 or a 1. The next bit to be hidden is also a 0 or a 1. This means that, on average, only half the least-significant bits are modified by the bits being hidden. Each pixel is represented by three bytes, so changing half the bytes means changing 1 or 2 least-significant bits per pixel, on average.

Small changes in the color of a few isolated pixels may be noticeable if the cover image $A$ has large uniform areas, which is why $A$ has to be carefully chosen. It should contain a large number of small details with many different colors. Such an image is termed *busy*.

If image $A$ is to be compressed later, the compression should, of course, be lossless. If the owner forgets this, or if someone else compresses image $A$ with a lossy compression algorithm, some of the hidden data would be lost during the compression.

The hiding capacity of this method is low. If one data bit is hidden in a byte, then the amount of data hidden is $1/8 = 12.5\%$ of the cover image size. As an example, only 128K bytes can be hidden in a 1Mbyte cover image.

An extension of this method hides the bits of the data not in consecutive bytes of the image but only in certain bytes that are selected by a key. If the key is, say, 1011, then data bits are hidden in the least-significant bits of bytes 1, 3, and 4 but not of byte 2, then in bytes 5, 6, and 8, but not in byte 7, and so on. A variation of this method hides one bit in the least-significant position of a byte of $A$ and hides the next bit in the second least-significant position of the next byte of $A$.

A more sophisticated technique is to scan the image byte by byte in a complex way, not row by row, and hide one bit of the data in each byte visited. For example, the image bytes may be visited in the order defined by a space-filling curve, such as the Peano curve or the Hilbert curve. Such curves visit each point of a square area exactly once.

A marked improvement is achieved by considering the sensitivity of the eye to various colors. Of the three colors red, green, and blue, the eye is most sensitive to green and least sensitive to blue. Thus, hiding bits in the blue part of a pixel results in image modification that's less noticeable to the eye. The price for this improvement is reduced hiding capacity from 12.5% to 4.17%, since one bit is hidden in each group of 3 image bytes.

Better security is obtained if the data is encrypted before it is hidden. An even better approach is to first compress the data, then encrypt it, and finally hide it. This increases the hiding capacity considerably.

An audio file can also serve as a cover. Such a file consists of audio samples, each typically a 16-bit number, so one bit of data can be hidden in the least-significant bit of each sample. The difference is that an

image is two-dimensional, and so can be scanned in complex ways, whereas an audio file is one-dimensional.

### 3.20.2 BPCS Steganography

BPCS (bit-plane complexity segmentation) steganography is the brainchild of Eiji Kawaguchi. Developed in 1977, this method hides data in a cover image (color or grayscale) and its main feature is large hiding capacity [BPCS 01]. The size of the hidden data is typically 50–70% the size of the cover image, and hiding the data does not increase the size of the cover image.

Figure 3.43 shows a grayscale image of parrots, together with five of its bitplanes (bitplane 8, the most-significant one, bitplane 1, the least-significant one, and bitplanes 3, 5, and 7). It is clear that the most-significant bitplane is somewhat similar to the full image, the least-significant bitplane is random (or at least seems random) and, in general, as we move from the most-significant bitplanes to least-significant ones, they become more random. However, each bitplane has some parts that look random. (BPCS converts the pixels of the cover image from binary to *Gray codes*, but this feature will not be discussed here, see Section 2.24.)

The principle of BPCS is to separate the image into individual bitplanes, check every $8\times 8$ bit square region in each bitplane for randomness, and replace each random region with 64 bits of hidden data. Regions that are not random (*shape-informative* regions in BPCS terminology) are not modified. A special complexity measure $\alpha$ is used to determine whether a region is random. A color image where each pixel is represented by 24 bits has 24 bitplanes (8 bitplanes per color). A grayscale image with 8 bits/pixel has eight bitplanes. The complexity measure $\alpha$ of an 8×8 block of bits is defined as the number of adjacent bits that are different. This measure (see note below) is normalized by dividing it by the maximum number of adjacent bits that can be different, so $\alpha$ is in the range $[0, 1]$.

An important feature of BPCS is that there is no need to identify those regions of the cover image that have been replaced with hidden data, because the hidden data itself is transformed, before it is hidden, to a random representation. The BPCS decoder identifies random regions by performing the same tests as the encoder, and it extracts the 64 data bits of each random region. The hidden data is made to look random by first compressing it. Recall (from Section 3.10) that compressed data has little or no redundancy, and therefore looks random. If a block of 64 bits (after compression) does not pass the BPCS randomness test, it goes through a *conjugation operation* that increases its randomness. The conjugation operation computes the exclusive-OR of the 64-bit block (which is regrouped as an $8\times 8$ square) with an $8\times 8$ block that has a checkerboard pattern and whose upper-left corner is white. This transforms a simple pattern to a complex one and changes the complexity measure of the block from $\alpha$ to $1-\alpha$. The encoder has to keep a list of all the blocks that have gone through the conjugation operation, and this list is also hidden in the cover image. (The list is not long. In a 1K×1K×24-bit image, each of the 24 bitplanes has $2^{14}=16K$ regions, and so contributes 16K bits (2K bytes) to the conjugation list, for a total of $24\times$2K = 49,152 bytes, or 1.56% the size of the image. The algorithm does not specify where to hide the conjugation list and any practical implementation may have a parameter specifying one of several places to hide it.)

Note (for those who want the entire story). Figure 3.44 shows how the BPCS image complexity measure $\alpha$ is defined. Part (a) of the figure shows a 4×4 image with the 16 pixels (1 bit each) numbered. The complexity of the image depends on how many adjacent bits differ. The first step is to compare each bit in the top row (row 1) to its neighbor below and count how many pairs differ. Then the four pairs of bits of rows 2 and 3 are compared, followed by the pairs in rows 3 and 4. The maximum number of different pairs is $4\times 3$. The next step is to compare adjacent bits horizontally, starting with the four pairs in columns 1 and 2. This can also yield a maximum of $4\times 3$ different bit pairs. The image complexity measure $\alpha$ is defined as the actual number of bit pairs that differ, divided by the maximum number, which for a $4\times 4$ block is $2\cdot 4(4-1)=24$ and for an $8\times 8$ block is $2\cdot 8(8-1)=112$.

Figure 3.44b shows a checkerboard pattern, where every pair of adjacent bits differ. The value of $\alpha$ for such a block is 1. Figure 3.44c shows a slightly different block, where the value of $\alpha$ is

$$\alpha=\frac{(3+4+4)+(3+4+4)}{2\cdot 4\cdot 3}=0.917.$$

BPCS considers a block random if it satisfies $\alpha\geq 0.5-4\sigma$, where $\sigma=0.047$ is a constant. The value of $\sigma$ was determined by computing $\alpha$ values for many $8\times 8$ blocks and plotting them. The distribution of the $\alpha$ values was found to be Gaussian with mean 0.5 and standard deviation 0.047.

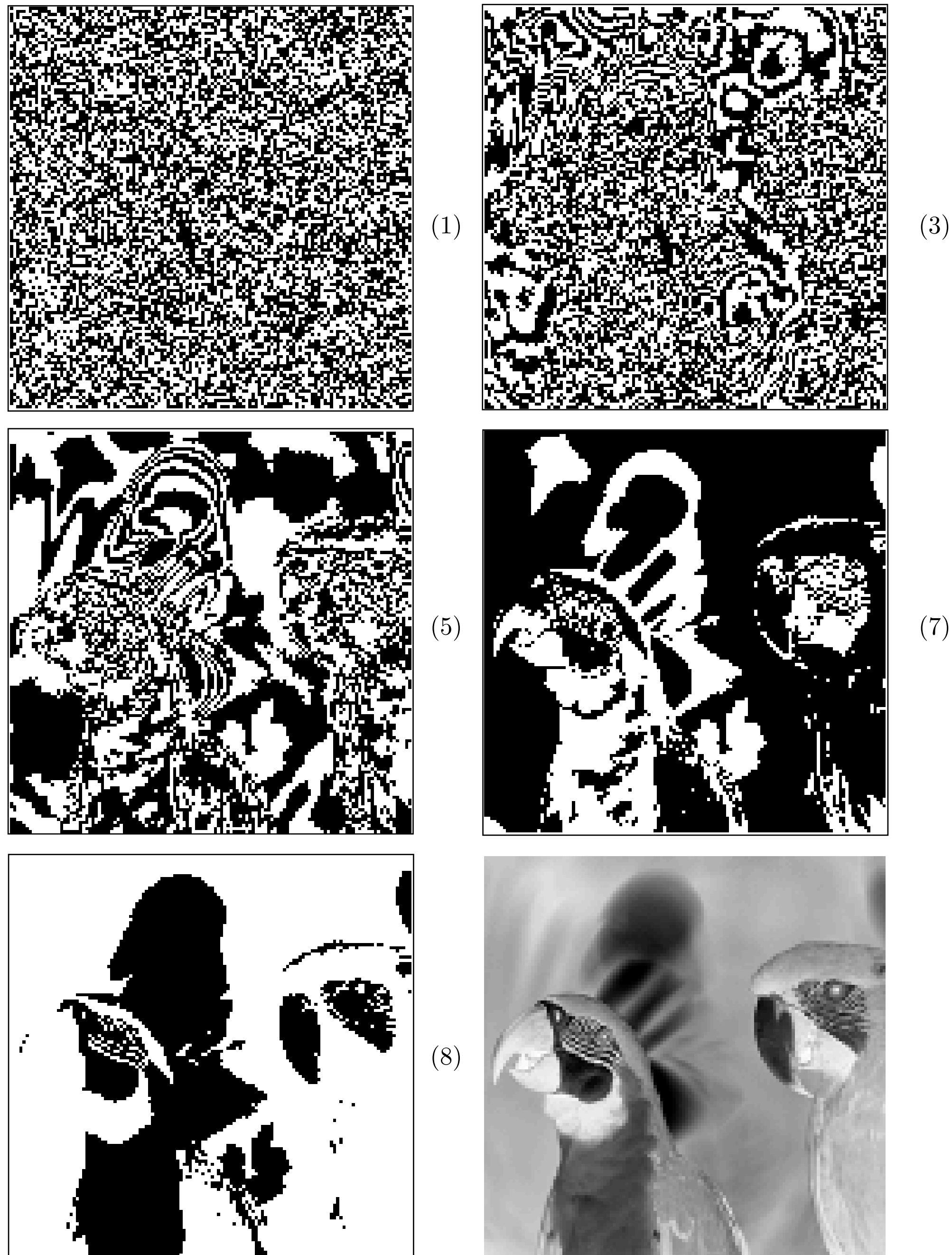



**Figure 3.44:** Bitplanes 1, 3, 5, 7 and 8 of the Parrots Image.

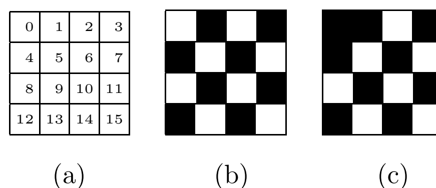



**Figure 3.45:** BPCS Image Complexity Measure.

### 3.20.3 Watermarking

The term *watermarking* refers to any technique used to embed ownership information in important digital data. The watermark can be used to identify digital data that has been illegally copied, stolen, or altered in any way. a typical example is a map. It takes time, money, and effort to create a road map of a city. Once this map is digitized, it becomes easy for someone to copy it, make slight modifications, and sell it as an original product. A watermark hidden in the map can help the original developer of the map identify any attempts to steal the work. The watermark itself should not increase the size of the original data by much, and must also be robust, so it does not get destroyed by operations such as filtering, compressing, and cropping.

Watermarking can be done by steganography. The watermark may be a string with, for example, the name of the owner, repeated as many times as needed. This string is hidden in the original data (image or audio) and can be extracted by the original owner by means of a secret key.

There is, however, a fundamental difference between steganography and watermarking. In the former, the hidden data is important, while the cover image is not. The embedding capacity of the algorithm is important, but the hidden data may be fragile (it may be destroyed by transforming or compressing the cover image) In watermarking the cover image is valuable, while the hidden data is not (it can be any identifying data). The embedding capacity is unimportant, but the hidden data has to be robust. As a result, watermarking should use special steganographic techniques.

The related concept of *fingerprinting* should also be mentioned. Fingerprinting refers to embedding a secret serial number in each copy of some important digital data. A commercially-sold computer program, for example, is easy to copy illegally. By embedding a secret serial number in each copy of the program sold, the manufacturer can identify each buyer with a serial number, and so identify pirated copies.

## 3.21 Computer Communications

Marshall McLuhan, known as the metaphysician of media, is the author of the phrase "the medium is the message," by which he meant that the way we acquire information affects us more than the information itself. The modern field of telecommunications, however, distinguishes the message (text, images, or audio sent in bits) from the medium it is sent on (copper wire, microwave links, etc.)

The roots of computer communications can be traced to Claude Shannon, the founder, in 1948, of information theory. In his 1948 landmark paper, Shannon dismissed the universally accepted view that messages were synonymous with the waveforms by which they were sent over lines, and that different lines were needed for the transmission of voice (i.e., telephone) and of telegraph messages. Telephone engineers saw the waveform as both the medium and the message, but Shannon recognized that by treating all messages as strings of binary digits—or bits as he called them—it was possible to distinguish the message from the medium it travelled on. This allowed communications engineers to focus on the delivery mechanism without worrying about the message itself, which could be voice, data, or telegraph signals.

The remainder of this chapter discusses computer communications in three parts. Section 3.22 discusses serial I/O, including modem operation, modulation techniques, protocols, and related topics. Section 3.26 discusses local area networks. The Internet and the World Wide Web are discussed starting with Section 3.27.

## 3.22 Serial I/O

All the I/O processors discussed earlier in this chapter operate in parallel. They send or receive $N$ bits in parallel on the data bus. This section discusses *serial I/O*, the case where bits are sent on a *single* transmission line, one at a time. Serial I/O is obviously slower than parallel I/O, so the first point to consider is why it is used.

The reason for using serial I/O is the importance of long distance transmission of data between computers (Figure 3.45). This is an important computer application, and its most expensive part is the transmission line.

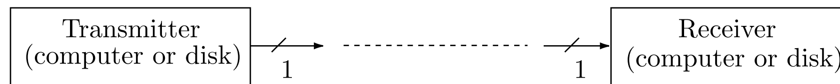



**Figure 3.46:** A typical long-distance configuration

In long distance transmission of information (telecommunications), existing communications lines, such as telephone lines, are often used. Using long distance lines may be more expensive than the combined costs of the computers and the I/O devices on both ends of the transmission line. As a result, we use the minimum number of lines, which is 1. (In the case of fiber optic lines, only one line is used. However, with electrical lines, a pair of lines have to be used, in order to have a closed circuit. We ignore this technical point and we consider serial I/O to use just one line.)

The speed of serial I/O is measured in bits per second or *baud*, a term derived from the name Baudot (page 83). The speed of parallel I/O is measured in bytes/s.

Other terms used to indicate speed of serial transmission are T1 and T3 (Section 3.25). The term T1 was coined by AT&T to indicate transfer rates of 1.544 megabits per second. T3, in spite of its name, is not three times, but is almost 30 times T1. It indicates transfer rates of 44.736 megabits of digital data per second. For some historical reason, T2 has been bypassed altogether.

The computer components discussed so far (processor, memory, and I/O processor) are not designed with serial I/O in mind. They use buses on which they move data in *parallel*. To do serial I/O on a computer, it is necessary to perform special operations. On the transmitting side, it is necessary to unpack a byte and send it on the single transmission line bit by bit. The receiver has to do the opposite, namely receive individual bits, pack each group of eight bits into a byte, and send the byte on the data bus to the CPU. These operations are performed by a piece of hardware (an optional part of the I/O processor) called a *serial port* or a *serial interface*. Often, the serial interface is an optional part of the computer, and can be purchased separately (in the form of a card), plugged into the computer, and connected to the single communications line. Figure 3.46 is a diagram of such a configuration.

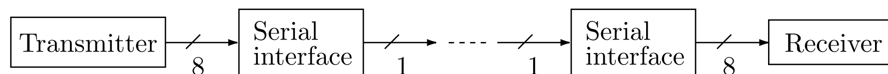



**Figure 3.47:** A long-distance configuration with serial interfaces on both sides

The configuration of Figure 3.46 has one disadvantage, namely the distance between the transmitter and the receiver cannot be too large. The reason for this is that computers and I/O devices are designed for short distance communications. They generate low voltage signals that cannot penetrate through long wires. A typical signal generated by a computer can travel, on a wire, a distance of a few hundred yards. Beyond that, the signal gets too weak to be received reliably. This is why the configuration of Figure 3.46 is used only when the distances involved are short, such as in a local area network (LAN, Section 3.26.1).

For long distances, public lines, such as telephone lines, have to be used. In such a case, the digital low-voltage signals generated by the computer have to be converted to signals that can travel a long distance over the phone line. At the receiving end, the opposite conversion must take place.

### 3.22.1 Modulation

This naturally leads to the question of how the telephone system works. How does the telephone send voice signals over long distances on phone lines? The telephone cannot produce high-voltage signals (because they are dangerous), and a weak signal fed into a transmission line is completely absorbed by the internal resistance of the line after traveling a short distance.

The solution used by the phone system is to generate a signal that varies regularly with time, a wave (Figure 3.48a), and to feed that wave into the transmission line. This solution uses the fact that the resistance of a line to electrical signals is not constant but depends on the frequency of the signal. This phenomenon is known as *impedance* and it is an important feature of modern telecommunications. The telephone generates a wave whose amplitude and frequency change according to the voice pattern. The frequency, however, must always be in the limited range where the impedance of the line is lowest. Even so, signals on phone lines must be amplified every 20 miles or so. The wires commonly used for voice-grade telephone lines feature low impedance for frequencies in the range 2–3 KHz, which is why many modem standards specify a frequency of 2400 Hz or 300 Hz.

The wave sent on the telephone line has to be varied (modulated) to reflect the bits being sent. Two different wave shapes are needed, one for a binary 0 and the other for a binary 1. This can be done in a number of ways, the most important of which are outlined below. Figure 3.47 shows a transmitter-receiver configuration with special hardware to implement modulation. Note that on the transmitter side we need a modulator, and on the receiver side we need a demodulator. However, during transmission, the role of transmitter and receiver may be reversed several times, so each side needs both a modulator and a demodulator. Such devices are therefore made so they can perform either function and they are called *modems* (for MOdulator, DEModulator).

Since the distance between the serial interface and the modem is short, they are connected with several lines, but only one is used for the data; the rest are used for control signals.

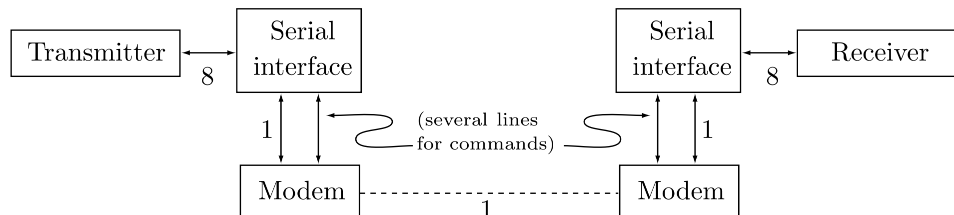



**Figure 3.48:** Serial I/O with serial interfaces and modems

Here are the common modulation techniques used by current modems.

Amplitude modulation (AM). The wave is modulated by varying the amplitude between a zero and a one (Figure 3.48b). The frequency remains constant. This method, sometimes called *amplitude shift keying* or ASK, is simple but not very reliable, because electrical interference and internal resistance of the telephone line can corrupt the original amplitude of the wave.

Frequency modulation (FM). Zeros and ones are distinguished by varying the frequency of the wave (Figure 3.48c). Two frequencies are used, both in the region where the impedance is lowest. The amplitude is kept constant. This method is also called *frequency shift keying* or FSK, and is less sensitive to electrical noise during transmission. A typical example of FSK is the old Bell 103 modem standard (Table 3.54), used in old, 300 baud modems. This standard uses the frequencies 1070 and 1270 Hz to send binary 0's and 1's, and the frequencies 2025 and 2225 Hz to receive binary 0's and 1's, respectively.

Phase modulation (PM, also phase shift keying, PSK). The phase of the wave is reversed each time the data changes between a 0 and a 1 (Figure 3.48d). Technically, we say that the phase is advanced by 180° (or $\pi$ radians). The advantage is that it is easy for the receiver to detect each phase change in the transmission, so this type of modulation is less susceptible to electrical interference and noise. A similar version of PM reverses the phase only when a binary 1 is transmitted. A sequence of all zeros results in a uniform, unmodulated wave, but a sequence of ones results in waves, each of which is phase reversed relative to its predecessor.

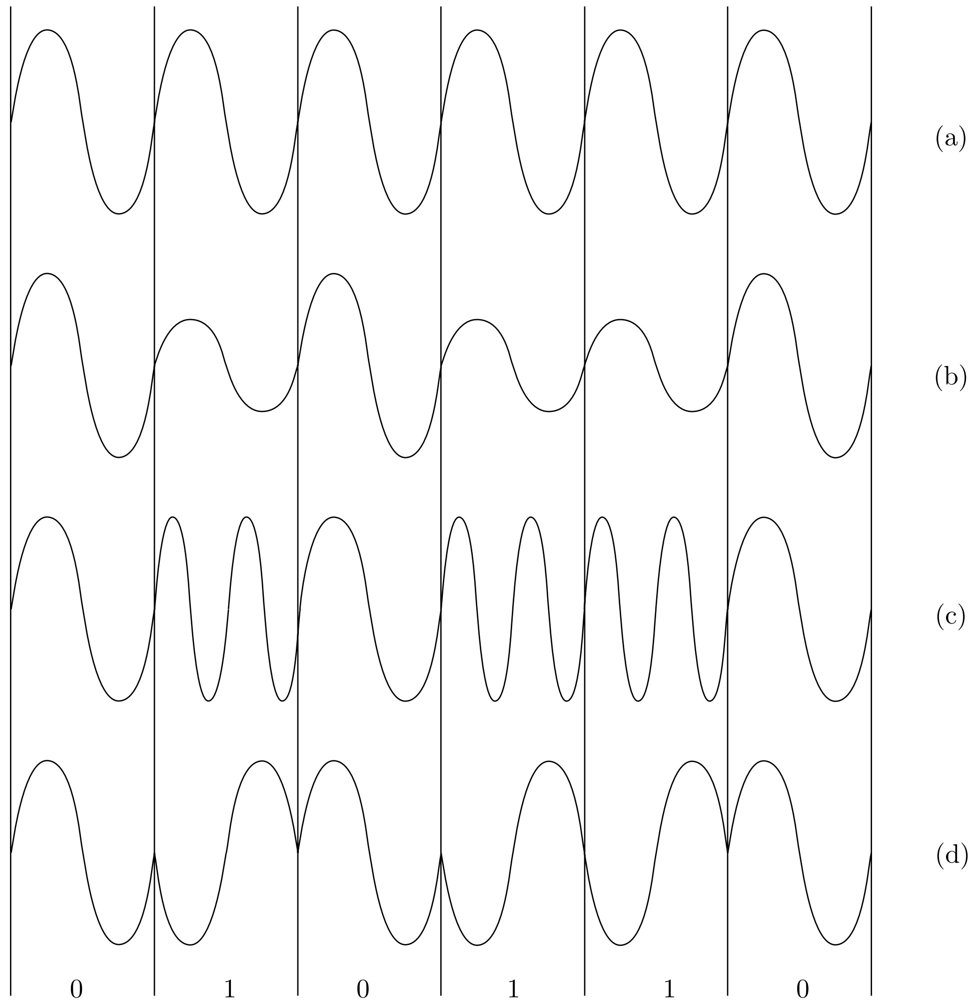



**Figure 3.49:** Three basic modulation techniques

These three basic modulation techniques make it possible to transmit one bit per time unit. In such cases, the term "baud" indicates both the number of time units per second (the *signaling rate*) and the number of bits transmitted per second. Recent advances in telecommunications, however, allow the use of sophisticated modulation techniques where more than one bit can be transmitted in a time unit. Such techniques involve phase modulations at various angles, not just 180°, and may also employ different amplitudes. Figure 3.49a shows a sine wave and the results of shifting it through various angles. This diagram makes it easy to understand a modulation technique where in each time unit the phase of the wave can be shifted by one of the four angles 0°, 90°, 180°, and 270°. This is equivalent to transmitting, in each time unit, one of four symbols. Each of the four symbols can be assigned a 2-bit code, with the result that this modulation technique transmits two bits in each time unit. It is called *phase amplitude modulation* or PAM. The term "baud" now refers to the signaling rate, and the number of bits per second (bps or the *bitrate*) is now twice the baud.

A simple extension of PAM uses two distinct amplitudes. The wave transmitted in each time unit can be in one of four phases and one of two amplitudes. This is equivalent to sending one of eight symbols, or three bits, in each time unit. The resulting bitrate is three times the baud. Such a method is called *quadrature amplitude modulation*, or QAM. The constellation diagram of Figure 3.49b is a graphics representation of the 4-phase 2-amplitude QAM modulation technique. The four inner points represent the four phases with small amplitude, and the four outer points correspond to the large amplitude. Table 3.50 lists the three bits assigned to each of the eight symbols, and Figure 3.49c is an example of the modulated wave required to transmit the bit sequence 010|111|000|100 (notice the discontinuities in the wave). The symbols sent in the four time units are 1: 90°, 2: 270°, 1: 0°, and 1: 180°.

| Amp | Phase | Bits |
|---|---|---|
| 1 | 0 | 000 |
| 2 | 0 | 001 |
| 1 | 90 | 010 |
| 2 | 90 | 011 |
| 1 | 180 | 100 |
| 2 | 180 | 101 |
| 1 | 270 | 110 |
| 2 | 270 | 111 |

**Table 3.51:** Eight state QAM.

▸ **Exercise 3.18:** QAM extends the three basic modulation techniques by using multiple amplitudes and phases but not multiple frequencies. What is the reason for that?

The QAM technique can be extended to more than four phases and two amplitudes. Figure 3.51 shows (a) a 16-QAM scheme with four phases and four amplitudes, (b) an 8-QAM with eight phases and a single amplitude (this is used in 4800 bps modems), (c) a 16-QAM with 12 phases and three amplitudes (used in 9600 bps modems), and (d) a 32-QAM consisting of 20 phases and four amplitudes (this modulation is used by V.32 modems). However, as the constellation diagram gets denser, it becomes easier for even minor disturbances along the way to corrupt the modulated wave and result in incorrect demodulation. This is why the smallest phase shift currently used in QAM is 22.5° and why QAM modulations don't exceed nine bits per baud (512-QAM). Two solutions to the reliability problem are (1) improved modem hardware and line quality (resulting in higher signal to noise ratio) and (2) the use of error-correcting codes. A further speed increase is achieved by compressing the data before it is transmitted.

Typical examples of amplitudes are 2, 4, and 6 volts. A typical time unit is 0.833 ms. This translates to a signaling rate of 1200 time units per second. At two bits per time unit, this results in a bitrate of 2400 bps, while at four bits per unit, the bitrate is 4800 bps. An example of a higher speed is a signaling rate of 3200 baud combined with nine bits per baud (achieved by 512-QAM). This results in a bitrate of 28800 bps.

Currently, the fastest telephone modems work at 56.6Kbps. Such a modem uses a D/A circuit to convert 8-bit digital data from the transmitting computer into a wave with a signaling rate of 8 KHz. A 256QAM modulation is used to pack eight bits into each time unit, resulting in a bitrate of $8000\times 8 = 64000$ bps. The

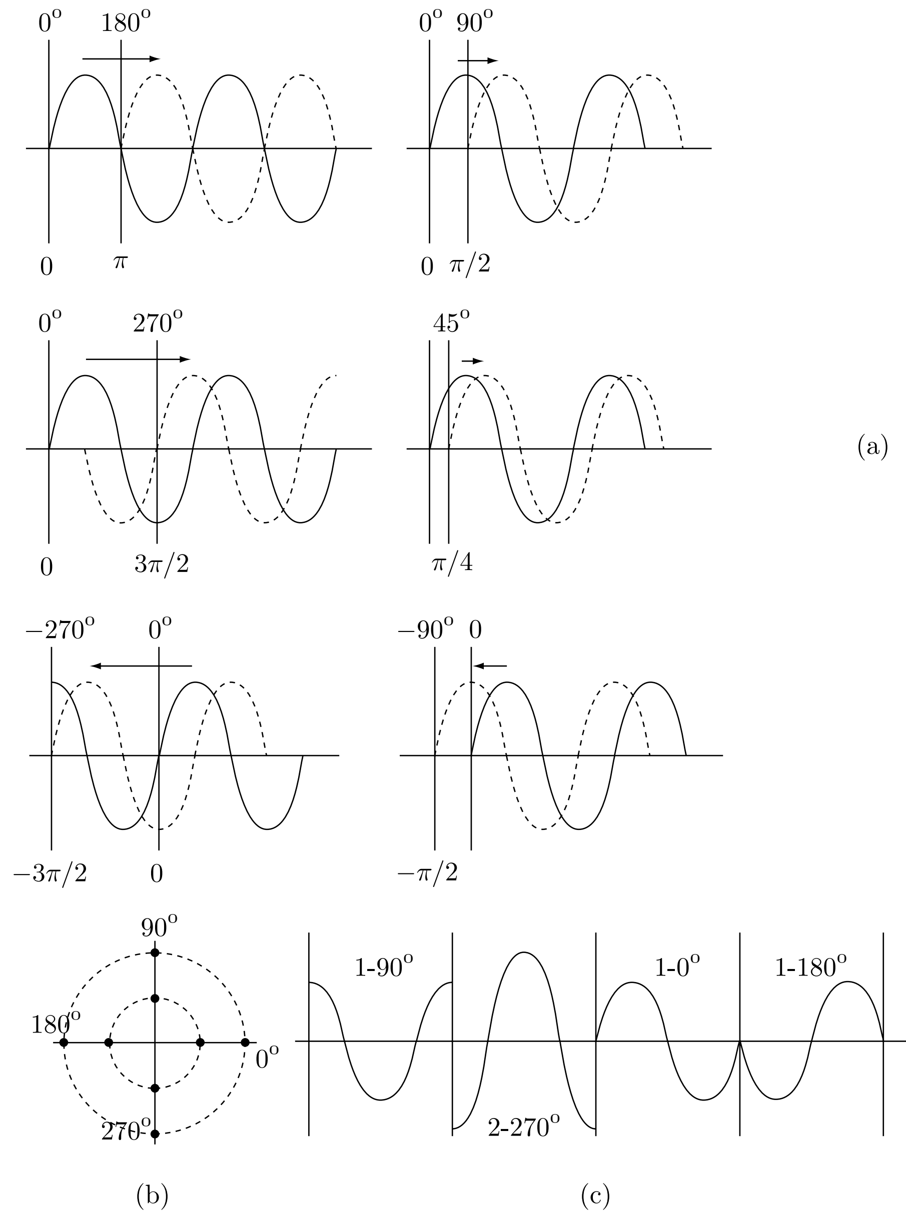



**Figure 3.50:** Quadrature Amplitude Modulation

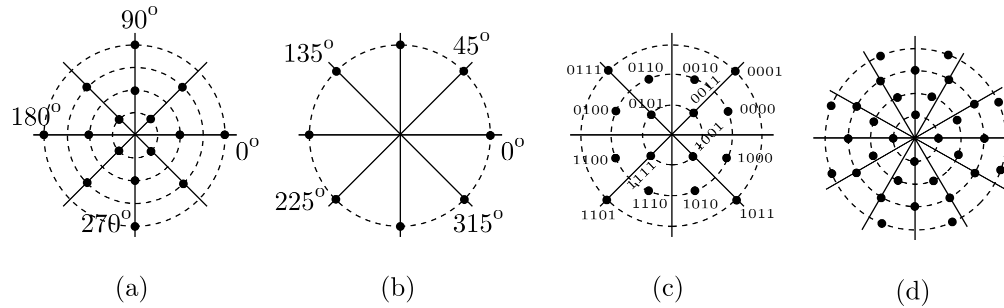



**Figure 3.52:** Various QAMs

receiving modem, however, has to use an A/D circuit to demodulate the wave, and such circuits are far from perfect. A typical D/A can resolve the incoming modulated wave to only seven bits per time unit, resulting in an effective bitrate of $8000 \times 7 = 56000$ bps.

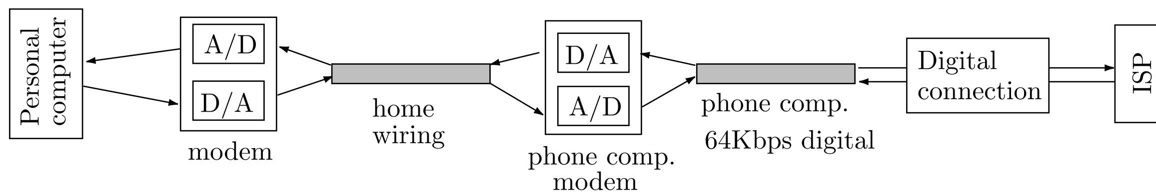



**Figure 3.53:** Analog and digital data paths for 56K communications

Figure 3.52 shows a typical data path between a home computer and an ISP's computer, using the (relatively new) digital wires. Uploading (from home to ISP) involves sending the data from the home computer to the D/A circuit in the home modem. From there, through wires in the home, to the A/D in the telephone company modem (normally located on the outside wall of the house, or on a nearby utility pole). This A/D circuit is the weakest point in the entire path. From that modem, the data is sent on a digital wire to the ISP's computer, which normally has direct digital connection.

Downloading (from ISP to home) is done in the opposite direction, so the only A/D circuit involved is the one in the home modem. In principle, we can purchase a high-quality modem with a high-precision A/D circuit. In practice, most modems have an average performance A/D, which becomes the weak link in the download path.

### 3.22.2 Receiving Bytes in Serial I/O

The next step in understanding serial I/O has to do with the packing of bits by the receiver. The serial interface on the transmitting side has no problem in unpacking 8-bit bytes and sending out a long stream of bits on the transmission line. The serial interface on the receiving side, however, has to break up the incoming stream of bits into groups of 8 bits and pack each group so that it can be sent to the receiving computer.

The problem stems from the fact that there is just one communication line, through which both the data and synchronization signals have to be sent. Thus, the receiver has no way of knowing when the transmission is going to start. When the receiver is first turned on, the line is either quiet or the transmission has already started and part of it may have been lost already.

Only rarely does it happen that the receiver is turned on at the very moment the transmission arrives from the transmitter. This is why it is important to make sure that the receiver can identify the beginning of the transmission or, at least, the beginning of some byte in the transmission.

This problem has two solutions, *synchronous serial I/O* and *asynchronous serial I/O*.

### 3.22.3 Synchronous Serial I/O

The main principle of this method is that the transmitter should transmit all the time, at a constant rate. This is not as simple as it sounds. The transmitter may be a keyboard, operated by a human. Even the

fastest human typist cannot possibly exceed 10 characters/s and maintain it for a while, when using a keyboard. This rate, however, is an extremely slow one for the hardware. From the point of view of the serial interface, such a keyboard *rarely* has anything to send. This is why the synchronous method uses another rule: When a synchronous transmitter has nothing to send, it sends the ASCII character `SYN` (code $0001\,0110 = 16_{16}$).

As an example, suppose the user types `SAVE 2,TYPE(cr)`. The serial interface on the keyboard would generate a sequence of ASCII codes similar to

← `16 16 16 53 41 16 56 45 16 20 16 32 2C 16 16 54 59 50 45 16 16 0D 16 16` ←

(in hex, see the ASCII code table on page 81). This way the transmission is sent at the same rate, even when there is nothing to transmit.

The receiver knows the rules of synchronous transmission and its only problem is, therefore, to identify the first bit of the first data byte (the `53` in our example). Once this is done, the receiver can simply receive the bits at the same constant rate, count modulo 8, and pack consecutive groups of eight bits into bytes. If a group turns out to be a `SYN` character, it is ignored by the receiver.

At worst, the receiver should be able to identify the first bit of a data byte (perhaps not the first data byte). In such a case, only part of the transmission would be lost. In the above example, if the receiver identifies the first bit of byte `56`, then it has lost the first two data bytes `53` and `41`, but the rest of the transmission can still be received correctly.

To enable the receiver to identify the first data byte, the transmitter starts each transmission with at least two `SYN` characters. Whenever it is turned on, the receiver automatically switches into a special mode called the *hunt mode*. In this mode the receiver hunts for a `SYN` character in the transmission. It starts by receiving the first eight bits from the line and storing them in a shift register. It then compares the contents of the shift register with the code of `SYN` ($00010110_2$). If the two match, the receiver assumes that the current eight bits are, in fact, a `SYN` character, and the next bit to be received is thus the first bit of a byte (we show later that this assumption is not always correct). If the comparison fails, the receiver inputs the next, shifts the register to the left, enters the new bit on the right end of the register, and compares again. The process is repeated until either a match is found or the transmission is over.

To illustrate this process, we use the example above. It starts with the following bits:

00010110 00010110 00010110 01010011 01000001 ...

(without the spaces). We assume that the receiver is turned on a little late, at the point marked with the vertical bar

00010110 00010|**110 00010**110 01010011 01000001 ...

The shift register will initially contain 11000010. The first three comparisons will fail and the receiver will read and shift the next three bits. The shift register will contain, in successive stages: 10000101, 00001011, and 00010110. After the third shift, the shift register will match the code of `SYN` and the receiver will switch to its normal mode. In this mode, the receiver (actually, the serial interface on the receiving side) receives bits in groups of 8, packs each group, and sends it to the computer or the I/O device on the receiving side. If a group equals 00010110, it is ignored.

Two things can go wrong in this simple process.

1. If the receiver is turned on too late, it may miss part of the transmission. In the case

00010110 00010110 0|**0010110 0**1010011 01000001 ...

the receiver will stay in the hunt mode until it finds the `16`

`16 16 16 53 41` **`16`** `56 45 16 20 ...`

and will therefore miss the first two data bytes `53` and `41`.

2. Sometimes the shift register may contain, during a hunt, the pattern 00010110 even though no `SYN` is involved. For example, if the receiver is turned on just before the word `TYPE` is received, it will see the stream `16 54 59 50 45 16` or, in bits 01011000|01011001.... The shift register will start with 01011000 and, after six shifts, will contain 00|010110. Two bits are left over from the `T` and the other six came from the `Y`. The receiver, however, will conclude that the shift register contains a `SYN` and will assume that the next bit (the seventh bit of the `Y`) is the first bit of the next byte. In such a case, all synchronization is lost and the rest of the transmission is corrupted.

To help the receiver in such cases, each byte should include a parity bit. When the receiver loses synchronization, many parity bits will be bad, and the receiver should ask for a retransmission of the entire

block.

### 3.22.4 Asynchronous Serial I/O

This method is based on a different principle. When the transmitter has nothing to send, it generates an unmodulated wave. The receiver receives it and interprets it as a uniform sequence (an idle sequence) of all zeros (or, possibly, of all ones). When the transmitter has a character to send, it first breaks the sequence of zeros by sending a single 1 (a start bit) followed by the eight data bits of the character.

The task of the receiver is to wait for the start bit (the break in the transmission) and to receive the data bits that follow. They are packed and sent, after a parity check, to the computer on the receiving side. To allow the receiver enough time to do the packing and checking, the transmitter sends at least two zeros (stop bits) following the data bits. After the two stop bits, the transmitter may either send a sequence of zeros or the next data byte (again sandwiched between a start bit and two stop bits). A typical example is

$$00\ldots001ddddddd p001ddddddd p00\ldots001ddddddd p000\ldots$$

Where *ddddddd* stand for the seven data bits and $p$ is the parity bit. The idle sequences can be as long as necessary, but each character is now transmitted as 11 bits. The seven data bits, the parity bit and the three start/stop bits.

Since each character now requires more bits, asynchronous transmission is, in principle, slower than synchronous transmission. On the other hand, the asynchronous method is more reliable since it can never get out of synchronization. If a character is received with bad parity, the rest of the transmission is not affected.

### 3.22.5 Serial I/O Protocols

The last step in understanding serial I/O is the description of a complete transfer, including data and control characters. Such a transfer follows certain conventions and must obey certain rules in order for the transmitter and receiver may understand each other. This convention, or set of rules, is called the *serial I/O protocol*.

There are many different protocols, and the one illustrated here is typical. When first turned on, the two serial interfaces at both ends of the transmission line switch themselves to the receiving mode. At a certain point, one of them receives data in parallel, from the computer, to be transmitted serially. It changes its mode to "transmit" and starts the protocol.

It may happen, in rare cases, that both serial interfaces start transmitting at the same time. In such a case, none will receive the correct response from the other side, and sooner or later, one side is going to give up and switch back to receiving mode. (Section 3.26.1 discusses methods for the prevention of such conflicts.)

The first step in the protocol is to determine whether the receiver is really connected at the other end and is ready to receive. The receiver may be disconnected, turned off, busy, or out of order. The transmitter therefore starts by sending an enquire (`ENQ`) character (ASCII code $05_8$). If the receiver is ready, it responds with an acknowledge (`ACK`) character (ASCII code $06_8$). If the receiver cannot receive, it responds with a negative acknowledge (`NAK`, code $15_8$).

If the transmitter receives a `NAK`, it has to wait, and perhaps try again later. If the response is positive (an `ACK`), the transmitter sends the first block of data, bracketed between the two control characters `STX` (start text, ASCII code $02_8$) and `ETX` (end text, code $03_8$). Each character in the data block has its own parity bit, the entire block (in fact, the entire protocol) is transmitted either in the synchronous or the asynchronous mode, and the receiver has to be preset, when first turned on, to the speed of the transmitter.

Following the `ETX` there is usually another character that provides a check, similar to a parity check, for the entire block. This character is called the *cyclic redundancy character* (CRC) and is discussed below (see also page 93).

Following the transmission of the first block, the receiver responds with either an `ACK` or, in case of parity errors, with a `NAK`. Upon receiving an `ACK`, the transmitter sends the next block, using the same format (`STX`, data bytes, `ETX`, CRC character). In response to a `NAK`, the transmitter retransmits the entire block. If the same block has to be retransmitted several times, the transmitter may give up and notify the computer or I/O device on its side by sending an interrupt signal.

After sending the last block, the transmitter waits for a response and, on getting an `ACK`, sends an

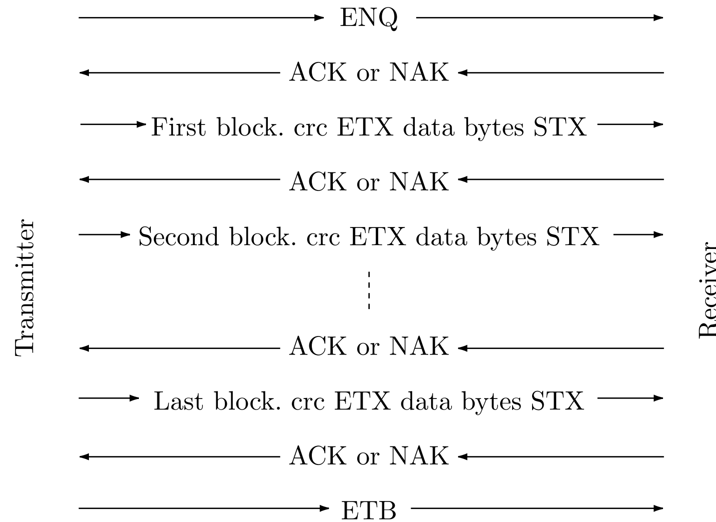



**Figure 3.54:** A typical serial protocol

`EOT` (end of transmission) character. Following the `EOT`, the transmitter switches its mode to receiving. Figure 3.53 shows a typical, simple protocol.

The CRC character is the output of the cyclic redundancy checking algorithm, a method similar to, but much more powerful than, parity. All the characters in the message block are treated as one long stream of bits representing a (large) binary number. That number is divided modulo 2 by another, pre determined binary number $b$, and the remainder is the CRC character. If $b = 256$, then the remainder is always in the range $0 \ldots 255$ and is therefore an 8-bit number. More information on the cyclic redundancy check method can be found in [Press and Flannery 88] and [Ramabadran and Gaitonde 88].

As an example of a simple protocol, we describe the Simple Mail Transfer Protocol (SMTP, [RFC821 82]), commonly used for email messages. We assume that the message is sent from a computer located on a network where there is a central computer (called the sender) that's responsible for all communications. The message is sent by the sender to another network (making several stops on the way, if necessary), where a similar computer (the receiver) receives it and sends it to the recipient computer. The main point is that the same message may be sent to several recipients served by the same receiver. In such a case, only one copy of the message data should be sent. The steps of the protocol are as follows:

1. The sender sends a MAIL command, indicating that it wants to send mail. This command contains a reverse path, specifying the addresses the command went through. The reverse path ends with the sender's IP address
2. If the receiver can accept mail, it responds with an OK reply.
3. The sender sends a RCPT commands, identifying one of the recipients of the mail.
4. If the receiver can accept mail for that recipient, it responds with an OK. Otherwise, it sends a reply rejecting that recipient, but not the entire email transaction.
5. The sender and receiver may negotiate several recipients this way.
6. When the recipients have been identified and accepted by the receiver, the sender sends the mail data, terminated with a special sequence.
7. If the receiver understands the data, it responds with an OK reply.

The following are the SMTP commands:

```
HELO <SP> <domain> <CRLF>
MAIL <SP> FROM:<reverse-path> <CRLF>
RCPT <SP> TO:<forward-path> <CRLF>
DATA <CRLF>
RSET <CRLF>
SEND <SP> FROM:<reverse-path> <CRLF>
```

```
SOML <SP> FROM:<reverse-path> <CRLF>
SAML <SP> FROM:<reverse-path> <CRLF>
VRFY <SP> <string> <CRLF>
EXPN <SP> <string> <CRLF>
HELP [<SP> <string>] <CRLF>
NOOP <CRLF>
QUIT <CRLF>
TURN <CRLF>
```

The following example (from [RFC821 82]) is a typical SMTP transaction scenario. It shows mail sent by Smith at host USC-ISIF, to Jones, Green, and Brown at host BBN-UNIX. Here we assume that host USC-ISIF contacts host BBN-UNIX directly. The mail is accepted for Jones and Brown. Green does not have a mailbox at host BBN-UNIX.

```
R: 220 BBN-UNIX.ARPA Simple Mail Transfer Service Ready
S: HELO USC-ISIF.ARPA
R: 250 BBN-UNIX.ARPA

S: MAIL FROM:<Smith@USC-ISIF.ARPA>
R: 250 OK

S: RCPT TO:<Jones@BBN-UNIX.ARPA>
R: 250 OK

S: RCPT TO:<Green@BBN-UNIX.ARPA>
R: 550 No such user here

S: RCPT TO:<Brown@BBN-UNIX.ARPA>
R: 250 OK

S: DATA
R: 354 Start mail input; end with <CRLF>.<CRLF>
S: Blah blah blah...
S: ...etc. etc. etc.
S: .
R: 250 OK

S: QUIT
R: 221 BBN-UNIX.ARPA Service closing transmission channel
```

### 3.22.6 Communications Lines

There are three types of communication lines, *simplex, half-duplex, and full-duplex*. A simplex line is unidirectional. Information can only be sent from the transmitter to the receiver and nothing, not even an acknowledge, can go in the other direction. Such a line can be used only in cases where the receiver is not expected to respond. Radio and television transmissions are common examples of simplex lines.

A half-duplex line is bidirectional but can only carry one transmission at a time. Thus, before the receiver can send a response, the transmitter has to send a control character (part of the protocol) to signal a change in the direction of the line. Such a direction change can be time consuming. After sending the response, the receiver should send another control character to again change the direction of the line, which slows down the whole transmission.

A full-duplex is bidirectional and can handle transmissions in both directions simultaneously. Both sides may transmit data on such a line at the same time without the need to change directions. Such lines are, of course, faster.

The following sections discuss terminology, protocols, and conventions used by modern modems:

### 3.22.7 The V.xxx Modem Standards

Since modems are made by many manufacturers, there is a need for standards in this area. The ITU-T, part of the International Telecommunications Union (ITU, Section 3.13), is the international organization responsible for developing standards in all areas of telecommunications. It has developed several standards, known as the "V" protocols, for modems.

Any `V.xxx` modem standard specifies speed, data compression, or both. Reliable connections (especially high-speed ones) require both modems to support the same standards. V.FC (Fast Class) is an early version of the 28.8Kbps asynchronous international communications standard, designated V.34. Because of market competitiveness, V.FC modems have been rushed to the market while the ITU-T was slowly and carefully working on the design and development of V.34. Those who released the early V.FC modems were participants of the ITU-T Study Group who cooperatively developed V.34. Today there is no reason to buy a modem that supports only V.FC

V.32bis: Allows raw data transfers of up to 14.4Kbps.

V.42: Provides real-time error detection.

V.42bis: Data compression that can push V.32bis to 57.6Kbps.

V.32terbo: AT&T's standard to push V.32bis beyond its ITU (International Telecommunications Union) limits.

V.32FC: Developed by Rockwell International as a precursor to V.34 (which was bogged in committee for many months). Similar to V.34, but uses a modified handshaking protocol.

V.34: 28.8Kbps without compression. V.34 can top out at 115.2Kbps with compression. The V.34 standard operates at a top speed that is twice that of the previous generation of modems (which execute the V.32-bis protocol and operate at 14,400 baud), and three times the speed of the generation before that (V.32 protocol, for 9,600 baud modems).

It is rare to get consistent 28.8Kbps connections. Speeds of 28.8kbps require pristine phone line quality along the entire length of the connection. V.34 modems are capable of automatically changing the connection speed to 21.6K, 24K, and even 26.4Kbps.

The V.34 standard is smart enough to perform the so-called "channel probe." This is a frequency response and signal-to-noise ratio test of frequencies at various points across the bandpass. During the modem handshake, the modems send a series of tones to each other, at known signal levels and specific frequencies. The modem calculates the level of the received signal at each frequency, and therefore can determine the maximum bandwidth available for use.

During normal operation, a V.34 modem goes through a "retrain" from time to time. A retrain means the two modems suspend operations and renegotiate the best possible connection all over again. The V.34 standard calls for probes 25 frequencies across the channel. Because the frequencies are spaced closely, the frequency response profile (i.e., the channel probe) is accurate. That is a main reason why V.34 connections are reliable (there is accurate line problem detection). The channel probe occurs during initial modem negotiation, and during training and retraining. Additionally, line noise and the line's signal-to-noise ratio is remeasured continually during the connection.

A V.34 modem also does a cooperative and nearly instantaneous speed shift (also called a "fallback"), which hosts can better tolerate. This rate renegotiation procedure allow rapid switching ranging from 4.8Kbps up to 28.8Kbps, as line conditions vary.

In spite of all this a V.34 modem can still suffer from many errors because of a noisy phone line, and may automatically disconnect itself (drop the line). Here are some things you can do to get a better connection.

Try calling a different location. Line quality differs from region to region, and noise may be completely different at a different location, with a different modem.

Try connecting with a local call. Sometimes the connections within a long distance call can cause impairments. (If this isolates the problem, you can try switching long-distance companies.)

Try plugging the modem to a different phone line or wall jack.

Try eliminating all telephone extensions, phone line surge suppressors, line switches, answering machines, fax machines, and anything else on the line with the modem.

If you know someone else in your area with a high-speed modem, ask what type of connections they make. Try making the connection from their location. If you encounter the same low connection rates,

the problem may be resulting from impairments along the lines running to the local telephone company or within your home or office. Your telephone company or a private consultant may be able to help.

Modem Data Compression. No matter how fast the modem, file transfers will always go faster with compression, unless the file is already compressed. There are two basic standards for compression: V.42bis and MNP5. MNP compression is older, but today's modems support both methods.

V.90: The original standard for 56Kbaud modems. In practice, speeds reached by this protocol are only about 36Kbaud.

V.92: A new protocol for 56K modems currently (summer 2000) being considered for ratification by the ITU-T. The four main features of this protocol are as follows:

1. "Modem-on-hold." This feature allows the modem and the server to enter a sleep mode, restoring the data without the need to redial the server. This is useful in cases where a single phone line is used for both data transfers and speech. When the modem uses the line and the user notices an incoming call on the same line, the user places the modem on hold and accepts the call.

2. A "Quick Connect" feature reduces the time spent on negotiating a connection between a modem and a server. To implement this feature, the modem has to "learn" the capabilities of the home phone line and its maximum transmission rates.

3. Data transmission rates (as opposed to receiving) are improved. Generally, modems receive data faster than they transmit. Current data transmission rates on 56Kbaud modems are about 28.8Kbaud. The V.92 standard will be able to improve them to about 40Kbaud, depending on line conditions.

4. A new, efficient data compression standard, V.44, has been developed by Hughes Network Systems in parallel with V.92. It has been specifically designed for the compression of HTML documents, and it is expected to increase the effective data receiving rates of V.92 modems to over 300Kbaud, about double that of the standard V.42bis protocol. This will very likely speed up web surfing by 20-60%, and perhaps up to 200% in some extreme cases

Modem Error Control. Modern modems support the error correction standards V.42 and MNP 2-4. Any of these standards assures that your data will be received as accurately and as quickly as possible.

Table 3.54 is a summary of the most important modem standards.

## 3.23 Modern Modems

The story of the modern modem starts with Dennis C. Hayes, a telecommunications engineer formerly with Financial Data Sciences from Georgia. Hayes started making modems in the 1970s, and founded Hayes Microcomputer Products, Inc., in 1978. The company makes modems and other products under the brand names of Hayes and Practical Peripherals. Hayes' main idea was to make *programmable modems*. A programmable device is one that can receive and execute commands. All modern modems are Hayes compatible and execute commands based on Hayes' original design. A typical modem can execute upwards of 100 commands.

In 1981, Hayes started making the Smartmodem 300, the first intelligent modem that executed Hayes Standard AT Command Set. In 1982, the Smartmodem 1200 was introduced. In 1987, Hayes introduced the Express 96, a 9600 baud modem. In 1988, the Hayes Smartmodem 9600 became the first Hayes V.32 modem introduced for use with data communications networks.

A modern modem is a complex machine that performs many tasks. However, since it does not contain any moving parts, it is easy to manufacture once a prototype is ready. This is why, by the middle 1980s, there were many modem makers competing with Hayes, developing new ideas, standards and products. Currently, the main modem makers are AT&T, Global Village, Motorola, Multi-Tech Systems, Racal Electronics, Supra, and U.S. Robotics.

A modern modem performs the following tasks:

- Modulation/Demodulation. This is the main task.
- Dialing the phone (also answering it, see below).
- Sending tones to the modem on the other side (and receiving tones) to determine the quality of the phone line at several frequencies.
- Creating and verifying error-correcting codes.

| standard | bitrate (bps) | signaling rate | modulation technique |
|---|---|---|---|
| 103 | 300 | 300 | FSK |
| V.21 | 300 | 300 | FSK |
| 202 | 1200 | 300 | FSK |
| 212A | 1200 | 600 | DPSK |
| V.22 | 1200 | 600 | DPSK |
| 201 | 2400 | 1200 | DPSK |
| V.22bis | 2400 | 600 | QAM |
| V.26ter | 2400 | 1200 | DPSK |
| 209 | 9600 | 2400 | QAM |
| V.29 | 9600 | 2400 | modified |
| V.32 | 4800 | 2400 | QAM |
| | 9600 | 2400 | TCM |
| V.32bis | 14400 | 2400 | modified |
| V.32terbo | 19200 | 2400 | QAM |
| V.34 | 28800 | 2400 | TCM |
| | | 3000 | TCM |
| | | 3200 | QAM |
| | | 2743 | TAM |
| | | 2800 | TCM |
| | | 3429 | TCM |
| V34+ | 33600 | | QAM |
| V.90 | 56600* | | QAM |

* 56K receiving, 33K transmitting

**Table 3.55:** Common modem standards.

- Compressing the data sent. Decompressing data received.

- Negotiating with the modem on the other side features such as baud rate, synchronous/async, and the precise method for data compression and error-correction.

- Renegotiate the transmission speed "on the run," depending on how many errors have been received.

Answering the phone is one important feature of modern modems. You can leave your computer running, start a communications program, send the command "ATS0=2" to the modem, and it will answer the phone after two rings. This is important for an individual wanting to set up a web server, and also for a computer center servicing remote users. The character string `AT` is the modem's *attention* command. The string `S0` is the name of an internal register used by the modem to decide when (after how many rings) to answer the phone.

The newest features in modems today are speeds of 56,600 baud (achieved mainly through the use of efficient compression), cable-tv modems, and simultaneous voice/data modems (DSL). DSL (Section 3.24) is such a promising technology that many predict that the current generation of modems will be the last one and modems will be completely phased out in the future.

See also this book's web site for a very detailed modem glossary and dictionary. Another useful source of information on modems is `http://www.teleport.com/~curt/modems.html`.

Modern modems can also send and receive fax. See `http://www.faximum.com/faqs/fax` for more information on fax modems.

## 3.24 ISDN and DSL

ISDN (Integrated Services Digital Network) is an international telecommunications standard for transmitting voice, video, and data over digital lines at a speed of 64Kbaud. The telephone companies commonly use a 64Kbps channel for digitized, two-way voice conversations. ISDN service is available in most parts of the U.S.

ISDN uses 64 Kbps circuit-switched channels, called B (or "bearer") channels, to carry voice and data. It uses a separate D (for "delta") channel for control signals. The D channel is used to signal the telephone company computer to make calls, put them on hold and activate features such as conference calling and call forwarding. It also receives information about incoming calls, such as the identity of the caller.

DSL (short for Digital Subscriber Line) is a technology that dramatically increases the data transmission rates of the ordinary telephone lines used in homes and offices. Speeds achieved by DSL depend on the distance between the customer and the telephone exchange. There are two main types of DSL. Asymmetric DSL (ADSL) is suitable for Internet access, where fast downstream (from server to client, i.e., from web site to user) is required, but slow upstream is acceptable. Symmetric DSL (including technologies such as SDSL and HDSL) is suitable for short connections where high speeds in both directions are needed.

Unlike ISDN, which is also digital but travels through the switched telephone network, DSL data signals travel from the customer's home to the nearest telephone exchange (see Table 3.55 for distance limitations), and is then sent on special lines to the ISP.

An important, useful feature of DSL is that it is always on. Data sent with DSL from the client's computer, arrives at the telephone exchange, is aggregated in a unit called the DSL Access Multiplexor (DSLAM), and is then forwarded to the appropriate ISP or data network. The same happens with data that arrives in the exchange in the opposite direction, on its way to the client.

DSL is a relatively new technology, having started in the late 1990s. In spite of this, there currently are many DSL versions, the most important of which are summarized here.

Asymmetric DSL (ADSL) shares the same line as the telephone. Voice is transmitted on the line in low frequencies and data is transmitted at higher frequencies. However, a POTS splitter (plain old telephone system or service) must be installed on the customer's premises to separate the frequencies. A version of ADSL—known variously as "G.lite," "Universal ADSL," "ADSL Lite," and "splitterless ADSL"—is geared to the consumer. It eliminates the splitter and associated installation costs, but all telephones on the line must plug into lowpass filters to isolate them from the higher ADSL frequencies.

Because of the use of different frequencies, ADSL allows digital data to "ride over" the same wires as voice frequencies. The signals are combined and split apart at both sides. At the customer's site, the splitting is done either with an external device installed by the telephone company, or it is built into the DSL modem.

High Bit Rate DSL (HDSL) is a symmetric technology (i.e., has the same transmission rate in both directions). HDSL is currently the most mature DSL variety, because it has been used to provide T1 transmission over existing twisted-pair line without requiring the additional provisioning required for setting up T1 circuits. HDSL requires two cable pairs and is limited to a range of 12000 feet, while HDSL-2 requires only one cable pair and works at distances of up to 18000 feet. HDSL does not allow line sharing with analog phones.

Symmetric DSL (SDSL) is similar to HDSL. It uses only one cable pair and is offered in a wide range of speeds from 144Kbps to 1.5Mbps. SDSL is a rate adaptive technology, and like HDSL, SDSL cannot share lines with analog telephones.

Table 3.55 and Figure 3.56 summarize the main features and components, respectively, of symmetric and asymmetric DSL (PSTN stands for "public switched telephone network").

## 3.25 T-1, DS-1 and Their Relatives

Digital telephones transmit voice signals in digital form (bits) at a speed of 64Kbaud. This is designated DS-0 (sometimes called DS-0 channel, the term "DS" stands for "data service"). The next step is DS-1 (or more accurately DS-1/T-1). In North America, the notation T-1 indicates the line (a shielded, twisted pair of wires) and DS-1 indicates the signal, 24 DS-0 channels bundled on the single T-1 line. DS-3/T-3 consists of a fiber optic cable and is 30 times faster, offering transfer rates of 44.736 Mb of digital data per second.

Asymmetric DSL (can share line with analog telephone)

| Type | Maximum upstream speed | Maximum downstream speed | Cable pairs | Maximum distance |
|---|---|---|---|---|
| ADSL | 1M | 8M | 1 | 18000 |
| RADSL | 1M | 7M | 1 | 25000 |
| G.lite | 512K | 1.5M | 1 | 25000 |
| VDSL | 1.6M | 13M | 1 | 5000 |
| | 3.2M | 26M | 1 | 3000 |
| | 6.4M | 52M | 1 | 1000 |

Asymmetric DSL (can share line with phone)

| Type | Upstream and downstream speed | Cable pairs | Maximum distance |
|---|---|---|---|
| HDSL | 768K | 2 | 12000 |
| | 1.544M | 2 | 12000 |
| | 2.048M | 3 | 12000 |
| HDSL-2 | 44M (T1) | 1 | 18000 |
| | 2.408M (E1) | | 18000 |
| SDSL | 1.5M | 1 | 9000 |
| | 784K | 1 | 15000 |
| | 208K | 1 | 20000 |
| | 160K | 1 | 22700 |

**Table 3.56:** Features of symmetric and asymmetric DSL

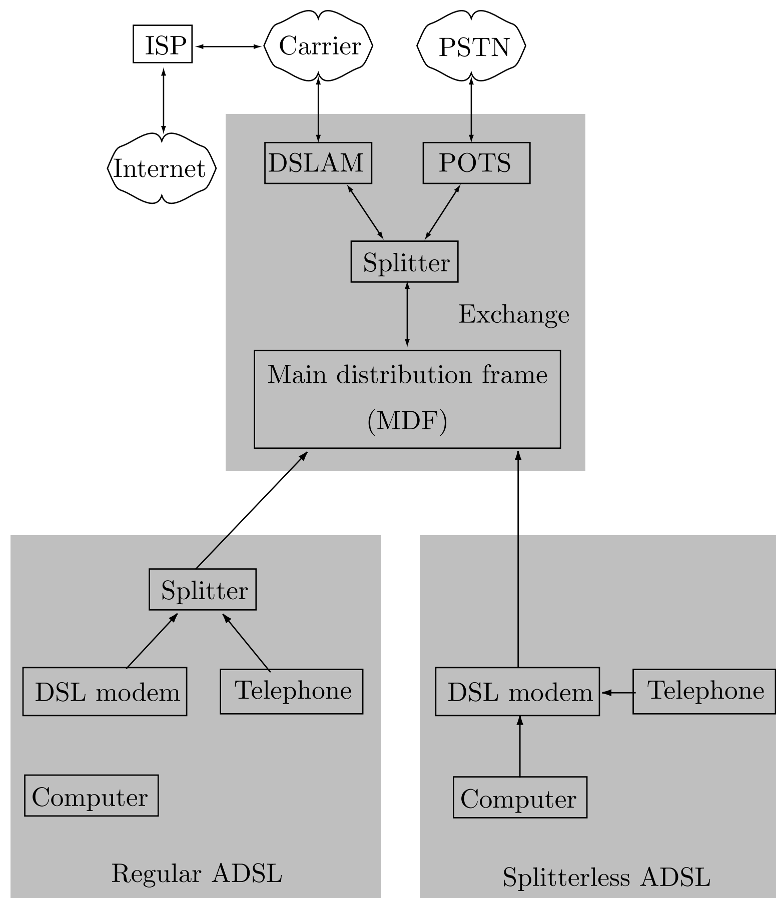



**Figure 3.57:** Main components of DSL

For some historical reason, T-2 has been bypassed altogether. The rest of the world sends a bundle of 30 DS-0 signals on an E-1 line and calls this a DS-1/E-1 channel.

In North America, the following terms are used to indicate standards of serial transmissions:

| | |
|---|---|
| DS-1(T-1) | 1.544 Mbps 24 DS-0 (voice channels). |
| DS-1C(T-1C) | 3.152 Mbps 48 DS-0 (voice channels). |
| DS-2(T-2) | 6.312 Mbps 96 DS-0 (voice channels). |
| DS-3(T-3) | 44.736 Mbps 672 DS-0 (voice channels). |
| DS-4(T-4) | 274.176 Mbps 4032 DS-0 (voice channels). |

The following terms are used in Japan:

| | |
|---|---|
| DS-1 | 1.544 Mbps 24 DS-0 (voice channels). |
| DS-2 | 6.312 Mbps 96 DS-0 (voice channels). |
| DS-3 | 32.064 Mbps 480 DS-0 (voice channels). |
| DS-4 | 97.728 Mbps 1440 DS-0 (voice channels). |
| DS-5 | 400.352 Mbps 5760 DS-0 (voice channels). |

In Europe the following applies:

| | |
|---|---|
| DS-1 (E-1) | 2.048 Mbps 30 DS-0 (voice channels). |
| DS-2 (E-2) | 8.448 Mbps 120 DS-0 (voice channels). |
| DS-3 (E-3) | 34.368 Mbps 480 DS-0 (voice channels). |
| DS-4 (E-4) | 139.268 Mbps 1920 DS-0 (voice channels). |
| DS-5 (E-5) | 565.148 Mbps 7680 DS-0 (voice channels). |

North American Standards: A T-1 is only one way of bundling DS-0s together. In general, control data has to be transmitted in addition to the voice signals. This data is needed for call connection, caller ID, and record-keeping/billing information. On a T-1 line, this data is transmitted by robbing 8 Kbaud from the 64 Kbaud of each of the 24 DS-0 channels, effectively reducing them to 56 Kbaud channels. This is called a "robbed bit DS-1." The degradation in voice quality is not noticeable.

Most of us still have analog telephone lines used in our homes. Such lines are referred to as analog Central Office (CO) lines. In the 1970's, in response to the explosive growth of personal computers and computer communications, AT&T decided to convert the entire telephone network to digital. This was how ISDN (Integrated Services Digital Network), was developed. Initially, ISDN was not popular, but it started to grow together with the internet. There currently are two types of ISDN connections as follows:

Type 1. This ISDN connection is common in residences and businesses. It is called a Basic Rate Interface (BRI) and consists of two 64 Kb voice lines and one 16 Kb data line on one pair of regular phone cable. It is sometimes called 2B + D (two bearer plus data). This is slowly catching on among internet users because the two bearer lines can be bound together to form one 128 Kb line, but a special modem is needed. It is also possible to send and receive slow data such as e-mail over the 16 Kb data line without the phone lines being tied up or even making an actual phone call. ISDN BRI uses the same phone wire as regular phones but transmits digitally, so a different card is needed at the Central Office or at the company's PBX (Private Branch Exchange). Also, a special phone is needed, or a special box to plug regular telephones into.

Type 2. This type connects the customer's PBX to the telephone company's central office. It is called a Primary Rate Interface (PRI) and consists of a DS-1 connection. An ISDN PRI connection comes on a T-1 type circuit, but there are many more services offered to the customer. Also, an ISDN PRI is not a robbed bit service. It provides 23 DS-0 channels and uses the 24th channel as the data channel for all 23 DS-0s.

SONET (Synchronous Optical NETwork) is a fiber optic network standard that telephone company central offices use to communicate. It originated in the US but has been adopted by the CCITT as an international standard, so equipment made by different manufacturers would work together. SONET defines how different pieces of equipment interface and communicate, but not what they say to each other. As a result, different types of networks can be used on a sonet ring. With sonet, the term OC-1 connection is used instead of DS-3/T-3 connection (the term "OC" stands for "optical carrier"). Currently, sonet connections

are very expensive and are consequently used only by large customers. The different OC speed standards are as follows:

| | | |
|---|---|---|
| OC-1 | 52 Mbps | 28 DS-1's or 1 DS-3. |
| OC-3 | 155 Mbps | 84 DS-1's or 3 DS-3. |
| OC-9 | 466 Mbps | 252 DS-1's or 9 DS-3. |
| OC-12 | 622 Mbps | 336 DS-1's or 12 DS-3. |
| OC-18 | 933 Mbps | 504 DS-1's or 18 DS-3. |
| OC-24 | 1.2 Gbps | 672 DS-1's or 24 DS-3. |
| OC-36 | 1.9 Gbps | 1008 DS-1's or 36 DS-3. |
| OC-48 | 2.5 Gbps | 1344 DS-1's or 48 DS-3. |
| OC-96 | 5 Gbps | |
| OC-192 | 10 Gbps | |

ATM (Asynchronous Transfer Mode) is a packet switching network that runs on SONET. It runs at the OC-3 and OC-12 speed. ATM has been slated by CCITT as the future backbone of Broadband ISDN or B-ISDN, but it could also be used as a private network. ATM is not even a finished or officially adopted standard but manufacturers are currently making products promising that they will comply with the finished standard when it is approved. The products are still very expensive, but higher end customers may be looking at it. ATM would be their private network and they would connect to other offices through a leased OC-3 or OC-12 connection. A cheaper alternative may be for a company to buy connections to a public ATM network and just use the ATM network as the company's WAN backbone.

## 3.26 Computer Networks

Early computers could only send their output to a printer, a punched card, or a punched paper tape machine. Magnetic tapes and drums were added in the 1950s, and magnetic disks have been used since the second generation of computers. In the 1960s, when computers became more popular, computer designers and users started thinking of ways to send information between computers, to make computers communicate. Two things are necessary for computer communication, a communication line and programs that can send and receive information in a standard way.

> Network: Anything reticulated or decussated, at equal distances, with interstices between the intersections.
>
> —Samuel Johnson, *Dictionary of the English Language*

Today, the Internet (Section 3.27) is an important part of our lives, but local area networks are also important, so this section starts with a discussion of LANs.

### 3.26.1 Local Area Networks

When several computers are located within a short distance (a few hundred yards) of each other, it is possible to connect them with a cable, so that they form a *local area network* (LAN). Such a network connects computers in the same room, in the same building, or in several nearby buildings. A typical example is a multipurpose building on a university campus, with offices, classrooms, and labs, where a cable runs inside the walls from room to room and from floor to floor, and all the computers are connected in a LAN.

To reduce interference, a special type of cable, such as a coaxial cable or a twisted pair, has to be used. Each computer on the network sends output on the wire, or receives input from it, as if the other end of the wire was connected to an I/O device. The only problem is *conflicts*, the case where several computers try to send messages on the same wire at the same time. This problem can be solved in a number of ways, two of which are described in Sections 3.26.2 and 3.26.3.

The discussion of serial I/O in Section 3.22 implies that every computer on a LAN has to have a serial port installed. The problem of conflicts has to be solved somehow, and this also requires special hardware. As a result, each computer in a LAN must have a special piece of hardware called a *network card* or a *communications interface*. This hardware acts as a serial port, packing and unpacking bytes as necessary, and also resolves conflicts in communications over the network.

### 3.26.2 Ethernet

This type of LAN was developed in 1976 by Bob Metcalfe of Xerox [Metcalfe and Boggs 76], and is currently very popular. A single cable, called the *ether*, connects all the computers through an *ethernet interface* (also called an ethernet interface) in each computer. The ethernet interface is a special piece of hardware that performs the data transfers. It executes the ethernet protocol and knows how to avoid conflicts.

A computer sends a message to its ethernet interface as if the interface were an I/O device. The interface listens to the ether to find out if it is in use. If the ether is in use, the interface waits until the ether is clear, then sends the message immediately. It is possible, of course, that two or more interfaces will listen to the ether until it becomes clear and then send their messages simultaneously. To solve this problem, an interface has to monitor the ether while it is sending a message. If the signal on the ether is different from the one the interface is sending, the interface stops, waits a random time interval, then tries again.

Once a message is sent on the ether, it is received by every other ethernet interface on the network. The message must therefore start with an address, which each receiving interface compares with its own. If the two addresses are not equal, the interface ignores the message, otherwise, it received it and sends it, as input, to its computer.

An ethernet network can easily reach speeds of about 10 Mbaud.

Ethernet addresses are 48 bits long, organized in six bytes. They must be unique, so they are assigned by the IEEE. Each ethernet manufacturer is assigned three bytes of address, and the manufacturer, in turn, assigns these three bytes, followed by a unique, three-byte number, to each ethernet card they make.

▸ **Exercise 3.19:** How many ethernet cards can a manufacturer make?

Ethernet messages must have the following format:

1. An 8-byte preamble.
2. A 6-byte destination address.
3. A 6-byte source address.
4. A 2-byte ‘type’ field.
5. The data itself (must be between 45 and 1500 bytes).
6. A 4-byte crc field.

Figure 3.57a shows an ethernet installed in a three-story building. The ethernet cable goes from room to room and from floor to floor, strung inside the walls and ceilings. Computers in various rooms are hooked up to this cable by short lines, resembling telephone wires. Notice that only one cable is needed, regardless of the size of the network.

The downside of this configuration is that if the main cable breaks or gets damaged, the entire network may be affected. This is why practical ethernets are normally arranged as in Figure 3.57b. One room (a communications closet) is dedicated to the local network, and the ethernet cable is strung in that room. Each computer is connected to the network by a (long) cable run from its room to the main (short) cable. Such a network is sometimes referred to as a *hub*, but this is a misleading term, since ethernet is linear, not circular.

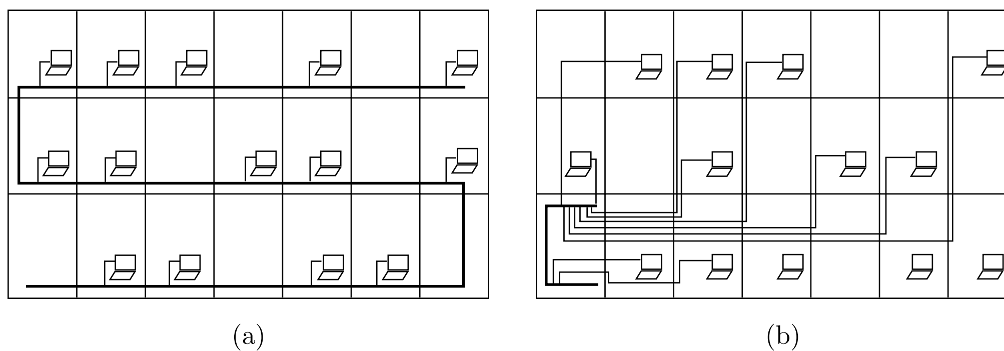



**Figure 3.58:** (a) Ethernet, (b) ethernet with hub

**History of Ethernet**

The following are excerpts from the draft version of the upcoming book, *Überhacker!*, by Carolyn Meinel.

The history of Ethernet is important because this is a networking technology that's currently used on the vast majority of all the local area networks on this planet.

May 22, 1973, at the Xerox Palo Alto Research Center (PARC), the world's first Ethernet LAN transmitted its first packet (chunk of data). The proud inventors were Bob Metcalfe and David Boggs. For years they labored in the laboratory to improve their invention. By 1976 their experimental network was connecting 100 devices.

The turning point came in 1979. That year Gordon Bell of Digital Equipment Corp. (DEC) phoned Metcalfe to suggest that they work together to make a commercial product out of Ethernet. Metcalf's employer, Xerox, loved the idea. DEC would build Ethernet hardware, and Intel would provide chips for DEC's Ethernet network interface cards (NICs). The idea was that this trio of industrial titans would keep the technology to itself, so that anyone who would want to use Ethernet would have to buy the equipment from their combine.

There was one problem with this idea—if Ethernet were to become the dominant networking technology someday, this combine would violate US antitrust laws designed to curb monopolies. Back then, no one used Ethernet outside the laboratory. So for these people to be thinking about the danger of becoming a monopoly was either arrogant—or prescient.

Metcalfe, Bell and associates chose to avoid an Ethernet monopoly. They began working with the Institute of Electronics and Electrical Engineers (IEEE) to create an open industry standard for Ethernet. That meant that anyone would be free to create and sell Ethernet hardware or design network operating systems that would use it. Persuading Xerox, DEC and Intel to make Ethernet free for anyone to build, ensured that Ethernet would become inexpensive and widely available. For this they deserve credit for creating one of the keystones of today's Internet. In June of 1979, Metcalfe left Xerox to found 3Com Corp. By March 1981, 3Com shipped its first Ethernet hardware to the public. Ethernet had finally emerged from the laboratory.

In 1982, 3Com shipped its first Ethernet adapter for a personal computer—the "Apple Box." Some 18 months later 3Com introduced its first Ethernet internal card, the Etherlink ISA adapter for the PC. This card used "Thin Ethernet" cabling, a technique that is still popular today.

In 1983, the IEEE published the Ethernet standard, 802.3. Xerox turned over all its Ethernet patents to the nonprofit IEEE, which in turn will license any company to build Ethernet hardware for a fee of $1000. This was yet another act of corporate generosity that helped make Ethernet the most widely used local area networking technology. In 1989, the Ethernet standard won international approval with the decision of the International Standards Organization (ISO) to adopt it as standard number 88023.

Why all this history? The important thing with Ethernet is that it became a world-wide recognized standard in 1989. That means if you set up an Ethernet LAN in your home, you can be certain that much of what you learn from it will work on Ethernet LANs anywhere else on the planet. Also, if you ever invent something truly wonderful, please remember this story and make your invention freely available to the world, just as Metcalfe and Boggs did.

### 3.26.3 A Token Ring LAN

This type of LAN was developed at about the same time as the ethernet. Each interface is connected to a ring through a 1-bit buffer that it can read and write. The ring consists of a cable, with both ends connected together. When the network starts, the network controller, a special piece of hardware, injects a special binary pattern, called the *token*, into the ring. The token is continuously rotated, so each interface can see the individual bits. When an interface has a message to send, it first makes sure that the entire token is rotating on the ring. This means that no other interface is using the network. The interface then damages the token by inverting its last bit. This guarantees that no other interface will try to use the ring. The interface can now send the message, bit by bit, into the ring. Following that, the interface sends the original token bits, enabling other interfaces to use the ring.

The message should be sent in a special code, to guarantee that no message should contain a bit pattern identical to the token.

### 3.26.4 Wide Area Networks

When the computers in a network are separated by long distances, private lines cannot be used, either because they are too expensive to install or because of legal problems. In such a case, the network uses telephone lines, either voice-grade lines, or special lines that provide higher transmission rates.

A problem with long range transmission is that computer-generated signals are physically different from the signals sent on telephone lines. The solution is to either use a modem (a device that can translate between the two forms of signals) or use special lines, dedicated to data transmission.

The development of modems and special communications lines made it possible to build wide area networks (WANs). However, what made today's large networks possible—and, in hindsight, inevitable—is the human need to communicate. People like to communicate; a fact proven again and again when the telephone, radio, television, fax machine, and computer networks came into being.

In the 1960s, ARPA (a U.S. government agency that supports advanced research projects) commissioned Bolt, Beranek, and Newman (BBN) Inc., a consulting firm in Cambridge, Mass., to design and build an experimental network that became known as the ARPANET, to explore the possibilities of connecting computers made by different manufacturers, and to develop standards for computer communications. The nodes initially connected by ARPANET were the following: On the east coast, BBN, Harvard University, MIT, and MIT's Lincoln lab. In Cleveland, Case Western Reserve University. In Pittsburgh, Carnegie-Mellon University. In Salt Lake city, The University of Utah. On the west coast, SRI (in menlo park), Stanford University, UCLA, UC Santa Barbara, the RAND Corporation in Santa Monica, and SDC (system development corporation) in Los Angeles. ARPANET started operations in 1968, was immediately recognized as a success, and BBN had to develop standards for message routing and for the allocation of addresses.

Such a standard is called a *protocol*. It is a set of rules that two computers must follow in order to exchange data reliably. It describes how the data should be organized, and what other information (such as parity bits) should be sent with the data.

Over time, many nodes on the ARPANET became centers of small networks, necessitating more standards for the interoperation of the smaller networks with ARPANET. The combined standards became known as the Internet Protocol (IP). When other networks were established, mostly in the early 1980s, they also used IP to interoperate. The worldwide collection of all the networks that are interconnected and use IP, is known today as the Internet.

URL `http://www.isoc.org/guest/zakon/Internet/History/HIT.html` is an excellent source for the history of the internet. In addition, The Internet Society (ISOC) and its technical groups such as The Internet Engineering Task Force (IETF), have web sites with much information.

The Internet is made up of member networks (also called backbones, Section 3.28), each of which may consist of smaller networks (subnetworks or regional networks) that have still smaller networks as members, and so on, for several levels. At the bottom level there are small networks whose members are individual computers. Currently, many millions of computers around the world are connected this way to various networks, to become *nodes* in the Internet. Some networks are local and may consist of a few computers located in the same building or in the same installation; others may consist of a large number of computers spread worldwide.

As an example of the current nesting of networks, imagine a user on the campus of the Colorado State University, in Fort Collins, Colorado. He is part of a small network consisting of his department or school on campus. This small network is, in turn, part of the overall campus network, the CSUNET. The CSUNET is implemented by means of an ethernet cable connecting the individual LANs on campus to one computer, the network server, that is dedicated to network operations. The CSUNET is part of the SUPERNET, the State of Colorado network, that connects campuses and research institutes throughout Colorado. Thus, there is a dedicated fiber optic cable connecting the gateway on the CSU campus to the University of Colorado, Boulder. Every message coming out of the CSU campus goes on that cable to Boulder and, from there, either to another part of the SUPERNET, or outside, to one of the gateways of the Internet. SUPERNET itself is part of WESTNET, a network covering the western states of Arizona, Colorado, Southern Idaho, Utah, New Mexico, and Wyoming. WESTNET, in turn, is one of the regional networks of the NSFNET,

and is connected to the NSFNET through gateways in Boulder (at the National Center for Atmospheric Research, NCAR), and Salt Lake City (at the University of Utah). The NSFNET is a member network of the Internet.

In addition to IP, three more protocols are currently used in the Internet. They are collectively known as TCP/IP (Transmission Control Protocol. Internet Protocol) and are fully described in RFC-1140.

- Simple Mail Transfer Protocol (SMTP), is the Internet standard protocol for transferring E-mail messages.

- File Transfer Protocol (FTP), is used to transfer files between Internet nodes.

- Telnet is the standard protocol for remote terminal connection. It allows a user to interact with a remote computer as if he were connected to it directly.

The point where a network is connected to another one used to be called a *gateway*, but the term *network server* is now common. This is usually a computer dedicated to message routing, but a server can also be a custom piece of hardware. It may be connected to one or more nodes in each of the networks, and it must have some knowledge of the organization of both, so it can route each message in the right direction.

A message is sent from a node to the nearest server. It is then forwarded from server to server, until it reaches a server connected to the destination node. The message is sent as a packet, and the entire process is known as *packet switching*. The term 'packet,' however, is loosely used and may describe different things. On the Internet, a packet is a block of data, also called a *datagram*, whose format is defined by the IP.

Some servers on the Internet have a total picture of the network. They know how to reach every domain on the Internet. They are called *root servers*, and are updated each time a domain is added to or deleted from the Internet. Every other server has to get up-to-date information from a root server periodically.

An interesting special case is when one of the networks is small, such as a university campus. A network server in such a network knows all the nodes on the network, so it can easily route each incoming message to its destination. However, all outgoing messages may be routed by the server to the same node in the larger network, where there is more information about the outside world, helping to decide where to forward the message.

When all the nodes of a network are physically close (within a radius of up to a few thousand meters), they can be connected directly, without the need for modems and phone lines, to form a local area network. Examples are a department located in the same building, or a small facility housed in a few adjacent structures.

The Internet as a whole is managed by two centers. The Information Sciences Institute (ISI) of the University of Southern California, located in Los Angeles, and ICANN (Section 3.30), a US government corporation. ISI is in charge of developing standards and procedures. ICANN is the Internet Assigned Numbers Authority. It maintains the Network Information Center (InterNIC), where users can connect to get information about available and assigned addresses and about existing and proposed standards.

## 3.27 Internet Organization

How does a network of computers work? How is information sent from a source computer and gets to a destination computer? A good way to understand this is to think in terms of *layers*. A network is made up of layers, the lowest of which is the hardware. All other layers consist of software. Lower layers consist of programs that use the hardware directly. Programs in higher layers invoke the ones in the lower layers, and are easy to use, since they don't require detailed knowledge of how the network operates.

The lowest layer, the hardware, is, of course, necessary. Information can be sent, in bits, on wires and can be created and processed by hardware circuits. Hardware, however, is not completely reliable, and is tedious to use directly (think of the difference between machine language and higher-level languages). This is where the low software layers come in. Software is necessary to decide how to route the information, and to check and make sure that all the bits of a message have been received correctly.

Computer networks resemble the phone network in a superficial way because of the following:

- They use existing phone lines to send information, and they rent dedicated lines from telephone and telecommunications companies.

■ Using a network from a computer is similar to the way we use a phone. A connection has to be opened first, information is then moved both ways; finally, the connection is closed.

It therefore comes as a surprise to learn that computer networks operate more like the postal service than the phone network. When $A$ calls $B$ on the phone, the phone network creates a direct connection between them, and dedicates certain lines to that connection. No one else can use these lines as long as $A$ and $B$ stay connected. These two users monopolize part of the phone network for a certain duration. On the other hand, when $A$ sends a letter to $B$, the postal service does not dedicate any routes or resources to that letter. It sorts the letter, together with many others, then sends it, by truck or plane, to another post office, where it is sorted again and routed, until the letter arrives at its destination.

We say that the phone system is a *circuit switched* network, whereas the postal service uses *packet switching*. Computer networks also use packet switching. A user does not monopolize any part of the network. Rather, any message sent by the user is examined by network software. Depending on its destination address, it is sent to another part of the network where it is examined again, sent on another trip, and so on. Eventually, it arrives at its destination, or is sent back, if the destination address does not exist.

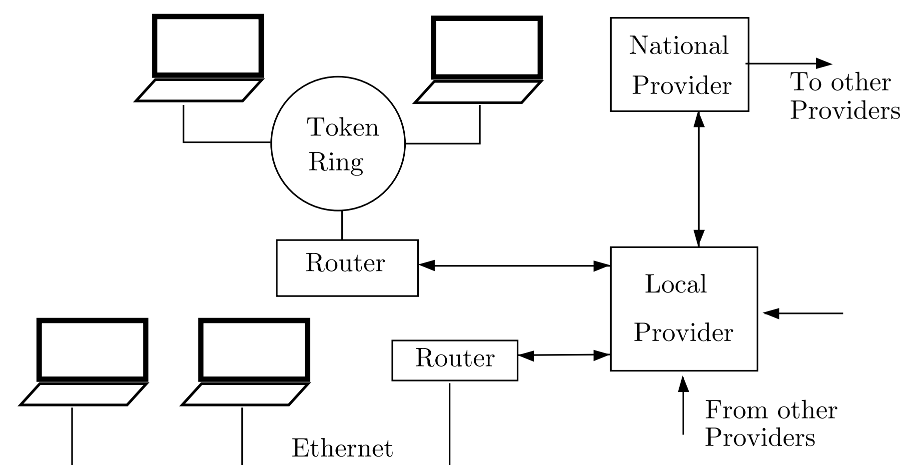



**Figure 3.59:** Internet organization

This structure is illustrated by Figure 3.58. It shows two local networks, a token ring and an ethernet, connected to a local provider. This may be a computer at a local university or a local computer center. The connections consist of dedicated lines run between computers called *routers*. A router is a computer performing all the network operations for a local network. A router is normally dedicated to network applications and does not do anything else.

The router at the local network provider is connected to a number of local routers, and to at least one bigger network provider, perhaps a national one. The national network providers are connected with high-speed dedicated *backbone* lines, to complete the network.

### 3.28 Internet: Physical Layout

The key to understanding the physical structure of the Internet is to think of it as a hierarchical structure. At the bottom of this hierarchy are the internet users (PCs, workstations, mainframes). Each is connected to a local Internet Service Provider (ISP), which is the next higher level in the hierarchy. The local ISPs are in turn connected to regional ISPs (or regional backbone providers), which are in turn connected to national and international backbone providers. These providers are the highest level in the hierarchy and are connected together at interconnect points. New branches and connections can be added to any level.

An Internet backbone (sometimes called a network) is a chain of high-speed data lines. In a national or international backbone, these lines span great distances. The national backbones typically have high-bandwidth transmission links, with bandwidths of 1Gigabit/s and higher. Each such backbone also has numerous hubs that interconnect its links and at which regional backbone operators can tap into the national

backbone. In a regional backbone, the lines span a limited region, such as a city, a county, a state, or several states. In order to provide communications throughout the internet, each backbone must be connected to some other backbones at several interconnect points. These points are also called Network Access Points (NAPs), Metropolitan Area Exchanges (MAEs) points, and Federal Internet Exchange (FIX) points.

The ten major internet backbone providers in the US (as of mid 2002) are UUNET/WorldCom (27.9%), AT&T (10.0%), Sprint (6.5%), Genuity (6.3%), PSINet (4.1%), Cable & Wireless (3.5%), XO Communications (2.8%), Verio (2.6%), Qwest (1.5%), and Global Crossing (1.3%).

Several backbones (we'll call them networks) may be connected at an interconnection point, but not every network at the point is connected to every other one. The owner of a network $A$ may have agreements with the owners of networks $B$ and $D$ but not with the owners of network $C$ to exchange data at an interconnection point. Such agreements are called *peering*. By peering, networks $A$ and $B$ open their data lines to one another, while $A$ and $C$ do not communicate even though they are both connected to the interconnection point.

An interconnection point may be owned by a commercial entity or by a government (the San Francisco interconnect point is owned by Pacific Bell, the New York interconnect point is owned by SprintLink, and the Maryland interconnect point is owned by the federal government). The point consists of computers and routers that are needed to receive and transmit data via high-speed data lines. The lines themselves are owned by local and long-distance telephone carriers. The two types of high-speed lines currently in use are data service (DS), and optical carrier (OC). The various speeds of these lines are discussed in Section 3.25. There are more than 40 national backbone operators operating in the United States today (2000). A national backbone operator in the US is generally defined as one that has operations (points of presence, POPs) in at least five states and has at least four peering agreements as access points. Most national backbone operators in the US also have lines that stretch coast to coast.

Once the main lines of a national backbone have been installed by a national backbone operator, the operator extends the backbone by placing Points of Presence (PoPs) in a variety of communities. Low-speed data lines are leased by the operator to users in the community, turning the operator into the ISP of these users. Physically, the PoP is a router and most users use a telephone to dial to the router in order to connect themselves to the internet. The PoPs become part of the backbone and the national backbone becomes a part of the Internet by peering with similar national backbones at interconnect points.

The two magazines [ispworld 2002] and [boardwatch 2002] are good references for ISPs and Internet backbones.

One step below the national backbones are the regional backbones. Such a backbone extends over a single state or an area that covers a few states. A regional backbone is connected to the Internet through a national backbone (or even at an interconnect point, if one is available in its area). The regional backbone operator does business by setting up local PoPs and leasing low-speed lines to ISPs.

The ISPs, the next lower level in the Internet hierarchy, exist between the backbone providers (national or regional) and the end users (the dial-up home and small business owners). A typical ISP may offer Internet connection to some hundreds of users for $15–20 a month. The ISP has modems, a router and a server. The server is a computer dedicated to communications. It is the domain name server and it is responsible for assigning dynamic IP numbers to the many customers. The router receives the actual data from the customers and sends it to the server, to be sent to the router at the backbone. The router also receives data from the server and sends it to the right customer.

An ISP can be a small, "mom and pop" local company, but it can also be part of a university campus or a large corporation. Even a backbone operator can serve as an ISP (Pacific Bell is a good example). [List 2000] is a URL listing many thousands of local, regional, and backbone ISPs. Another source of information is [Haynal 2000].

Considering how the Internet is organized, it should be obvious that any computer hooked up to the Internet can serve as an ISP. A small, private computer user who pays an ISP can become a PoP and start providing Internet service to others by simply purchasing a router, several modems, and server software. Such a service would likely be too slow, but it is possible in principle and it shows how the Internet is organized and how it grows.

The lowest level in the Internet hierarchy is the end user. An end-user connects to the Internet through a dial-up connection or a dedicated line. The dedicated line can be DSL (Section 3.24), a television cable,

or a special leased line. The latter requires a server computer on the user's end, and can cost thousands of dollars a month.

Once the physical layout of the Internet becomes clear, we realize that the various ISPs and backbone operators own part of the Internet, its physical part. This part of the Internet grows and spreads all the time without the need for approval from a higher authority. The other part of the Internet, the protocols used by the software to transfer the information, is not owned by anyone, but is subject to approval.

> The most beautiful thing we can experience is the mysterious.
>
> —Albert Einstein
>
> Who owns the internet? No one.
> Where is it? Everywhere.
>
> Sounds mysterious? that's because it's beautiful.

### 3.28.1 Internet Protocol (IP)

We continue with the analogy of the post office. In order for a letter to arrive at its destination, it must include the receiver's address at a certain place on the envelope. It should also have the sender's address, in a different place, in case it has to be sent back. Similarly, a message sent on the network must have both the receiver's and sender's addresses, and they must appear at certain places. Since the addresses are examined by software, they should be numeric, not strings of characters.

The Internet therefore specifies a set of rules that tell users how and where to include addresses in a message. These rules are called the *Internet Protocol*, or IP. A message generated in a computer is sent by it to the router of its local network. The router may be connected to only one other router (at the local provider), in which case it must send all messages to that router. The router at the local provider executes IP programs that examine the address, and decide how to route the message. This process continues, and the message 'hops' from router to router. At a certain point, the message arrives at a national router, which forwards it to a smaller, local provider. From there it is sent to a local network, where the local router finally sends it to the destination computer (or waits for the destination computer to query it, looking for messages).

IP addresses are 32-bit numbers, allowing for up to $2^{32}$ computers (that's about a billion) connected to the Internet. Each computer on the Internet has a unique IP address (but see the discussion of domain names in Section 3.28.3). Each address is divided into four groups of 8 bits each. When an organization joins the Internet, it receives the most significant part of its number (the network part) from ICAN (Section 3.30). The organization then allocates the least significant part (the host part) to its individual members. For large organizations, the network part is short, leaving more room for the host part.

A 32-bit IP address is divided into a most-significant network part and a least-significant address part in one of three ways, designated A, B, and C:

Class A addresses start with a single bit of 0, followed by a 7-bit network address, followed by a 24-bit address. This allows for 128 networks (only 126 are allowed in practice, because 0 and 127 are reserved), each having $2^{24} \approx 16.7$ million addresses. An example is address `96`, which has been assigned to the Army Finance and Accounting Office (ASN-JTELS) located at Fort Ben Harrison in Indianapolis, Indiana (zip code 46249-1136).

Class B addresses start with the two bits 10, followed by a 14-bit network address, followed by a 16-bit address. This allows for $2^{14} = 16$K networks, each having $2^{16} = 64$K addresses. An example is address `128.166`, which has been assigned to Boeing Aerospace Corporation (NET-BAC-NET) located at Kent Space Center, 20403 68th Avenue, South Kent, Washington 98032.

Class C addresses start with the three bits 110, followed by a 21-bit network address, followed by an 8-bit address. This allows for $2^{21} \approx 2$ million networks, each having $2^8 = 256$ addresses (actually, only 254 addresses). An example is address `192.100.5`, which has been assigned to Zitel Corporation of Milpitas, California.

The four octets normally specify '`bigsubnet.smallsubnet.lan.user`'. For example, the Northridge campus of the California State University (CSUN), has been assigned a '`bigsubnet.smallsubnet`' code of `130.166`.

With 32 bits, there can be $2^{32} \approx 4.3$ billion IPs. This is a large number, but the number of available IPs is steadily shrinking. In the future, the IP size may be increased to 128 bits, resulting in $2^{128} \approx 3.4 \times 10^{38}$ IPs, a confortably large number.

▸ **Exercise 3.20:** With about six billion people on Earth (mostly poor), why are so many IP numbers needed?

Another part of the IP standard is the packet size. A long message is broken, by the IP programs at the sending router, into packets that are sent independently. Each packet contains the two addresses and looks like a complete message. Long packets mean fewer packets, and less work for the routers. However, an error may be discovered in a packet when it is received, requiring a retransmission. This is why packets should not be too long. Also, when a network is heavily used, users sending fewer long packets would get better service than users sending many short ones. For fairness, IP limits the packet size. Typically, packets do not exceed about 1500 bytes.

### 3.28.2 Transmission Control Protocol (TCP)

Packets sent in a certain order may be sent along different routes, and may arrive out of order. Also, packets may be lost, or may get damaged along the way. Here again the post office is a good analogy. This is why IP is enough in principle, but not in practice. Another protocol, the transmission control protocol (TCP), is used on the Internet to help with packet organization.

TCP software at the sending router breaks a single long message into packets, and adds a serial number and a checksum to each packet. At the receiving router, similar software combines arriving packets according to their numbers, and checks for missing or bad packets.

Most network literature talks about IP and TCP as if they were on the same level. In fact, TCP 'exists' on a layer higher than IP. A very detailed reference for IP and TCP is [Comer 89].

The combined efforts of IP and TCP create the effect of Internet resources being dedicated to each user when, in reality, all users are treated equally. Note that, in practice, most users don't know anything about IP/TCP, and use software on a higher layer. A typical program used for Internet communications, such as a web browser, starts with a menu of standard Internet operations, such as Email, Telnet, FTP, Finger, and Whois (see [Krol 94] for details). The user selects an option, types in an Internet address to connect to, or a file name to send, and the software does the rest.

### 3.28.3 Domain Names

All software used for communications over the Internet uses IP addresses. Human users, however, rarely have to memorize and type such numbers. We normally use text-based addresses like `http://www.JohnDoe.com/`. Such an address is called a *universal resource locator* (URL). The part `JohnDoe.com` is called the *domain name*. The domain name system was developed in 1984 to simplify email. It has grown exponentially and it is currently used to identify Internet nodes in general, not just email addresses. Here are some reasons for why domain names are important:

1. People find it more convenient to memorize and use text rather than numbers. We find it more natural to use `http://www.JohnDoe.com/` instead of an IP number such as `123.234.032.321`.

2. IP numbers may change, but internet users prefer to stay with the same domain names. An organization with URL `http://www.GeneralCharity.org/` (where the domain name is `GeneralCharity.org`) may move its headquarters from town $A$ to city $B$. Its location on the internet will change, it will switch to a different internet service provider (ISP), and so will be assigned a different IP address. It prefers, however, to keep its domain name.

3. Since an IP address is 32 bits, there can be $2^{32}$ IP addresses. This is a large number (about 4.3 billion), but the amount of available IP addresses is dwindling fast because of the rapid growth of the Internet. The number of valid domain names, however, is much bigger, and it is possible to associate several domain names with one IP address.

This is why domain names are important and why the internet needs two central data bases, one with all the assigned IP addresses and the other with all the assigned domain names. Internet standards and software implementors should also make it easy for internet software to find the IP address associated with any domain name.

URL `http://ws.arin.net/cgi-bin/whois.pl` can be used to find out who owns a given IP address or a chunk of addresses. This URL is part of ARIN (see below). Users need to input an IP address or

its most-significant part. Try, for example, `130.166`. An alternative is the specialized search engine at `http://www.whonami.com/`.

The data bases are part of the InterNic (network information center) and are maintained by Network Solutions Inc., under a contract to ICANN (Section 3.30). Address translation is provided by numerous computers around the world. Such a computer is a *domain-name server* (DNS) and is normally dedicated to this task. When a user inputs a domain name into any internet software, such as a browser, the software has to query a domain-name server to find the IP address associated with that domain name. Any individual domain-name server is familiar with only some of the many existing domain names, typically the names and numbers of all computer users in its corner of the Internet. As a result, a domain-name server must sometimes send a query to other name servers. The various DNSs should be able to communicate with each other and especially with the "root" DNSs, which are part of the InterNic. These root DNSs are: `A.ROOT-SERVERS.NET.`, `B.ROOT-SERVERS.NET.`, `C.ROOT-SERVERS.NET.`, `D.ROOT-SERVERS.NET.`, and `E.ROOT-SERVERS.NET.`.

If the DNS used by computer $X$ has a problem and stops working, computer $X$ will still be connected to the internet, but its software won't be able to resolve any domain names into IP addresses, and would therefore stop functioning. This is why each domain name should be known by at least two DNSs. When a new domain name is registered with ICANN, the registrar (a company authorized by ICANN) must place the name in at least two DNS computers. A list of accredited registrars can be found at `http://www.icann.org/registrars/accredited-list.html`.

As an example, the registrar `Domains.com` maintains the two DNSs `NS1.DOMAINS.COM` (at IP address 208.169.214.68) and `NS2.DOMAINS.COM` (at IP address 208.169.214.25).

Notice that the domain name is not owned by the registrar. It has an owner (an individual, a corporation, an institute, an organization, or the government) that's responsible for paying the fees to the registrar to keep the name current. The owner has to interact with ICANN through a registrar. The owner selects an available domain name through the registrar and pays the registrar to register the name with ICANN.

The owner then looks for an ISP and becomes a client of the ISP. The ISP gets a chunk of IP addresses from ICANN and assigns them to its clients. The owner sends the registrar its new IP address, and sends the ISP its domain name. The registrar's DNS computers associate the new IP address with the owner's domain name. When a message is sent to the owner, the sender knows just the domain name. That name is sent by the software (such as a browser or an email program) to several DNSs until one of them finds the name and its associated IP address. The message is then sent with the IP address (and also the domain name) over the Internet. When the message arrives at the ISP, the ISP's computers route it to the right client based on the domain name, since the ISP may have allocated the same IP address to several of its clients. Here are three variations on this simple theme:

1. An internet-savvy individual $A$ has discovered that domain name `abc.com` is available. The name is registered by $A$ immediately, even though he does not need it. He figures that there is (or will be) an individual or an organization $B$ whose initials are `abc` and they might be interested in this name. *Selling* the name is what $A$ hopes to do. Since $A$ does not need to use the name, he only has to *park* it, so that anyone interested would be able to find it. Many registrars provide this service for a small fee or for free. Anyone browsing that domain name will find a "For Sale" sign on it. There are even companies that will conduct a public auction for a desirable name such as `business.com`. Notice, however, that it is illegal to register someone else's registered trademark (such as IBM) as a domain name (such as `ibm.com`), then try to sell it to them. Such behavior is known as *cyber-squatting*, and it even includes reserving names of celebrities as domain names.

[For a list of domain names that are currently for sale see `http://www.domainnames-forsale.net/`.]

2. An individual already has a web page, such as `http://www.ibm.com/larry`, at his place of work. He now decides to have a side business. He registers the domain name `MonkeyBusiness.com`, but wants to *forward* anyone surfing there to his original web page at IBM without the surfer's knowledge. Some registrars provide a forwarding service for a small fee.

3. It is expensive to get authorized as a registrar by ICANN (see "How to Become an ICANN-Accredited Registrar" at `http://www.icann.org/registrars/accreditation.htm`). A small company that wants part of the act but cannot afford to be a registrar, can become a *subregistrar*. A company $A$ may attract clients by offering low prices. When a client wants to register a domain name, $A$ pays a registrar $B$ to actually

register the name. *A* then charges the client for the registration, parking, forwarding, disk space for a web page, web-page design, and other services.

As an example of point 3, here is a domain that was registered by its owner at `DomainsAreFree.com` (a subregistrar). They, in turn, referred it to Networks Solutions Inc., a registrar that actually registered it with the InterNic. This information is easy to find at `http://www.internic.net/whois.html`.

Domain Name: BOOKSBYDAVIDSALOMON.COM
Registrar: NETWORK SOLUTIONS, INC.
Whois Server: whois.networksolutions.com
Referral URL: www.networksolutions.com
Name Server: NS1.DOMAINSAREFREE.COM
Name Server: NS2.DOMAINSAREFREE.COM
Updated Date: 29-mar-2000

When a domain name moves to a different IP address, its owner has to notify the registrar to update their DNS computers. When a domain name moves to another DNS, the owner (through the registrar) has to notify InterNic. The InterNic data base for domain names is then updated, to associate the domain name with the new DNSs. The "root" InterNic DNS servers are also updated, by Network Solutions Inc., of Herndon, Virginia, USA, so they can provide reliable referrals to other DNSs.

A large organization may maintain its own DNS. Local administrators can then modify their DNS information in order to block access of users from the organization to certain domain names.

It is clear that the quality of internet communications depends heavily on the ISP. This is why Inverse Internet Technology Inc. regularly provides ratings of some of the big name ISPs, using measures such as "average time to login," "call failure rate," etc. It is possible to get quite a bit of information from their assessment, which is located at `http://www.inversenet.com/products/ims/ratings/`.

With so many web pages and domain names, how can anyone find anything? The answer is: Search engines. Several companies maintain computers that scan the internet and collect the contents of millions of web pages. Anyone can use those search engines freely. Anyone who owns a domain name, has a web page, and wants to advertise it, can submit their URL to many of these search engines. Here are two examples. Altavista is one of the most popular search engines. At the bottom of their initial page "`http://www.altavista.com/`" there is an "`Submit a site`" menu item that leads to information on adding and removing URLs from their search engines. The Yahoo start web page also has a button labeled "`How to suggest a site`" which has a similar function.

A domain name may consist of letters, digits, and hyphens. It may not start or end with a hyphen. The maximum size is 63 characters (this tells us that the sizes are stored in the InterNic data base as 6-bit numbers, with size zero reserved). The number of valid domain names is thus huge. The number of strings of one letter is 26. There are $26^2 = 676$ two-letter strings, and the number of strings of 63 letters is $26^{63} \approx 1.39\times 10^{89}$. This number is much greater than our estimate of the number of elementary particles in the observable universe and is not much smaller than the legendary googol. Here is an example of a really long domain name

`this-is-the-real-domain-name-that-I-have-always-wanted-all-my-life.com`

The domain name is just one of the fields that make up a URL. The general format of a URL is specified by RFC1738, and it depends on the particular protocol used. As an example, the URL syntax for an `http` protocol is `http://site.name:port/path#fragment`, where:

1. `http:` (which stands for *Hypertext Transfer Protocol*) is the protocol that web browsers and web servers use to communicate with each other.

2. `site` is any string of letters, digits and periods. Often, it is just `www`, but long site names such as `hyperarchive.lcs` (used in `http://hyperarchive.lcs.mit.edu/`) and `www.irs` (used in government URL `http://www.irs.ustreas.gov/`), are common.

3. `name` is a domain name or an IP address.

4. `:port` is a port number to connect to. The default value of `:port` for an `http` URL is 80.

5. `path` is a directory path, consisting, as usual, of subdirectory names separated by slashes. The last name may be a file name.

6. If the last name in the `path` is a file name, then `#fragment` is the point (in the file) that should be placed by the browser at the top of the screen (if possible). If `#fragment` is not used, the browser positions the top of the file at the top of the screen.

The URL syntax for the `ftp` protocol is `ftp://site:<password>@name:port/path`. It is similar to `http` with the following differences:

1. The default value of `name` is `anonymous`.

2. `:<password>@` is an optional password for sites that have limited access. The default value of `:password@` is the email address of the end user accessing the resource.

3. The default port is 21.

Table 3.59 lists some popular internet protocols.

| Name | Description |
|---|---|
| ftp | File Transfer protocol |
| http | Hypertext Transfer Protocol |
| RTSP | Real-time streaming protocol |
| gopher | The Gopher protocol |
| mailto | Electronic mail address |
| news | USENET news |
| nntp | USENET news using NNTP access |
| telnet | Reference to interactive sessions |
| wais | Wide Area Information Servers |
| file | Host-specific file names |
| prospero | Prospero Directory Service |
| dict | accessing word definitions |

**Table 3.60:** Various internet protocols

**Using water as an organic network between two computers.**

StreamingMedia is an interactive data sculpture that employs a new Internet protocol (H20/IP) I developed that uses water to transmit information between computers. H20/IP functions in a similar way as TCP/IP but focuses on the inherent viscous properties of water that are not present in traditional packet networks. The StreamingMedia demonstration of H20/IP exists as an installation of two computers at different heights where one captures an image and transmits it to the second computer in the form of modulated water drops. The project attempts to show how digital information can be encoded and decoded into organic forms to create a physical network between digital devices.

—Jonah Brucker-Cohen, Media Lab Europe, 2002, `jonah@coin-operated.com`

IANA (the internet assigned numbers authority, Section 3.30) is the organization responsible for establishing top-level domain identifiers.

Any extension to the domain name, such as `com` or `edu`, is a top-level domain name (TLD). There are three types of top-level domains:

1. Generic top-level domains (gTLDs). For years, these were just `com`, `net`, and `org`. On 16 November 2000, ICANN approved the three TLDs `biz`, `info`, and `name`. The gTLDs are not affiliated with any country and are "unrestricted"—anyone from anywhere can register them. In the future there may be more generic top level domains, such as `arts`, `shop`, `store`, `news`, `office`, `lib`, `private`, and `sex`. Notice that gTLDs are not limited to three letters.

2. Limited top-level domains. The original ones are:
`edu` for educational institutions.
`int` for international entities (`http://www.iana.org/int.html`)
`gov`, for US government agencies (`http://www.nic.gov/`)
`mil` for US military bases and units (`http://nic.mil/`).

On 16 November 2000, ICANN approved the seven additional TLDs `museum`, `aero`, `coop`, `biz`, `info`, `name`, and `pro`. Only certain entities can register domain names with these TLDs.

3. Country-specific domains. They consist of two letters such as `au` for Australia and `cn` for China. These are operated by separate registry authorities in 184 different countries. About one-half of these country-specific domains are "unrestricted"—anyone anywhere can register them, just like `com`, `net`, and `org` (although some are expensive). The rest are "restricted" and require a local presence and/or company documentation. Table 3.60 lists some of these codes and Table D.1 lists all of them.

Popular country codes are `at` (Austria), `cc` (Cocos Island, formerly Keeling), `nu` (Niue), `to` (Tonga), `md` (Moldavia), `ws` (Samoa, stands for "web site"), and `tv` (Tuvalu). Somewhat less popular are `ac`, `li`, `sh`, `ms`, `vg`, `tc`, `gs`, `tf`, `ky`, `fm`, `dk`, `ch`, `do`, and `am`. Try, e.g., `http://go.to/`, `http://stop.at/`, `http://vacation.at/hawaii`, and `http://www.tv/`.

See `http://www.iana.com/cctld.html` and `http://www.iana.com/domain-names.html` for more information on top-level domains.

The US Domain Registry is administered by the Information Sciences Institute of the University of Southern California (USC-ISI). The main ISI web page is at `http://www.isi.edu`. Detailed information about the US domain can be found at `http://www.nic.us/`.

ARIN (American Registry for Internet Numbers) is a nonprofit organization established for the purpose of administration and registration of IP addresses for the following geographical areas: North America, South America, the Caribbean, and sub-Saharan Africa. Their web site can be found at `http://www.arin.net/` and a WhoIs search is provided at `http://www.arin.net/whois/index.html`.

ARIN is one of three Regional Internet Registries (RIRs) worldwide, which collectively provide IP registration services to all regions around the globe. The others are: RIPE NCC—for Europe, Middle East, and parts of Africa and APNIC—for Asia Pacific.

| Country | A 2 | A 3 | Number | Country | A 2 | A 3 | Number |
|---|---|---|---|---|---|---|---|
| AFGHANISTAN | AF | AFG | 004 | CENTRAL AFRICAN REPUBLIC | CF | CAF | 140 |
| ALBANIA | AL | ALB | 008 | CHAD | TD | TCD | 148 |
| ALGERIA | DZ | DZA | 012 | CHILE | CL | CHL | 152 |
| AMERICAN SAMOA | AS | ASM | 016 | CHINA | CN | CHN | 156 |
| ANDORRA | AD | AND | 020 | CHRISTMAS ISLAND | CX | CXR | 162 |
| ANGOLA | AO | AGO | 024 | COCOS (KEELING) ISLANDS | CC | CCK | 166 |
| ANGUILLA | AI | AIA | 660 | COLOMBIA | CO | COL | 170 |
| ANTARCTICA | AQ | ATA | 010 | COMOROS | KM | COM | 174 |
| ANTIGUA AND BARBUDA | AG | ATG | 028 | CONGO | CG | COG | 178 |
| ⋮ | | | | ⋮ | | | |
| SOLOMON ISLANDS | SB | SLB | 090 | VIRGIN ISLANDS (U.S.) | VI | VIR | 850 |
| SOMALIA | SO | SOM | 706 | WALLIS AND FUTUNA ISLANDS | WF | WLF | 876 |
| SOUTH AFRICA | ZA | ZAF | 710 | WESTERN SAHARA | EH | ESH | 732 |
| S. GEORGIA & S. SANDWICH | GS | SGS | 239 | YEMEN | YE | YEM | 887 |
| SPAIN | ES | ESP | 724 | YUGOSLAVIA | YU | YUG | 891 |
| SRI LANKA | LK | LKA | 144 | ZAMBIA | ZM | ZMB | 894 |
| ST. HELENA | SH | SHN | 654 | ZIMBABWE | ZW | ZWE | 716 |

**Table 3.61:** Some ISO country codes

## 3.29 CSUN in the Internet

The Northridge campus (CSUN) of the California State University system (CSU) has more than 8000 computers, ranging from a mainframe (IBM S/3-90), to powerful workstations (and a parallel computer), to personal computers. With all this, the entire campus is just one small dot (a node) in the internet. Figure 3.61 shows how CSUN is connected to the rest of the internet. A single communications line connects the campus to the southern California hub of the 4CNET in Los Alamitos. (There is also a backup line connecting the campus to Qwest, Inc., in Anaheim.) This is an OC-3c line (optical carrier, 155Mbits/s). The 4CNET connects the CSU campuses and community colleges in California, and has another hub in northern

California at the California State Hayward campus. That hub connects the Humboldt, Sonoma, Chico, Sacramento, Stanislaus, San Jose, and Monterey bay campuses, as well as many community colleges. It is connected to the rest of the internet through UUNET (which is owned by MCI Inc.). The southern California hub connects the San Luis Obispo, Channel Islands, Bakersfield, Northridge, LA, Dominguez Hills, Pomona, San Bernardino, Fullerton, San Marcos, and San Diego campuses (and many community colleges). This hub has an OC-48 Sonet ring line (2.4Gbits/sec). The communications lines range from T1 (1.544 Mbits/sec) to OC-3c. The 4CNET hub in Los Alamitos is, in turn, connected to the rest of the Internet through UUNET (MCI). There is also a cable connecting the Bakersfield, Fresno, and Stanislaus campuses, and another one between San Luis Obispo and Monterey bay. These two complete the two backbones of the 4C network.

Figure 3.62 shows the main parts of the CSUN communications network. The single cable from 4CNET is hooked up to a router (through an ATM switch made by FORE). Currently, this is a CISCO 4700 router. The main task of the router is to confirm that the IP addresses of all incoming messages start with 130.166. The CISCO router is connected to a Cabletron Smart Switch 8600 router (through an FDDI switch). The 8600 router has 16 slots with two, 1Gbits/s ports each. This router can therefore handle bandwidths of up to 32Gbits/s. This equipment is located in the MDF (main distribution frame) building. One line goes from the 8600 router to the campus' main computer room, where the main campus computers, CSUN1 and CSUN2, are located (and also other computers dedicated to communications). In addition, the 8600 router is connected, by underground 1Gbit/s fiber optic cables, to many campus buildings. Each building has its own local area network(s).

The third octet of all the `130.166` IP numbers designates the campus subnet. Subnet 1 connects the main campus computers, such as IBM S/3-90, CSUN1, and CSUN2. Each local area network is a subnet. Each subnet has an ethernet router that sends the incoming message to one of its computers based on the 4th octet.

In the engineering building, for example, there are subnets 2, 12, 40–46, 67, and 68. When a router in that building receives a message, it routes it to one of these networks, where it finally gets to its destination computer based on its fourth octet.

Each of the four octets of the IP number consists of 8 bits, so it can have values in the range 0–255. On our campus, the values are allocated as follows: 0 and 255 are unused. 1–19 are reserved. 20-229 are available for general use. 230–249 are used for network equipment, and 250–254 are used for test equipment.

The computer center on campus allocates dynamic IP numbers. Any computer used for communications has to have a communications card (such as an ethernet card) installed in it. Such a card has a unique, 48-bit id number (referred to as MAC, for *media access control*) hardwired in it (see also Section 3.26.2). The leftmost 24 bits identify the card manufacturer and the rightmost 24 bits are unique to the card. A subnet must have a computer that stores the id numbers of all the computers connected to the subnet (these numbers may change since computers and communications cards change). This computer matches the (permanent or semi-permanent) 48-bit id numbers with the (temporary) 24-bit IP numbers in incoming and outgoing blocks of data.

## 3.30 ICANN and IANA

The internet is distributed (i.e., decentralized). It is not owned by any country, company, or individuals. However, IP addresses, as well as domain names, must be unique. There is therefore a need for a central body that will allocate IP addresses and domain names, and will save them in a single data base. Also, all internet software must be compatible, so someone must develop standards for internet communications. These functions are now carried out by ICANN, a body that is the closest thing the internet has to a government.

The Internet Corporation for Assigned Names and Numbers (ICANN, `http://www.icann.org/`) is the new nonprofit corporation that was formed to take over responsibility for the IP address space allocation, protocol parameter assignment, domain name system management, and root server system management functions now performed under U.S. Government contract by IANA and other entities.

ICANN deals only with the points mentioned above. It does not deal with other important issues of internet use and abuse, such as privacy, pornography, and spamming.

The Board of ICANN is composed of 19 directors, nine at-large directors, nine to be nominated by supporting organizations, and the president/CEO (ex officio). The nine at-large directors of the initial

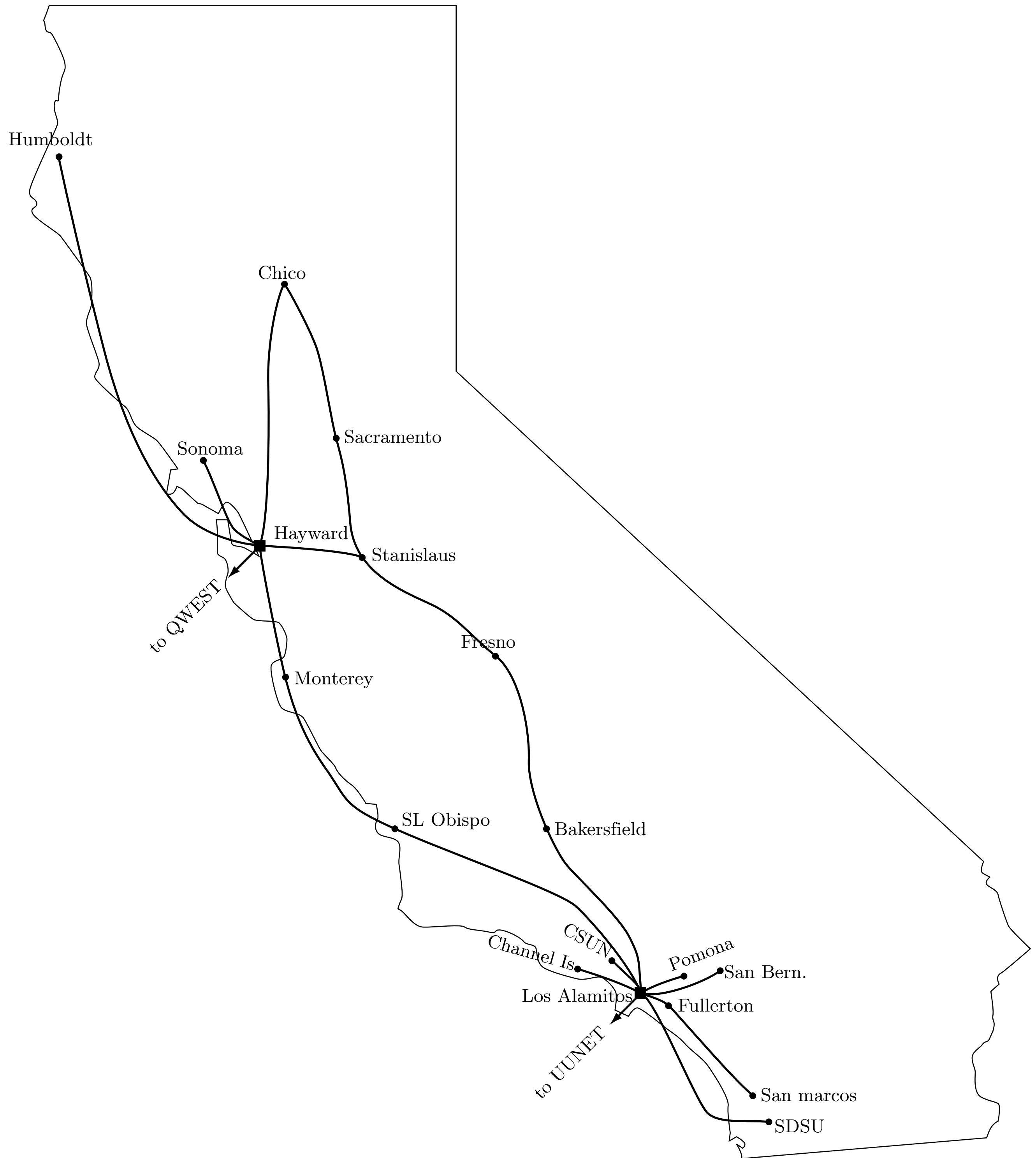



**Figure 3.62:** The two hubs of the 4CNet in California

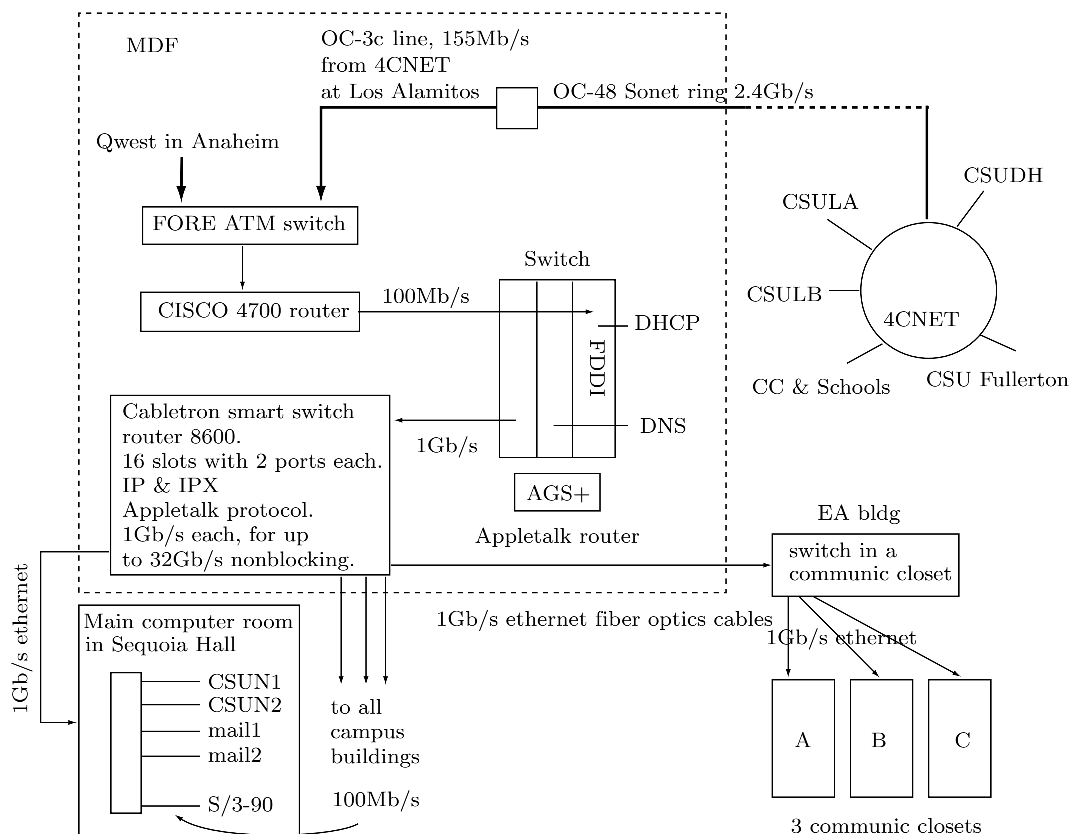



**Figure 3.63:** The CSUN campus network

board serve one-year terms, and are succeeded by at-large directors elected by an at-large membership organization. See `http://www.icann.org/general/abouticann/htm` for biographies of the 19 directors.

Formed in October 1998, ICANN is a nonprofit, private sector corporation formed by a broad coalition of the Internet's business, technical, and academic communities. ICANN has been designated by the U.S. Government to serve as the global consensus entity to which the U.S. government is transferring the responsibility for coordinating four key functions for the Internet: the management of the domain name system, the allocation of IP address space, the assignment of protocol parameters, and the management of the root server system.

Critics of ICANN, some of whom want to change it and others who want to tear it down, met around the issue of "governing the commons: the future of global internet administration," sponsored by computer professionals for social responsibility, a public interest group.

ICANN is the organization to which the U.S. government transferred administration of the technical functions of the Internet, including the single root system. The nonprofit group is also charged with promoting competition in domain names `com`, `org`, and `net`, that until recently were handled exclusively by Network Solutions Inc., of Herndon, Virginia, USA.

Several other companies are now registering domain names and more are accredited by ICANN all the time.

I CANN but you cannot.

See also the IANA web site at `www.iana.org/`

## 3.31 The World Wide Web

The world wide web originated by Tim Berners-Lee at CERN, the European Laboratory for Particle Physics, in 1990. The idea was to develop a world-wide collection of hypermedia documents.

Currently, the main organization devoted to web development is the W3 consortium. It is headed by the laboratory for computer science at the Massachusetts Institute of Technology. Its members are industry representatives and its mission is to develop standards and encourage interoperability between WWW products. See `URL:http://www.w3.org` for more information.

The basic principle behind the web and html is surprisingly simple. A text file is augmented by including pointers to other files. The file becomes a hypertext document. A user can select certain items in such a document, to receive more information about them. This additional information is included in other (plain or hyper) documents, possibly written by other authors and even located on other computers. The pointers can link any number of documents that contain images, movies, and audio, in addition to text.

The web requires two types of participants, authors and user-browsers. An author uses special software such as an editor, a word processor, or a drawing program, to create a hypertext file. Such a file starts with text, images, audio, and video data. Pointers and formatting commands are then embedded in the file using special notation called HTML (hypertext markup language). The file becomes a web page. An author may have a number of web pages linked by pointers, and this collection is referred to as a web site. The author places the hypertext files in a computer called a web server. A user-browser also uses special software, a browser program, to view hypertext files and follow links. In the web terminology, the user is called a client.

The main use of the web is with http (hyper text transfer protocol). When this protocol is used, the user starts by typing a URL into the browser, and the software locates the IP address of the URL. If this IP address is the address of a directory, the browser displays its content. If it is the address of a file, the browser displays the file (text, images, and audio) according to the HTML commands embedded in the file. When the user selects a pointer, the browser gets a URL from the pointer and repeats the process.

The standard browser-server interaction is defined by RFC 822. Following is a simple example. The browser on the client's side (Netscape, Internet Explorer, or other) requests a document from a web server and displays it. When the user selects a pointer (link) in the document, the browser starts with the URL of that link (a domain name and a file name). The browser queries a DNS (Section 3.28.3) to find the IP address associated with the domain name. When this is found, the browser sends a request to that IP address, in order to retrieve the file specified in the URL. The http protocol specifies a format for the request as follows:

```
GET /abc.html HTTP/1.0
Accept: www/source
Accept: text/html
Accept: image/gif
User-Agent:  Netscape/3.7  libwww/2.14
From:  dxs@ecs.csun.edu
     * a blank line *
```

The "GET" keyword indicates the file the client wants (`abc.html`) and discloses that it is using HTTP version 1.0 to communicate. The request also lists the Multipurpose Internet Mail Extension (MIME) types it will accept in return, and identifies itself as a Netscape browser. (The "Accept:" list has been truncated for brevity.) The request also identifies the user in the "From:" field. The request terminates with a blank line.

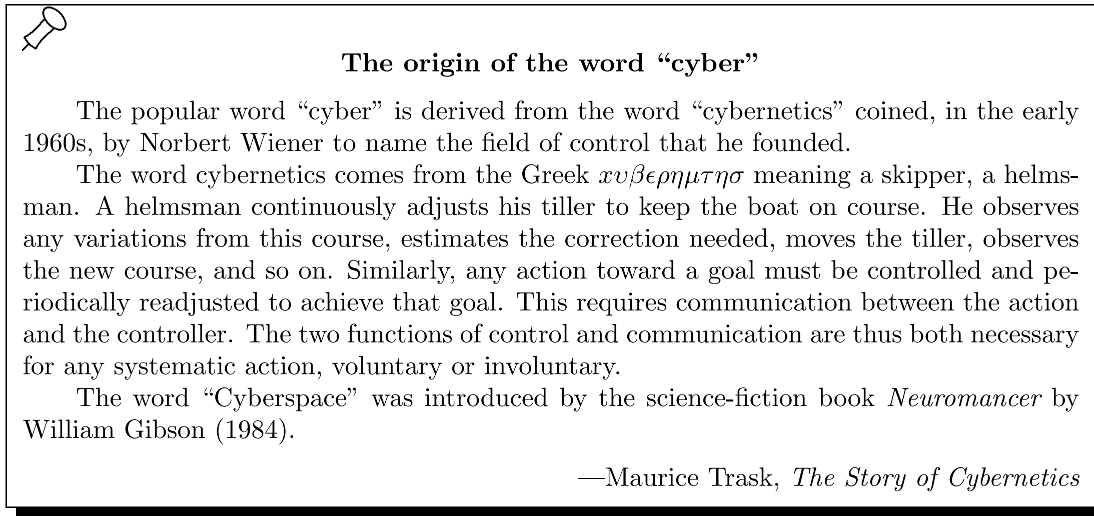
The server then responds by sending:

```
HTTP/1.0 200 OK
Date: Wednesday, 26-Jun-00 16:38:12 GMT
Server: NCSA/1.1
MIME-version: 1.0
Last-modified: Monday, 26-Aug-98 21:15:16 GMT
Content-type: text/html
Content-length: 128566
```

```
    * a blank line *
the html file follows here
```

The server agrees to use HTTP version 1.0 for communication and sends the status 200 indicating it has successfully processed the client's request. It then sends the date and identifies itself as an NCSA HTTP server. It also indicates it is using MIME version 1.0 to describe the information it is sending, and includes the MIME-type of the information about to be sent in the "Content-type:" header. Next, it sends the number of characters it is going to send, followed by a blank line and the html file (the requested data).

A web browser can also access files with other protocols such as FTP (for file transfer), NNTP (a protocol for reading news), SMTP (for email) and others.



**The origin of the word "cyber"**

The popular word "cyber" is derived from the word "cybernetics" coined, in the early 1960s, by Norbert Wiener to name the field of control that he founded.

The word cybernetics comes from the Greek $x\upsilon\beta\epsilon\rho\eta\mu\tau\eta\sigma$ meaning a skipper, a helmsman. A helmsman continuously adjusts his tiller to keep the boat on course. He observes any variations from this course, estimates the correction needed, moves the tiller, observes the new course, and so on. Similarly, any action toward a goal must be controlled and periodically readjusted to achieve that goal. This requires communication between the action and the controller. The two functions of control and communication are thus both necessary for any systematic action, voluntary or involuntary.

The word "Cyberspace" was introduced by the science-fiction book *Neuromancer* by William Gibson (1984).

—Maurice Trask, *The Story of Cybernetics*

Because of the popularity of the web, there is a huge amount of material of all types available for browsing and downloading. The problem is to locate material of interest, and this is done by using a search engine. Such an engine scans the web all the time, and collects information about available documents. A user specifies a search topic, such as "introduction to html", and the search engine lists all the web pages that contain this string. Two popular search engines are `http://www.google.com/` and `http://www.infoseek.com/`.

A word about cgi. A cgi script is a program that's executed on the web server in response to a user request. Most information traffic on the web is from a server to a client. However, because of the importance and popularity of the web, html and the http protocol have been extended to allow for information to be sent back. This is important, for instance, when shopping online. A customer should be able to select merchandise and inform the seller how payment will be made. An html document may display a form that the user fills out. The form includes a button for the user to click when done. That button specifies the name of a program that will process the information in the form. When the user sends the form back to the server, that program is invoked. Such a program is referred to as a "common gateway interface" or a cgi script.

### 3.31.1 Web Search Engines

Thw world wide web is immense, encompassing millions of pages and billions of words on many topics ranging from poetry to science and from personal thoughts to government regulations. There is much useful information, which raises the question of finding items of interest. Much as a library would lose much of its value without a catalog, the web would lose much of its usefulness without the various search engines. A search engine is a software system that keeps a huge amount of information about web pages and can locate any piece of information quickly.

Since there is so much information on the web, it makes sense to search the web itself to find how search engines work. URL `http://www.searchenginewatch.com/webmasters/work.html` was found by such a search.

The two main tasks of a search engine are: (1) To go out to the web, collect information, and update it often and (2) to quickly respond to any query with a *ranked list* of URLs. These two tasks ae discussed here.

Going into the web and collecting existing pages is done by programs called spiders, crawlers, or robots. Such a program finds web pages, inputs their entire texts (sometimes with images), and stores this data in the search engine's data base together with the address (URL) of each page. The main problem is to cover the entire Internet (or at least most of it). The spider has to employ a search strategy that will find all (or most) of the pages in the world wide web. One component of such a strategy is the many *links* found in web pages. Once a spider has found a web page and has input its content, the spider can use the links in that page to find more pages, collect them, then use links found in them to dig deeper and find more web pages. Such a strategy is useful, but does not guarantee that all, or even most, of the existing web pages have been found. It is possible that starting from another web page, new links would be found, extending the coverage. The amount of material found by a spider therefore depends on the choice of initial URLs.

A spider starts with an initial list of URLs that have many links. Examples of such URLs are `yahoo.com` for general reference, the Internet movie data base (`http://www.imdb.com/`) for movie information, `http://vlib.org/` for literary resources, and `http://hometown.aol.com/TeacherNet/` for educational resources. Such a list can be started by using another search engine to search for web pages containing phrases such as "many links," "the most links," "art resources," or "all about music." Once an initial list of URLs has been established, the spider can follow every link from each of the URLs on the list. When those web pages are input into the data base, the number of links in each is counted, and those pages with the most links are added to the initial list.

Such a strategy takes into account the important fact that the web changes continuously. Web pages appear, are modified, then may disappear. It is therefore important to modify the initial list all the time.

The second important task of a search engine is ranking the results. The aim is to present the user with a list of web pages containing the term or phrase searched for, suh that the items first on this list will be the most useful to the user. None of the present methods used to rank results is fully satisfactory, since there is a big difference between how humans think and how computers work. The main methods used for ranking are the following:

1. Term frequency. The search engine locates all the web pages in its data base containing the term requested by the user. Each of those pages is checked for multiple occurrences of the term and is ranked according to how many times the term appears. The assumption is that a page where the term appears many times is likely to be relevant. On the other hand, just repeating a term does not guarantee that the page would have important information on the term.

This ranking method is simple and fast, but might generate irrelevant items in cases where the term is very common or has several meanings.

2. A close variant is a method that considers both the number of occurrences of the term and their positions in the document. If the term is mentioned in the header or anywhere at the beginning of the web page, this method considers the page relevant.

3. Another variant asks the user to supply relevancy terms. For example, searching for "cork," produces web pages about bottles, other cork products, and also about Ireland. When the user specifies "cork" as the search term and "bottle" as the relevancy term, most web pages about te city of Cork in Ireland will not be included in the results list. Some web pages may discuss corks and bottles in Ireland, and they can also be avoided if the search engine allows specifications such as: Avoid pages with the term "Ireland."

4. A different approach to ranking considers the number of times a web page has been referenced in other pages. The idea is that a page that's referenced from many other pages must be relevant to many readers even if it contains the search term just once. To implement this approach, the search engine must count the number of times each web page in its data base is referenced by other pages, a time-consuming task.

5. Imagine a user searching for a common term such as "heart." This may result in a long list of matching web pages. The search engine may spy on the user, trying to determine what pages are selected by the user, and assigning those pages high ranks next time someone searches for the same term.

In addition to the spider-based search engines, there are also *web directories.* Two important ones are Yahoo and ODP. They use human editors to compile their indexes, which are clean and well-organized but

not as extensive as the results produced by a search engine.

More information? Easy, just use a search engine to search the web for the phrase "how do search engines work."

ASCII stupid question, get a stupid ANSI.

—Anonymous

# 4 Microprogramming

In a conventional computer, the control unit fetches the next instruction, decodes it, then executes it (Sections 1.2 and 1.4 should be reviewed for the details of this loop). The control unit has an execution circuit for each machine instruction, and the decoding process determines which circuit to use for the current instruction. The instructions are thus executed by hardware circuits, which is the fastest way the computer can run.

Microprogramming constitutes an entirely different approach to fetching and executing instructions. It has already been introduced in Section 1.6. This chapter discusses many more aspects of microprogramming, together with a running example that illustrates practical microprogramming concepts and techniques.

## 4.1 Basic Principles

The principle of microprogramming is to have the control unit fetch and execute, not machine instructions, but microinstructions. The microinstructions are similar to machine instructions but are simpler and operate on a lower level (they are closer to the hardware). Instead of individual execution circuits for each machine instruction, the control unit needs just one circuit, to execute the microinstructions. (This is an advantage of microprogramming since it simplifies the design of the control unit.) After fetching the next machine instruction and decoding it, the control unit executes it by executing several microinstructions. These microinstructions constitute the *microprogram* for the particular machine instruction, and we say that the microprogram is the description of the machine instruction.

(It should be noted that microinstructions can be used to also decode the current machine instruction after it has been fetched. This is the approach taken by Tanenbaum in his excellent book *Structured Computer Organization* (3rd edition). The example in this chapter, however, uses a hardware-based approach to decoding, see Section 4.7.)

For each machine instruction, the control unit needs a microprogram that describes it. All the microprograms are stored in a special memory called the *control store*, which is separate from main memory. The entire content of the control store is sometimes referred to as the *microcode*. Notice that the control store differs from main memory in the following aspects:

1. It is smaller. Main memory must have room for programs, which may be very large. The control store need only have room for the microprograms, and they are normally very short, just a few microinstructions each.

▸ **Exercise 4.1:** How many microprograms are there in the control store?

2. It must be faster. Executing a single machine instruction involves fetching several microinstructions from the control store and executing them. Microprogramming is thus inherently slow, and one obvious way to speed it up is to have a fast control store. Fortunately, this does not increase the price of the computer significantly, since the control store is small.

3. It is normally read-only. Different programs are loaded into main memory all the time, but the microprograms have to be written and loaded into the control store only once. They also should stay in the control store permanently, so ROM is the natural choice for this storage.

The microprograms for a given, microprogrammed computer are normally written by the manufacturer when the computer is being developed and are never modified. Recall that the microprograms describe the machine instructions, so modifying a microprogram is the same as changing the meaning of a machine instruction. A user modifying the control store may discover that none of their programs executes properly.

Two important features of microprogramming should now be clear to the reader; it is slow (a disadvantage), and it simplifies the hardware (a small advantage, since computer hardware isn't expensive). However, microprogramming has another, more important advantage; it makes the instruction set (in fact, the entire design of the control unit) more flexible. In a conventional, hardwired computer, changing the way a machine instruction operates requires changing the hardware. In a microprogrammed computer it only requires a modification of the microcode. We have already seen that under normal use, the machine instructions are never changed, but there are two good examples of cases where they need to be changed.

1. Imagine a computer manufacturer making and marketing a computer X. If X proves successful, the manufacturer may decide to make a better model (perhaps to be called X+ or X pro). The new model is to be faster than the existing machine X, have bigger memory, and also an extended instruction set. If X is microprogrammed, it is easy to design the extended instruction set. All that it takes is adding new microprograms to the control store, and perhaps modifying existing microprograms. In contrast, if X is hardwired, new control circuits have to be designed and debugged, a much slower and more expensive process.

2. Traditionally, when a new computer is developed, it has to be designed first and a prototype built, to test its operation. Using microprogramming, a new computer Y can be developed by simulating its instruction set on an existing, microprogrammed computer X. All that it takes is to replace the control store of X with a new control store containing the microprograms for the instruction set of Y. Machine Y can then be tested on machine X, without having to build a prototype. This process, a simulation of a computer by microinstructions, is called *emulation*.

## 4.2 A Short History of Microprogramming

The idea of microprogramming is due to Maurice Vincent Wilkes of Cambridge University, an early pioneer of the computing field. He used it in the design of the EDSAC computer, built by his group at Cambridge in the early 1950s. In 1951 he published his objective "to provide a systematic and orderly approach to designing the control section of any computing system." He compared the step by step execution of a machine instruction to the execution of a program instruction by instruction. His approach to microprogramming, however, was hardware-based and is not the one used today. In the same year, L. W. van der Poel, of Lincoln lab described the principle of horizontal microprogramming using the modern, software-based approach. In the 1950s, however, computer memories were too slow for a practical implementation of microprogramming. The February, 1964 issue of *Datamation* reviewed the state of microprogramming and the consensus was that the future of this field "is somewhat cloudy."

Just a few months later, in April 1964, the IBM/360 series of computers—the first important commercial application of microprogramming—was announced. The individual members of the family were developed from the same basic design using different microprograms. This was possible because of technological advances in memory design that both increased speed and reduced costs. The term "firmware" was coined by A. Opler in 1967. The term "emulation" seems to have been coined by R. Rosin in 1969. His ideas led to the development, by Nanodata Inc., of two experimental microprogrammed machines, the QM1 and QM2, designed primarily for research in microprogramming. The first serious book on microprogramming was written by S. Husson in 1970, and the July 1971 issue of the *IEEE Transactions on Computers* was exclusively devoted to this topic.

The Borroughs 1800 computer represents a novel approach to microprogramming. This was an attempt to build a higher-level language computer, a machine whose machine language is higher-level. The 1800 was originally microprogrammed such that its instruction set was identical to RPG (an old programming language; RPG stands for "Report Generator"). Later, two more sets of microcode were developed, to program this machine directly in Cobol and in Fortran.

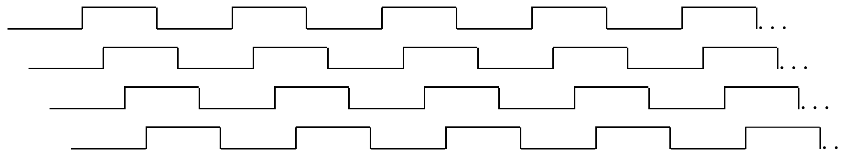
Today, many modern microprocessors are microprogrammed, with the control store as an integral part of the chip, and usually with this feature being completely transparent to the users.

[Smotherman 99] is a more detailed history of microprogramming.

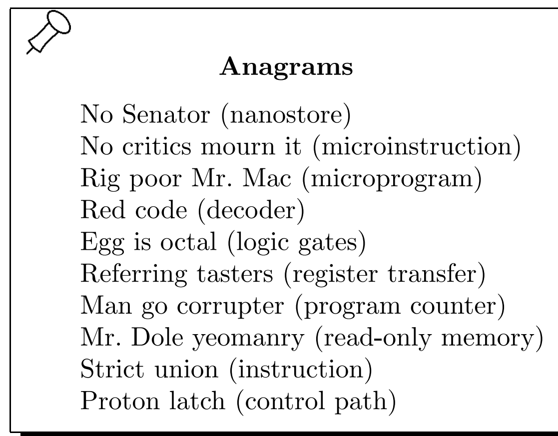
## 4.3 The Computer Clock

Electronic components are fast, but not infinitely fast. They still require time to perform their operations. When, e.g., the inputs of a latch change states, it takes the latch a certain amount of time to sense this, change its internal state, and update its output. The response time of a simple digital device is measured in nanoseconds; it is thus very short but it is not zero. [A nanosecond (ns) is defined as $10^{-9}$ of a second, or one thousandth of a microsecond. See also pages 9 and 359.] As a result it is important to design the computer such that each operation should wait for the previous one to finish before it can start. This is why every computer has a clock that supplies it with timing information. The clock is an oscillator that produces an output signal shaped like a square wave. During each cycle, the clock pulse starts low, then goes up (through a *rising edge*) then goes down again (through a *falling edge*). The clock pulse is sent to all the other parts of the processor, and the rising and falling edges of each cycle trigger the individual components to perform their next operation.

In the microprogramming example described in this chapter, we assume a clock with four output lines, all carrying the same signal, but at different phases as follows:



These outputs divide each clock cycle into four subcycles, not necessarily of the same size, and the four rising edges can be used to trigger the processor to start four operations each clock cycle (Section 4.6). Notice how three of the clock outputs are generated as delayed copies of the first one, using delay lines.




## 4.4 An Example Microarchitecture

Armed with this simple background of digital logic, we are ready to learn about microprogramming by looking at a simple example that incorporates many important features and concepts found in real microprogrammed computers.

The example involves a small, 16-bit computer, with 4 general-purpose registers, 4 special-purpose registers and a small main memory of size $4K \times 16$. The control unit contains the usual PC and IR registers, but it does not have any circuits to execute machine instructions. Instead it has a control store for the microinstructions (with an MPC and MIR registers, whose use is similar to that of the PC and IR) and one circuit to execute the microinstructions. There is also a simple ALU. This computer is hypothetical but

is very similar to the microprogramming example in Tanenbaum's book *Structured Computer Organization* (3rd edition, p. 170) which, in turn, is based on existing commercial bit slices. The main differences between our example and Tanenbaum's are.

1. Tanenbaum's example uses microinstructions to isolate and check the individual bits of the opcode. Our example uses extra hardware to move the opcode to the MPC.

2. Tanenbaum's computer is based on a stack, where all the important operations take place. Our example uses 4 general-purpose registers.

3. Our example adds hardware to make it possible for the microinstructions to generate and use *constants*. Tanenbaum's example is limited to a few registers which contain permanent constants. No other constants can be used.

We describe the machine in several steps: the data path, the microinstructions, the microinstruction timing, the control path, the machine instructions, and the microcode.

The data path is shown in Figure 4.1. The central part is a block of 8 registers, four 16-bit general-purpose and 4 special-purpose. The latter group includes the 16-bit IR, the 16-bit auxiliary register AX, and the two 12-bit registers PC and SP. In addition there are: an ALU (with 2 status flags, Z and N), an MAR, an MBR and a multiplexor (the AMUX). These components are connected by 16-bit data buses (the thick lines; some are only 12 bits wide) along which data is moved in the computer. The main paths for moving data are:

1. The A and B buses. Any register can be selected, moved to either the A bus or the B bus, and end up in the A latch or the B latch. The B latch is permanently connected to the right-hand input of the ALU, but it can also be sent to the MAR. The left-hand input of the ALU comes from the AMUX, which is fed by the A latch and the MBR, and can be switched left or right.

2. The C bus. The output of the ALU can be moved to the C bus (and from there, to any of the registers) or to the MBR (and from there, to main memory). It is important to realize that this output can also be moved to **both** the C bus and the MBR, and it can also be ignored (i.e., moved to none of them). There are therefore 4 ways of dealing with the ALU output.

▸ **Exercise 4.2:** Why would anyone want to go to the trouble of using the ALU to create a result only to disregard it?

## 4.5 The Microinstructions

In addition to the registers and buses, there are many gates, decoders, control lines and other hardware components which are not shown in Figure 4.1. This diagram is kept simple in order to show the basic operations of the computer. Here is what can be done by this computer:

1. Two registers can be selected and sent to the two latches.

2. A 3-bit code can be sent to the ALU, requesting an ALU operation. Also, the B latch can be sent to the MAR.

3. The ALU output can be sent to the C bus, to the MBR, to both places or to none of them.

4. A memory operation (read or write) can be specified.

Since our computer is microprogrammed, the control unit does not have circuits to execute the machine instructions, and they are executed by the microinstructions. Each microinstruction should therefore be able to specify any of the four operations above. It has been mentioned that microinstructions are similar to machine instructions but are simpler, so it takes several microinstructions (a microprogram) to execute one machine instruction. Writing a microprogram is similar to writing a program in assembler. In particular, the microinstructions in a microprogram should be able to **jump** to each other. This is why another, fifth, operation is needed namely,

5. Jump to another microinstruction in the control store.

Any microinstruction should be able to perform the five operations above. Figure 4.2 shows the format of the microinstructions. All the fields where no width is indicated are 1-bit wide. Notice that they all have the same format, specifically, the microinstructions don't have an opcode, which is one reason why they are simpler than machine instructions. This means that a microinstruction can be fetched from the control store and can be executed immediately, without any opcode decoding.

Here is the meaning of the individual fields:

- The AXL and AXR fields control the way the AX register is loaded from the ADDR field (Section 4.7).

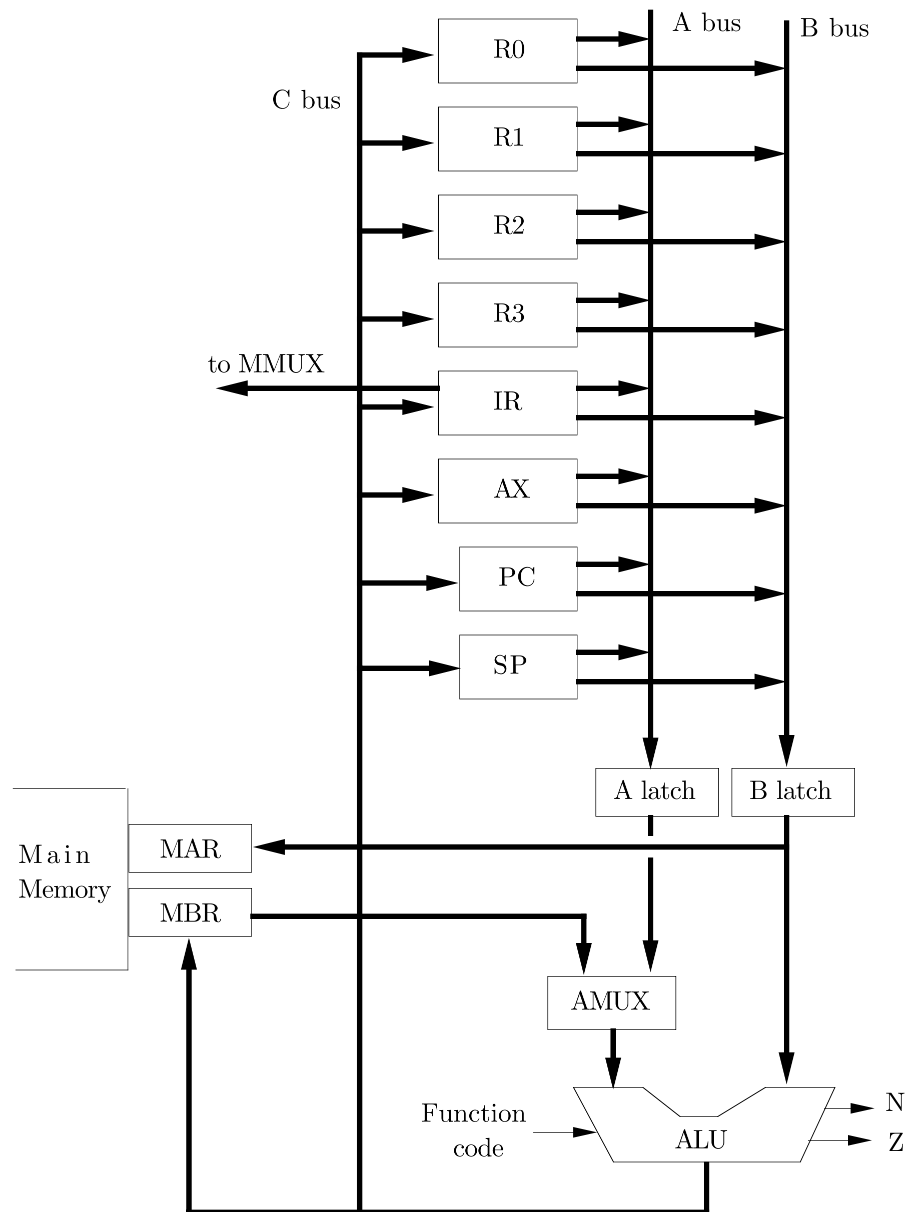



**Figure 4.1:** The data path

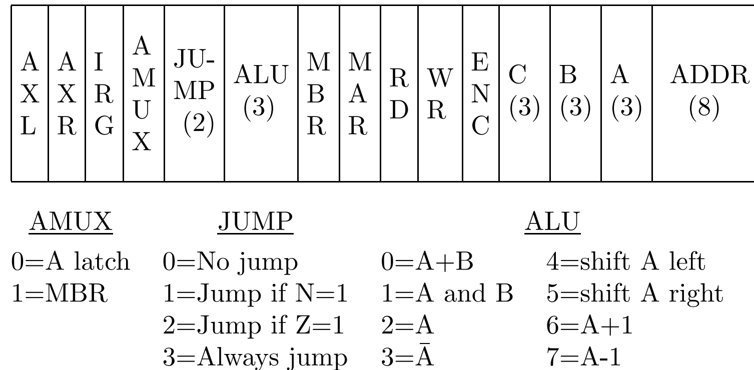



**Figure 4.2:** The microinstruction format

▪ The IRG bit controls loading the opcode from the leftmost 4 bits of the IR to the MPC (Section 4.7).

▪ The AMUX field controls the setting of the A-multiplexor. Any microinstruction that wants to flip the A-multiplexor to the left (MBR) should have a one in this field.

▪ The 2-bit JUMP field specifies the type of jump the microinstruction requires. There are 2 conditional jumps, based on the values of the status flags Z and N, one unconditional jump, and one value for no jump. The jump address is contained in the ADDR field.

▪ The ALU field specifies 1 of 8 possible ALU operations. Code 2 means the ALU will simply pass the value of the A latch through, and disregard the input from the B latch. Code 3 is similar, except that the value being passed is 1s complemented while going through the ALU. Code 6 increments the A input by 1. It can be used to increment any register but it has been included mostly to increment the PC. Code 7 decrements the A input by 1. It can be used to decrement any register but it has been included mostly to decrement the SP.

▸ **Exercise 4.3:** Programs tend to have bugs in them, and microprograms have a similar nature. When the microprograms are written for a new, microprogrammed computer they always have bugs. Yet when the computer is sold, the microprograms must be completely debugged, otherwise no programs will run. How can the manufacturer guarantee that no microinstructions will contain any invalid values?

▪ The MBR and ENC (enable C) fields specify the destinations of the ALU output. Each of these two bits can be 0 or 1, allowing for four ways to route the output, as discussed earlier.

▪ The MAR field indicates whether the B latch should be moved to the MAR. This is done when a microinstruction needs to load the MAR with an address.

▪ The RD and WR fields are set to 1 by any microinstruction that needs to read memory or write to it. Since memory can only do one thing at a time, it is invalid to set both RD and WR to 1 in the same microinstruction.

▪ The A, B and C fields are used to select registers. Each field can select 1 of the 8 registers, which is how a microinstruction selects source registers to be sent to the A and B latches, and a destination register for the ALU output.

▸ **Exercise 4.4:** Our microinstructions are 31 bits long, but we know that the word size in modern computers is a multiple of 8. Why weren't the microinstructions designed to be 32 bits long?

▸ **Exercise 4.5:** How big is the control store of our example computer?

## 4.6 Microinstruction Timing

Even though any microinstruction can perform all five operations above, it does not make sense for it to perform them simultaneously. After selecting two registers and sending them to the A and B latches, the voltages at the latches should be given time to stabilize. Only then can the ALU operate reliably. The ALU, obviously, should also be allowed some time to finish its operations. Any ALU results can then be moved further. This is why timing is important and should be considered in detail even in this simple example.

We assume that each microinstruction is fetched and is fully executed in one clock cycle. It has already been mentioned that, in our example, a clock cycle is divided into 4 subcycles, so the rest of this section is concerned with what happens during each subcycle. Here is the complete description:

1. Subcycle 1 is fully devoted to fetching the microinstruction from the control store. The MPC already points to the next microinstruction (it has been updated in the previous subcycle 4) and the control store already tries to output the next microinstruction to the MIR. The clock output line 1 opens the gates between the control store and the MIR, moving the next microinstruction into the MIR (Figure 4.3). Notice that all the fields of the MIR are active at the end of this subcycle. Those fields that are used in later subcycles should thus be blocked by means of gates.

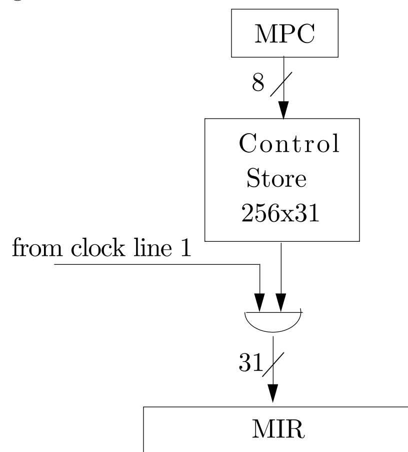



**Figure 4.3:** Fetching a microinstruction

2. In subcycle 2, two registers are selected, are sent to the A and B buses and, from there, to the two latches. The voltages at the latches are given the rest of this subcycle to stabilize. Figure 4.5 shows the details of selecting a register and moving it to the A bus. The 3-bit 'A' field in the MIR becomes the input of the A-decoder. Only one of the 8 decoder outputs can be high, and that one opens a gate (actually, 16 gates) that move the selected register to the A bus and from there, in subcycle 2, to the A latch.

Figure 4.4a shows the details of moving data (16 bits) from the MBR to the AMUX. This is controlled by 16 AND gates which are opened when the 'MBR' field in the MIR is 1. This field is updated in subcycle 1 so, if it is 1, the MBR will move to the AMUX and from there, (if the AMUX is switched to the left) to the ALU, during subcycle 2.

The end of subcycle 2 sees stable inputs at the ALU.

3. Subcycle 3 is devoted to the ALU. Notice that there are no gates at the ALU inputs and no way to enable/disable it. The ALU is always active, it always looks at its inputs and uses them to generate some output. Most of the time the inputs are wrong, producing wrong output. The trick of using the ALU is to disregard its output until the moment it is supposed to be correct. The earliest time the ALU inputs are right is the start of subcycle 3, which is why the control unit should disregard the ALU output until the start of subcycle 4.

▸ **Exercise 4.6:** Consider the 3 'function' input lines to the ALU. They tell the ALU which of its 8 operations to perform. When does the ALU look at these inputs?

Loading the MAR from the B latch is also done during this subcycle. Figure 4.4b shows the details of moving the B latch to the MAR. This is done through an AND gate (actually 12 gates) that opens only

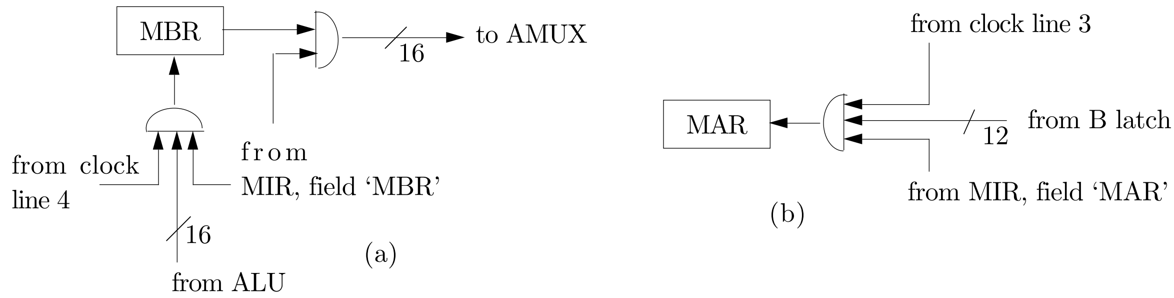



**Figure 4.4:** The details at the MAR and MBR

when subcycle 3 starts AND when the microinstruction selects this particular move (when the 'MAR' field in the MIR is 1).

4. Since the ALU output is now stable, it is only natural to use subcycle 4 to moving the ALU output to its destination. There are two destinations, the C bus and the MBR.

Figure 4.4a shows the details of moving the ALU to the MBR. This move is controlled by 16 gates which open only when the 'MBR' field in the MIR is 1 AND it is subcycle 4.

Figure 4.5 shows the details of moving the C bus to one of the registers. Notice that the C-decoder is normally disabled. It is enabled only when the 'ENC' field in the MIR is 1 AND it is subcycle 4.

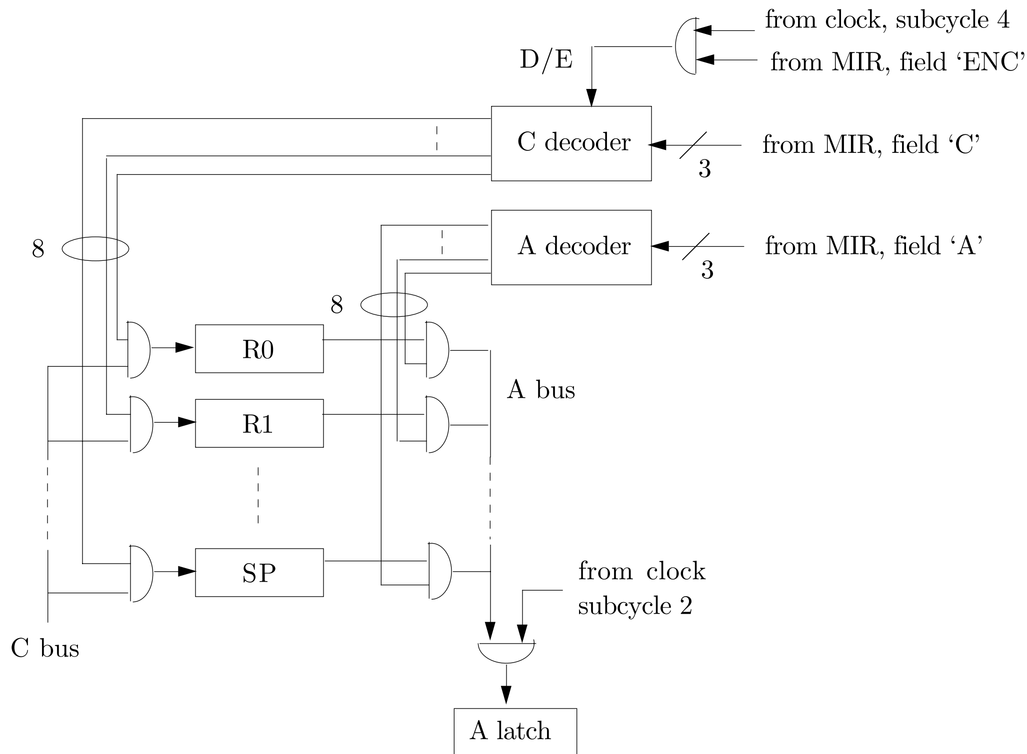



**Figure 4.5:** The details at the A and C buses

A microinstruction jump also takes place during subcycle 4. A jump in a program is always done by resetting the PC. Similarly, a jump in a microprogram is done by resetting the MPC. Figure 4.6 shows the details at the MPC. The value of the MPC is always incremented by 1 and is sent back to the MMUX

multiplexor, from which it goes back to the MPC in subcycle 4 (the MMUX is normally switched to the left). This is true for all microinstructions that don't need to jump. If a microinstruction decides to jump, it simply switches the MMUX to the right, resetting the MPC to the value of the 8-bit 'ADDR' field in the MIR.

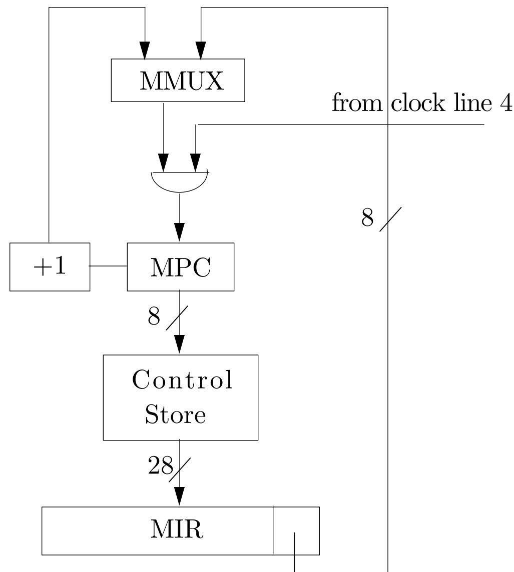



**Figure 4.6:** Details at the MPC

Recall that a microinstruction will jump, conditionally or unconditionally, to another microinstruction depending on the value of its 'JUMP' field and on the values of the status flags Z and N. Extra hardware is therefore needed, to examine the 2 bits of the 'JUMP' field (which are labeled L and R) and the status flags, to decide whether or not a jump should take place and, if yes, to flip the MMUX to the right. This hardware is called the *micro sequencing logic* (Figure 4.7). It is actually very simple, consisting of just 3 AND gates and 1 OR gate, since a jump should take place if any of the following three conditions is met: (1) The 'JUMP' field equals 3 (L=1, R=1); (2) 'JUMP'=1 (L=0, R=1) AND N=1; (3) 'JUMP'=2 (L=1, R=0) AND Z=1.

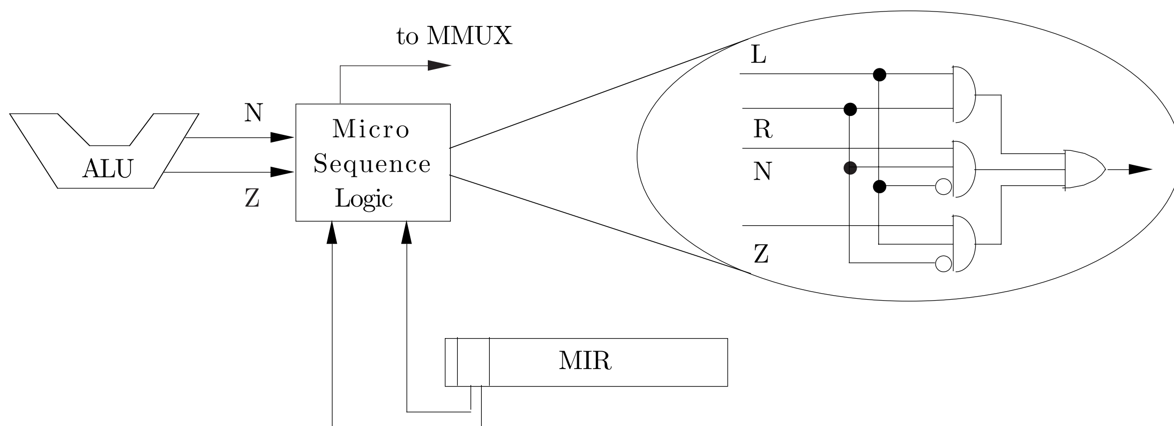



**Figure 4.7:** Micro sequencing logic

## 4.7 The Control Path

This section discusses a few more features that enhance the operation of our example computer and that haven't been mentioned so far. The first one has to do with decoding the current machine instruction.

In a real hardwired computer there is an opcode decoder (Figure 4.8a) whose inputs are the bits of the opcode. Each output of the opcode decoder triggers one of the individual execution circuits. Depending on the current opcode, one of the decoder's outputs goes high, triggering the corresponding execution circuit.

In a real microprogrammed computer there are no execution circuits. After the next machine instruction is fetched, the control unit has to execute its microprogram. It therefore needs to know where that microprogram starts in the control store. This is usually done by means of a small, separate ROM containing the start addresses of all the microprograms in the control store. The control unit picks up the opcode of the current instruction from the IR and sends it to the address bus of that ROM. When sending an address to a ROM, the corresponding data comes out on the data bus of the ROM (there is no need to send a 'read' signal, since a ROM is unwritable). The control unit moves that data, which is the start address of the right microprogram, to the MPC. Figure 4.8b shows an example of such a ROM. We assume a hypothetical computer having instructions with opcodes 0, 1, 3, and 5, whose microprograms start at control store locations 25, 56, 74 and 129, respectively. Notice that not all locations in the ROM are used since a computer may not use every available opcode. (For example, a computer with 8-bit opcodes does not always have 256 instructions.) In such a case it is sometimes cheaper to use a *Programmable Logic Array* (PLA) instead of a ROM. A PLA is similar to a ROM but is manufactured such that only locations that are actually used exist. Any memory location that's not going to be used is simply not fabricated.

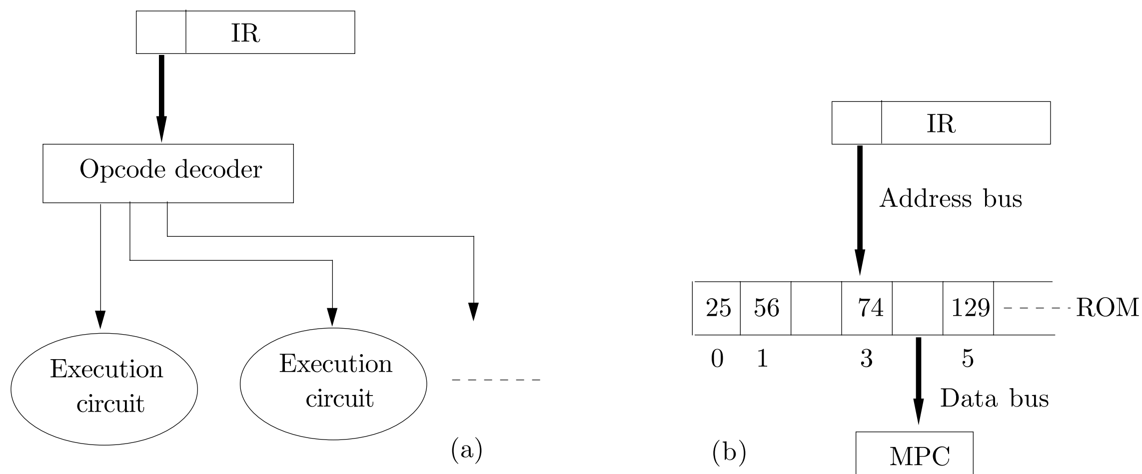



**Figure 4.8:** Decoding the opcode

In our example computer, no ROM is used. Instead, the 4-bit opcode is moved from the IR to the MPC each time the next instruction is fetched from main memory. Our opcodes (Table 4.12) have values 1 through 12, implying that the next microinstruction will be fetched from one of these locations (locations 1 through 12) in the control store. All that's necessary is to place '`goto`' microinstructions at those locations, that will go to the start of the right microprograms. The PUSH instruction looks different in Table 4.12 since it has a 6-bit opcode. However, Section 4.9 shows how the last 2 bits of this opcode are isolated and analyzed by microinstructions. The control unit can therefore treat this instruction as if it had the 4-bit opcode 1011.

Figure 4.9 shows the details of moving the opcode of the current machine instruction from the IR to the MPC through the MMUX. The four leftmost bits of the IR are moved to the MMUX, through AND gates followed by OR gates. The microinstruction which does that has to satisfy the following three conditions: (1) it has to open the AND gates; (2) it should have an ADDR field of all zeros, so as not to send anything through the OR gate; (3) it has to make sure that the MMUX is flipped to the right. All three conditions are

achieved by the simple microinstruction '`irg; goto 0;`' (see Section 4.9 for this notation). A little thinking should convince the reader that a '`goto 0;`' combined with an '`irg`' does not cause a jump to location zero of the control store. Rather it moves the eight bits 0000*pppp* (where *pppp* are the opcode bits from the IR) to the MMUX and from there, to the MPC. The MPC is thus reset to 0000*pppp*, causing the next microinstruction to be fetched from location *pppp* of the control store. Since *pppp* is a 4-bit number, it can be between 0 and 15 (actually it should be between 1 and 15, see below). Locations 1 through 15 of the control store should therefore be reserved for special use. Location L should contain the microinstruction '`goto P`' where P is the start address, in the control store, of the microprogram for the machine instruction whose opcode is L. Location 0 of the control store is reserved for a '`goto`' microinstruction that starts the fetch sequence of the first machine instruction (Section 4.9).

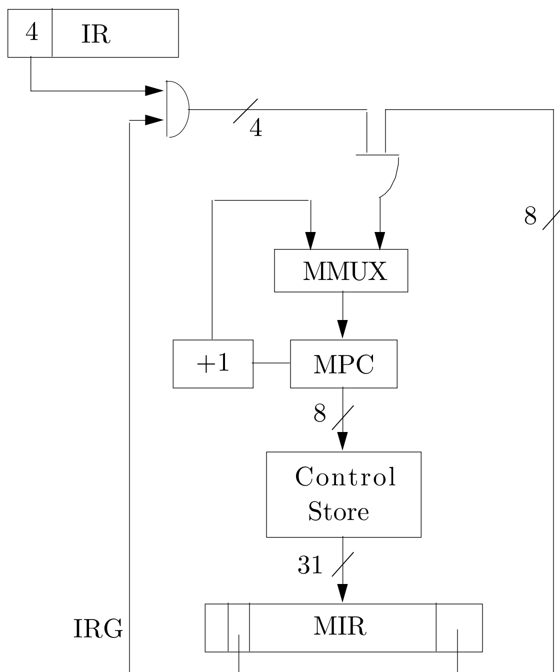



**Figure 4.9:** Moving the opcode to the MPC

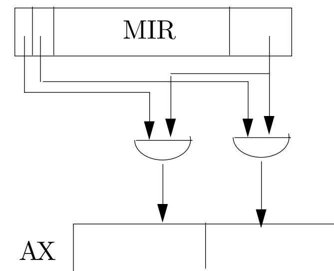



**Figure 4.10:** Moving a constant to the AX register

The next feature discussed in this section has to do with the use of constants. When writing machine instructions, many computers offer the immediate mode (Section 2.6), making it easy to use constants. Our microinstructions do not use any addressing modes and can only use the existing registers and gates. Since constants are useful, one more feature has been added to our microinstructions, making it possible to create a 16-bit constant in the AX register. The two fields AXL and AXR of every microinstruction can be used to move the 8-bit ADDR field to the left or right halves of the AX register. Recall that the ADDR field normally contains a jump address; it now has a second use as a constant. Figure 4.10 shows the details of the gates involved in this move.

▸ **Exercise 4.7:** When does this move occur (in what subcycle)?

Figure 4.11 shows the entire control path. Areas surrounded by circles or ellipses indicate details shown in previous diagrams.

**Figure 4.11:** The control path

## 4.8 The Machine Instructions

A real computer should have a set of machine instructions that includes all the important types (Section 2.11). Our computer is just an example, so it has been assigned a small, incomplete instruction set, specifically designed to illustrate the microprograms. All instructions are 16 bits long and all but one have a 4-bit opcode. A few, simple addressing modes are used.

A look at Table 4.12 shows that some instructions use register R1. On a real computer, each of those should be able to use any of the 4 general-purpose registers. Important instructions, such as STORE, ADD, SUBTRACT and SHIFT are missing since their microprograms are similar to the ones for LOAD. Another important point to notice is that the machine instructions can use the 4 general-purpose registers and the SP, but they cannot use the PC, IR or AX. The latter registers are used only by the microinstructions. This is another aspect of the microinstructions being low-level. They can access hardware components that even machine instructions cannot.

## 4.9 The Microcode

The microinstructions operate at a low level; they are close to the hardware. Nevertheless, it is convenient to write them using a high-level notation, similar to PASCAL or C, since this reduces the number of bugs in the microinstructions and also makes it easy to read the microcode. We can imagine a microassembler reading microinstructions in this notation and translating each into a 31-bit number. We use the convention that each microinstruction is written on one line, with the individual phrases separated by semicolons. Following is the notation used for the various operations a microinstruction can perform.

The operations of selecting 2 registers, specifying an ALU operation, and routing the result to another register are specified by phrases such as, '`r1:=r1+r0;`', '`r2:=band(r3,r1);`', '`r4:=r3;`', '`r0:=inv(r3);`',

| Name | Mnemonic | Opcode | Operand | Description |
|---|---|---|---|---|
| Load direct | LODD R1 | 0001 | 12-bit x | R1:=M[x] |
| Load relative | LODR R1 | 0010 | ” | R1:=M[x+PC] |
| Load indirect | LODN R1 | 0011 | ” | R1:=M[M[x]] |
| Load immediate | LODI R1 | 0100 | ” | R1:=x |
| Unconditional Jump | JMP | 0101 | ” | PC:=x |
| Indirect jump | JMP R1 | 0110 | none | PC:=R1 |
| Conditional jump | JZER R1 | 0111 | 12-bit x | if R1=0, PC:=x |
| Conditional jump | JNEG R1 | 1000 | ” | if R1< 0, PC:=x |
| Procedure call | CALL | 1001 | ” | SP:=SP−1; M[SP]:=PC; PC:=x |
| Procedure return | RET | 1010 | none | PC:=M[SP]; SP:=SP+1 |
| Push register | PUSH Ri | 1011ii* | ” | SP:=SP−1; M[SP]:=Ri |
| Increment SP | INSP | 1100 | 8-bit y | SP:=SP+y |

* ii are 2 extra opcode bits indicating the register to be pushed.

**Table 4.12.** The machine instructions

and ‘`r3:=lshift(inv(r3));`’, where `band` stands for “boolean AND”, `inv`, for “inverse” and `lshift`, for “left shift.” The phrase ‘`alu:=r1;`’ means selecting R1, moving it to the A latch and, from there, to the ALU, where it is passed through without any processing (ALU function-code 2) and then discarded. Nothing is moved to either the C bus or the MBR. The result of such an operation is new values of the status flags Z and N, so such a phrase is usually combined in the microinstruction with a conditional jump phrase.

▶ **Exercise 4.8:** What notation should be used to move R2 to **both** the C bus and the MBR?

A jump is specified by either ‘`goto abc;`’ or ‘`if N goto xyz;`’.

The microinstruction ‘`axr 127;`’ moves the 8-bit constant 127 (= $01111111_2$) to the right half of AX, without modifying the left half. If this is followed by ‘`axl 0;`’, then the AX register is loaded with 9 zeros on the left followed by 7 ones on the right; a number useful as a mask to isolate 7 bits out of 16. Another example is the microinstruction ‘`axl axr 15;`’ It loads the AX with the 16-bit pattern ‘0000111100001111’. We assume that the microassembler can handle binary and hex constants, in addition to decimal. Thus the microinstructions ‘`axl 15;`’, ‘`axl F’X;`’ and ‘`axl 1111’B;`’ are all identical.

Table 4.13 shows some microinstructions written in this notation, together with their numeric fields.

| Microinstruction | AXL | AXR | IRG | AMUX | JUMP | ALU | MBR | MAR | RD | WR | ENC | C | B | A | ADDR |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| mar:=pc; rd; | 0 | 0 | 0 | x | 0 | 2 | 0 | 1 | 1 | 0 | 0 | x | 6 | x | xx |
| rd; pc:=pc+1; | 0 | 0 | 0 | 0 | 0 | 6 | 0 | 0 | 1 | 0 | 1 | 6 | x | 6 | xx |
| ir:=mbr; | 0 | 0 | 0 | 1 | 0 | 2 | 1 | 0 | 0 | 0 | 1 | 4 | x | x | xx |
| irg; goto 0; | 0 | 0 | 1 | 0 | 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 00 |
| axl 12; alu:=r1; | 1 | 0 | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 0 | 0 | x | x | 1 | 12 |
| alu:=r0; if Z goto 10; | 0 | 0 | 0 | 0 | 2 | 2 | 0 | 0 | 0 | 0 | 0 | x | x | 0 | 10 |
| r2:=lshift(r2); | 0 | 0 | 0 | 0 | 0 | 4 | 0 | 0 | 0 | 0 | 1 | 2 | x | 2 | xx |
| r3:=band(r2,ax); | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 3 | 5 | 2 | xx |
| r1=mbr:=r2; mar:=pc; | 0 | 0 | 0 | 0 | 0 | 2 | 1 | 1 | 0 | 0 | 1 | 1 | 6 | 2 | xx |

**Table 4.13:** Microinstruction examples (x=don’t care)

▶ **Exercise 4.9:** What is the difference between the two microinstructions ‘`r1:=mbr; goto fech;`’ and ‘`goto fech; r1:=mbr;`’?

Before we look at the entire microcode, here are some examples of machine instructions and their microprograms.

1. A direct load, `LODD R1,123`. The 12-bit `x` field in the machine instruction is a direct address, and the content of memory location `x` are moved to register R1. This instruction uses the direct mode (Section 2.4) since the entire 12-bit address is included in the instruction. The microprogram is,

```
mar:=ir; rd;
rd;
r1:=mbr;
```

Notice that the MAR is a 12-bit register while the IR has 16 bits. All 16 bits are moved to the B bus and the B latch, but only the 12 least significant ones are moved from there to the MAR. Notice also how the 'read' signal to memory is maintained during two microinstructions, i.e., during two clock cycles. This makes sense since memory is always slower than the processor. We assume that our processor can perform 4 operations per cycle, while memory needs 2 cycles to complete one of its operations.

This instruction is basically a memory-read operation, where the result is sent to R1. A real computer would, of course, have similar instructions for all 4 registers, and also the similar STORE instructions, which are memory-write.

▸ **Exercise 4.10:** Write the microprogram for the Store Direct instruction STOD R1.

2. A load in the relative mode, `LODR R1,14`. The 12-bit `x` field in the machine instruction is considered a relative address, and the effective address is the sum of `x` and the PC.

```
axl F'X;
axr FF'X;          [ax=0FFF hex]
ax:=band(ax,ir);   [ax becomes the 12 lsb of the ir]
ax:=ax+pc;         [ax becomes the effective address]
mar:=ax; rd;
rd;
r1:=mbr;
```

Four microinstructions are needed just to compute the effective address. Those familiar with the relative mode (Section 2.5) may remember that in this mode the displacement is signed, which introduces a slight problem. After the first 3 microinstructions isolate the 12 least significant bits of the IR and move them to the AX register, that register contains `0000sxxx...x`, where `s` is the sign bit of the 12-bit relative address. This sign bit should be moved to the leftmost position to create the pattern `sssssxxx...x` in the AX. We conveniently ignore this point since it is tedious to do by our microinstructions. We also ignore the complications resulting in the ALU from the different sizes of the two registers being added (the AX is 16 bits long, whereas the PC is only 12 bits).

3. Indirect load, `LODN R1,13`. The 12-bit `x` field in the machine instruction is considered an indirect address, and the effective address is the content of `m[x]` (we denote EA=m[x]).

```
mar:=ir; rd;       [indirect address is sent to memory]
rd;
ax:=mbr;           [ax becomes the effective address]
mar:=ax; rd;       [effective address is sent to memory]
rd;
r1:=mbr;
```

▸ **Exercise 4.11:** Why is it necessary to read the effective address to the AX register and then send it from there to the MAR. Why not move the EA directly from the MBR to the MAR with a microinstruction such as '`mar:=mbr`'?

4. Immediate load, `LODI R1,17`. The 12-bit `x` field in the machine instruction is considered an immediate quantity to be moved to R1. There is no effective address and no memory operations.

```
axl F'X;
axr 255;           [ax=0FFF hex]
r1:=band(ax,ir);   [r1 becomes 0000+the 12 lsb of the ir]
```

▶ **Exercise 4.12:** The next important addressing mode, after direct, relative, indirect and immediate, is the index mode. Design a reasonable format for the '`LODX R1,indx`' (Load Index) instruction, and write the microprogram.

5. Increment SP, `INSP 12`. This instruction uses an 8-bit constant, which has to be extracted from the IR before any addition can take place. The microprogram is straightforward.

```
axl 0;
axr FF'X;
ax:=band(ax,ir);
sp:=sp+ax;
```

▶ **Exercise 4.13:** The previous examples show that the AX register is used a lot as an auxiliary register. What other registers can the microinstructions use as auxiliaries?

6. Push a register into the stack, `PUSH Ri`. This is an example of an instruction with a 6-bit opcode. The opcode starts with 1011, followed by two more bits that indicate the register to be pushed into the stack. The microprogram is much longer than the average, since 9 microinstructions are necessary just to isolate the extra two opcode bits and decide what register to use.

```
      sp:=sp-1;
      axl 00001000'B;
      axr 0;          [ax:=0000100...0, a mask]
      ax:=band(ax,ir); if Z goto r0r1; [isolate bit ir(11)]
r2r3: axl 00000100'B;  [ir(11)=1, check ir(10)]
      axr 0;
      ax:=band(ax,ir); if Z goto r2;
r3:   mbr:=r3; mar:=sp; wr; goto writ;
r2:   mbr:=r2; mar:=sp; wr; goto writ;
r0r1: axl 00000100'B;  [ir(11)=0, check ir(10)]
      axr 0;
      ax:=band(ax,ir); if Z goto r0;
r1:   mbr:=r1; mar:=sp; wr; goto writ;
r0:   mbr:=r0; mar:=sp; wr;
writ: wr; goto fech;   [Done!]
```

▶ **Exercise 4.14:** Show how an ADD instruction can be included in our instruction set. Write the microprograms for two types of ADD; an ADD R1, which adds m[x] to register R1, and an ADD R1,Ri, which adds register Ri to R1.

One more thing should be discussed before delving into the details of the microprograms namely, what to do at the end of a microprogram. A look at Table 4.14 shows that every microprogram ends with a '`goto fech`'. Label '`fech:`' starts a microprogram that fetches the next machine instruction. This is another important feature of microprogramming. The control unit does not need a special circuit to fetch the next machine instruction, since this operation can easily be done by a short microprogram.

The microcode is thus one big loop. It starts at control store location 0, where a '`goto fech`' causes the very first machine instruction to be fetched. Following that, the microcode moves the opcode to the MPC, resetting the MPC to a value between 1 and 12, and causing a jump to the start of the right microprogram. After executing the microprogram, a '`goto fech`' is executed, sending control to the microprogram that fetches the next machine instruction.

▶ **Exercise 4.15:** The microcode is a loop, but there is no HALT microinstruction. How does it stop? Also, how can we be sure that the loop will start at control store location 0 (i.e., that the MPC will contain zero when the computer starts)?

Table 4.14 shows the entire microcode for our example. It is not hard to read since the machine instructions are simple.

```
0: goto fech; [just to fetch the first instruction]
1: goto lodd;
```

```
   2: goto lodr;
   3: goto lodn;
   4: goto lodi;
   5: goto jmp;
   6: goto ijmp;
   7: goto jzer;
   8: goto jneg;
   9: goto call;
  10: goto ret;
  11: goto push;
  12: goto insp;

[13-15 reserved for future use]

fech: mar:=pc; rd;  [fetch next machine instruction]
f1:   pc:=pc+1; rd; [increment pc while waiting for memory]
      ir:=mbr;
      irg; goto 0;  [decode: mpc:=4 msb of ir]

lodd: mar:=ir; rd;  [load direct]
      rd;
      r1:=mbr; goto fech;

lodr: axl F'X;           [load relative]
      axr FF'X;          [ax=0FFF hex]
      ax:=band(ax,ir);   [ax becomes the 12 lsb of the ir]
      ax:=ax+pc;         [ax becomes the effective address]
      mar:=ax; rd;
      rd;
      r1:=mbr; goto fech;

lodn: mar:=ir; rd;       [indirect address is sent to memory]
      rd;                [read ea]
      ax:=mbr;           [ax becomes the effective address]
      mar:=ax; rd;       [effective address is sent to memory]
      rd;
      r1:=mbr; goto fech;

lodi: axl F'X;           [load immediate]
      axr 255;           [ax=0FFF hex]
      r1:=band(ax,ir); goto fech; [r1:=0000+the 12 lsb of the ir]

jmp:  pc:=ir; goto fech; [pc:=12 lsb of the ir]

ijmp: pc:=r1; goto fech; [indirect jump]

jzer: alu:=r1; if Z goto jmp;     [jump conditionally]
      goto fech;          [no jump]

jneg: alu:=r1; if N goto jmp;     [jump conditionally]
      goto fech;          [no jump]

call: sp:=sp-1;           [procedure call]
```

```
      mar:=sp; mbr:=pc; wr;
      pc:=ir; wr; goto fech;

ret:  mar:=sp; rd;          [return]
      rd; sp:=sp+1;
      pc:=mbr; goto fech;

push: sp:=sp-1;       [push ri. should identify ri first]
      mar:=sp;
      axl 00001000'B;
      axr 0;          [ax becomes the mask 0000100...0]
      ax:=band(ax,ir); if Z goto r0r1; [isolate bit ir(11)]
r2r3: axl 00000100'B; [ir(11)=1, check ir(10)]
      axr 0;
      ax:=band(ax,ir); if Z goto r2;
r3:   mbr:=r3; wr; goto writ;
r2:   mbr:=r2; wr; goto writ;
r0r1: axl 00000100'B; [ir(11)=0, check ir(10)]
      axr 0;
      ax:=band(ax,ir); if Z goto r0;
r1:   mbr:=r1; wr; goto writ;
r0:   mbr:=r0; wr;
writ: wr; goto fech;  [Done!]

insp: axl 0;          [Increment sp]
      axr FF'X;       [ax:=00FF]
      ax:=band(ax,ir);[ax:=8 lsb of ir]
      sp:=sp+ax; goto fech;
[end of microcode]
```

**Table 4.14.** The microcode

▸ **Exercise 4.16:** The short sequence

```
axl F'X;
axr FF'X;
ax:=band(ax,ir);
```

occurs often in our microcode. When such a thing happens in a program, we write it as a procedure. Discuss the use of procedures in the microcode.

The following should be noted about the microcode.

1. Microprograms should be efficient, squeezing maximum performance out of every last nanosecond, but they don't have to be readable. Remember that they are written and debugged once, then used often, perhaps over years, without any changes.

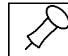
2. One example of microprogram efficiency is the microprogram for JZER which says '`if Z goto jmp;`'. It branches to the microprogram for JUMP, which saves one microinstruction in the microcode. Another example is '`pc:=pc+1; rd;`'. It increments the PC while waiting for the slow memory. Our microcode has quite a few microinstructions that say '`rd`' or '`wr`' and just wait for a memory operation to complete. A real microprogrammed computer can perform many useful background tasks while waiting for slow memory. Examples are: refresh dynamic RAM, check for enabled interrupt lines, refresh the time and date displayed on the screen.

3. Another example of how the microprograms can be made more efficient is to start fetching the next machine instruction *before* the current microprogram is complete. As an example, here is a modified version of the microprogram for LODD.

```
lodd: mar:=ir; rd;  [load direct]
```

```
rd;
r1:=mbr; mar:=pc; rd; goto f1;
```

The last line finishes the load operation by saying 'r1:=mbr;', and also starts fetching the next machine instruction by saying 'mar:=pc; rd;'. It then goes to 'f1' instead of to 'fech'. Other microprograms can be modified in a similar way. This is a primitive form of *pipelining*, where the control unit performs several tasks on several consecutive machine instructions simultaneously.



Using computer lingo one can say: the mind of a mathematician works in the background, while in the foreground, he/she is communicating with the world.

—Andrew Vázsonyi

4. Our microprograms tend to be short, since the machine instructions are so simple. Real microprograms, however, may be quite long. A good example is the microprogram for PUSH. It is 17 microinstructions long (compared to 4 microinstructions which is the average of all the other microprograms) because it has to isolate and check individual bits.

▸ **Exercise 4.17:** A microinstruction can only perform certain types of data movements, since the data path (Figure 4.1) is limited and does not allow arbitrary register transfers. Microinstructions such as 'r1:=r2;' or 'mbr:=r2;' perform one move each. It's been mentioned earlier that the notation 'r1=mbr:=r2;' can be used to move R2 to **both** R1 and the MBR (notice the use of the equal sign '=' and the assignment operator ':='). The question is, Can one microinstruction perform 3 data moves?

▸ **Exercise 4.18:** Consider adding a 'shift' instruction to our small instruction set. The instruction should have a 4-bit opcode followed by a 12-bit x field. It should shift R1 to the left x positions. Discuss the problem of writing a microprogram for this instruction.

## 4.10 Final Notes

One way to intuitively understand microprogramming is to consider it an additional *level* in the computer. It is possible to consider the computer a multilevel machine where the lowest level, level 0, is the hardware and each level uses the ones below it. In a conventional (hardwired) computer, the level above the hardware, level 1, is the machine instructions. If we write a program in machine language, we need the hardware of level 0 to execute it. Level 2 is the assembler, which isn't hardware, but is very useful. Level 2 is used to translate programs into level 1, to be executed by the hardware of level 0. Level 3 is the higher-level languages.

With this picture in mind, we can consider microprogramming an additional level, inserted between the hardware and the machine instructions. In a microprogrammed computer, the hardware constitutes level 0, the microinstructions, level 1, the machine instructions constitute level 2, and the assembler makes up level 3. The hardware no longer executes the machine instructions but the microinstructions. The microinstructions, in turn, execute the machine instructions, except that the word "execute" isn't the right one. There is a difference between the way the hardware executes the microinstructions and the way they execute the machine instructions. We therefore say that the hardware *interprets* the microinstructions, and they *emulate* the machine instructions. The word "simulate" is reserved for simulation by instructions (software), while the word "emulate" is used to indicate simulation by microinstructions (sometimes called *firmware*).

▸ **Exercise 4.19:** How about the assembler and the microassembler. Do they simulate or emulate?

In principle it is possible to write a compiler that would translate a program in a higher-level language directly into microinstructions. Such a compiler would generate a microprogram for each machine instruction that it normally generates. The trouble with this approach is that the microprograms would have to be stored in a large memory (since programs can be very large) and large, fast memories are still very expensive. A large slow memory would result in slow execution. This is why contemporary computers use the paradigm of compiler translation, followed by microcode emulation, in a double-memory configuration (a large, slow main memory, and a small, fast control store).

One promising approach is RISC. A RISC machine has simple instructions; an approach which makes the instructions easy to execute by hardware but requires more instructions to do the same job. A RISC computer, by definition, does not use microprogramming, which makes it faster and more than compensates for the larger number of instructions that have to be fetched and executed. (Some microprogrammed computers are advertised as RISC, but experts agree that "pure" RISC should not utilize microprogramming. A microprogrammed computer can therefore at best be called "RISC-like.")

Another interesting point to consider is speeding up the microprogrammed computer by adjusting the subcycles. Up until now we assumed that each cycle is the same length and is divided into 4 subcycles in the same way. This is clearly not efficient. ALU operations require different times, so subcycle 3 should only last for as long as the ALU needs. Subcycle 4 may require different times depending on how the ALU output is routed and whether a jump is necessary.

How can we change the duration of subcycles. Speeding up and slowing down the clock frequency is not practical since the clock is driven by a crystal that gives it precise frequency. The solution is to use a faster clock and have each subcycle last a certain number of clock cycles. We now use the term "period" instead of subcycle, so every microinstruction is fetched and executed in 4 periods, each lasting a certain number of clock cycles (Figure 4.15). Period 3, for example, may last between say, 1 and 20 clock cycles on our example computer, depending on the ALU operation.

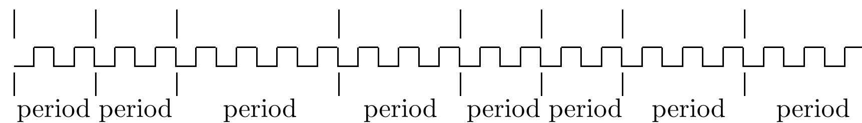



**Figure 4.15.** Two cycles (eight different periods) of a fast clock.

How does the control unit know how long each period should be? One approach is to add 4 fields to each microinstruction, containing the lengths of the 4 periods for the microinstruction (in clock cycles). The microprogrammer should know how long each ALU operation takes, and how long period 4 takes for every possible output move and every possible jump. That information should be determined by the microprogrammer and added to every microinstruction in the microcode. Another approach is to have all the timing information stored in a ROM (or in a PLA). The ALU function code is sent to the address bus of the ROM at the start of period 3, and out, on the data bus of the ROM, comes the necessary duration of the period.

A microprogrammed computer designed for speed should use this approach even though it complicates the hardware and results in a higher price for the computer. Computers designed to be inexpensive normally use fixed-size periods, so each period should be long enough to accommodate the slowest operation possible in that period.

The next important point to discuss has to do with the encoded fields in our microinstructions. The 'A', 'B', 'C', 'ALU' and 'JUMP' fields are encoded, and have to be decoded when the microinstructions are executed. We have seen the A-, B- and C decoders. The ALU has its own internal decoder, to decode its three "function" input lines. The two bits of the 'JUMP' field are decoded by the gates in the micro-sequencing-logic circuit. It is possible to redesign our microinstructions without any encoded fields (Section 4.11). The 'A', 'B' and 'C' fields would be 8-bit wide each. The 'ALU' field would be 5 bits wide (even though our ALU can execute 6 functions) and the 'JUMP' field would be 3 bits wide. This would eliminate the need for decoders and would consequently speed up the hardware. The microinstructions would be longer (51 bits instead of the present 31). Such an approach to microprogramming is called *horizontal* because the control store can be visualized as wide and flat (Figure 4.16a). The opposite approach is *vertical* microprogramming, where a microinstruction has just a few, highly encoded fields. The microinstruction is short, but more of them are needed for the microcode. The control store looks like Figure 4.16b.

The fields of a vertical microinstruction typically include an opcode, source operand (a register or a memory address), destination operand (the same), ALU function, and jump. All encoded, and requiring decoders to execute the microinstruction. Tanenbaum's book discusses such an example on pp. 194–199.

Vertical microinstructions are generally less powerful than horizontal ones, so more are needed. Horizontal microinstructions are longer and harder to write and debug. This is why most real microprogrammed

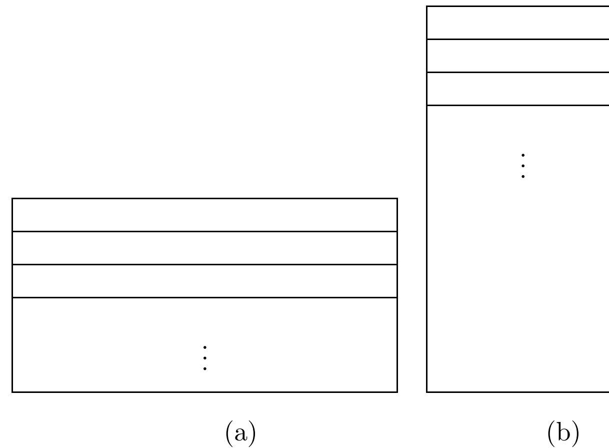



**Figure 4.16.** Horizontal and vertical control stores

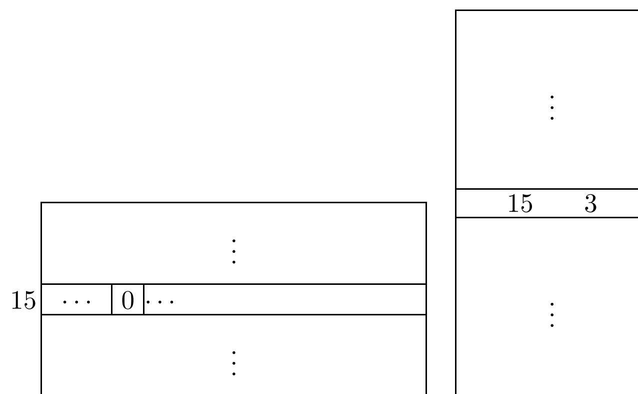



**Figure 4.17.** Nanostore and control store

computers use microinstructions that are in-between these two extremes. A typical microinstruction has some encoded fields and some explicit (decoded) ones. This is why our example, while being simple, is also representative.

In the late 1960s, a research team under Robert Rosin at the State University of New York at Buffalo came up with a combination of the two methods. They envisioned a design where the hardware fetches and executes extremely simple horizontal *nanoinstructions* from a nanostore. The nanoinstructions are organized into nanoprograms, each for the execution of a microinstruction. The microinstructions, which are vertical, are thus fetched and executed by the nanoinstructions. The microinstructions themselves fetch and execute the machine instructions.

This approach adds two levels to the computer, and is therefore slow. Its main advantage is that it is now easy to change the microinstructions! Changing the microinstructions is like changing the instruction set, something that users never do. It may, however, be useful for a computer manufacturer or a research lab, where one computer could be used to develop, compare and implement different instruction sets. Two computers based on those ideas, the QM-1 and QM-2, were actually built and sold in small quantities during the 1970s by Nanodata Inc. Two references describing this approach are [Rosin 69] and [Salisbury 76].

The terms *nanoprogramming*, *nanoinstructions* and *nanostore* are also used to describe another approach to microprogramming. The nanostore contains the set of all different microinstructions. The control store contains pointers to the microinstructions in the nanostore. Thus a microprogram in the control store is a set of pointers to microinstructions in the nanostore. This approach is slow since it is dual-level, but it may make sense in cases where the microinstructions are very long (horizontal) and there are relatively few

different microinstructions.

This idea can be generalized to the case where several microinstructions are very similar, and can be made identical by changing one field into a parameter. Imagine a set of microinstructions that have a field indicating a register. One representative microinstruction from this set, with zeros in the register field, is placed in the nanostore. When a microprogram needs one of these microinstructions for, say R3, it places a pointer and the parameter 3 in the control store (Figure 4.17).

The last important point to mention is that some modern microprogrammed computers are only partially microprogrammed. In such a computer, the control unit has execution circuits for those machine instructions that are simple to execute, and a control store for the more complex instructions. This results in relatively simple hardware, fast execution for the simple instructions and slow execution for the complex ones. The ALU, registers and the entire control unit, including the control store, are fabricated on one small chip, so users don't even know if the computer is microprogrammed, and to what extent.

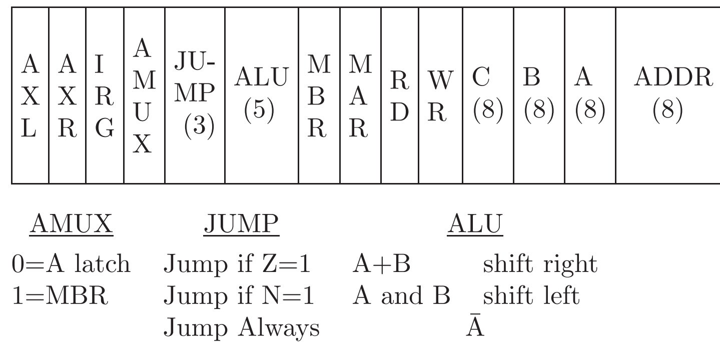



**Figure 4.18:** Horizontal microinstruction format

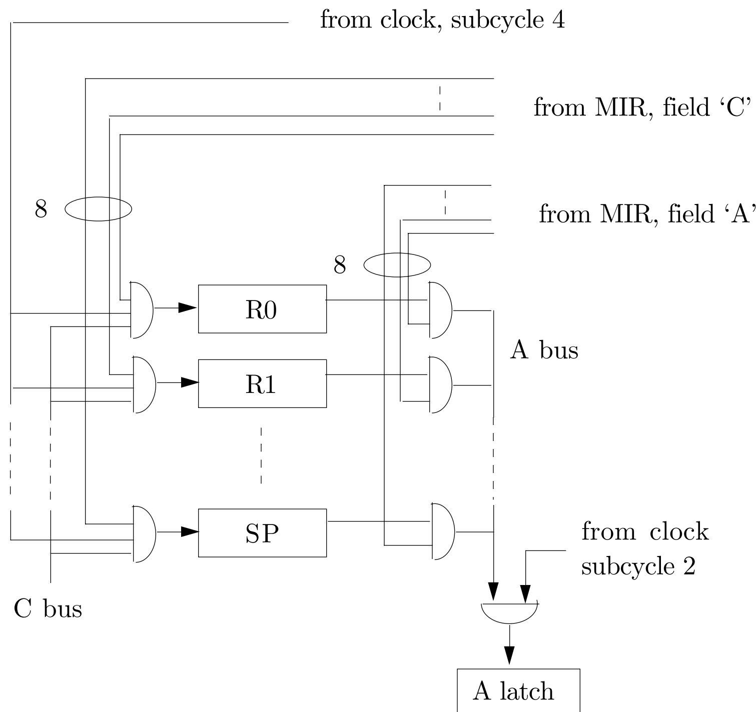



**Figure 4.19:** Horizontal example: the A and C buses

## 4.11 A Horizontal Example

Our microprogramming example can easily be converted to horizontal, since only 5 fields of our microinstructions are encoded. Figure 4.18 shows the new format of the microinstructions, which are 48 bits long.

The JUMP field is three bits long, indicating, from left to right, "Jump if Z=1", "Jump if N=1" and "Jump Always". There is no need for a fourth bit to indicate no jump. The microinstruction will not jump if all three bits of the JUMP field are zero. Similarly, the ALU field is 5, not 6 bits long, since one ALU operation namely, "Let A through" takes place when all 5 bits are zero. Notice that there is no need for the ENC field. A microinstruction where all 8 bits of the C field are zero will not move anything to the C bus.

The main differences in the control circuits are in the details at the three buses and the micro-sequence-logic. Figures 4.19 and 4.20 are the new versions of Figures 4.5 and 4.7.

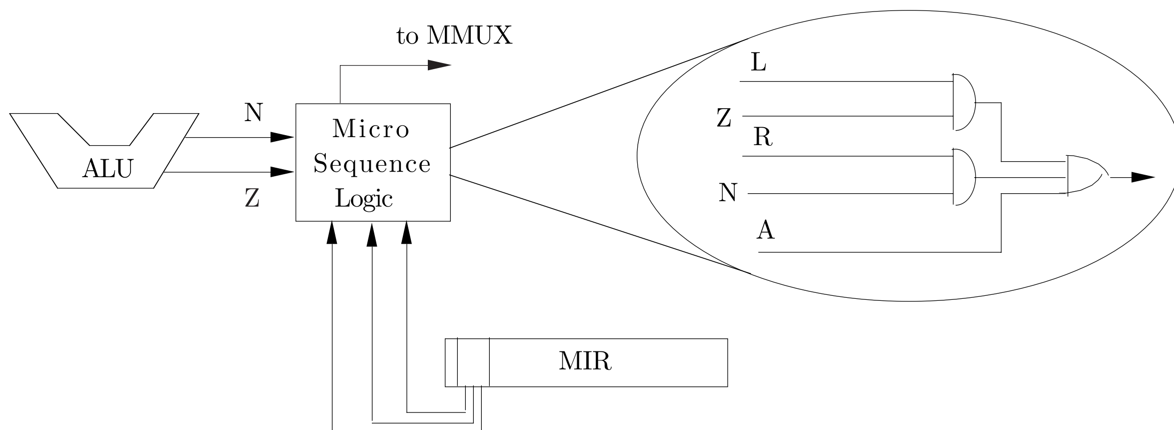



**Figure 4.20:** Horizontal micro sequencing logic

The three bits of the JUMP field are denoted L, R and A in Figure 4.20, which stand for "left" (jump if Z=1), "right" (jump if N=1) and "always". Only 5 bits are shown for ALU operations but they can be increased to 8 or any other number.

The microcode does not change, which shows that our original example was very close to horizontal. Converting this example to pure vertical would result in completely different microinstructions, and consequently different microcode. The interested reader should consult Tanenbaum, pp. 195–201.

> Let us honour if we can
> The vertical man,
> Though we value none
> But the horizontal one.
>
> —W. H. Auden, *Collected Poems II*

# 5
# Parallel Computers

## 5.1 Introduction

This chapter covers the main parallel architectures in some detail, and also discusses several examples of parallel computers. To begin with, let's look at two of the main problems of parallel processing, namely why is parallel processing useful, and how it is achieved. Parallel processing is useful because of the following:

- There are—and, for the foreseeable future, will be—applications requiring faster computers than we have today.
- Conventional (sequential) computers are reaching fundamental physical limits on their speed.
- With VLSI devices, it is possible to produce a computer with hundreds or even thousands of processors, at a price that many users can afford.
- Parallel computers can provide an enormous increase in speed, using standard components available today.

## 5.2 Classifying Architectures

This short section discusses coarse-grained and fine-grained architectures, scalar, superscalar and vector architectures, and the Flynn classification.

A parallel computer with coarse-grained architecture consists of a few units (processors or PEs), each relatively powerful. A fine-grained architecture consists of a large number of relatively weak units (typically 1-bit PEs). The concept of graininess is also used to classify parallel algorithms. A coarse-grained algorithm involves tasks that are relatively independent and communicate infrequently; a fine-grained algorithm involves many simple, similar subtasks that are highly dependent and need to communicate often. Often, they are the individual repetitions of a loop. The fine-grained algorithms are the ones we consider parallel, but the concept of graininess is qualitative. It is impossible to assign it a value, so sometimes we cannot tell whether improving an algorithm changes it from coarse-grained to fine-grained.

A scalar computer is one that can only do one thing at a time. In a superscalar computer there is hardware that makes it possible to do several things at a time. Examples are (1) executing an integer operation and a floating-point operation simultaneously, (2) executing an addition and a multiplication at the same time, and (3) moving several pieces of data in and out of different memory banks simultaneously, using separate buses. The PowerPC, the Intel i860, and the Pentium are examples of superscalar architectures.

A vector computer has even more hardware, enabling it to operate on several pieces of data at the same time, in a pipeline located in the ALU.

The Flynn classification is based on the observation that a parallel computer may execute one or more programs simultaneously, and that it may operate on one or more sets of data. As a result, the following

table classifies computer architectures in four types:

| instr/data | single | multiple |
|---|---|---|
| single | SISD | SIMD |
| multiple | MISD | MIMD |

The SISD group includes all conventional computers. Such a computer executes one program (a single stream of instructions, SI) and that program operates on a single set of data (SD). The MIMD class includes computers with several processors (with or without shared memory). An MIMD computer thus executes several programs simultaneously (MI), and each program operates on its own set of data (MD). An SIMD computer executes one program (SI) that operates on several sets of data (MD). The computer has several processing elements (PEs) that operate in parallel. Each of these groups includes many different architectures. Various types of architecture can be assigned to the fourth group, MISD, but none of them is important, so this group will not be discussed here.

## 5.3 Parallel Processing Methods

Parallel processing is done either by generalizing the von Neumann architecture (as in multiprocessors, multicomputers and array processors) or by designing non von Neumann architectures (such as associative computers and data flow computers) that are inherently parallel. The following four methods for parallel processing are covered in this chapter:

▪ MIMD computers. These are further divided into multiprocessors and multicomputers. A multiprocessor is a set of several processors, sharing a common memory. It can be used in one of two ways. One way is to have several independent programs executed in parallel by the different processors. Another way is to have several, related parts of the same program execute in parallel and communicate all the time.

▪ Array processors. They consist of several processing elements (PEs) working in lockstep. At any given time, all the PEs execute the same instruction but on different sets of data. Such a computer is special purpose, since only certain applications lend themselves to this kind of execution.

▪ Associative computers. They solve the von Neumann bottleneck problem (Section 5.13) by eliminating the concept of a memory address. On such a computer, a memory word is accessed by its content and not by an address. This represents a major departure from the conventional, sequential design, and is therefore hard to program.

▪ The data flow computer, is another approach to a non von Neumann computer design. The flow of control is not sequential but is driven by the data. This leads to a computer that is parallel in nature, and is conceptually different from any other kind of information processing machine.

The above four types of parallel computers are discussed in this chapter with examples. First, however, we present some design criteria for parallel computers.

## 5.4 Design Criteria

One important design parameter for a parallel computer is the concept of granularity. A parallel computer can be fine-grained, coarse-grained, or anything in between. A coarse-grained architecture is one based on a few nodes (processors or PEs), each relatively powerful. It is suitable for general purpose computers. A fine-grained architecture is the opposite. It consists of many nodes, each very weak (with very limited computing power). It is a good choice for certain special purpose computers. The concept of granularity is not rigorously defined and is qualitative rather than quantitative. The granularity is not assigned a specific numeric value but is specified by words such as 'coarse', 'fine', 'very fine.' Nevertheless, granularity is an important architectural feature of parallel computers.

Another important design parameter is the use of memory. In a parallel processor, memory can be shared or distributed. Shared memory is global, it can be used by every node of the machine. Distributed memory is local, each node has its own memory, which cannot be accessed directly by any other node. Shared memory has the following advantages:

▪ It allows for very easy communications between the individual nodes of the computer. They can leave each other messages in memory.

■ It can be shared. If, at a certain time, one node needs just a little memory, the other nodes can get more memory. A while later, the memory allocation can be changed.

■ Files can be shared between programs. An important data base, for example, used by several programs, needs to be loaded into memory only once.

A third design issue is general-purpose vs. special-purpose. A general-purpose computer is generally more useful. A special-purpose computer, however, is much faster for certain applications. experience shows that there are major computer users, with special problems, willing to pay for the design and development of a special-purpose computer. Another aspect of this design issue is the nature of the individual nodes. They may all be identical, or they may be different, some may be designed for special tasks.

## 5.5 The Hypercube

This is a very common architecture for parallel computers, and it deserves a special mention [Siegel and McMillan 81]. In a hypercube, the individual nodes are located at the corners of an $n$-dimensional cube, with the edges of the cube serving as communication lines between the nodes. The hypercube has two important properties (1) It has an intermediate number of connections between the individual nodes. (2) The nodes can be numbered in a 'natural' way.

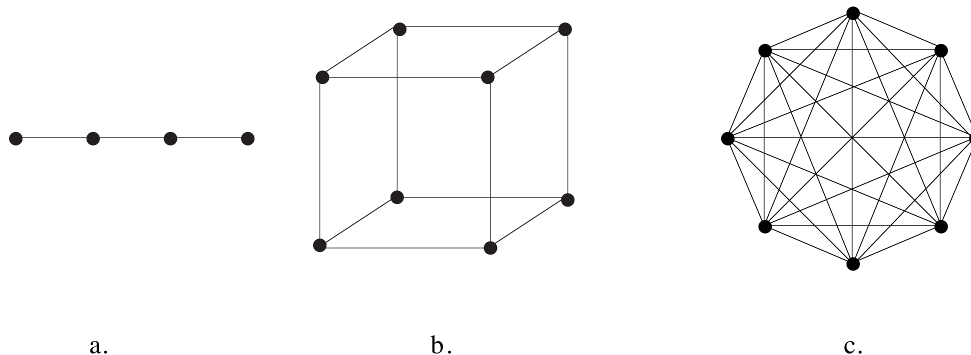



**Figure 5.1. Various topologies**

The first property is easy to understand by comparing the three connection schemes of Figure 5.1. In Figure 5.1a, four nodes are connected linearly. The number of connections is three (in general, for $n$ nodes, it is $n-1$). In Figure 5.1b, eight nodes are connected in a three-dimensional cube. Each node is connected to three other ones, and the total number of connections is $8\times 3/2 = 12$. In general, for $n$ nodes, this number is $n\log_2 n/2$. In Figure 5.1c the eight nodes are fully connected. Each is connected to all the other ones. The total number of connections is $8\times 7/2 = 28$ and, in general $n(n-1)/2$.

The more connections between the nodes, the easier it is for individual nodes to communicate. However, with many nodes (large $n$), it may be impossible to fabricate all the connections, and it may be impractical for the nodes to control their individual connections. This is why the hypercube architecture is a compromise between too few and too many connections.

The second important property of the hypercube has to do with the numbering of its nodes, and is discussed later. Figure 5.2 shows cubes of 1-, 2-, 3-, and 4 dimensions. Several cube properties immediately become clear.

■ An $n$-dimensional cube is obtained (Figure 5.2a) by generating two $(n-1)$-dimensional cubes and connecting corresponding nodes.

■ An $n$-dimensional cube consists of $2^n$ nodes, each connected to $n$ edges.

■ There is a natural way to number the nodes of a cube. The two nodes of a one-dimensional cube are simply numbered 0 and 1. When the one-dimensional cube is duplicated, to form a two-dimensional cube, the nodes of one copy get a zero bit appended—to become 00 and 01—and the nodes of the other copy

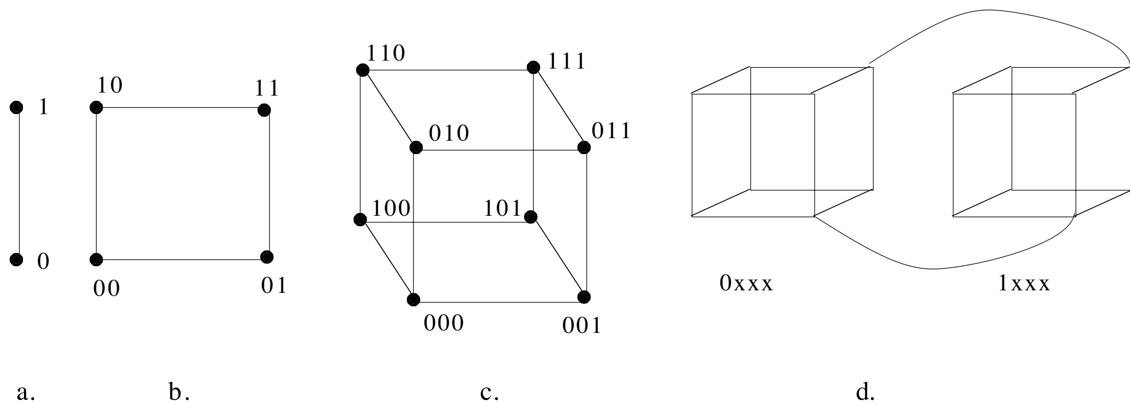



**Figure 5.2. Various hypercubes**

get a 1 bit appended in the same way—to form 10 and 11. The additional bit can be appended on either the left or the right. Figure 5.2c shows how the node numbers are carried over from a two-dimensional to a three-dimensional cube. Again, the new, third bit can be added on the left, on the right, or in between the two existing bits. This numbering method is not unique. Numbers can be carried over from the $n-1$-dimensional to the $n$-dimensional cube in several different ways. However, the method has one interesting property, namely each bit in the node number corresponds to a dimension of the cube. In Figure 5.2c, all four nodes on the top of the cube have numbers of the form $x1x$, while the numbers of the four nodes on the bottom are $x0x$. The middle bit of the node number thus corresponds to the vertical dimension of the cube. In a similar way, the leftmost bit corresponds to the depth dimension, and the rightmost bit corresponds to the horizontal dimension.

▸ **Exercise 5.1:** How can we use our numbering method to generate different node numbers for the four-dimensional cube in Figure 5.2d?

This property helps in sending messages in a hypercube computer. When node $abcd$ in a four-dimensional cube decides to send a message to its 'next door' neighbor on the second dimension, it can easily figure out that the number of this neighbor is $a\bar{b}cd$ where $\bar{b}$ is the complement of bit $b$.

▸ **Exercise 5.2:** Summarize the above mentioned properties for a 12-dimensional cube.

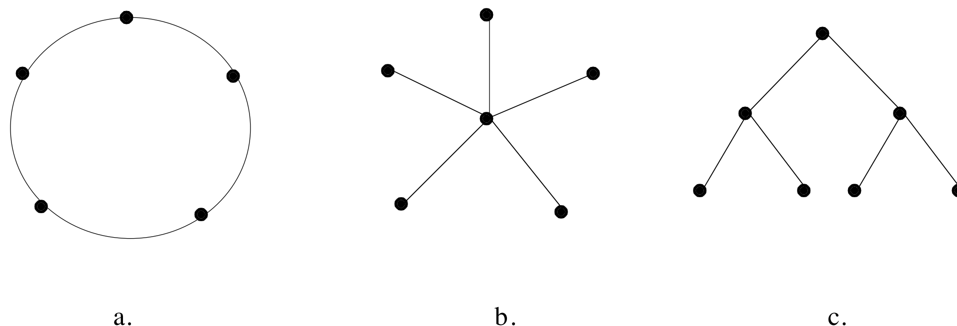



**Figure 5.3. Different node connections**

It should be noted that there are many other ways to connect individual nodes in a parallel computer. Figure 5.3 shows a few possibilities. The star configuration is especially interesting since it has both a short communication distance between the nodes and a small number of connections. It is not very practical, however, because its operation depends heavily on the central node.

## 5.6 Array Processors

The term SIMD refers to one of the three classes in the Flynn classification. This term stands for Single-Instruction, Multiple-Data, so computers in this class execute a single instruction stream (one program), but execute each instruction on multiple data items. The most important property of SIMD computers is that they are **special-purpose**. They are useful only for certain types of problems, and are ineffective or even useless for general use. This is why they are relatively unknown, and why only a few of them have ever been built and used.

An array processor is one type of an SIMD computer (the other type is called an ensemble and will not be discussed here.) Its organization is summarized in Figure 5.4.

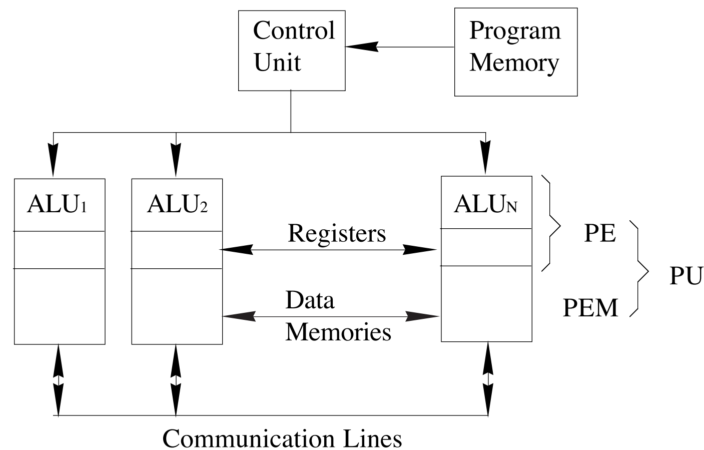



**Figure 5.4. Organization of an array processor**

The program memory holds one program, and instructions are fetched, one by one, by the control unit. The control unit decodes each instruction, executes some of them, and sends the rest to be executed by the ALUs. Instructions such as Enable Interrupts, `JMP`, and `CALL` are executed by the control unit. Other instructions, involving operations on numbers, are executed by the ALUs. What makes the array processor special is that each such instruction is executed by *all* the ALUs simultaneously. The ALUs, of course, operate on different numbers. The communication lines can be used to transfer data between the ALUs.

A word about terminology. Since the ALUs operate on different data, each should also have its own registers and data memory. An ALU with registers is usually called a PE (processing element). A PE with data memory (PEM) is called a PU (processing unit).

The principle of operation of this computer is that all the PUs execute the same instruction simultaneously (they operate in *lockstep*), but on different data items. This, of course, makes sense for certain applications only. Examples of applications that don't lend themselves to this kind of execution are: text processing, the compilation of programs, and most operating systems tasks. Examples of applications that can benefit from such execution are: payroll, weather forecasting, and image processing.

In a typical payroll application, the program performs a loop where it inputs the record of an employee, calculates the amount of net pay, and writes the paycheck to the output file, to be printed later. The total processing time is, therefore, proportional to the number of employees $M$.

An array processor also involves a loop, but it starts by reading the records of the first $N$ employees (where $N$ is the number of PUs) into the data memories of the PUs. This input is performed serially, one record at a time. The array processor then executes a program to calculate the net pay which, of course, calculates $N$ net pays simultaneously. The last step is to write $N$ paychecks to a file, one at a time, to be eventually printed. The entire process is repeated for the next group of $N$ employees. The last group may, of course, contain fewer than $N$ employee records. The main advantage of this type of processing is the run time, that is proportional to $\lceil M/N \rceil$. However, the following points should be carefully considered, since they shed important light on the operation of the array processor.

■ The input/output operations are not performed by the PUs. These operations can be performed by the control unit, but most array processors have a conventional computer, called a *host*, attached to them, to perform all the I/O and the operating system tasks. The host computer treats the PE memories (and the program memory) as part of its own memory, and can thus easily move data into and from them.

An interesting result is that an array processor can be viewed in two different ways. One way is to view the array as the central component, and the host computer as a peripheral. The other way is to think of the array as a peripheral of the host computer. Figure 5.5 illustrates the two points of view.

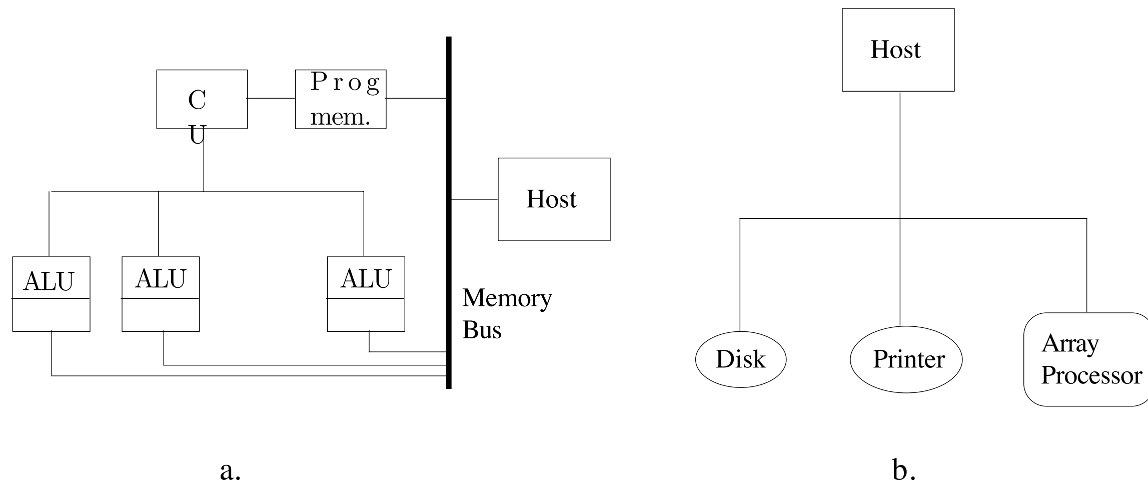



**Figure 5.5. Two views of an array processor**

■ It is now clear that a program for an array processor includes three types of instructions: control unit instructions, such as a `JMP`, PU instructions (operations on numbers), and host instructions (input/output and memory reference). After fetching an instruction, the control unit decodes it and either executes it or sends it to the host or the PUs for execution. To simplify the coding process, the control unit should wait with the next instruction until it is sure that the current instruction has been completed.

■ In the payroll example above, the last group of employees normally has fewer than $N$ records. In the case of a payroll, this does not present a problem. The extra PUs perform meaningless operations and end up with meaningless results, that are ignored by the program. In other cases, however, it is important to disable certain PUs. Consider, for example, the program fragment (a) below:

```
     .          .          .          .
     .          .          .          .
     JMP L1     BPL L1     DPL        BPL L1
     ADD ..     ADD ..     ADD ..     BMI L2
     .          .          .          .
     .          .          .          .
  L1 SUB ..  L1 SUB ..     ENA     L1 BCC L3
     .          .          SUB ..     .
     .          .          .          .

     (a)        (b)        (c)        (d)
```

This is easy to execute, since `JMP` is a control unit instruction. The control unit executes it by resetting its PC to the address of the `SUB` instruction. If, however, the `JMP` is changed to a conditional jump, such as a `BPL` (branch on plus, see (b) earlier), the situation becomes more complicated. `BPL` means “branch if the most recent result is positive (if the S flag is zero).” Since the PUs operate on different numbers, some have positive results while others have negative results. The first group should branch to `L1` while the remaining PUs should continue normally.

This problem is solved by adding a *status bit* to each PU. The PU can be either enabled or disabled, according to the value of its status bit. Instead of a `BPL`, the compiler generates a `DPL` instruction (disable if plus), that is sent to all the PUs. All PUs with an S flag of zero disable themselves. The `ADD` instruction and the instructions that follow are only executed by the PUs that remain enabled. At point `L1` the compiler should generate an `ENA` (enable all) instruction, to enable all the PUs. This is case (c) earlier.

What if the program employs a complex net of conditional branches, such as in (d) earlier? In such a case, there may not be any way to compile it. This is one reason why array processors are special-purpose computers. They cannot execute certain programs that conventional computers find easy.

An important class of applications for array processors is image processing. With many satellites and space vehicles active, many images arrive at receiving stations around the world every day, and have to be processed.

An image taken from space is digitized, i.e., made up of many small dots, called pixels, arranged in rows and columns. The picture is transmitted to Earth by sending the values of the pixels row by row, or column by column. The value of a pixel depends on the number of colors sensed by the camera. It can be as little as a single bit, in case of a black and white (monochromatic) image. If the image contains many colors, or many shades of grey, the value of each pixel consists of several bits.

Image processing may be necessary if certain aspects of the image should be enhanced. A simple example is contrast enhancement. To achieve more contrast in an image, the program has to loop over all the pixels; bright pixels are made brighter, and dark ones are made darker. Another example is enhancing certain colors, which involve a similar type of processing.

Image processing is a highly-developed art, and is advancing all the time. Typically, an image processing program consists of a loop where all the pixels are scanned, and each pixel undergoes a change in value, based on its original value and the values of the pixels around it. A typical example is:

$$p[i,j] := p[i-1,j] + p[i+1,j] + p[i,j-1] + p[i,j+1] - 4p[i,j].$$

where $i$ and $j$ go over the rows and columns of the picture, respectively.

On a conventional computer, such a computation is done by a double-loop, over the rows and columns of the picture. On an array processor, however, the process is different, and much faster. The PUs are arranged in a two-dimensional grid, with each pixel connected to its four nearest neighbors. The image is read in and pixel $p[i,j]$ is stored in (say, location 32) of the memory of the PU in row $i$ and column $j$. The program executes one instruction of the form "get the contents of memory location 32 of your four nearest neighbors, add them and subtract 4 times your own location 32. Store the result as the new value of your location 32." This instruction is executed simultaneously by all PUs, thus processing $N$ pixels in the time it takes a conventional computer to process one pixel.

Another group of $N$ pixels may be stored in location 33 of all the PUs, and may be processed by a single instruction, similar to the one above.

This is a typical example of image processing, and it shows that a computer designed for such applications should have its PUs arranged in a two-dimensional array, with as many connections between the PUs as possible. Other applications may require a one-dimensional or a three-dimensional topology, but the two-dimensional ones are the most common. Figure 5.6 shows a typical array processor featuring a two-dimensional array of PUs with nearest neighbor connections.

If an application requires more than just nearest neighbor connection, several instructions may be necessary, each moving data between near neighbors.

## 5.7 Example: MPP

The Massively Parallel Processor (MPP) is a typical example of an array processor. It is large, consisting of 16K PUs. Each PU, however, is only 1-bit wide. The MPP is thus an example of a fine-grained architecture.

The MPP was designed and built for just one application, image processing of pictures sent by the landsat satellites. This is a huge task since each picture consists of eight layers, each recorded by a sensor sensitive to a different wavelength. Each layer is made of 3000×3000 7-bit pixels. Each picture thus consists of $504 \times 10^6$ bits, and many pictures are recorded each day. Anticipating the huge amount of data processing needed, NASA, in 1975, awarded a contract to the Goodyear Aerospace company to develop a special-purpose computer to do the task. The MPP was delivered in 1983.

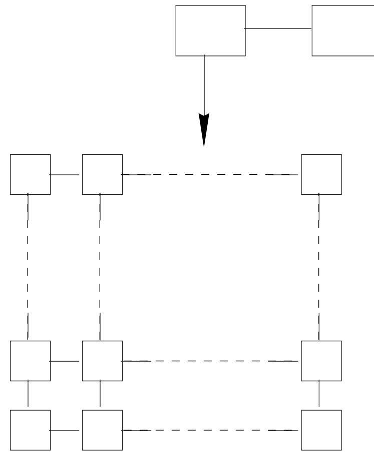


Figure 5.6. A two-dimensional array of PUs

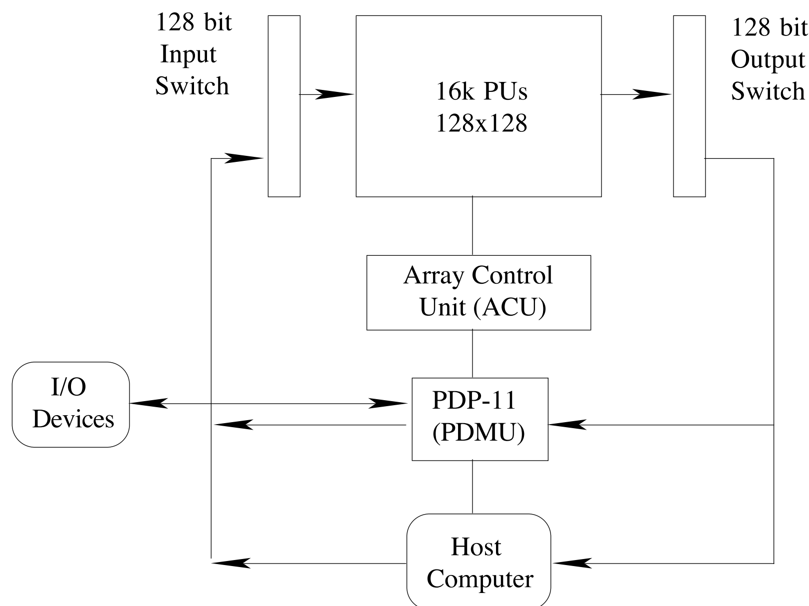



Figure 5.7. Block diagram of the MPP

The 16K PUs are arranged in a $128\times128$ grid (Figure 5.7) and are controlled by the Array Control Unit (ACU). The ACU itself is controlled by the Program and Data Management Unit (PDMU), which is a PDP-11 computer, and by the host computer, a VAX 11-780. The PDMU receives the compiled program from the host and sends it to the ACU instruction by instruction. It also is responsible for handling the disk, printer and other peripherals of the MPP. A third task of the DPMU is to move data from the output switch back to the input switch.

The host computer compiles MPP programs. It is also responsible for receiving the data from the satellite and sending it to the staging memory. From that memory, the data goes to the input switches which hold 128 bits. From the input switches, 128 bits are fed to the leftmost column of the array, and the program is written such that each PU receives a bit of data from its neighbor on the left, processes it, and sends it to its neighbor on the right. In normal operation, groups of 128 bits appear at the rightmost column all the time, and are sent to the output switches, from which they can be sent back to the host computer or to the staging memory, to be fed again into the leftmost column.

It is interesting to look at the architecture of a PU on the MPP. It is a simple, 1-bit wide, circuit

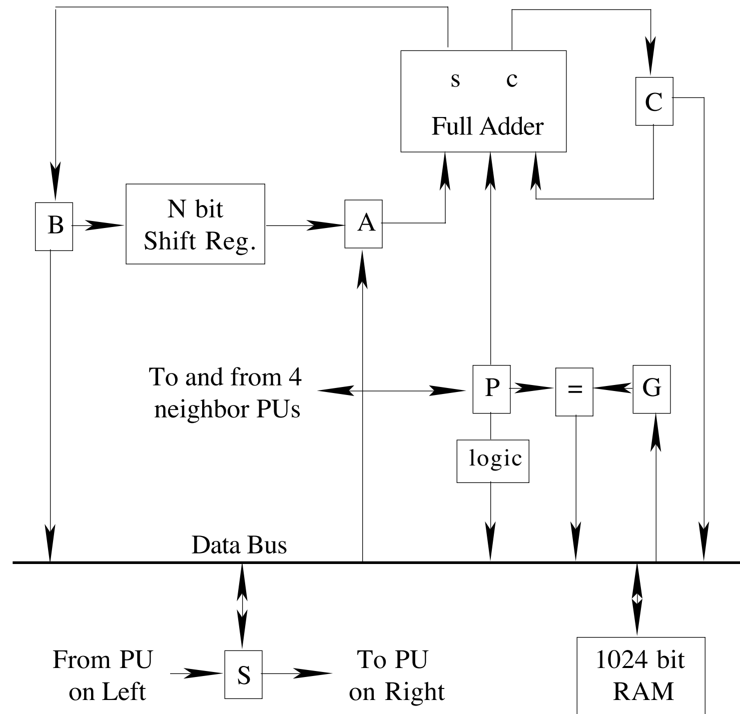



**Figure 5.8. A PU on the MPP**

consisting of six registers, a full-adder, a comparison circuit, a logic circuit, a and 1K×1 data memory.

The main operation performed by a PU (Figure 5.8) is adding numbers. The full adder adds three bits (two input bits plus an input carry) and produces two bits, a sum and a carry. Numbers of different sizes can be added using the shift register. It can be adjusted to 2, 6, 10, 14, 18, 22, 26, or 30 bits, and an $n$-bit number can be stored by adjusting the shift register size to $n-2$ bits, and holding the other 2 bits in the `A` and `B` registers. The other $n$-bit number to be added should come from the `P` register, bit by bit.

The `P` register is used for routing and for logic operations. It is possible, e.g., to instruct all the PUs to send their `P` register to their neighbor on the left. Each PU thus sends its `P` to the left and, at the same time, receives a new `P` from its neighbor on the right. The box labeled 'logic' can perform all the 16 boolean operations on two bits. Its inputs are the `P` register and the data bus. Its single output bit stays in the `P` register.

The `G` register is the mask bit. If `G`=0, the PU is disabled. It still participates in data routing and arithmetic operations, but not in the logic operations.

The `S` register is connected to the `S` registers of the neighbors on the left and right. The `S` registers of the extreme PUs on the left and right of the array are connected to the input and output switches.

The box labeled '=' is used for comparisons. The `P` and `G` registers are compared and the result is sent to the data bus.

## 5.8 Example: The Connection Machine

This is a very fine-grained SIMD computer with 64K PUs, each 1-bit wide, arranged as a 12-dimensional hypercube, where each of the 4K nodes is a cluster of 16 PUs. The entire machine is driven by a host computer, typically a VAX. The Connection Machine was designed by W. Daniel Hillis, as a Ph.D. thesis [Hillis 85], around 1985. The machine was designed to overcome the main restriction of older SIMD computers namely, rigid array connections.

Traditionally, the PUs of SIMD computers were connected in a two-dimensional grid. In such a configuration, it is easy for a PU to send a piece of data to any of its four nearest neighbors. Sending data to any other PU, however, requires several steps, in which the data is sent between near neighbors.

The main idea in the design of the connection machine was to make it easy for PUs to send *messages* to

each other, thereby simulating all kinds of connections. It turns out that this is easy to do in a hypercube, because of the natural numbering of the nodes. However, before we go into the details of message routing, let's look at the main architectural components of the machine.

The 64K PUs are organized as a 12-dimensional cube, with 4K corners called clusters. Each cluster is thus directly connected to 12 other clusters. A cluster is made up of a central circuit, called a *router*, and 16 PUs.

Each PU (Figure 5.9) is one bit wide, has sixteen 1-bit registers called *flags*, and a 4K×1 memory. There is a simple logic circuit to execute the instructions, but there is no ALU.

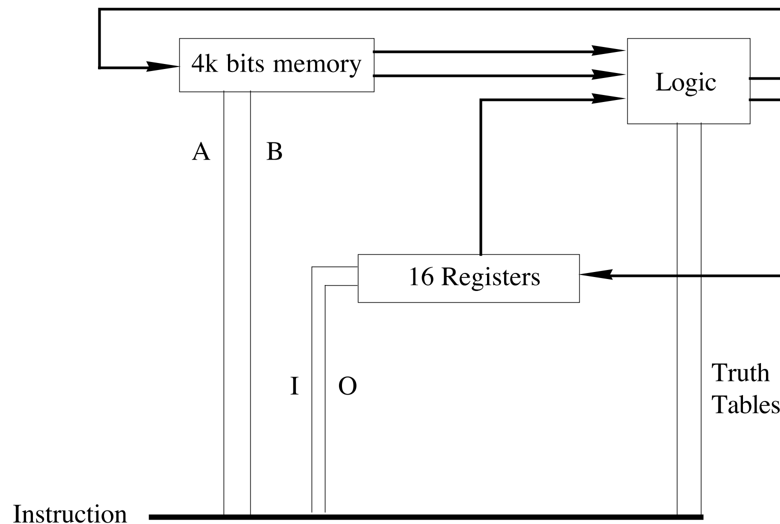



**Figure 5.9. A PU in the connection machine**

Since there is no ALU, instructions sent to the PU have no opcode. Instead, each instruction specifies three input bits, two output bits, and two truth tables, one for each output bit. A truth table is made up of eight bits, specifying the output bit for every possible combination of the three input bits. The instruction has two 12-bit address fields `A` and `B` and two 4-bit register fields `I` and `O`. The three input bits come from locations `A`, `B` in memory, and from the register specified by `I`. The two output bits go to location `A` and the register specified by `O`.

Examples: The two truth tables 01110001 and 00001111 create two output bits that are the sum and the carry of the three inputs. The truth tables 01111111 and 00000001 create the logical OR and logical AND of the three inputs.

The main feature of the connection machine, however, is the message routing mechanism. The program contains instructions to the PUs to send a message, and, operating in lockstep, they all create messages and send them simultaneously. The 16 PUs in a cluster are connected in a 4×4 grid. Thus, in the cluster, messages can be sent to nearest neighbors easily. However, if a message has to be sent to a PU in another cluster, than it should consist of an address part and a data part. The data bits are, of course, the important part, but the address part is necessary. It consists of a 12-bit destination cluster address, a 4-bit PU number within the destination cluster, and a 12-bit memory address of the destination PU.

The important part is the 12-bit cluster address. It is supplied by the instructions and may be the same for all PUs. It is a relative address. Each of its 12 bits specifies one of the 12 dimensions of the hypercube. Thus address 000110010000 should cause the message to be sent from the sending cluster—along the fifth, ninth, and tenth dimensions—to the destination cluster.

The message is sent by each PU to its router, and is accumulated by the router, bit by bit, in an internal buffer. When the router has received the entire message from the sending PU, it scans the 12-bit destination address. In our example, the first nonzero bit is the fifth one. The router clears this bit and sends the message to the fifth of its 12 router neighbors. That router does the same and ends up sending the message

to the ninth of its 12 immediate neighbors. When a router receives a message with all 12 bits cleared, it sends it to the destination PU in its own cluster (actually, to the memory of that PU).

It should be emphasized that the PUs operate in lockstep, but the routers do not! While messages are sent in the machine, all the buffers of a certain router may be full. If a message arrives at that router, it may have to refer it to another router. Referring means that the router sets one of the 12 bits of the destination cluster address, and sends the message in that direction. That message will therefore take longer to arrive at its destination. Message routing, therefore, is not done in lockstep and, once the program initiates the process of sending a message, it should wait until all the PUs have received their messages.

## 5.9 MIMD Computers

An MIMD Computer consists of several processors that are connected, either by means of shared memory or by special communication lines. In the former case, the MIMD computer is called a *multiprocessor*. In the latter case, it is called a *multicomputer*. Their main components are summarized in Figs. 5.10 and 5.11.

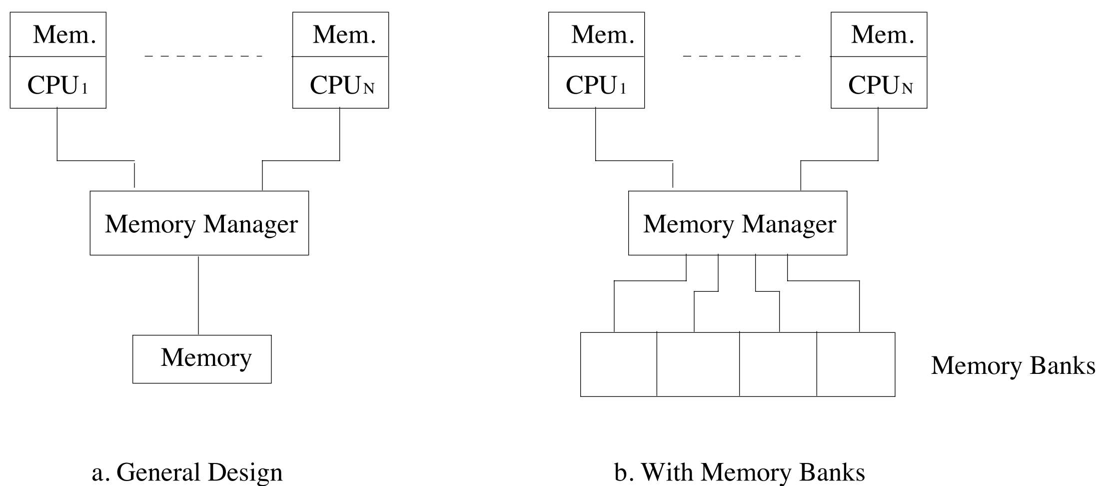



**Figure 5.10. A multiprocessor**

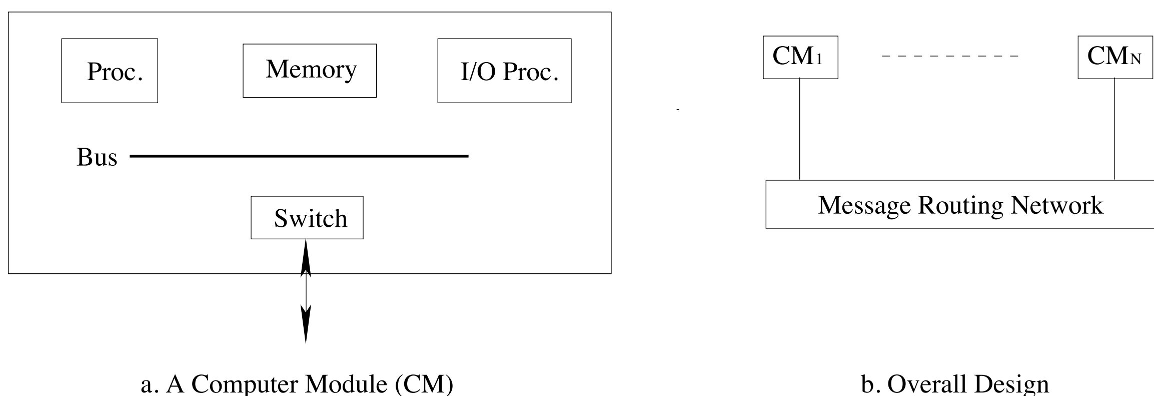



**Figure 5.11. A multicomputer**

A multiprocessor is conceptually simple. It is made up of units—each a processor with, perhaps, some local memory—all connected to the shared memory through the memory manager, a piece of hardware that uses priorities to determine which processor will get the memory next. A computer module consists of a

processor, its local memory, if any, its I/O processor, and a switch connecting the module to the rest of the multiprocessor.

The main problem in such a computer is memory conflicts. If the number of processors is large, a processor may have to wait an unacceptably long time for memory access. The solution is to use memory banks (Figure 5.10b) interleaved on the most significant bits. In the ideal case, processors use different banks, and never have to wait for memory. In practice, programs have different sizes, and as a result, some processors need less than one bank, others need more than a bank, and there is a certain number of memory conflicts.

In a multicomputer, all the memory is local, and the individual processors communicate by passing messages between them. The hypercube architecture is a popular way to connect processors in this type of computer.

## 5.10 Parallel Algorithms

The key to an understanding of MIMD computers is an understanding of parallel algorithms. We therefore start by looking at examples of algorithms designed specifically for MIMD computers. We describe one algorithm for multiprocessors and several for multicomputers.

### 5.10.1 Parallel Rank Sorting

1. This is an example of a shared-memory algorithm, to be run on a multiprocessor. We assume that $n$ numbers are to be sorted (with no repetitions), and $n^2$ processors are available (an overkill, but it simplifies the algorithm). The original numbers are stored in array $V$, and the sorted ones will be stored in array $W$. We declare a matrix $A_{nn}$, and assign each processor to one of the matrix elements (each processor is assigned indexes $i$ and $j$). The processors communicate only through shared memory. They are not connected in a grid or any other topology, and they do not send messages to each other.

- Step 1. Each processor calculates one matrix element:

$$A[i,j] = \begin{cases} 1, & \text{if } V[i] \le V[j]; \\ 0, & \text{otherwise;} \end{cases}$$

(this takes time 1 if we ignore memory conflicts). Note that $A$ is antisymmetric, so only $n^2/2$ processors are actually needed. Each calculates one matrix element and stores it in positions $A_{ij}$ and $A_{ji}$.

- Step 2. $n$ processors are selected out of the $n^2$, and each computes one of the sums $T[i] = \sum_{j=1}^{n} A[i,j]$. This takes time O($n$). Note that $T$ contains the *rank* of each element.

- Step 3. Scatter $V$ into $W$ by $W[T[i]] \Leftarrow V[i]$ ($n$ processors do it in time 1).

The total time is thus O($n$). Excellent, but requires too many processors. We can decrease the number of processors, which will make the method a little less efficient. Example: $V = (1, 5, 2, 0, 4)$ ($n = 5$). The results are:

$$\begin{bmatrix} 1 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 \end{bmatrix} \begin{matrix} 4 \\ 1 \\ 3 \\ 5 \\ 2 \end{matrix} \qquad \begin{bmatrix} 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 1 \end{bmatrix} \begin{matrix} 2 \\ 5 \\ 3 \\ 1 \\ 4 \end{matrix}$$

$$A[i,j] = V[i] \le V[j]; \quad A[i,j] = V[i] \ge V[j];$$

The sums on the right show how the numbers are to be placed in $W$. The matrix on the left results in descending sort, and the one on the right, in ascending order. (End of example.)

### 5.10.2 Parallel Calculation of Primes

2. Calculating primes. Here we assume that the multicomputer has a linear topology. Each processor (except the two extreme ones) has two immediate neighbors. We use a parallel version of the sieve of Eratosthenes. All the processors, except the first one, execute the same algorithm: A processor receives a sequence of numbers from its left neighbor, it assumes that the first number is a prime, and it prints it (or writes it on a disk, or sends it to a host, or stores it in shared memory). All the other numbers are compared to the first

number. Any that are multiples of the first number are ignored; those that are not multiples are sent to the neighbor on the right.

The first processor (that could also be a host) prints the number 2 (the first prime) and creates and sends all the odd numbers, starting at 3, to the right. The second processor receives and prints the number 3. It then receives 5, 7, 9, 11, . . ., discards 9, 15, 21, . . . and sends the rest to the right. The third processor receives and prints the number 5. It then discards 25, 35, 55, . . . and so on. An elegant algorithm.

Note that more and more processors join the computations, and work in parallel to eliminate the non-primes, to print one prime each, and to send more work to the right.

▸ **Exercise 5.3:** The amount of primes that can be calculated equals the number of processors. How can this limitation be overcome?

### 5.10.3 Some Short Examples

3. Sum 16 numbers. On a sequential computer this takes 16 (well, 15) steps. On a parallel computer, it can be done in $\log_2 16 = 4$ steps. We imagine 16 processors arranged in a binary tree. A host sends two numbers to each of the eight leaves of the tree. Each leaf sums one pair, and sends the sum to its parent. The parent adds two such sums and sends the result to *its* parent. Eventually, the root adds the two sums that it receives, and the process is over.

The root is also processor 0, which is responsible for the overhead (sending the data to the leaf processors). It does not seem to make sense to send many messages all the way down the tree just to add numbers. In practice, however, the tree of processors may be embedded in a larger topology (such as a hypercube), so the distance between processor 0 and the leaves cannot exceed the diameter of that topology.

The process can be visualized as a wave of activity, propagating from the leaves to the root. If more than 16 numbers are to be summed, the algorithm can easily be extended. The host sends the first 16 numbers to the leaves, waits one step (for the leaves to send their results to their parents), sends the next group of 16 numbers, and so on.

4. Sorting. We again assume a binary tree topology. The array of numbers to be sorted is divided into equal parts that are sent, by the host, to all the nodes. Each node sorts its part, waits for its sons to send it their parts (if it has sons), merges all the sorted streams, and sends the result to its parent. When node 0 is done, the process is complete.

Note that the higher a node is in the tree, the longer it has to wait for input from its sons. A possible improvement is to divide the original array of numbers into *unequal* parts and to send the bigger parts to the higher-placed nodes in the tree.

▸ **Exercise 5.4:** Discuss example 5, a parallel algorithm for numerical integration.

It is clear that these algorithms are similar. The host executes a program that sends information to the nodes by means of messages. The nodes execute identical programs (different from that of the host) that receive instructions and parameters from the host, perform calculations, combine the results with results received from the sons, and send the total result to the parent. When node 0 finishes, the process is over.

Sometimes internal communication isn't necessary, and individual nodes communicate directly with the host. Examples are:

6. Matrix multiplication. We assume no special topology. In this algorithm individual nodes communicate directly with the host. If $A$ and $B$ are matrices, then $C = A \times B$ is given by $C_{ij} = \sum_k A_{ik} B_{kj}$. Each node receives matrix $B$ and a row of $A$ from the host, calculates a row of $C$ and sends it either to node 0 or to the host.

These are examples of problems that can be solved by breaking them up into subproblems that can be done in parallel, with a lot of communication between them. There are problems that can be broken up into subproblems requiring very little communication. There are, of course, problems that don't lend themselves to parallel execution at all. In general, we wish we had a multicomputer where the individual processors could be arranged in any configuration (topology).

> "Lines that are parallel
> meet at infinity!"
> Euclid repeatedly,
> heatedly,
>   urged
> until he died.
> and so reached that vicinity:
> in it he
> found that the damned things
>   diverged.
>
> —Piet Hein (dedicated to Martin Gardner)

## 5.11 The Intel iPSC/1

This is a simple example of a hypercube. It was first delivered in 1985, based on experience gained by the Cosmic Cube at Caltech [Seitz 85]. The iPSC/1 is based on a host controlling a hypercube of nodes:

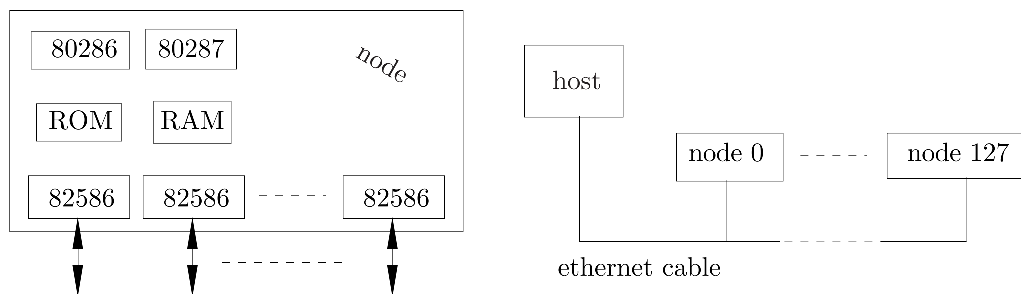



**Figure 5.12. The iPSC/1**

The host features an 80286 CPU (8MHz), 2Mb DRAM, 140Mb hard drive, a 1.2Mb floppy drive, a terminal, 60Mb tape, and Xenix operating system.

Each node contains (Figure 5.12) An 80286 (8MHz), 80287, 64Kb PROM, 512Kb DRAM, seven 82586 communication channels (LAN controllers) connected to the seven nearest neighbors, and one 82586 connecting the node to the host by means of an ethernet cable. An NX mini OS resides in each node.

A cube of 32 nodes is called a *unit*. There may be 1, 2, or 4 units in the iPSC/1, resulting in cubes of 5, 6, or 7 dimensions. A cube can be partitioned into subcubes.

Communications between the nodes is point-to-point. A node may also send a message directly to the host by specifying an id of −32768. The host can send messages to the nodes by means of the ethernet cluster. This is a multiplexor that connects the single host to all the nodes. The host may select either one node, several nodes, or all the nodes, and send a message.

To send or receive a message, a node executes a subroutine call. The subroutines are part of the OS, and execute 82586 commands. Here are the steps for sending a message:

- The program prepares the message in a buffer and calculates the number of the destination node. It calls the appropriate subroutine.

- The subroutine (part of the OS) decides which nearest neighbor will be the first to receive the message. It creates 82586 commands with the buffer address, the number of the receiving node, and the number of the neighbor. If the message is sent in the 'blocked' mode, the OS does not immediately return to the user program, but waits for an interrupt from the 82586.

- The 82586 reads the message from the buffer and sends it on the serial link. When done, it interrupts the 80286.

▪ The OS responds to the interrupt. If the message was sent in the 'blocked' mode, it restarts the user program.

The steps for receiving a message are:

▪ The message comes into one of the seven 82586s, which writes it in a buffer in the system area, and interrupts the CPU.

▪ The OS processes the interrupt. It gets the RAM address from the 82586, and checks to see if the message is for that node. If not, it automatically sends the message to one of the nearest neighbors.

▪ If the message is for this node, the OS checks to see if the user program is waiting for a message of the same type as the received message (i.e., if it is waiting in a blocked subroutine call). If yes, the OS transfers the message from its buffer to the user's buffer. Otherwise, the message stays in the system buffer, which may eventually overflows.

A 'blocked' send stops the sending program until the message is sent by the OS (but not necessarily received). Using 'unblocked send' is faster but, before sending the next message, the program must ask the OS (through function `STATUS`) whether the message has been sent (whether the buffer is available). Similarly, a 'blocked' receive is safe, but may delay the receiving program since, if no message has been received, the OS will put the receiving program on hold. In an 'unblocked' receive, the program first has to check if there is a message to be received.

Here is a list of the important procedures and functions used for message passing and cube management (we use Fortran syntax, but the same routines can be called from C).

▪ `CRECV(TYPE,BUF,LEN)` receives a message in the blocked mode. Parameter `TYPE` is a 32-bit cardinal. The programmer may assign meanings to several different types, and a program may want to receive only a certain type of message at any time. The type is the only clue that the receiving program has about the sender of the message. `BUF` is an address in the receiving node's memory, where there is room for `LEN` bytes. If the incoming message is longer than `LEN` bytes, execution halts.

▪ `CSEND(TYPE,BUF,LEN,NODE,PID)` sends (in the blocked mode) a message of length `LEN` and type `TYPE` from address `BUF`. The message is intended for process with id number `PID`, running on node # `NODE` (note that more than one process may be running on a node, so each process must be assigned an id by the host). A node number of $-1$ will send the message to all the nodes, but they will accept it selectively, depending on the type and current pid. To send a message to the host, write '`myhost()`' in the `NODE` field.

▪ `KILLCUBE(NODE,PID)` kills process `PID` running on node `NODE`. This is necessary since many node programs don't halt. They end by waiting for a message that never arrives, so they have to be killed.

▪ `LOAD('FILENAME',NODE,PID)` loads a program file into the memory of `NODE`, and assigns it the id number `PID`. This is used by the host to load nodes with (identical or different) programs. A `NODE` of $-1$ selects all nodes.

▪ `SETPID(PID)` sets the id number of a process. This is supposed to be used only by the host program.

▪ `WAITALL(NODE,PID)` waits for certain process (on a certain node) to finish. This used mostly by the host. Both `NODE` and `PID` can be $-1$.

The following are functions. They don't have nay parameters, and are used, e.g., as in '`i=infocount()`':

▪ `INFOCOUNT()` can be used to return the size (in bytes) of the most recent message received.

▪ `INFONODE()`, `INFOPID()` and `INFOTYPE()` return the number of the node that sent the most recent message, the process id, and the type of that message, respectively.

▪ `INFOPID()` is the same for process id.

▪ `MYNODE()`, `MYHOST()` and `MYNPID` return the node numbers of the node executing the call, the node number of the host, and the process id number of the current process, respectively.

▪ `NUMNODES()` returns the total number of nodes in the cube.

There is also a `SYSLOG` function where a node can send data to the system log file. This is useful for debugging.

▸ **Exercise 5.5:** How can a non-parallel program be run on the iPSC/1?

The iPCS/1 is now obsolete. In 1987, the iPSC/2 was delivered, based on the 80386. It has the same architecture as the iPSC/1, so all the procedures and functions described earlier apply to the iPSC/2. In 1990, the iPSC/860 was introduced, with a peak rate of 7.6Gflops.

## 5.12 Vector Processors

The term vector processor means a computer in which certain ALU circuits operate as a pipeline (Figure 5.13). The best example is adding floating-point numbers. This operation, discussed in Chapter 2, requires the following steps: 1. Compare the exponents. 2. Shift one of the fractions to the right. 3. Add the fractions. 4. Combine the sum with the common exponent, and normalize the result.

In a conventional ALU, a pair of numbers is entered into the floating-point adder, and goes through each of the four stages. When one of the stages is active, the other ones idle. In a vector processor, pairs of numbers are fed into the floating-point adder all the time, and all the stages are active, each operating on a different pair of numbers. This way, all four stages are continuously active, thereby speeding up the computer.

This kind of operation is feasible only when many pairs of floating-point numbers should be added; it is not useful for adding one pair. Fortunately, this is a common case. A loop of the form

```
for i:=1 to 100 do x[i]:=x[i]+y[i];
```

is very common in scientific programs, where operations on arrays are done all the time. In a conventional computer, the statement above is compiled into a loop consisting of several machine instructions, and the loop is repeated 100 times. In a vector processor, the statement is compiled into one machine instruction, and this instruction is executed by sending the 100 pairs of numbers to the ALU as fast as possible, so that all stages of the floating-point adder are kept busy.

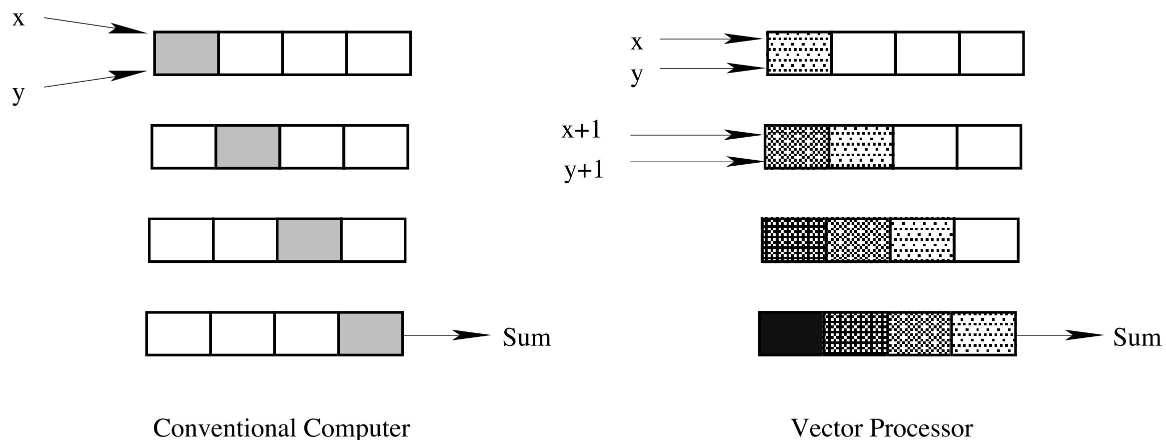



**Figure 5.13. Pipelining in the ALU**

There are two main types of vector processors, register-register and memory-memory. In the first type, numbers are sent to the ALU from special vector registers. In the second type, they are sent from memory. The CRAY computers are an example of the first type; the Cyber 205 is an example of the second one.

The CRAY-1 computer has eight `V` registers, each with room for 64 numbers. A typical vector instruction is '`FADD V1,V3(55)`', which adds the first 55 pairs of numbers in registers `V1` and `V3` as floating-point numbers. On the Cyber 205, the instruction '`FADD A,B(55)`' works similarly, except that it fetches the 55 pairs from arrays `A` and `B` in memory.

A register-register vector processor seems faster since the operands are already in registers, but how do they get there? In some cases the operands go into the registers as results of previous operations. More commonly, though, they are fetched from memory. Since only one number can be read form memory at any time, this seems to defeat the purpose of vector processing. This is why vector processors use memory banks interleaved on the *least significant bit*. In such a memory, successive memory words are located in

consecutive banks, speeding up the process of reading an entire array. To read the elements of an array from memory, the processor initiates a read from the first bank and, without waiting for the result, initiates a read from the second bank, and so on. When results start arriving from the memory banks to the processor, they do so at a rate much greater than the memory bandwidth.

Figure 5.14 illustrates the time it takes to performs an operation on $n$ pairs of numbers, as a function of $n$. Three types of computers are compared, a conventional (also called serial, sequential, or scalar, a reg-reg vector processor, and a mem-mem vector processor.

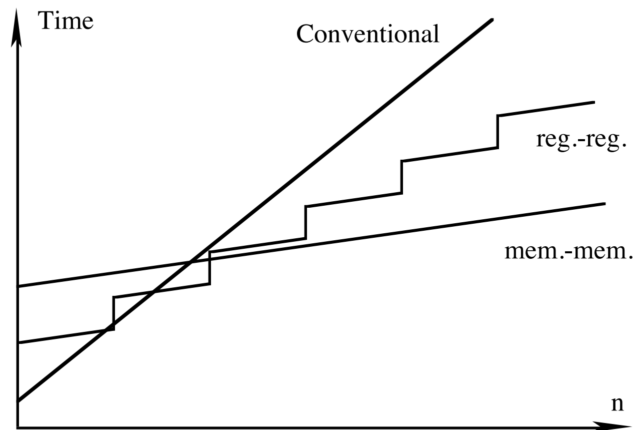



**Figure 5.14. A timing diagram**

The curve for a scalar computer is a straight line. Such a computer takes twice as long to operate on 24 pairs as it takes to operate on 12. The curve for the mem-mem vector is also a straight line, but with a much lower slope. Such a computer fetches the numbers from memory at a constant rate, sends them to the ALU at the same rate, and the results are obtained at that rate. However, it should be noted that, for small values of $n$, the line is higher than the one for the scalar computer. This is because it takes time to initialize (or to setup) the vector circuits before they can be started.

The situation with the reg-reg vector is different. Assuming that each vector register has room for 64 numbers, it takes a linear time to operate on values of $n$ between 1 and 64. In this range the curve is a straight line with about the same slope as in the mem-mem case. However, if the program needs to operate on 65 pairs, two vector instructions are necessary, which increases the execution time required. This is why the curve looks like a staircase, where each part is a straight line, slightly sloping.

It should be noted that a good compiler is an essential part of any vector processor. A good compiler, of course, is always good to have, but on a vector processor, it can make a significant difference. The compiler should identify all the cases where vector instructions should be used, which is not always easy. Obviously, vector instructions should not be used in cases such as `z:=x+y`, and should be used in cases such as

```
for i:=1 to 100 do x[i]:=x[i]+y[i];
for i:=1 to 100 do x[i]:=x[i]+x[i];
for i:=1 to 100 do x[i]:=x[i]*5;
```

However, it is not easy for a compiler to recognize that in a case such as
`for i:=1 to 100 do x[i]:=x[i]+x[i-1],`
a vector instruction cannot be used. Such a case should be compiled into a loop where each pair is added only after its predecessor has been completely added.

## 5.13 The Von-Neumann Bottleneck

The von Neumann design of a computer is more than fifty years old and is still used today, even though technologies have changed a lot. It is based on a processor, memory, and a bus connecting them. Those working on non von Neumann computers call this bus the von Neumann bottleneck because of the way the machine works. At any given time, a single word can be sent between the processor and memory, preceded by its address. The address is also sent on the same bottleneck. The task of a computer program, any program, is to change the contents of memory in some major way. To do this, it has to pump addresses and words of data through the single bus back and forth, all the time, which certainly justifies the name bottleneck (see page 10). It so happens that a large part of the traffic in the bottleneck is not even useful data but names of data (i.e., addresses), and numbers and operations needed only to compute addresses.

Before a piece of data can be written to (or read from) memory, its address must be sent. The processor can get the address in one of three ways as follows:

1. It may have the address stored in a register. This is simple.

2. The address may be stored in memory and should be read from memory (as data) before it can be used (i.e., before it is sent to memory as an address). In order to read the address from memory, *its* address must first be sent to memory. Its address is obtained in one of the three ways described here.

3. The address has to be calculated by the processor. The processor either performs a simple operation, such as incrementing the PC, or the processor must execute some instructions. In the latter case, it has to fetch those instructions through the bus, which means it has to send their addresses first, through the same bus.

We thus end up with a design for a computer which must push huge quantities of data and addresses through the same bus sequentially, one word at a time. This has the drawback that von Neumann computers are sequential and therefore inherently slow. It also has another important consequence, namely the intellectual bottleneck that has kept us, the users, tied to word-at-a-time way of thinking, instead of encouraging us to think in terms of larger units of the problem. Traditional programming is therefore mostly involved with the details of moving the large quantities of words through the bus, and of finding places in memory to store them. This is also why traditional programming languages use variables, why it is so hard for us to imagine a programming language without variables, and why people keep developing higher level languages that are supposed to take some of the burden (of organizing the data items) off the back of the programmer.

## 5.14 Associative Computers

One approach to the design of a non von Neumann computer is the associative computer. This design is based on the concept of associative memory (or content addressable memory) whose definition is: a memory where a word is accessed by its content rather than by its address. In an associative memory there are no addresses and a word is accessed (read or written into) by specifying its content or part of it. An associative computer is a special-purpose machine, since not every application lends itself to this kind of execution. Certain tasks, such as word processing, are easier to program on a conventional computer with addressable memory. However, given the right kind of problem, an associative computer can be fast, because certain associative memory operations are executed in parallel.

The most important associative memory operation is the search. In this operation, the program searches the memory for a certain pattern, and all the memory words containing this pattern respond to the search. The search is done in parallel on all the words, which is one of the main advantages of the associative computer. The pattern specifies binary values in certain bit positions in the word. A possible pattern is, for example, a 1 in bit position 3, a 1 in position 7, a 0 in position 8, etc. The pattern is sent, on a bus, to all the words in memory, and each word has logic to compare its content to the pattern. If a match is found, the word is considered a *responder*. Memory read and write are performed on responders only, which is why the two concepts of search and response are important.

The above discussion shows the two main features of associative memory. Its size, in principle, is unlimited, and it requires more hardware than a conventional, addressed memory.

The size of a conventional memory is limited because a conventional computer has to operate on addresses. There must be an address bus and registers, such as the PC, that contain addresses. The sizes of the address bus and those registers limits the maximum size of an address and this, in turn, limits the

maximum size of memory. In an associative memory, there are no addresses and, therefore, no limit on the size of memory. There are, of course, practical limits to the size of those memories. Note that, even though words do not have addresses, they are still ordered. There is such a thing as the first word in the associative memory, the second word, the last one, and so on. This order is important and is used in certain 'read' operations. However, it is impossible to refer to a word in memory by its position. A word can only be accessed by its content.

Since an associative memory performs certain operations in parallel, it requires more hardware than a conventional memory. Each word must have enough hardware to perform the search and the other parallel operations. Conventional memory, in contrast, requires less hardware since it is sequential and operates on one word at a time.

The pattern is specified by means of two registers, the comparand and the mask. The comparand contains the values searched for, and the mask contains the bit positions. In the example above, the comparand should contain a 1 in bit position 3, a 1 in position 7, a 0 in position 8, etc., while the mask should have 1's in positions 3, 7, 8, ... Two registers are needed, since with only a comparand, it would be impossible to tell which comparand bits are part of the pattern and which are unused. Figure 5.15 shows the two registers connected to all the memory words by means of the memory bus.

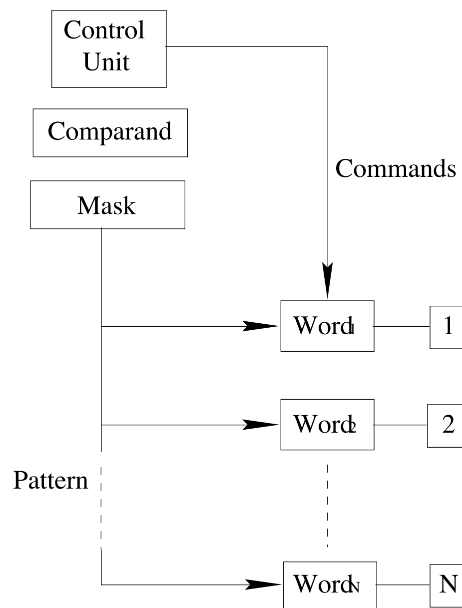



**Figure 5.15. The comparand and mask registers**

Before getting into the details of the various memory operations, let's try to convince ourselves that the entire idea is useful, by looking at certain applications.

Example 1. Finding the largest number. On a conventional computer, finding the largest of $M$ numbers takes $M$ steps. We will show that, on our associative computer, the same process takes fixed time. We start by loading both the comparand and the mask register with the pattern $10\ldots00$. When the process ends, the comparand will contain the largest number. We start by performing a 'search', and ask if there are 'any responders'. If the answer is yes, then we know that the largest number has a 1 in the leftmost position. We accordingly keep the 1 in that position of the comparand. If the answer is no, then we clear the leftmost position of the comparand. The comparand now contains $x0\ldots00$, where $x$ equals the leftmost bit of the largest number. Next, we set the comparand to $x10\ldots00$ and shift the mask to the right $010\ldots00$. We repeat the process and get a value of $xy0\ldots00$ in the comparand, where $y$ equals the largest number's second bit from the left. The loop is repeated $N$ times, where $N$ is the word size, and thus takes fixed time. The execution time is independent of $M$, the number of words.

▸ **Exercise 5.6:** Describe the process for finding the smallest number.

Example 2. Sorting $M$ numbers is now a trivial process. The largest number is selected as shown earlier, and the comparand is stored at the start of an $M$-word array in conventional memory. Since the comparand contains a copy of the largest number, a 'search' will identify that word in memory, and it should be cleared or somehow flagged and disabled (see below). Repeating the process would get the second largest number and so on. The entire sort thus takes $M$ steps, instead of a best case of $M \log M$ steps on a conventional computer. Since sorting is such a common application of computers, this example is a practical one.

Example 3. Payroll is a typical example of a business application that requires a sort. In a simple payroll program, records are read from a master file—each containing the payroll information of one employee—pay checks are prepared, and the (modified) records are written to a new master file. The master file is sorted by employee id number (or by social security numbers). As a record is read from the master file, the program compares it to *modification records* stored in memory, to see if there are any modifications to the payroll information of the current employee. On a conventional computer, the modification records are either sorted in memory or are written, in sorted order, on a file. On an associative computer, the modification records may be stored in memory in any order. When an employee record is read off the master file, the social security number is used as a pattern to search and flag all the modification records for that employee. Those modification records are then read, one by one, and used to produce the pay stub and the modified record of the employee.

This example is very general, since there are many business applications which work in a similar way, and may greatly benefit from parallel searches in an associative memory. It also shows the importance of an architectural approach which combines an associative and a conventional computer.

These examples, although simple, illustrate the need for certain instructions and features in an associative computer. The most important of those are listed below.

- Instructions are needed to load the comparand and the mask, either from other registers, from a conventional memory, or with a constant. It should also be possible to perform simple arithmetic operations, logical operations, and shifts on the two registers. It should also be possible to save those registers in several places.

- After a search, a special instruction is needed to find out whether any memory words have responded.

- Conventional instructions are necessary to perform loops, maintain other registers, do I/O, etc.

- Since we rarely want to perform a search, a sort, or any other operation on *all* the words in memory, there should be more hardware for enabling and disabling words. As a result, the response store of each word should be extended from 1 to 2 bits. The first bit (the response bit), indicates whether the word is a responder to the current search. The second bit (the select bit), tells whether the word is enabled and should participate in the current operation, or whether it is disabled and should be ignored.

  Suppose that it is necessary to search a certain array. How can the program enable the words that belong to the array, and disable the rest of the associative memory? The key is to uniquely identify each data structure stored in the associative memory. Our array must be identified by a unique key, and each word in the array should contain that key. A search is first performed, to identify all the words of the array. Those words are then enabled by copying the response bits to the select bits of the response store.

  This discussion shows the importance of the response store, and as a result, it seems reasonable to add a third, auxiliary, bit to it. The auxiliary bit could be used to temporarily store either the response bit or the select bit. It also makes sense to add instructions that can save the response store, either in conventional memory or in the associative memory itself.

- As a result of the discussion above, it is clear that words should be large, since each should contain the data operated on, and a key (or several keys) to identify the word.

- There are two ways to read from the associative memory. A 'read' instruction can either read the first responder (in which case, it should be possible to turn off that responder after the read), or it can read the logical OR of all the responders. In the payroll example above, responders should be read one by one.

- As a result of these points, designers have concluded that a good way to design an associative computer is as a combination of a conventional computer and an associative one. In such an architecture, the

conventional computer is the host, and the associative computer is a peripheral of the host. The two computers can process different parts of the same application, with the conventional computer executing the sequential parts, sending to the associative computer those parts that involve parallel memory operations.

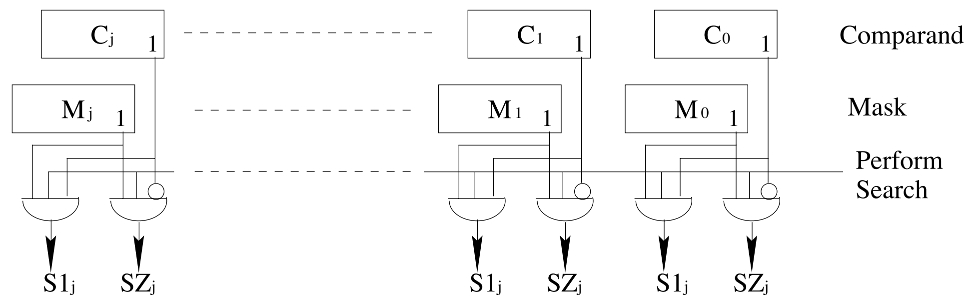



**Figure 5.16. Searching an associative memory (part 1)**

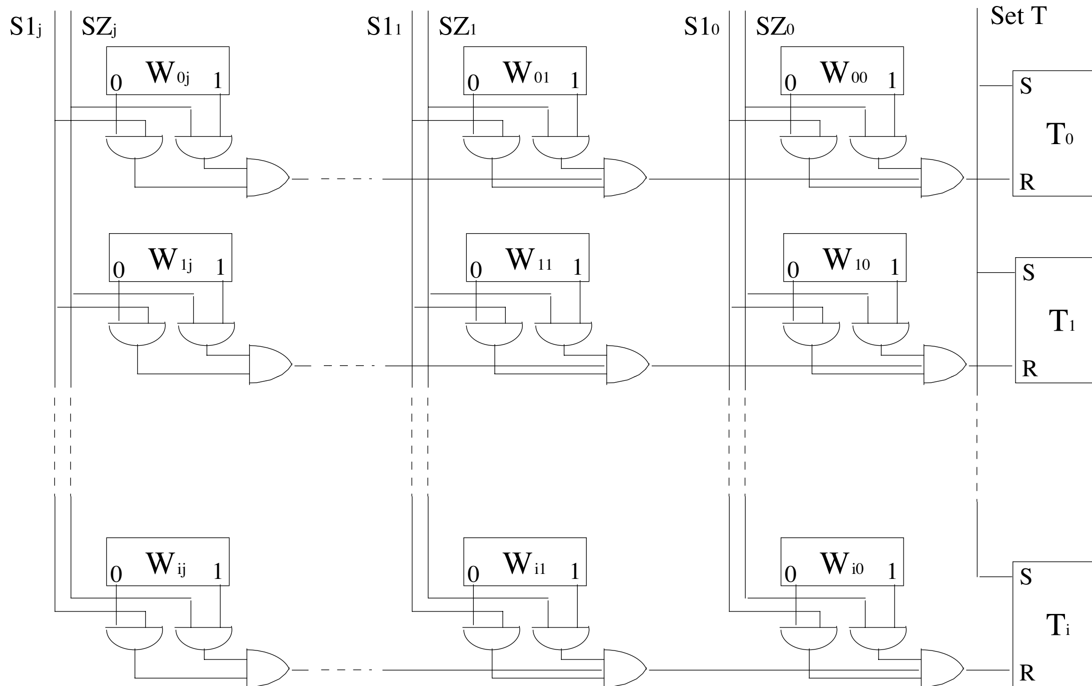



**Figure 5.16. Searching an associative memory (part 2)**

Figure 5.16 illustrates the search operation in an associative memory. The control unit executes a search by sending a 'set T' signal, followed by a 'perform search' signal. The first signal sets all the T bits of the response store. The second signal clears the response bits of all words that don't match the pattern.

## 5.15 Data Flow Computers

Another approach to the design of a non von Neumann computer is the data flow idea. A data flow computer resembles a von Neumann machine in one respect only, it has addressable memory. In all other respects, this computer is radically different. Figure 5.17 lists some of the differences between a von Neumann and a data flow computer and can serve as an informal definition of a data flow computer.

What is the earliest time that an instruction can be executed? Consider an instruction such as '`ADD A,B`'. In principle, this instruction can be executed as soon as both its operands `A`, `B` are ready. In a von Neumann machine, execution is sequential and our instruction may have to wait for its turn. In the sequence:

**A von Neumann Computer**

1. There is a single memory where both the instructions and the data (the operands) are stored
2. An instruction knows what its operands are and it fetches them during its execution
3. The operand of an instruction consists of just the data
4. Execution is sequential. An instruction is executed after its predecessor has been executed (except for jumps, which change the order of execution)
5. The von Neumann bottleneck limits the efficiency of the computer.
6. Only one instruction can be executed at a time.

**A Data Flow Computer**

1. There is an instruction store and a separate, *matching store* for the operands.
2. The operands know what their instruction is (what their destination address is), and they go to the instruction store and fetch a copy of the instruction.
3. The operands are called *tokens*.
4. Execution is not sequential. An instruction is executed at the *earliest possible time*.
5. There is nothing to cause a bottleneck (no main memory bus).
6. There are several ALUs, so several instructions can be executed simultaneously.

**Figure 5.17**. Comparing data flow and conventional computers

```
  .
STO R1,A
  .
STO R2,B
SUB R1,C
ADD A,B
```

both `A` and `B` are ready after the second `STO` and, at that point, the `ADD` instruction can, in principle, be executed. Let's assume, however, that the `SUB` instruction also becomes ready at this point. The `SUB`, of course, is executed first, because of the order in which the instructions were written. The `ADD` has to wait.

In a data flow computer, our `ADD` instruction wouldn't have to wait. It would be executed as soon as its operands are ready, which implies that it would be executed in parallel with the `ADD`. Thus, one important aspect of data flow computers is parallel execution. A data flow computer should have a number of PEs (processing elements) where several instructions can be executed simultaneously.

Research on data flow computers started in the late 1960s, both at MIT and at Stanford university. The first working prototype, however, was not built until 1978 (at the Burroughs Corp.). Since then, several research groups, at Texas Instruments, UC Irvine, MIT, University of Utah, and Tokyo University, have been exploring ways to design data flow computers, and to compile programs for them. Today, several working prototypes exist, but not enough is known on how to program them. A short survey of data flow computers is [Agerwala and Arvind 82].

Since the data flow concept is so much different from the sequential, von Neumann design, it takes a conceptual breakthrough to design such a machine. This section describes one possible implementation, the Manchester data flow computer, so called because it was developed and implemented in Manchester, England [Watson and Gurd 82].

To concentrate our discussion, we start with a simple, one-statement program. Figure 5.18a shows a common assignment statement. In a conventional (von Neumann) computer, such a statement is typically executed in four consecutive steps (Figure 5.18b). However, in a data flow computer, where several instructions can be executed simultaneously, we first have to identify the parallel elements of the program. The example is so simple that it is immediately clear that the two multiplications can be executed in parallel. The graph in Figure 5.18c exposes the parallelism in the program by displaying the two multiplications on the same level. Note that there are only three instructions. We later see that there is no need for a final `STO` instruction. Note also that, since execution is not sequential, instructions can be stored anywhere in memory. They do not have to be stored sequentially. Addresses 2, 7, and 11 have been assigned arbitrarily to the three instructions.

The first multiplication instruction '$*$', at address 11, generates a product ($=6$) that should become the left operand of the subtract '$-$' instruction at address 7. The '$*$' therefore generates a result that consists of

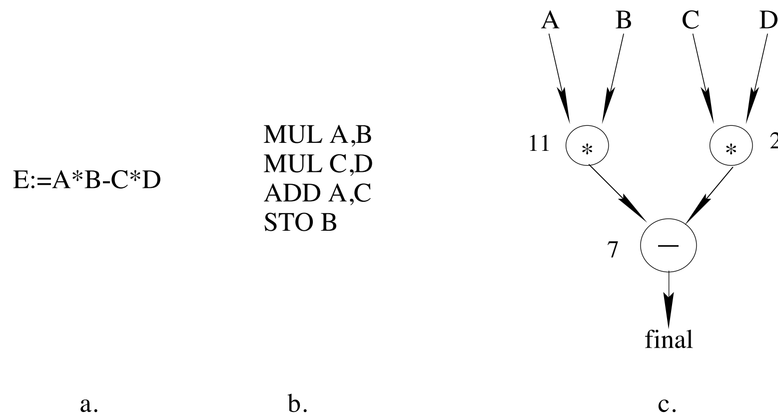



**Figure 5.18**. (a) The program, (b) conventional execution, (c) the data flow graph

the product and the destination address, 7. Such a result is called a *token*. A token is a result generated by an instruction. It can either be an *intermediate* result or a *final* one. If the token represents an intermediate result, then it should serve as an input operand to some other instruction and it should contain the address of that instruction. If it is a final result, however, then it should be stored in memory (or be output) and should contain the memory address where it should be stored. In either case, the token consists of the following fields:

- A value (which could be integer, floating-point, boolean, or any other data type).
- A type. This describes the type of the 'value' field.
- A one-bit field to indicate an intermediate or a final result.
- A destination address. If the token is intermediate, this is the address of an instruction in the instruction store, where the token is headed. Otherwise, this is an address (in the host's memory, see later), where the token will be stored for good.
- Handedness. Our example token goes to the '−' instruction at address 7 as its left input operand. The right input comes from the '*' instruction at address 2. Each (intermediate) token should carry its handedness, except if it is destined for a single operand instruction (such as a `DUP`, see later).
- Bypass matching store? A one-bit field indicating whether the token should go to the matching store (see later) or should bypass it.
- A *tag* field (see later).

The seven tokens used in our example are shown (first five fields only) in the table

| Number | Name | Value | Type | Interm/Final | Address | Hand. |
|---|---|---|---|---|---|---|
| 1 | A | 2 | integer | int. | 11 | L |
| 2 | B | 3 | integer | int. | 11 | R |
| 3 | C | 4 | integer | int. | 2 | L |
| 4 | D | 5 | integer | int. | 2 | R |
| 5 | 11–7 | 6 | integer | int. | 7 | L |
| 6 | 2–7 | 20 | integer | int. | 7 | R |
| 7 | final | -14 | integer | fin. | 100 | NA |

The next point to discuss is the matching of tokens. Tokens 1 and 2 in our example are the operands of the '∗' instruction at 11. Before attaching themselves to that instruction, they should somehow meet and make sure they constitute a pair. Tokens are matched by destination address and handedness. Tokens 1 and

2 match since they have the same destination address (= 11) and opposite values of handedness. Tokens 5 and 6 also match because of the same reason.

The matching is done in a special memory, the *matching store*. A token arriving at the matching store searches the entire store to find a match. If it does not find any, it remains in the matching store, waiting for a matching token. If it finds a match, it attaches itself to the matched token and the two tokens proceed, as a pair, to the instruction store.

▸ **Exercise 5.7:** What kind of memory should the matching store be?

In the instruction store, the two tokens pick up a copy of the instruction they are destined for, and proceed, as a *packet*, to be executed by an available PE. Figure 5.19 illustrates the basic organization of our computer so far. Note that the buses connecting the three main components are different since each carries a different entity.

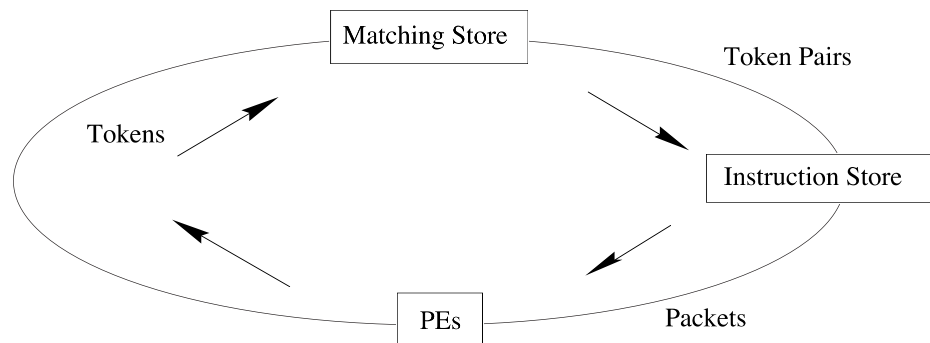



**Figure 5.19**. Data flow computer organization (1st version)

The basic organization of the particular data flow computer discussed here is a ring consisting of the three main components—the matching store, instruction store, and PEs. As with any other loop, it is important to understand how it starts and how it stops. A good, practical idea is to add a *host* computer to help with those and with other operations. The host performs tasks such as:

- Injecting the initial tokens (A, B, C, and D in our example) into the data flow machine (the main ring).
- Receiving the final tokens and storing them in its memory (to be printed, saved on a file, or further processed by the host).
- Compiling the program for the data flow computer and sending it to the instruction store.
- Executing those parts of the program that do not benefit from parallel execution.

In general, the host may consider the data flow computer an (intelligent) peripheral. The host may run a conventional program, executing those parts of the program that are sequential in nature, and sending the other parts to to be executed by the data flow computer. The results eventually arrive back at the host, as tokens, and can be saved, output, or further processed. The host is interfaced to the data flow computer by means of a switch, placed in the main ring between the PEs and the matching store (Figure 5.20).

The switch routes the initial tokens from the host to the matching store, where they match and proceed to the instruction store, starting the data flow operation. Any tokens generated in the PEs pass through the switch and are routed by it to one of three places. A token destined for a double-operand instruction is sent to the matching store. A token destined for a single-operand instruction is sent directly to the instruction store. A token representing a final result is routed to the host.

At the end of the program, the last final result is generated and is sent to the host. At that point, the data flow computer should have no more tokens, either on its buses on in the matching store. This, obviously, is the condition for it to stop. Note that certain bugs in the program can cause the data flow computer to run forever. It is possible, for example, to have unmatched tokens lying in the matching store,

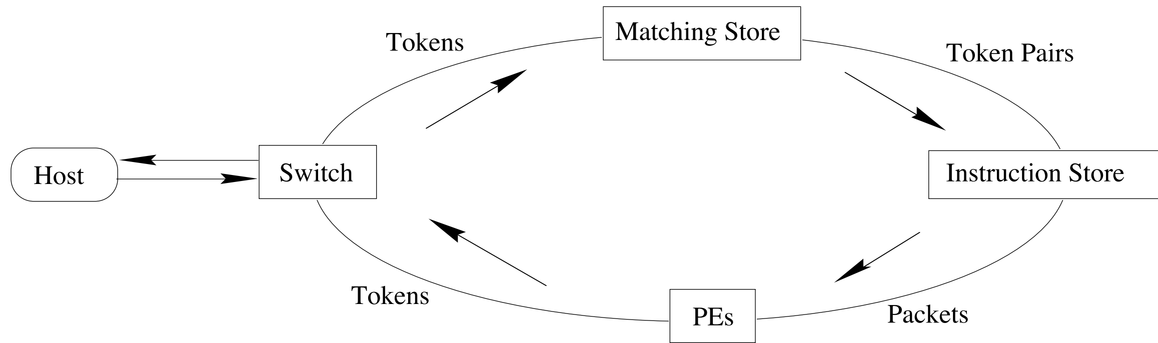



**Figure 5.20**. Data flow computer organization (2nd version)

waiting for a match that never comes. It is easy to create a situation where a token is executed by an instruction, generating an identical token, which causes an infinite loop. There can also be a case where tokens are duplicated without limit, flooding the matching store and clogging the buses.

Similar situations are also possible on a conventional computer and are dealt with by means of interrupts. A well-designed data flow computer should have interrupts generated by an overflowing matching store, overflowing queue (see below), a timer limit, and other error situations. However—since the data flow computer does not have a PC, and there are no such things as the next instruction or a return address—conventional handling of interrupts is impossible, and the easiest way to implement interrupts is to send them to the host.

The next step in refining the design of the main ring is to consider two extreme situations. One is the case where no PEs are available at a certain point. This may happen if the program has many parallel elements, and many instructions are executed simultaneously. In such a case, the next packet (or even more than one packet) has to wait on its way from the instruction store to the PEs. Another hardware component, a queue, should be added to the ring at that point. The PE queue is a FIFO structure that should be large enough not to overflow under normal conditions. If it does overflow, because of a program bug, or because of an unusual number of parallel instructions, an interrupt should be generated, as discussed earlier.

The other extreme situation is the case where a token arrives at the matching store while the store is busy, trying to match a previous token. This is a common case and is also handled by a queue, the *token queue*. All tokens passing through the switch go into the token queue, and the top of the queue goes to the matching store when the store becomes ready. The token queue should be large because it is common to have large quantities of tokens injected from the host, all arriving at the queue on their way to the matching store.

The final design of the ring is shown in Figure 5.21. Note the special bus connecting the token queue to the instruction store. This is used by tokens that do not need a match and should bypass the matching store. Another bus, not shown in the diagram, connects the host to the instruction store. It is used to load the program in the instruction store before execution can start.

To better understand how the data flow computer operates, we look at a few more simple programs.

Example 1. `E:=A*B+C/A+4×A↑2;` This is a short program consisting, as before, of one assignment statement. The difference is that variable `A` is used here three times. In a conventional computer, such a situation is simple to handle since `A` is stored in memory, and as many copies as necessary can be made.

```
LOD R1,A {\tenrm 1st use}
MUL R1,B
LOD R2,C
DIV R2,A {\tenrm 2nd use}
LOD R3,A {\tenrm 3rd use}
MUL R3,A {\tenrm 4th use}
MUL R3,\#4
ADD R1,R2
```

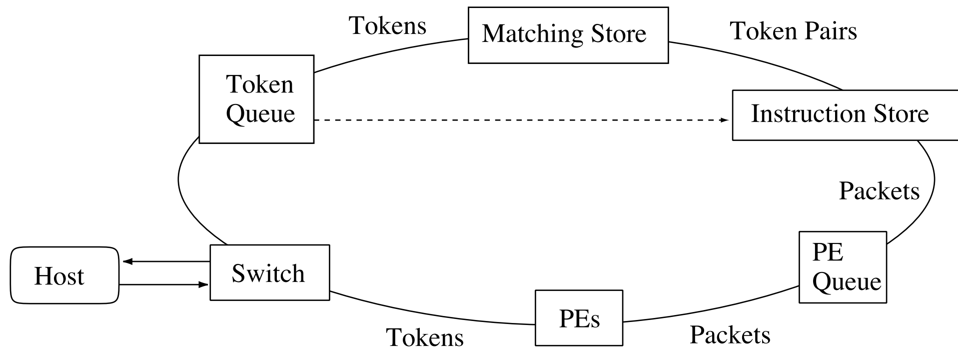



**Figure 5.21**. Data flow computer organization (3rd version)

```
ADD R1,R3
STO R1,E
```

In a data flow computer, however, there are no variables stored in memory; only tokens flowing along buses (arcs of the ring). In order to use `A` several times, extra copies of `A`'s token have to be made. We now realize that a new instruction, `DUP`, is needed, to duplicate a token. This is a common instruction in data flow computers and is an example of an instruction with one input and two output tokens.

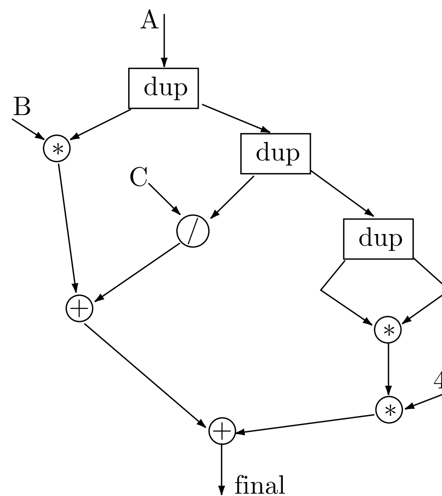



**Figure 5.22**. A simple data flow graph with DUP instructions

The data flow graph is shown in Figure 5.22 and it raises two interesting points. First, the graph is six levels deep, meaning that it takes six cycles of the ring to execute (compare this to the 10 instructions above). Second, one '$*$' instruction uses the immediate constant 4. Instead of injecting or generating a special token with a value of 4, it is better to have the instruction contain the immediate quantity.

We thus arrive at an instruction format for a data flow computer, containing the following fields (Figure 5.23):

| Opcode | Address | Hand. | I/F | M/B | 0 | Address | Hand. | I/F | M/B |
|---|---|---|---|---|---|---|---|---|---|

| Opcode | Address | Hand. | I/F | M/B | 1 | Immediate Input Operand |
|---|---|---|---|---|---|---|

**Figure 5.23**. Instruction formats on a data flow computer

1. OpCode
2. Destination address for the first output token.
3. Handedness for the first output token.
4. First token is an intermediate or a final result (I/F).
5. First token needs matching or should bypass the matching store (M/B).
6. A zero in this field indicates that fields 7–10 are a second output token; a one indicates they are a second, immediate, input operand.
7. Destination address for the second output token.
8. Handedness for the second output token.
9. Second token is intermediate or final result (I/F).
10. Second token needs matching or should bypass the matching store (M/B).

There are 10 fields, but fields 3–6, 8–10 consist of a single bit each. If field 6 is one (meaning that the second operand is an immediate input operand), then fields 7–10 contain the immediate quantity.

This format allows for instructions with either one or two outputs. If the instruction generates only one output, then one of its input operands can be immediate. Note that our instructions can have any number of input operands, although in practice they have just one or two.

Example 2. Any non-trivial program has to make *decisions*. In a conventional computer, decisions are made by the various comparison instructions and are executed by the conditional branch instructions. In a data flow computer, some 'compare' instructions, and a conditional branch instruction (`BRA`) have to be designed. The 'compare' instructions are straightforward to design. Each receives two input tokens, compares them, and outputs one *boolean* token. The two input tokens should normally be of the same type, and their values are compared. Several such comparisons should be implemented in a data flow computer, the most important ones being `EQ`, `GE`, and `GT`. Note that there is no need for a `LE` and `LT` instructions since a '`LE A,B`' instruction is identical to a '`GT B,A`'. Also, a `NE` instruction is not needed, since it can be replaced by the two-instruction sequence `EQ`, `NEG` (see Figure 5.24b for such a sequence). The 'negate' instruction `NEG` is quite general and can negate the value of a numeric or a boolean input.

Since those instructions have just one output token, they can have one immediate input. Thus instructions such as '`EQ A,0`'; '`GE B,2`'; '`GT C,-6`'; are valid.

The conditional branch instruction, `BRA`, is not that easy to design, since the data flow computer cannot branch; there is no PC to reset. One way to design a `BRA` is as a 2-input, 2-output instruction that always generates just one of its two outputs. Such an instruction receives an input token `X` and generates one of two possible output tokens, depending on its second input, that should be boolean. The two output tokens are identical to the input `X`, except for their destinations. Figure 5.24a shows this instruction and Figure 5.24b is an example of an absolute value calculation. A numeric token `X` is tested and, if negative, is negated to obtain its absolute value.

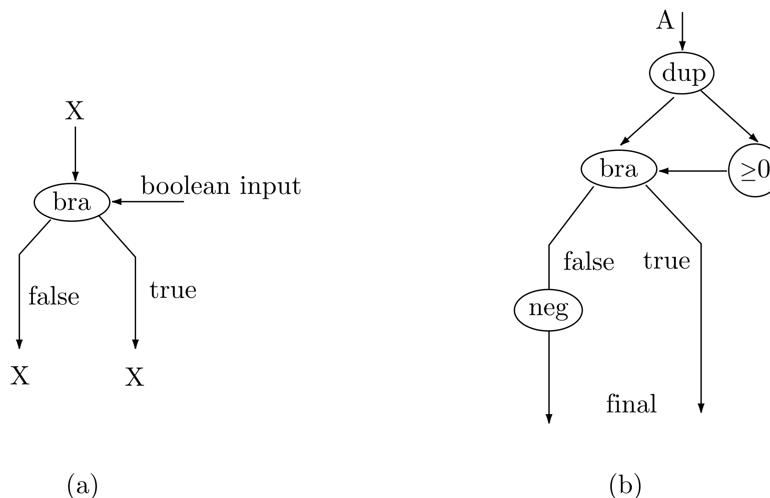



**Figure 5.24**. (a) A `BRA` instruction, (b) An `ABS` calculation

Example 3. *A loop* is an important construct in programs. Our understanding of a computer is incomplete until we know how to write a loop for it. We illustrate a simple loop that uses a counter. The counter, containing the number of iterations, is one of the initial tokens injected by the host. In a conventional computer, a loop is implemented by placing a 'branch' instruction at the end of the loop, to branch back to the start. In a data flow computer, however, the concept of branching does not exist and the loop has to implemented by means of generating and comparing tokens. This is why a loop in a data flow computer is different from our usual notion of a loop. A typical loop consists of *two* separate loops; one performs the necessary operations, and the other counts the iterations by decrementing the counter. The two loops communicate by means of tokens. When the counter, which is a token, reaches a certain, predetermined, value (such as zero or a negative value), a boolean 'false' token is generated and sent to the main `BRA` instruction, to terminate both loops.

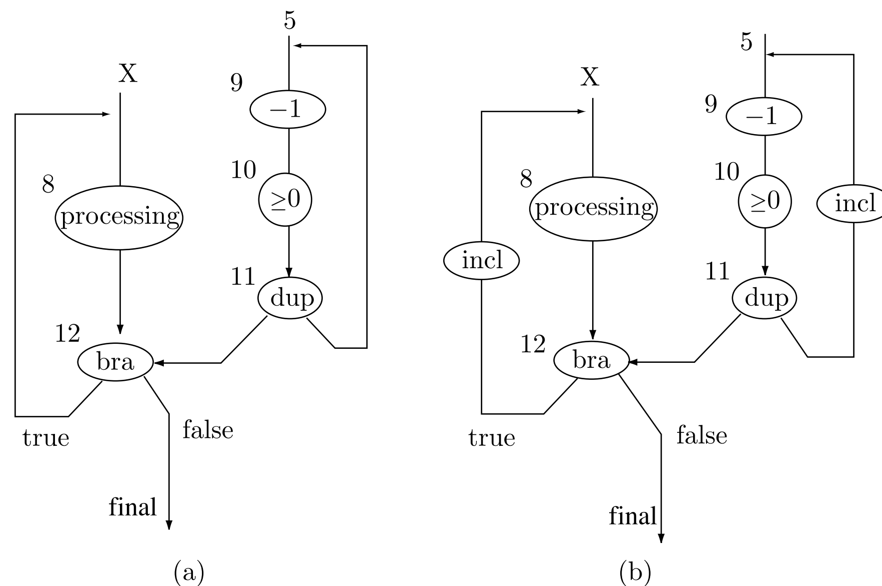



**Figure 5.25**. (a) The basic mechanism, (b) with labels added

The data flow graph (Figure 5.25a) illustrates a simple loop. We assume that the instruction at address 8 does some processing. The processing may, of course, require more than one instruction. The loop is straightforward except for one subtle point—the two loops do not run at the same rate. The main loop—consisting of instructions 8 and 12—is slow since it involves processing. The counting loop—instructions 9–11—is faster since it involves three simple instructions. This is an obvious but important point, since it may throw off the matching of tokens. Recall that tokens are matched, in the matching store, by destination and handedness. In our loop, however, tokens with the same destination and handedness are generated, one in each iteration, by the `BRA` instruction at address 12, part of the slow loop. We denote such a token, generated in iteration $i$, by $A_i$. Our $A_i$ is sent to the matching store to match itself to a token $B_i$ generated, in iteration $i$, by the "$\geq 0$" instruction at address 10 (part of the fast loop). When $A_i$ gets to the matching store, however, it may find several $B$ tokens, $B_i$, $B_{i-1}$, $B_{i-2}, \ldots$, generated by several iterations of the fast loop, already waiting. They all have the same destination and handedness, since they were all generated by the same instruction, but they may have different values. Most are 'true' but the last one should be 'false'. Our particular token $A_i$ should match itself to $B_i$ and not to any of the other $B_j$s.

We conclude that matching must involve the iteration number, and a general way of achieving this is to add a new field, a *label* field, to each token, and to match tokens by destination, handedness, *and* label. Figure 5.25b shows how this approach is implemented. Each iteration increments the label, thereby generating unique tokens. A token $A_i$ arriving at the matching store will match itself to a $B$ token with the same label. Such a token is $B_i$ (a $B$ token that was generated by the same iteration as $A_i$).

There is another approach to the same problem. The matching store can be redesigned such that

matching is done in order of seniority. A token $A_i$ will always match itself to the oldest token that has the same destination and opposite handedness. In our case this would mean that $A_i$ would match itself to $B_i$. In other cases, however, matching by seniority does not work.

▸ **Exercise 5.8:** What could be a good data structure for such a matching store?

Even the use of labels does not provide a general solution to the matching problem. There are cases, such as nested loops, where one label is not enough! A good design for a data flow computer should allow for tokens with several label fields, and there should be instructions to update each of them individually.

▸ **Exercise 5.9:** What could be another case where simple labels may not be enough to uniquely identify a token?

Parallel Processing is an efficient form of information processing which emphasizes the exploitation of concurrent events in the computing process

—K. Hwang and F. A. Briggs, *Computer Architecture & Parallel Proc.*

# 6 Reduced Instruction Set Computers

## 6.1 Reduced and Complex Instruction Sets

RISC (reduced instruction set computers) is an important trend in computer architecture, a trend that advocates the design of simple computers. Before explaining the word 'simple' and delving into the details of RISC architectures, let's look at the historical background of, and the reasons behind, RISC and try to understand why certain designers have decided to take this approach and implement simple computers.

one trend has been dominant in computer architecture since the early days of computer design, namely a trend toward CISC (complex instruction set computers). Larger, faster computers with more hardware features and more instructions were developed, and served as the predecessors of even larger, more complex computers. The following table shows a comparison of computer architectures from the mid 1950s to about 1976. It is clear that computer architects and designers did an excellent job designing and building complex computers.

| Model | Year | Relative performance | Word size | Memory size | CPU |
|---|---|---|---|---|---|
| IBM700 | 1954 | 1 | 36 | 32K core | sequential 12$\mu$s |
| IBM7000 | 1959 | 5 | 36 | 32K core | seq. transistorized 2.18$\mu$s |
| CDC6600 | 1965 | 25 | 60 | 128K core | multiple functional units, 1$\mu$s |
| CDC7600 | 1969 | 100 | 60 | 512K core | same, 27.5ns |
| CDC STAR100 | 1972 | 200 | 64 | 1M core | vector proc., 40ns |
| CRAY–1 | 1976 | 2000 | 64 | 2M semic. | vector proc., LSI, 12.5ns |

One important feature of CISC architectures is a large instruction set, with powerful machine instructions, many types, sizes and formats of instructions, and many addressing modes. A typical example of a powerful machine instruction is `EDIT`. This instruction (available on old computers such as the IBM 360, IBM 370, and DEC VAX), scans an array of characters in memory and edits them according to a pattern provided in another array. Without such an instruction, a loop has to be written to perform this task. Another example is a square-root machine instruction, available on the CDC STAR100 computer (now obsolete). In the absence of such an instruction, a program must be written to do this computation.

While most computer designers have pursued the CISC approach—performance evaluation studies, coupled with advances in hardware and software—have caused some designers, starting in the early 1970s, to take another look at computer design, and to consider simple computers. Their main arguments were:

- The complex instructions are rarely used. Most programmers prefer simple, conventional instructions and are not willing to take the time to learn complex instructions, which are not needed very often. Even compiler writers may feel that they are presented with too many choices of instructions and modes, and may decide not to take the trouble of teaching their compilers the proper use of the many addressing modes and complex instructions.

- Studies of program execution, involving tracing the execution of instructions, have shown that a few basic instructions such as load, store, jump, and integer add, account for the majority of the instructions executed in a typical program.

- Procedures are important programming tools and are used all the time in all types of programs. An efficient handling of procedures on the machine instruction level is one way of speeding up the computer.

- Complex hardware is slow to design and costly to debug, lengthening the development time of a new computer and increasing its price; thereby reducing the competitiveness of the manufacturer.

- Complex instructions are implemented by microprogramming which, while making it easy to implement complex machine instructions, takes longer to execute.

- In a microprocessor, where the entire processor fits on one chip, the hardware and control store needed to implement complex instructions take up a large part of the chip area, area that could otherwise be devoted to more useful features.

- Complex instruction sets tend to include highly-specialized instructions that correspond to statements in a certain higher-level language (for example, COBOL). The idea is to simplify the construction of compilers for that language. However, those instructions may be useless for compiling any other higher-level language, and may represent a lot of hardware implemented for a very specific use.

- Most older architectures up to the 1970s did not use pipelining. Pipelining was a relatively new feature in computers and RISC advocates considered it very useful and worth the cost of the extra hardware it requires.

Based on these arguments, RISC advocates have come up with a number of design principles that today are considered the basis—or even the definition—of RISC. They are:

- A small number of instructions (fewer than 100).

- Instructions should be simple and uniform; they should have a fixed-size opcode, few addressing modes, few types, and fixed size. This makes it easier to decode the instruction and thereby reduces the cycle time.

- Each instruction should execute in one clock cycle. However, It is clear that, even with simple instructions, some instructions take more time to execute than others. Floating-point operations being the prime example.

- No microprogramming should be used. Instructions should be executed by a hardwired, pipelined control unit. Microprogramming makes it easy to implement complex instructions, but it slows down execution. Microprogramming is therefore a bad feature in a computer with simple instructions.

- Memory access should be minimized, since memory reference instructions are slow. Few instructions should be able to access memory, and all the important operations should be performed on numbers in registers. This implies a large set of registers (large compared with other computers of comparable simplicity). It also implies that once an operand is loaded into a register, it should be used as many times as possible before the register is reloaded. This principle, of *reusing* operands, is considered by some designers the most important RISC principle.

- An optimizing compiler should be designed together with the hardware and should be considered part of the architecture of the machine. Instructions should be designed to make it easy to implement higher-level languages (HLL).

The concept of RISC, like most other concepts in computer architecture, is not a rigorous mathematical concept and is not precisely defined. The points above should be considered more as *guidelines* rather than

as a definition. A computer design following all those points is certainly a RISC. A design implementing just some of them may be considered RISC by some, and non-RISC by others. Perhaps a good analogy is the concept of fine-grained architecture vs. coarse-grained architecture, discussed in Chapter 5. Certain parallel computers have fine-grained architectures, others have coarse-grained architectures. Most parallel computers, however, are in-between and are hard to classify as belonging to either type.

The above points also serve to illustrate the advantages of RISC. Those are listed by RISC advocates as follows:

▪ Easy VLSI realization. Because of the simplified instruction set, the control unit occupies a small area on the chip. Generally, the control unit of a microprocessor occupies more than 50% of the chip area. In existing RISCs, however, that figure is typically reduce to around 10%. The available area can then be used for other features such as more registers, cache memory, or I/O ports.

▪ Faster execution. Because of the simple instruction set, the compiler can generate the best possible code under any circumstances. Also, instructions are faster to fetch, decode and execute, since they are uniform. RISC advocates also see the large number of registers as a point in favor of fast execution, since it requires fewer memory accesses.

▪ Designing a RISC is fast and inexpensive, compared to other architectures. This entails the following advantages:

- A short design period results in less expensive hardware.
- It gets the computer to the market faster, thus increasing the competitiveness of the manufacturer and resulting in products that truly reflect the state of the art. A delay in getting a computer to market may result in computers that are outdated the day they are first introduced.
- It contains fewer hardware bugs and new bugs found are easier to fix.

▪ It is easier to write an optimizing compiler for a RISC. To compile a given statement in a higher-level language, a compiler on a RISC usually has just one choice of instruction and mode. This simplifies the design of the compiler itself, making it easier to optimize code. Also, the large number of registers makes it easier for the compiler to optimize the code by generating more register-register instructions and fewer memory reference instructions. On a CISC, in contrast, the compiler has several choices of instructions and modes at any given point, which makes it much harder to implement a good optimizing compiler.

As with most other concepts in the computer field, it is impossible to absolutely prove or disprove the advantages listed here. RISC detractors were therefore quick to announce that they too have a list, a list of RISC disadvantages. Here it is:

▪ Because of the simple instruction set, certain functions that, on a CISC, require just one instruction, will require two or more instructions on a RISC. This results in slower RISC execution. RISC proponents, however, point out that, in a RISC, only the rarely-executed instructions are eliminated, keeping the commonly executed ones. It is well known, they say, that in a typical program, a few instructions account for more than 90% of all instruction executions. Therefore, eliminating rarely-used instructions should not affect the running time.

▪ Floating-point numbers and operations are important but are complex and slow to execute. Most RISCs today either do not support floating-point numbers in hardware or offer this feature as a complex, non-RISC, option.

▪ RISCs may not be efficient with respect to virtual memory and other OS functions. There is not enough experience with RISCs today to decide one way or the other.

Given the above arguments, one thing is clear; RISC is an important architectural topic today even if it is just because of the controversy it generates. This controversy has been going on since 1975 and will perhaps continue in the future. This is healthy, since it leads to more research in computer architecture and to the development of diverse computer architectures.

## 6.2 A Short History of RISC

The term "RISC" has first been proposed by D. Patterson of UC Berkeley and D. Ditzel of Bell laboratories [Patterson and Ditzel 80]. However, some of the RISC ideas were developed or, at least mentioned, by other people earlier. Seymour Cray, the well-known computer designer, is usually credited with the idea that register-register instructions lead to less use of memory and hence to faster execution. John Cocke of IBM has originated the ideas that led to the implementation of the first RISC, the IBM 801 [Radin 83].

In a sense, the name "RISC" is a misnomer, since the reduced instruction set is just one feature of RISC architectures, and is not even the most important one.

The first well-known RISC was the RISC I designed and implemented, as a university project, by Patterson, Sequin and their students at the Berkeley campus in 1981. It is described in some detail in Section 6.3. It was later succeeded by the RISC II, in 1983.

At about the same time, the MIPS project (Microprocessor without Interlocked Pipeline Stages) was carried out at Stanford, by Hennesy and his collaborators [Hennesy 81]. This computer was implemented mainly for research on the effect of pipelining on RISC.

Following the development of those computers, many designers, both in industry and academia, have started to think RISC and have come up with many RISC designs, some even implemented. The most notable examples are the IBM 6150 RT computer, A personal RISC; the Pyramid computer; the Ridge computer; and the transputer. A general reference for some of these is [Mokhoff 86].

## 6.3 The RISC I Computer

Designed as an experimental RISC, this 32-bit computer supports integer arithmetic only and has an instruction set consisting of just 31 instructions. The memory size is $2^{32} \times 8$ (4 Gb) but most data formats are either 16-bit *halfwords* or 32-bit *words*, and the data bus is 32 bits wide. The halfwords start at even addresses and the address of a word is always divisible by 4. There are 138 general-purpose registers but only 32 are available to the program at any time. The machine uses *register windows* (Section 6.3.2) which are particularly useful for implementing procedures.

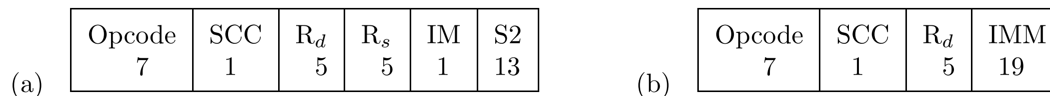



**Figure 6.1**. Instruction formats on the RISC I
(a) The short format, (b) The long format

### 6.3.1 The RISC I Instruction Set

All instructions are 32 bits long and occupy halfwords (they start on an even address). Most instructions have 3 operands, two of them registers and the third, a memory location. There are a few 2- and 1-operand instructions. Only two types of instructions (short and long, Figures 6.1 and 2.13) are supported, and only four addressing modes (three indexed and one PC-relative) exist. This is definitely a simple instruction set (see also Section 2.17). The instruction fields are:

For the Short Format instructions:

- Opcode. The fixed-size, 7-bit opcode allows for up to 128 instructions. The computer only has 31 instructions, so a 5-bit opcode would have been enough. However, when the RISC I was designed, the designers already had in mind the next model (RISC II), and decided to make the opcode longer than needed, for future expansion.

- SCC (Set Condition Code), a 1-bit field. When set, the results of the instruction affect the condition codes (status flags).

- $R_d$. The destination field, one of the 32 available registers.

- $R_s$. A source field, a register.

- IM. A 1-bit mode field. IM=0 implies that the second source is also a register. In such a case, the S2 field specifies a register and only 5 of the 13 bits of that field are used. IM=1 implies that the second source is a 13-bit signed constant used, by the index mode, to obtain the address.
- S2. The second source operand to the instruction. Either a register or an index.

The long instructions are used for all the branches and all the PC–relative instructions. The fields are:

- Opcode and SCC. Those are the same as for the Short Format instructions.
- $R_d$. This is interpreted either as a destination register or (for the conditional branch instructions) its four least-significant bits constitute the *condition code*, while the most-significant bit is unused.
- IMM. A 19-bit signed constant.

The addressing modes are:

- Indexed. EA = $R_x$ + S2, where $R_x$ is either $R_s$ or $R_d$, and S2 is a signed constant.
- Register Indirect. EA=$R_d$
- Array Indexed. EA=$R_x$+S2. Where S2 specifies a register.
- PC–relative. EA=PC+IMM.

The instruction set is divided into four groups. There are 12 arithmetic and logical instructions, 8 Load/Store instructions, 7 branch and call, and 4 miscellaneous ones. Following are some examples of RISC I instructions:

- Integer Add. It has fields $R_s$,S2,$R_d$. It adds $R_d \leftarrow R_s$+S2. All the arithmetic and logic instructions have this format.
- Load Long. The fields are $R_x$,S2,$R_d$. The description is $R_d \leftarrow$M[$R_x$+S2]. The first source field serves as an index register.
- Store Long. Its fields are $R_m$,($R_x$),S2. It stores $R_m$ (the first source field) in the memory location whose address is $R_x$+S2. The description is M[$R_x$+S2]$\leftarrow R_m$.
- Conditional Jump. COND,($R_x$),S2. The description is PC$\leftarrow R_x$+S2, but the jump is only executed if the status flags are agree with the COND field (which is the $R_d$ register). The following table summarizes the 16 COND values that can be used.

| Code | Syntax | Condition |
|---|---|---|
| 0 | none | Unconditional |
| 1 | GT | Greater Than |
| 2 | LE | Less or equal |
| 3 | GE | Greater or equal |
| 4 | LT | Less Than |
| 5 | HI | Higher Than |
| 6 | LOS | Lower or Same |
| 7 | LO or NC | Lower Than or No Carry |
| 8 | HIS or C | Higher or Same or Carry |
| 9 | PL | Plus |
| 10 | MI | Minus |
| 11 | NE | Not Equal |
| 12 | EQ | Equal |
| 13 | NV | No Overflow |
| 14 | V | Overflow |
| 15 | ALW | Always |

- Call Relative (and change window). Uses fields $R_d$,IMM. The call is done by saving the PC in the destination register and incrementing the PC by the signed, 19-bit constant IMM. Thus $R_d \leftarrow$PC, PC$\leftarrow$PC+IMM.

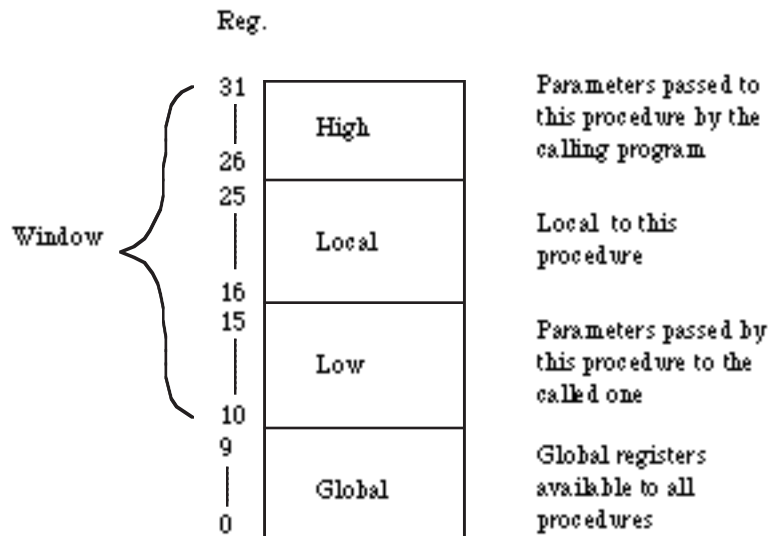



**Figure 6.2**. One register window

### 6.3.2 The RISC I Registers

The RISC I register set contains 138 32-bit registers, divided into two groups. The lower 10 registers—called the *global registers*—and the upper 128 registers—the *window registers*. At any given time, the program can access 32 registers: the global ones, and a set of 22 window registers (Figure 6.2).

Each time a procedure is called, the window moves down 16 registers; each time a procedure returns, the window moves up by the same amount (Figure 6.3). As a result, the lowest 6 registers of a window are common to the program and the procedures it calls. The program can store parameters in those registers and the procedure can access them directly because those same registers are the highest ones in *its* window.

When finished, the procedure can, of course, return results to the calling program in the same registers. When a procedure calls a nested procedure, the process is the same. There are 8 windows, allowing for nested procedure calls up to 8 deep. If all the register windows have been used and another procedure call occurs, an interrupt is generated (register file overflow) and the parameters of the new call are pushed into a stack in main memory. When the corresponding return is executed, another interrupt (register file underflow) is generated.

▸ **Exercise 6.1:** What could be a common reason for such deep nesting of procedure calls?

### 6.3.3 The RISC I Pipeline

Since the RISC I instructions are simple, its pipeline is also simple. It overlaps the execution of an instruction with the fetch of the next instruction. It is therefore a *two stage* pipeline. The important feature of this pipeline is the way it executes jumps. Normally, when a jump instruction is executed in a pipeline, all the instructions in earlier stages of the pipeline have to be flushed. In the RISC I, with a two-stage pipeline, this would mean flushing the instruction that has just been fetched. To avoid that, the designers have incorporated the new feature of *delayed jumps*.

In the RISC I computer, a jump is delayed one instruction. This means that when a jump is encountered, the control unit "saves" it, executes the next instruction (which is in the pipeline already), and then retrieves and executes the jump. The advantage is that the next instruction does not have to be flushed from the pipeline. The disadvantage is that sometimes the jump has to take place immediately and, in such a case, the next instruction should be a no operation (which, on the RISC I, is an '`ADD R0,R0,R0`'). The following table shows a short program fragment with a jump. The code is then modified, by adding a `NOP`, to accommodate a delayed jump. Finally, it is optimized by eliminating the `NOP`.

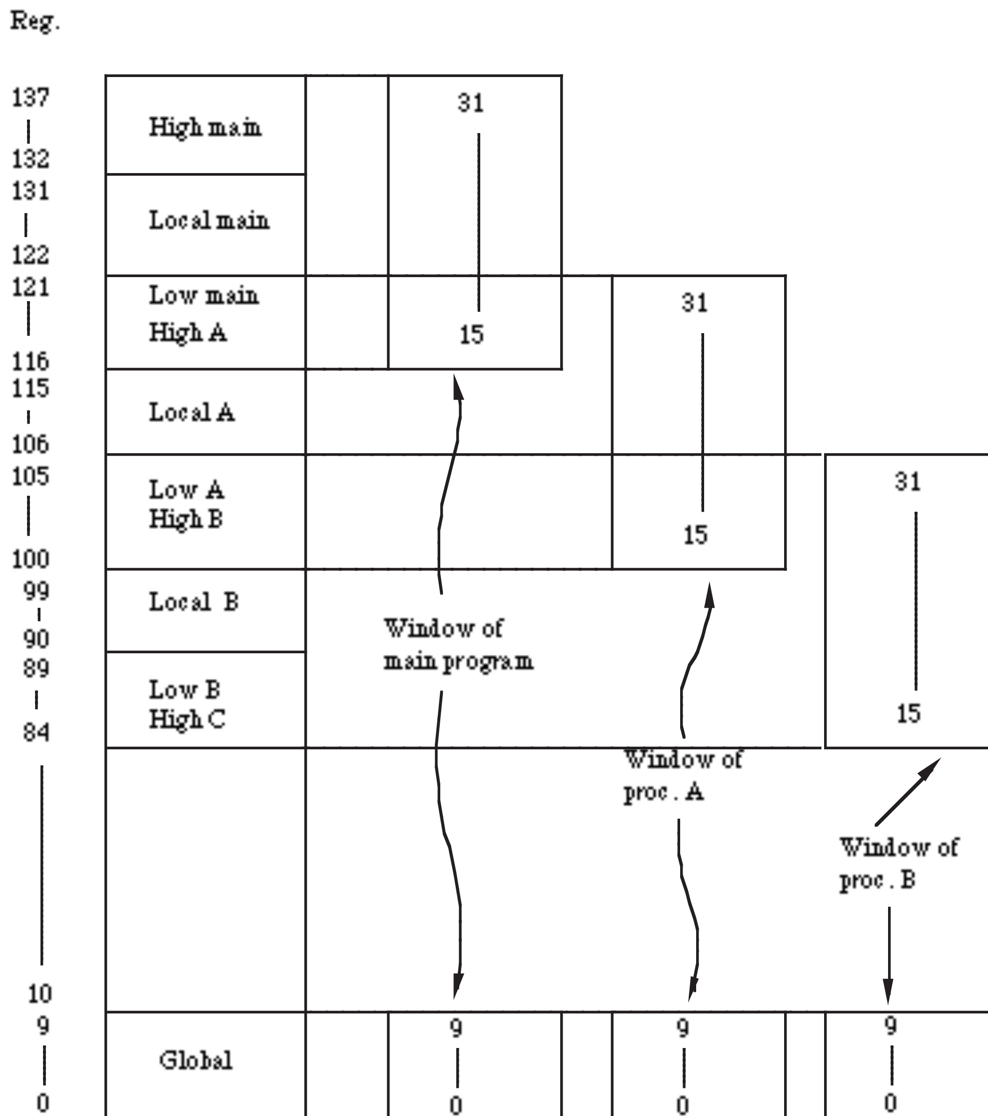



**Figure 6.3**. Moving the register window

| Address | Conventional Jump | Delayed Jump | Optimized Delayed Jump |
|---|---|---|---|
| 100 | `ADD A,B` | `ADD A,B` | `ADD A,B` |
| 101 | `SUB C,D` | `SUB C,D` | `SUB C,D` |
| 102 | `JMP 105` | `JMP 106` | `JMP 105` |
| 103 | `MUL X,Y` | `NOP` | `MUL X,Y` |
| 104 | `DIV M,N` | `MUL X,Y` | `DIV M,N` |
| 105 | ... | `DIV M,N` | ... |

This shows the importance of an optimizing compiler on the RISC I. A good optimizing compiler should be able to generate code with just a few `NOP`s, thereby making full use of the delayed jumps.

As a result, the compiler writer has to be aware of the pipeline, a fact that makes the RISC I pipeline an architectural feature. Normally, a pipeline is an organizational feature.

▸ **Exercise 6.2:** Why is pipelining an organizational, rather than an architectural feature?

### 6.3.4 RISC I Interrupts

Seven interrupts are supported, two external and five internal. The external interrupts are sent to the processor via special pins. They are "reset" and an IRQ. The reset interrupt is used to reset the computer. The IRQ interrupt is usually generated by an I/O device. The internal interrupts are: an illegal opcode, a privileged opcode used in user mode, an address misalignment, a register file overflow, and a register file underflow. Each interrupt has a dedicated vector in memory, where the program can store the address of the handling routine of the interrupt.

The "reset" interrupt cannot be disabled. All other interrupts can be disabled by clearing bit I in the PSW. When an interrupt is sensed during the execution of an instruction, and interrupts are enabled, the instruction is aborted, the PC is saved in R25, and the computer jumps to the routine whose address is fetched from the interrupt's vector. Upon finishing, the routine returns to the interrupted program, re-executing the interrupted instruction.

## 6.4 Conclusions

The RISC philosophy, which started as a reaction to complex computer architecture, has turned first into a debate and then into a field of research. Many good architectural ideas have been developed in an attempt to develop RISCs. None of those ideas is essential to RISC and some have even been adopted for use on CISCs. Perhaps the most important RISC principles are the close relationship between the hardware and the compiler, and the reuse of operands already in registers.

A general reference, presenting an overall view of the RISC concepts, as well as a description of several RISCs, is [Tabak 87].

The name RISC, reduced instruction set computer, focuses on reducing the number and complexity of instructions in the machine. Actually, there are a number of strategies that are employed by RISC designers to exploit caching, pipelining, superscalarity, and so on. You should be aware, however, that a given RISC design may not use all the techniques described in the following paragraphs.

—Vincent P. Heuring, *Computer Systems Design and Architecture*

# 7 Digital Devices

There are many different hardware circuits in the computer, they perform different functions, and they range from simple to complex. Most of them, however, are based on a small set of simple devices known as *logic gates*. Thus, an understanding of computer design should start with a description of the common logic gates, and of the most useful devices based on them.

A logic gate is a device that generates a binary output (i.e., an output signal that can have one of two values) from its inputs. The gate can have one or more inputs, and they are binary. Since the inputs and outputs are binary, it is possible to completely describe the operation of the gate by means of a table that lists the output for each set of inputs. This is the *truth table* of the gate. A logic gate also has a symbol associated with it that's used to represent the gate in logic diagrams. Figure 7.1a lists the seven most common logic gates, their symbols, names, and truth tables. Notice that the "Inverse" (sometimes also called "NOT") gate has just one input. The AND gate outputs a 1 when both its inputs are 1's and the OR gate outputs a 1 when either of its inputs (or both) are 1's. The XOR gate (for "exclusive OR") is an OR gate where the case "both inputs are 1's" is excluded.

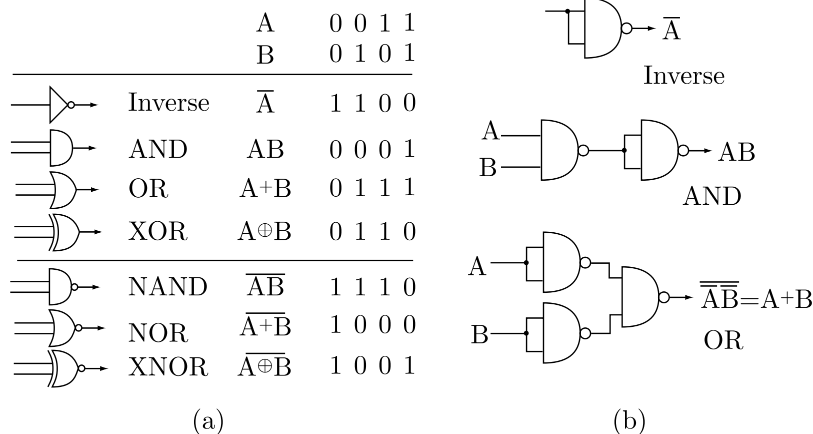



**Figure 7.1:** Logic gates (a) and universal NAND (b)

Logic gates, (with the exception of NOT) can have more than two inputs. A multi-input AND gate, for example, outputs a 1 when all its inputs are 1's.

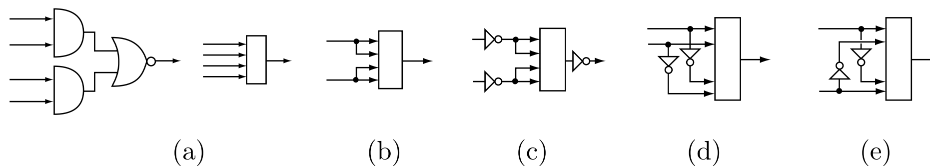
▸ **Exercise 7.1:** Explain the outputs of the OR and XOR gates with several inputs.

The last three gates of Figure 7.1a are NAND, NOR, and XNOR. They are combinations of AND, OR, and XOR, respectively, with a NOT. Such combinations are useful and are also universal (a universal logic gate is one that can be used to simulate all other gates). Figure 7.1b shows how NAND gates can be used to simulate a NOT, AND, and OR gates.

▸ **Exercise 7.2:** Show how NOR gates can be used to produce the same effects.

▸ **Exercise 7.3:** Show how the XOR gate can be constructed out of either NOR gates or NAND gates.

The AND-OR-INVERT (AOI) gate is not really a logic gate but a simple digital device that consists of two AND gates and a NOR gate (Figure 7.2a). If we label its four inputs $a$, $b$, $c$, and $d$, then its output is $\overline{ab+cd}$. The AOI gate is useful because of its universality. Many useful digital devices can be generated as combinations of these gates. Figure 7.2b,c,d shows how this device can be used to construct the NOR, NAND, and XOR gates, respectively. Figure 7.2e shows how it can simulate an XNOR.



(a) (b) (c) (d) (e)

**Figure 7.2:** The AOI gate

Logic gates are easy to implement in hardware and are easy to combine into more complex circuits. Since the gates have binary inputs and outputs, they are the main reason why computers use binary numbers. Another reason for the use of binary numbers is that the logic gates can perform only logic operations, but with binary numbers there is a close relationship between logic and arithmetic operations. Logic gates can therefore be combined to form circuits that perform arithmetic operations.

> Computers are composed of nothing more than logic gates stretched out to the horizon in a vast irrigation system.
>
> —Stan Augarten.

## 7.1 Combinational and Sequential Devices

The digital devices described in the rest of this chapter divide naturally into two categories as follows:

1. Combinational devices. The outputs of such a device are completely determined by its inputs at any given time. In particular, the outputs do not depend on any past states or past inputs. Such a device has no memory of the past. An example is a decoder (Section 7.6).

2. Sequential devices. The outputs of a sequential device depend on its current inputs and also on the current state. Such a device feeds its state back to its inputs and therefore has memory. An example is a latch (Section 7.2.1).

## 7.2 Multivibrators

The first example of an application of gates is the family of *multivibrators*. This family includes many different digital devices that have one feature in common; they have two states. There are three main types of multivibrators, astable (both states are unstable), monostable (one state is stable and the other isn't), and bistable (both states are stable).

### 7.2.1 Bistable Multivibrators

Bistables are the most important multivibrators. A bistable device (also called a *latch* or a *flip-flop*) has two stable states. Once placed in a state, it stays in that state (it *latches* the state) until an outside input flips the state. If we consider one state to be binary 0 and the other state to be binary 1, we can think of

the latch as a memory unit for a single bit. Registers and most types of computer memories are based on latches.

The simplest latch is shown in Figure 7.3a. It consists simply of two NOT gates connected in series. Once power is turned on, this simple device settles in a state where one NOT gate outputs a 1, which is input by the other NOT gate, which outputs a 0, which is input by the first NOT. Thus, the output of each gate reinforces the input of the other, and the state is stable. Figure 7.3b shows the same device arranged in a symmetric form, with the two NOT gates side by side.

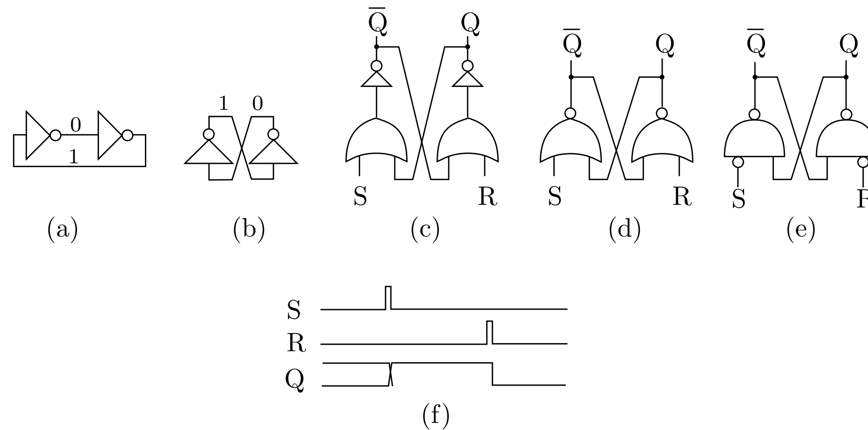



**Figure 7.3:** Latches

In order for this device to be practical, two inputs and an output are required. Figure 7.3c shows how one of the NOT gates is arbitrarily selected and its output labeled `Q`. The output of the other NOT is always the opposite, so it is labeled $\bar{\texttt{Q}}$ and is termed the *complementary* output. The two inputs are called `S` (for "set") and `R` (for "reset") and they are fed to the NOT gates through two OR gates. These inputs are normally low (i.e., binary 0), so they do not affect the state of the latch. However, if `S` is momentarily set to 1, the OR gate outputs a 1 regardless of its other input, this 1 is fed to the NOT on the left, which then outputs a 0, which is sent to the OR gate on the right, and through it, to the NOT on the right. This NOT starts outputting a 0, which is eventually fed to the NOT on the left. The `S` input can now be dropped, and the final result is that `Q` becomes 1 (and $\bar{\texttt{Q}}$, of course, becomes a 0). We say that the latch has been set. Similarly, a momentary pulse on the `R` input resets the latch. This behavior is illustrated in Figure 7.3f and this type of latch is called an SR latch (although some authors prefer the term RS latch). An SR latch is sometimes called an *asynchronous* device.

Feeding 1's through both inputs simultaneously results in an unpredictable state, so it is important to make sure that this does not happen.

Figure 7.3d shows how a pair of NOT and OR gates is replaced by a single NOR gate, and Figure 7.3e shows how a latch can be constructed from two NAND gates and inverted inputs. In this type of device, the inputs are normally held high and are pulled low momentarily to change the state.

In practical situations, it is preferable to have a latch that's normally disabled. Such a latch has to be enabled before it responds to its inputs. The advantage of such a device is that the designer does not have to guarantee that the inputs will always have the right values. The latch is disabled by a control line, and as long as that line is low, the inputs can have any values without affecting the state of the device. Figure 7.4a shows such a device. It requires two AND gates, and the `E` input controls the response of the device. Such a latch is said to be *synchronized* with the control line, and is called a *flip-flop*. Figure 7.4b shows a variation of an SR flip-flop where two new inputs `P` (or `PR`) and `CR` can be used to preset and clear the device even when it is disabled. Such a device is both synchronous and asynchronous as shown by the waveforms of Figure 7.4c.

Many times, the control line is the output pulse produced by the computer clock. Computer operations are synchronized by a clock that emits pulses of the form . . . , and operations are generally

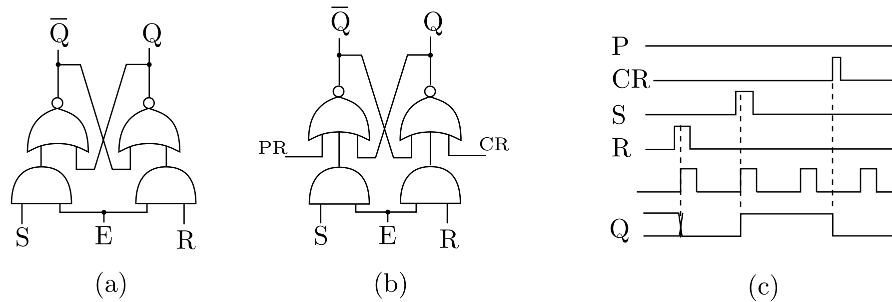



**Figure 7.4:** SR flip-flops

triggered by the rising (or falling) edge of the clock pulse. If the clock output is the control line, the flip-flop can change state only on a clock edge.

The symbol shown in Figure 7.5a indicates a flipflop that's enabled on a rising edge of the clock. In contrast, Figure 7.5b,c shows flipflops that are enabled on a falling edge of the clock.

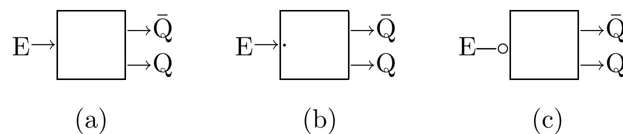



**Figure 7.5:** Flipflops enabled on the rising edge (a) and on the falling edge (b,c)

When both inputs of an SR latch are high, its state is undefined. Thus, the circuit designer has to make sure that at most one input will be high at any time. One approach to this problem is to design a latch (known as a JK flipflop) whose state is defined when both inputs are high. Another approach is to design a D latch, where there is just one input.

The `D` input of a D latch (`D` for "data") determines the state of a latch. However, a binary input line can have only two values (high and low), and it needs a third value to indicate no change in the current state. This is why the D latch has to have a control input `E`. When `E` is low, the `D` input does not change the state of the latch. Figure 7.6a shows the standard symbol of a D latch, Figure 7.6b shows the basic design, based on an SR latch (notice the asynchronous inputs; they should not be high simultaneously), and Figure 7.6c shows how a D latch can be constructed from an SR latch and a NOT gate. Figure 7.6d is a typical waveform.

Figure 7.6e illustrates a problem with this design. The `D` input goes high while the clock pulse is high, and as a result `Q` goes high immediately. In the computer, we want all states to change on a clock edge (rising or falling), so this design of a D latch is called *untimed*. Untimed bistables are generally called latches and timed ones are called flipflops. In a latch, the `E` input is called "enable," while in a flipflop it is called a "clock" and is denoted by `CK`. In a timed flipflop, the state can change only on a clock edge (leading or trailing), as illustrated in Figure 7.7a. One way to design such a device is to connect two SR latches in a master-slave configuration, as shown in Figure 7.7b.

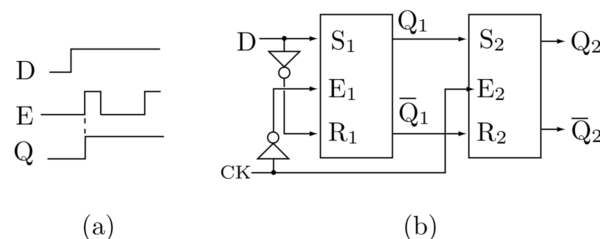



**Figure 7.7:** Timed D flip-flop

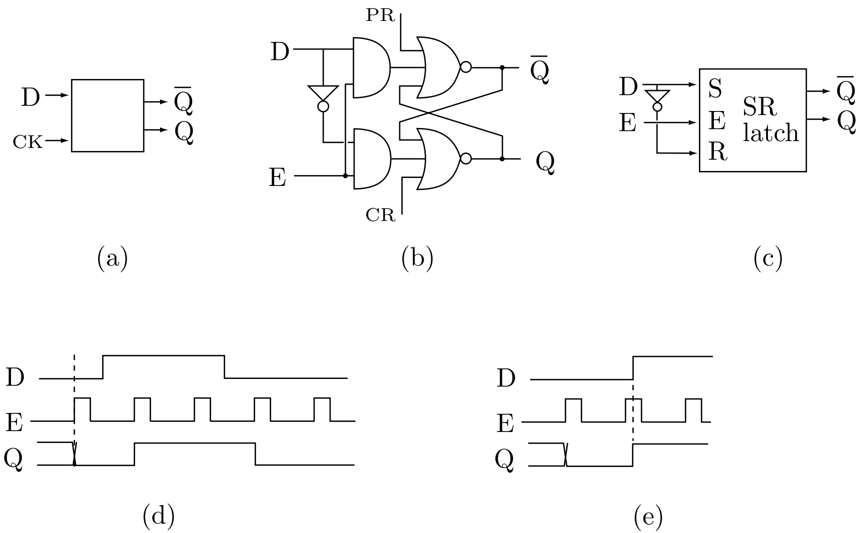



**Figure 7.6:** D flip-flops

This figure shows that the first SR latch is enabled only on a trailing edge of the clock. Therefore, $Q_1$ will change states only at that time. Output $Q_2$, however, can change states only during a leading edge. A high input at `D` would therefore pass through $Q_1$ during a trailing edge and would appear at $Q_2$ during a leading clock edge. This is a simple example of the use of the clock pulse to synchronize the propagation of signals in digital devices.

The D latch solves the problem of `S` = `R` = `1` by having just one input. A JK flipflop solves the same problem in a different way. This device is a timed SR flipflop with two inputs `J` (for "set") and `K` (for "reset"). When `J` = `K` = `1`, the device toggles its state. Figure 7.8a shows the standard symbol for the JK flipflop. Figure 7.8b shows how this device can be constructed from two SR flipflops, and Figure 7.8c illustrates typical waveforms.

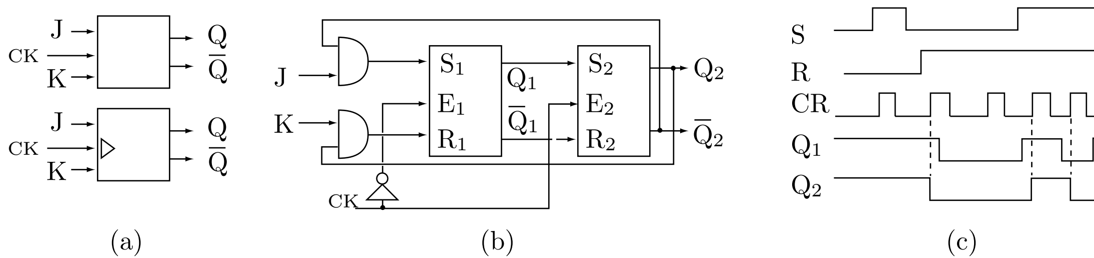



**Figure 7.8:** The JK flipflop

Another useful type of latch is the T flipflop. It has a single input labeled `T`. When `T` is low, the state of the T latch does not change. When `T` is high, the state `Q` toggles for each leading clock edge. Figure 7.9a shows how this type can be constructed from a JK flipflop, while Figure 7.9b shows typical waveforms. This type of latch is used in counters (Section 7.3).

### 7.2.2 Monostable Multivibrators

A monostable (also called a *one shot*) is a digital device that has one stable state (normally the 0 state) and one unstable state. In a typical cycle, a pulse on the input line switches the monostable to its unstable state, from which it automatically drops back (after a short delay) to its stable state. Figure 7.10a shows this behavior and Figure 7.10b shows how such a device can be built from two NOR gates and a delay line (this is an example of a device that requires more than just logic gates).

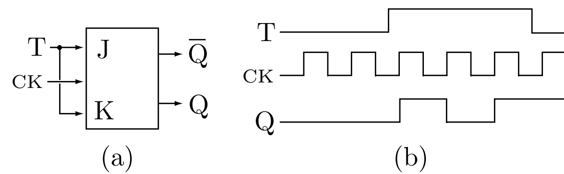



**Figure 7.9:** The T flipflop

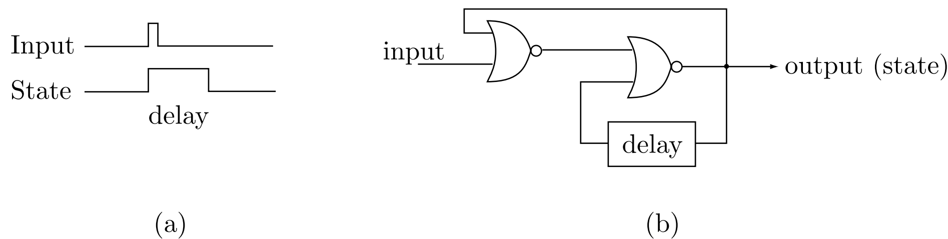



**Figure 7.10:** A Monostable multivibrator

### 7.2.3 Astable Multivibrators

An astable multivibrator is an oscillator. It has two states, neither of which is stable. As a result, it always switches from one state to the other, and is never "satisfied" with its present state. Such a device needs no inputs and can easily be constructed from an inverter combined with a delay line or from three inverters (Figure 7.11a,b).

When power is first turned on, line $b$ may end up low and, consequently, line $a$ becomes high. The high signal on $a$ propagates through the delay line and eventually sets $b$ to high. After a very short delay in the inverter, this signal is inverted to low, which becomes the new state of $a$ and starts propagating through the delay line.

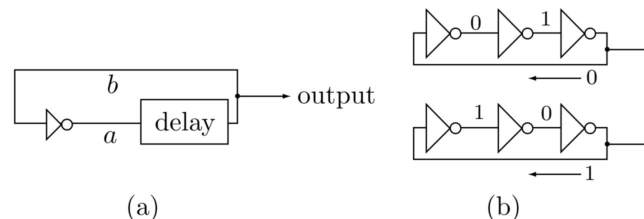



**Figure 7.11:** The Astable multivibrator

Other designs may have different delays for the two states and may also have a "reset" input that resets the state of the astable to low. This is a simple design for the computer clock, a design that emits a clock pulse of the form . . . .

## 7.3 Counters

A counter is a device (constructed from flipflops) that can perform the following functions:

1. Count pulses.
2. Divide a pulse train.
3. Provide a sequence of binary patterns for sequencing purposes.

A modulo-$N$ counter is a device that provides an output pulse after receiving $N$ input pulses. Figure 7.12a is the general symbol of a counter. The main input is denoted by `CK` (clock). A pulse on this input triggers the counter to increment itself. The `CR` (clear) input resets the counter. Counters use flipflops to

count and are therefore binary. As a result, it is easy to construct a counter that will count up to a power of 2. It is possible to modify a binary counter to count up to any positive integer.

In the computer, counters are used in the control unit and the ALU to control the execution of instructions. A simple example is the multiplication of two $n$-bit integers. Certain ALUs may perform this multiplication by a loop that iterates $n$ times and uses additions and shifts to multiply the numbers (Section 9.3). Such a loop is controlled by a counter that emits an output pulse after being triggered $n$ times, thereby terminating the loop.

Counters can be synchronous or asynchronous, the latter type is also called *ripple* counter. A ripple counter can be constructed from T flipflops by connecting the output of each flipflop to the clock (`CK`) input of its successor flipflop. Figure 7.12b shows a three-stage ripple counter where each stage toggles on the trailing edge of the input (this is implied by the small circle at the `CK` input). Each stage toggles its successor, since the `T` inputs are permanently high. Typical waveforms are shown in Figure 7.12c. Notice that all three outputs go back to their initial states after eight input pulses, which is why this counter is said to count modulo-8.

▸ **Exercise 7.4:** There is also a reason to claim that this counter counts to 4. What is it?

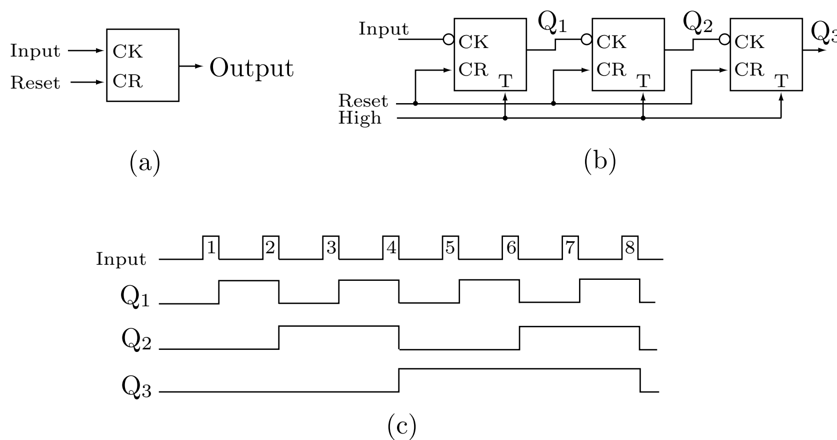



**Figure 7.12:** A Ripple Counter with `T` flipflops

▸ **Exercise 7.5:** Show how a 3-stage ripple counter can be designed with JK flipflops.

Sometimes, it is preferable to have a counter where only one output line goes high for each input pulse. Figure 7.13a shows how a 1-of-8 decoder (Section 7.6) can be connected to the outputs of a 3-stage counter, such that each input pulse causes the next decoder output to go high. Figure 7.13b shows how such an effect can be achieved explicitly, using AND gates instead of a decoder (this also illustrates a use for the complementary outputs $\bar{Q}_i$ of a flipflop).

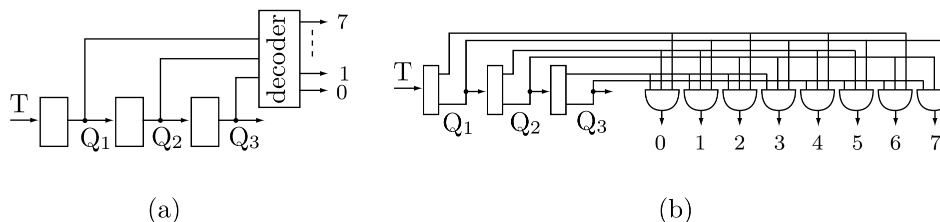



**Figure 7.13:** A counter followed by a decoder

A nonbinary ripple counter can be constructed by modifying a binary counter in such a way that it resets itself after the desired number of counts. A modulo-5 counter, for example, is easily constructed from a

modulo-8 counter where the three output lines $Q_1$, $\bar{Q}_2$, and $Q_3$ are connected to an AND gate whose output resets the counter after five input pulses (Figure 7.14). Notice that the use of $\bar{Q}_2$ is not really necessary, since the first time both outputs $Q_1$ and $Q_3$ will simultaneously be high is after the fifth input pulse.

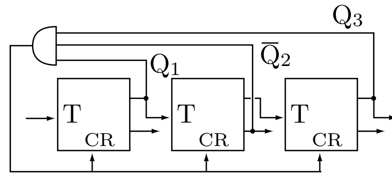



**Figure 7.14:** A modulo-5 counter

▸ **Exercise 7.6:** Show how to construct a nonbinary counter that counts up to any given positive integer $N$.

The counters described so far count up. They count from 0 to $N-1$ by a design that toggles output $Q_i$ whenever its predecessor $Q_{i-1}$ changes from 1 to 0. It is possible to design a counter that will count from $N-1$ down to 0. This is achieved by toggling output $Q_i$ when its predecessor $Q_{i-1}$ changes from 0 to 1. The idea is to use the complementary output line $\bar{Q}_{i-1}$ as the input to stage $i$ (Figure 7.15a).

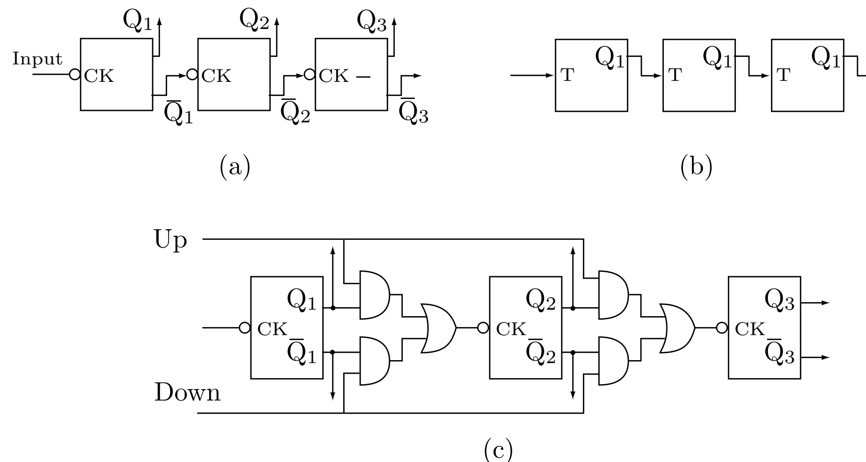



**Figure 7.15:** Down counters

▸ **Exercise 7.7:** Show how to use T flipflops to design a down counter.

A bidirectional (up or down) counter can be designed by connecting either $Q_{i-1}$ or $\bar{Q}_{i-1}$ (by means of an OR gate) to $Q_i$. A basic design is shown in Figure 7.15c.

The ripple counter is simple but slow, since the state propagates from stage to stage. Each flipflop in the counter has to wait for its predecessor to come to a stable state before its state can become stable and affect its successor. This problem is solved by the synchronous counter, the topic of the next section.

### 7.3.1 Synchronous Counters

In this type of counter, all the stages change their states simultaneously. The key to this behavior is the observation (illustrated in Table 7.16) that the state $Q_i$ of stage $i$ should be toggled on an input pulse only if the states of all the preceding stages are high, i.e., $Q_{i-1} = Q_{i-2} = \cdots = Q_1 = 1$. A basic design uses T flipflops where the `T` input of stage $i$ is the AND of the outputs of all the preceding stages. This, of course, is impractical for a counter with many stages, because it requires AND gates with many inputs (for an $m$-stage counter, the last AND gate has to have $m-1$ inputs). A compromise, not exactly synchronous, using only 2-input AND gates, is shown in Figure 7.17. This design uses $m-2$ AND gates, so the time it takes the last stage to come to a stable state is proportional to $m-2$.

| $Q_3$ | $Q_2$ | $Q_1$ | $Q_3$ | $Q_2$ | $Q_1$ |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 | 1 |
| 0 | 1 | 0 | 1 | 1 | 0 |
| 0 | 1 | 1 | 1 | 1 | 1 |

**Table 7.16:** Toggling states

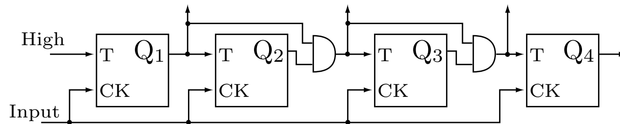



**Figure 7.17:** A synchronous counter

### 7.3.2 Ring Counters

A ring counter produces decoded outputs. Only one output is high at any time, and each input pulse moves the high signal to the next output line. Figure 7.18a illustrates a 3-stage ring counter based on D flipflops. Its three inputs vary in a rotating pattern from 100, through 010 and 001 back to 100. The last output, $Q_3$, is fed to the `D` input of the first stage. Therefore, when $Q_3$ becomes high, the high signal immediately appears at the first `D` input, and is reflected in output $Q_1$ in the next cycle.

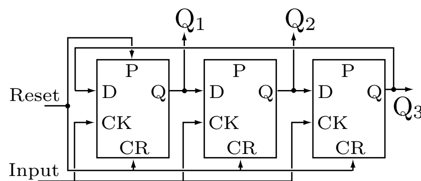



**Figure 7.18:** A 3-stage ring counter

This mechanism of starting the output cycle when the last output is high, may fail in the presence of errors. Imagine a 5-stage ring counter. If, as a result of an error, the output pattern gets corrupted into, say, 01001 (two 1's), then this bad pattern will continue rotating on the output lines indefinitely. A better design for a ring counter should be able to correct such an error automatically. Such a design is shown in Figure 7.18b. A high signal is fed into the `D` input only when $Q_1 = Q_2 = \cdots = Q_{m-1} = 0$ i.e., when all outputs except the last one are low. In such a counter, the bad output 01001 will go through the sequence 00100 (even though $Q_5$ is high, nothing is fed into the first input, since not all other outputs are low), 00010, 00001, 10000, etc.

It is also possible to design an $m$-stage ring counter that counts up to $2m$ or up to other values. Such devices are called *twisted ring counters.*

### 7.3.3 Applications of Counters

1. Digital clocks. The 60 Hz frequency of the power line is fed into a modulo-60 counter that produces an output pulse once a second. This is again divided by 60 by another counter, to produce an output pulse once a minute. These pulses are then used to increment a display. This is the principle of a simple consumer digital clock.

2. Clocking ALU operations. Some ALU operations are done in steps, and a counter can be used to trigger the start of each step.

3. Clocking the control unit. Each step performed by the control unit takes a certain amount of time. These times are expressed as multiples of one time unit (the clock cycle) and a counter can be used to count any desired number of clock cycles before the next step is started.

## 7.4 Registers

Intuitively, a register is a small storage device that can store one number. Physically, a register is a set of latches. Each latch stores one bit, so an $n$-bit register consists of $n$ latches, each with its two input and two output lines. There are two general types of registers as follows:

1. A latch register (also referred to as a buffer). This type is used for storage. The individual latches are not connected in any way, and modifying the state of any of them will normally not affect the others. General-purpose registers in a computer are usually of this type. Figure 7.19a is an example of an $n$-bit latch register.

2. A functional register. This type of register can store a number and can also perform an operation on it. Common examples of functional registers are a counter, an adder, and a shifter, but there are many more. In such a register, the individual latches are connected such that changing the state of one may affect other ones.

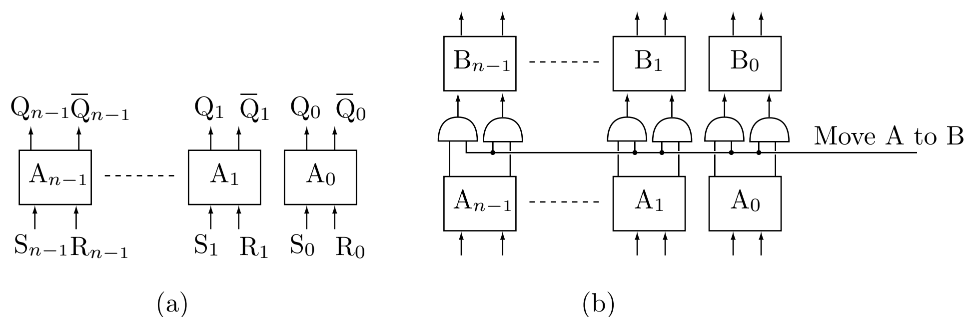



**Figure 7.19:** Registers

Most registers are parallel; the $n$ bits are sent to the register in parallel on $n$ lines, and are read from the register in parallel, on another set of $n$ lines. Serial registers are also possible. The $n$ bits are sent to such a register serially, one at a time, on the same line. Reading a serial register is also done serially.

A *register transfer* is a very common and important operation in computers. It is a task any computer performs many times each second. Given two registers $A$ and $B$, how can $A$ be transferred to $B$? We know from Section 1.3.2 that when data is moved inside the computer, only a copy is moved and the original always remains unchanged. This implies that transferring $A$ to $B$ makes register $B$ identical to $A$ and does not change $A$ in any way.

The principle of transferring $A$ to $B$ is to connect the outputs of every latch in $A$ to the inputs of the corresponding latch in $B$. This connection must be done through gates, since otherwise $A$ would always transfer to $B$, and $A$ and $B$ would be identical. Figure 7.19b shows how a register transfer is done. Two AND gates connect the outputs of each latch of $A$ to the inputs of the corresponding latch of $B$. The `Q` output is connected to the `S` input and the $\overline{\texttt{Q}}$ output, to the `R` input. The gates act as switches and are normally off, so the two registers are separate. When the control unit decides to move $A$ to $B$, it sends a short pulse on the line controlling the gates. The gates switch on momentarily, which copies each latch of $A$ to $B$. After the control line returns to low, the two registers are effectively disconnected. Notice that only one control line is necessary, regardless of how large the registers are.

▸ **Exercise 7.8:** If register $A$ can be transferred to $B$ (i.e., if all the necessary AND gates are in place), can $B$ also be transferred to $A$?

## 7.5 Multiplexors

A multiplexor is a digital switch that's used to switch one of several input buses to its output bus. Each bus consists of $n$ lines. Figure 7.20a shows a 2-to-1 multiplexor switching either bus $A$ or bus $B$ to the output bus. This multiplexor must have a control line that switches it up or down. Figure 7.20 shows the details of this simple device. Three gates, two AND and one OR, are required for each bus line. Thus, a multiplexor to switch 16-bit buses requires 48 gates. It is also possible to have a 4-to-1 multiplexor, switching one of four input buses to its output bus. Such a multiplexor requires two control lines.

▸ **Exercise 7.9:** (Easy.) How many control lines are needed in a multiplexor with $k$ inputs?

▸ **Exercise 7.10:** Show the design of a 4-to-1 multiplexor.

There is also a demultiplexor (Figure 7.20c), the opposite of a multiplexor. It switches its single input bus into one of several output buses.

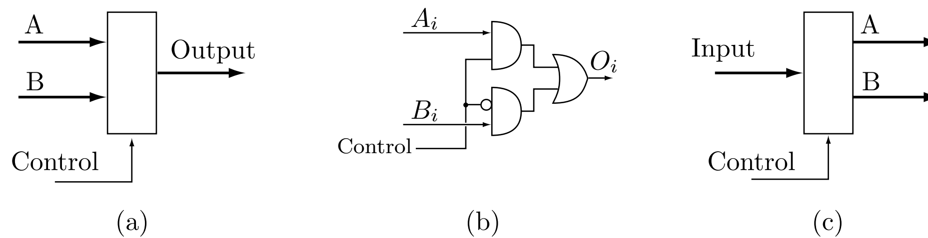
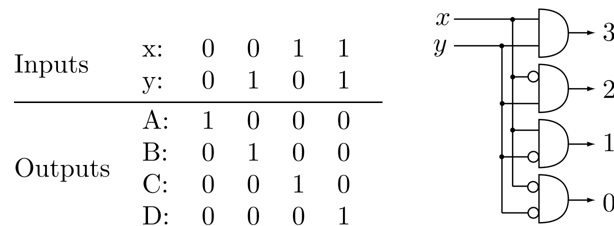



**Figure 7.20:** Multiplexor (a,b) and demultiplexor (c)

## 7.6 Decoders

A decoder is one of the most common digital devices. Figure 7.21 shows the truth table and the design of a decoder with two inputs `x` and `y` and four outputs `A`, `B`, `C`, and `D` (also denoted by 0, 1, 2, and 3). The principle of decoding is: For each possible combination of the input lines, only one output line goes high. We say that the decoder's input is *encoded* but its output is *decoded* which means that the input lines can have any binary values, but only one output line can be high at any time. A decoder has one more input called `D/E` (for disable/enable). When this input is high, the decoder operates normally. When this input is pulled low, the decoder is disabled; all its output lines are zero.



| | | | | | |
|---|---|---|---|---|---|
| Inputs | x: | 0 | 0 | 1 | 1 |
| | y: | 0 | 1 | 0 | 1 |
| Outputs | A: | 1 | 0 | 0 | 0 |
| | B: | 0 | 1 | 0 | 0 |
| | C: | 0 | 0 | 1 | 0 |
| | D: | 0 | 0 | 0 | 1 |



**Figure 7.21.** Truth table and design of a 1-of-4 decoder

Two input lines can have $2^2 = 4$ different states, since they are encoded. A decoder with two input lines should therefore have four output lines. It is called a "1-of-4 decoder" or a "2-to-4 decoder." In principle, it is possible to have a 1-of-$n$ decoder, where $n$ is any power of 2. In practice, $n$ is limited to small values, typically in the range $[4, 16]$.

## 7.7 Encoders

An encoder is the opposite of a decoder. Its inputs are decoded (i.e., at most one is high) and its output lines are encoded (i.e., in binary code). Generally, such a device has $2^n$ inputs and $n$ outputs. It is easy to design an encoder, but its use in computers is limited.

▸ **Exercise 7.11:** Show the design of an encoder with eight inputs.

▸ **Exercise 7.12:** An encoder with $n$ outputs generally has $2^n$ inputs. Can it have a different number of inputs?

### 7.7.1 Priority Encoders

A priority encoder is a special case of an encoder and it happens to be a useful device. It assigns priorities to its inputs, and its output is the encoded value of the highest-priority input. Table 7.22 is the truth table of a priority encoder with five inputs and three outputs (plus an extra output, `none`, that indicates the absence of any inputs). Input $I_5$ has the highest priority. When this input is high, the outputs are $Y_2Y_1Y_0 = 101$ regardless of the other inputs.

Such a device can be designed in a straightforward way, writing the logical expressions for each of the three outputs. We first define the five auxiliary entities

$$\begin{aligned}
P_1 &= \bar{I}_5 \cdot \bar{I}_4 \cdot \bar{I}_3 \cdot \bar{I}_2 \cdot I_1,\\
P_2 &= \bar{I}_5 \cdot \bar{I}_4 \cdot \bar{I}_3 \cdot I_2,\\
P_3 &= \bar{I}_5 \cdot \bar{I}_4 \cdot I_3,\\
P_4 &= \bar{I}_5 \cdot I_4,\\
P_5 &= I_5,
\end{aligned}$$

(where the dot indicates logical AND). Then define the four outputs by

$$Y_2 = I_5 + I_4, \quad Y_1 = P_3 + P_2, \quad Y_0 = I_5 + P_3 + P_1, \quad \text{None} = \bar{I}_5 \cdot \bar{I}_4 \cdot \bar{I}_3 \cdot \bar{I}_2 \cdot \bar{I}_1.$$

(where "+" indicates logical OR).

However, a more practical approach is shown in Figure 7.23. It uses identical circuits (although the leftmost one is slightly different) that generate the complements of the five auxiliary entities $P_i$. The four outputs can now easily be generated using just a few additional gates. Such a circuit is practical and easy to fabricate on an integrated circuit, because the same pattern is repeated several times.

| $I_5$ | $I_4$ | $I_3$ | $I_2$ | $I_1$ | $Y_2$ | $Y_1$ | $Y_0$ | none |
|---|---|---|---|---|---|---|---|---|
| 1 | $x$ | $x$ | $x$ | $x$ | 1 | 0 | 1 | 0 |
| 0 | 1 | $x$ | $x$ | $x$ | 1 | 0 | 0 | 0 |
| 0 | 0 | 1 | $x$ | $x$ | 0 | 1 | 1 | 0 |
| 0 | 0 | 0 | 1 | $x$ | 0 | 1 | 0 | 0 |
| 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |

**Table 7.22:** A priority encoder truth table

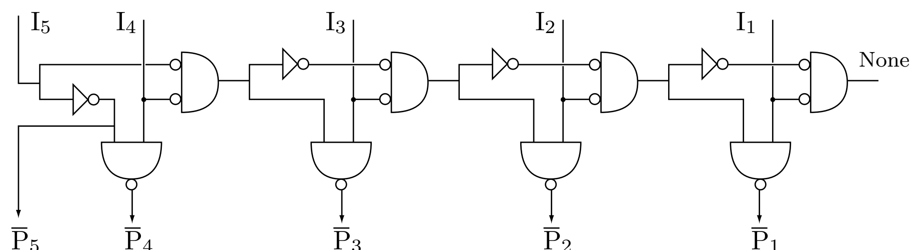



**Figure 7.23:** A priority encoder

This chapter covers some of the important digital devices used in computers. There are many more, less important digital devices, as well as more sophisticated versions of the devices described here. Those interested in more information on this important topic are referred to [Mano 97] and [Mano 91] as well as to the may other texts in this area.

Ian is really stretching the truth here since those
SmurFFs he's talking about are more flop than flip.

— Ruth M. Sprague, *Wild Justice*

# 8 The Memory

Memory is one of the three main units of the computer (the other two being the processor and the I/O processor). Section 1.3 discusses the basic features of memory, such as addresses and words, and how memory is accessed by the control unit. Section 3.4 explains that the DMA device is also a memory requestor. This chapter discusses the internal workings of the memory unit.

Memory can be classified in a number of ways as follows:

1. By its storage technique. Modern memories are semiconductor, but old computer memories used magnetic cores, delay lines, shift registers, and other techniques.

2. As random access or sequential. In a random access memory (RAM), accessing any memory location, regardless of its address, takes the same amount of time (although it may take different times to read and to write).

3. As read/write (RWM) or read only (ROM).

4. By whether it is volatile (i.e., loses its content when power is turned off) or nonvolatile.

5. Whether it is destructive or not (in a destructive memory, reading a word destroys its content).

▸ **Exercise 8.1:** Reading a word in a destructive memory erases its content. How can the word be reread?

## 8.1 A Glossary of Memory Terms

Memory is a computer system's primary workspace. It works in tandem with the CPU, to store data, programs, and processed information that can be made immediately and directly accessible to the CPU or to other parts of the computer. Memory is central to a computer's operation because it forms the critical link between software and the CPU. Computer memory also determines the size and number of programs that can be run simultaneously, and helps to increase the capabilities of today's powerful microprocessors.

There are several different types of memory, each with its own features and advantages. Unfortunately, with different types of memory, it can be easy to get them confused. This glossary is an attempt to help sort out the confusion and act as a quick reference.

The two main forms of RAM are DRAM and SRAM.

**DRAM** (Dynamic RAM). This is the most common type of computer memory. It is called "dynamic" because it must be refreshed, or re-energized, hundreds of times each second in order to retain the data in its words. Each bit in a word in DRAM is designed around a tiny capacitor that can store an electrical charge. A charged capacitor indicates a 1-bit. However, the capacitor loses its charge rapidly, which is why DRAM must be refreshed.

**SRAM** (Static RAM). This type of memory is about five times faster, twice as expensive, and twice as big as DRAM. SRAM does not need to be refreshed like DRAM. Each bit in SRAM is a flip-flop; a circuit that has two stable states. Once placed in a certain state, it stays in that state. Flip-flops are faster to read and write than the capacitors of DRAM, but they consume more power.

Because of its lower cost and smaller size, DRAM is used mostly for main memory, while the faster, more expensive SRAM is used primarily for cache memory.

**Cache RAM**. Cache is small, fast memory (usually SRAM) located between the CPU and main memory, that is used to store frequently used data and instructions. When the processor needs data, a special piece of hardware (the cache manager) checks the cache first to see whether the data is there. If not, the data is read from main memory both into the cache and the processor.

Cache works much like a home pantry. A pantry can be considered a "cache" for groceries. Instead of going to the grocery store (main memory) every time you're hungry, you can check the pantry (cache) first to see if the food you want is there. If it is, then you've saved a lot of time. Otherwise, you have to spend the extra time to get food from the store, leave some of it in the pantry, and consume the rest.

**FPM** (Fast page mode) DRAM. This type of DRAM memory is divided into equal-size chunks called *pages*. It has special hardware that makes it faster to read data items if they are located in the same page.

Using FPM memory is like searching in a dictionary. If the word you want is on the same page as the previous one, it is fast and easy to scroll down the page and find the definition. If you have to flip pages, however, it takes a little longer to find what you want.

**EDO** (Extended Data Out) DRAM. EDO DRAM is similar to FPM, the difference being that back-to-back memory accesses are much faster in EDO. Because EDO is easy to implement, it has become very popular. (EDO is sometimes called Hyper Page Mode DRAM.)

**BEDO** (Burst EDO) DRAM. This is a variation of EDO where a burst of data can be read from memory with a single request. The assumption behind this feature is that the next data-address requested by the CPU will be sequential to the last one, which is usually true. In BEDO DRAM all memory accesses occur in bursts.

**SDRAM** (Synchronous DRAM). This type of DRAM can synchronize itself with the computer clock that controls the CPU. This eliminates timing delays and makes memory reads more efficient.

**SGRAM** (Synchronous graphics RAM). SGRAM is an extension of SDRAM that includes graphics-specific read/write features. SGRAM allows data to be read or written in blocks, instead of as individual words. This reduces the number of reads and writes that the memory must perform, and increases the performance of a graphics controller.

**VRAM** (Video RAM). Graphics memory must be very fast because it has to refresh the graphics screen (CRT) many times a second in order to prevent screen "flicker." While graphics memory refreshes the CRT, the CPU or the graphics controller write new data into it in order to modify the image on the screen. With ordinary DRAM, the CRT and CPU must compete for memory accesses, causing a bottleneck of data traffic.

VRAM is a "dual-ported" memory that solves this problem by using two separate data ports. One port is dedicated to the CRT, for refreshing the screen. The second port is dedicated for use by the CPU or graphics controller, to modify the image data stored in video memory.

A good analogy to VRAM is a fast food drive-through with two windows. A customer places an order and pays at one window, then drives up and picks up the food at the next window.

Figure 8.1a shows a memory unit of size $M \times N$ and the buses connecting it to the rest of the computer (Section 1.3 shows why $M$ is a power of 2, $M = 2^k$). The two main types of memory used in modern computers are semiconductor RAM (read/write) and ROM (read only) and the two main types of RAM are static (SRAM) and dynamic (DRAM). Static RAM is the most important type of computer memory, and its design is the first topic discussed in this chapter.

## 8.2 Static RAM

Static RAM stores each bit of information in a latch. Thus, there may be millions of latches in a single SRAM memory integrated circuit (chip). A single latch and the gates around it are called a *bit cell* (BC, Figure 8.1b,c) and the main problem in RAM design is to construct a large memory with the minimum number of gates and wires. The individual BCs are organized in words, and the address lines are decoded, to become select lines that select the BCs of an individual word.

An intuitive way of combining individual BCs into a memory unit is shown in Figure 8.2a. Four words of three bits each are shown. The diagram is simple and easy to understand, but is not practical because of the multi-output decoder needed. Even a memory with 1K words (which is extremely small by today's standards) requires a decoder with 1024 output lines! The solution is to use two or more decoders, and to

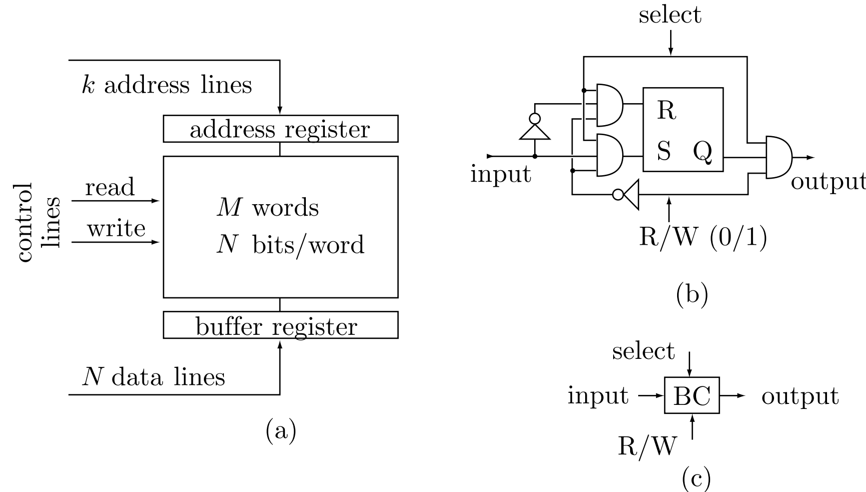



**Figure 8.1:** RAM (a) and a single BC (b,c)

construct a memory unit from bit-cell building blocks that are selected by two or more select lines each. Figure 8.2b shows an example of such memory, with an individual BC shown in Figure 8.2c. Such a memory unit has a capacity of $M \times 1$ (where $M$ can be in the millions) and is fabricated on a single chip. Assuming that the memory has 1K words, each decoder receives five of the 10 address lines and has 32 output lines. The total number of decoder outputs is therefore 64 instead of 1024. There is one output line and one input line for the entire memory.

▸ **Exercise 8.2:** The memory unit of Figure 8.2a has two control lines, R/W and enable. Show how to replace them with the two lines $R$ and $W$ (i.e., read and write, with no enable line).

▸ **Exercise 8.3:** Suggest a way to design a 1M memory unit with a small number of decoder outputs.

A complete $M \times N$ memory unit consists of $N$ such memory chips, as shown in Figure 8.3a. Figure 8.3b shows how several small memory chips can be connected to increase the number of words (i.e., the value of $M$). The diagram shows four memory chip of size 256K$\times N$ each, connected to form a 1M$\times N$ memory unit. Recall that 1M $= 2^{20}$, so a memory with 1M words requires 20 address lines. Each chip, however, contains 256K $= 2^{18}$ words, and requires only 18 address lines. The two most significant address lines are therefore used to select one of the four memory chips by means of a 1-of-4 decoder.

## 8.3 ROM

The simplest type of ROM is programmed when it is manufactured and its content cannot be changed later. Such ROM consists of a grid of $M$ word lines and $N$ bit lines, with no connections between word lines and bit lines at the grid points (Figure 8.4a). The $N$ outputs of such a grid are all zeros. In the last manufacturing step, the data is written in the ROM by fabricating a diode at each grid intersection where a 1 is required. This process is called *masking* and it sets some of the $N$ outputs to 1's, depending on which word is selected. The advantage of this fabrication process is that a customer can order a custom made ROM, and this affects only the last stage of manufacturing, not the entire process. If large quantities of identical ROMs are needed, the price per unit is small.

When only a small number of ROM units is needed, a programmable ROM (PROM) may be the solution. This type of ROM is manufactured as a uniform grid of words and bits, where every grid intersection has a small diode (Figure 8.4b). Thus, every word in the PROM consists of all 1's. The PROM is programmed by placing it in a special machine that fuses the diodes at those grid intersections where zeros are required. Once programmed, the content of a PROM cannot be changed (strictly speaking, bits of 1 can be changed to 0, but not the opposite).

For applications where the content of the ROM has to be modified several times, an erasable PROM (EPROM) is the natural choice. Such a device can be programmed, erased and reprogrammed several times. Programming is done by trapping electrical charges at those grid points where 1's are needed. Erasing is

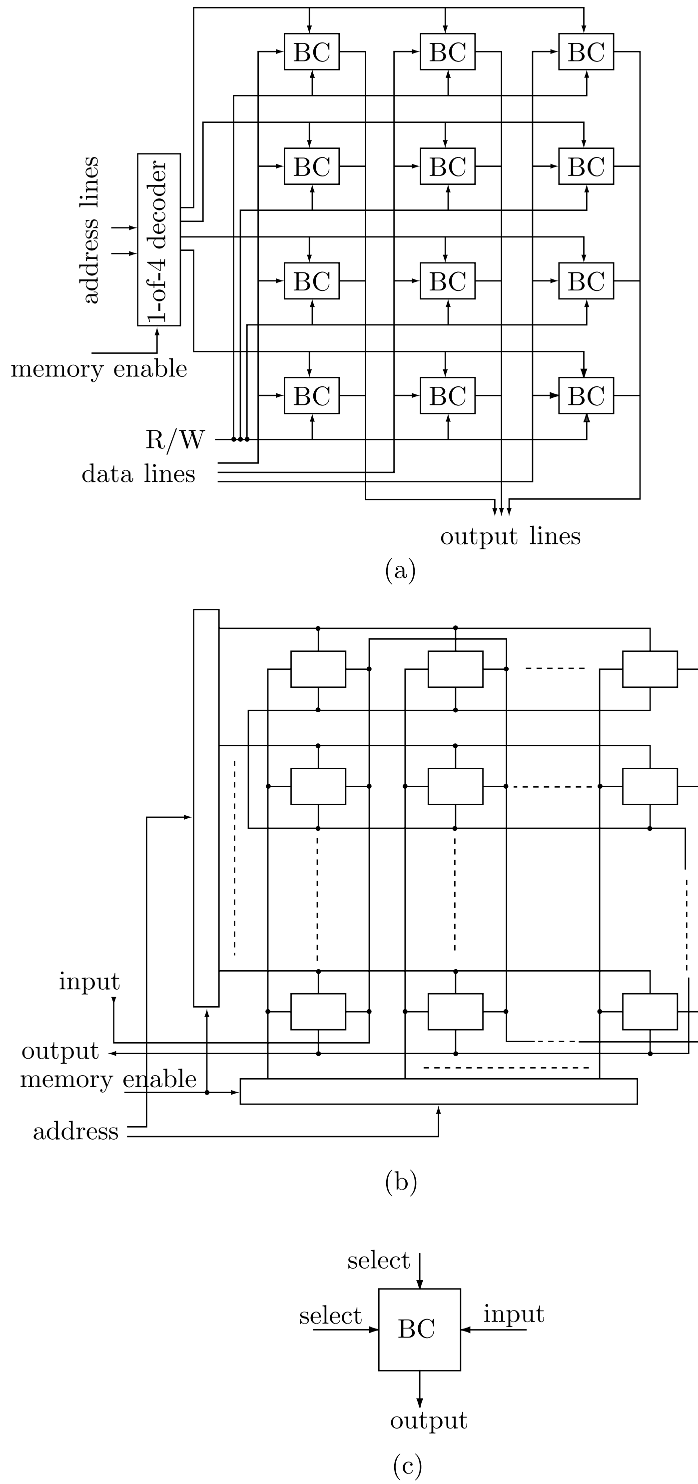



**Figure 8.2:** RAM (a,b) and a single BC (c)

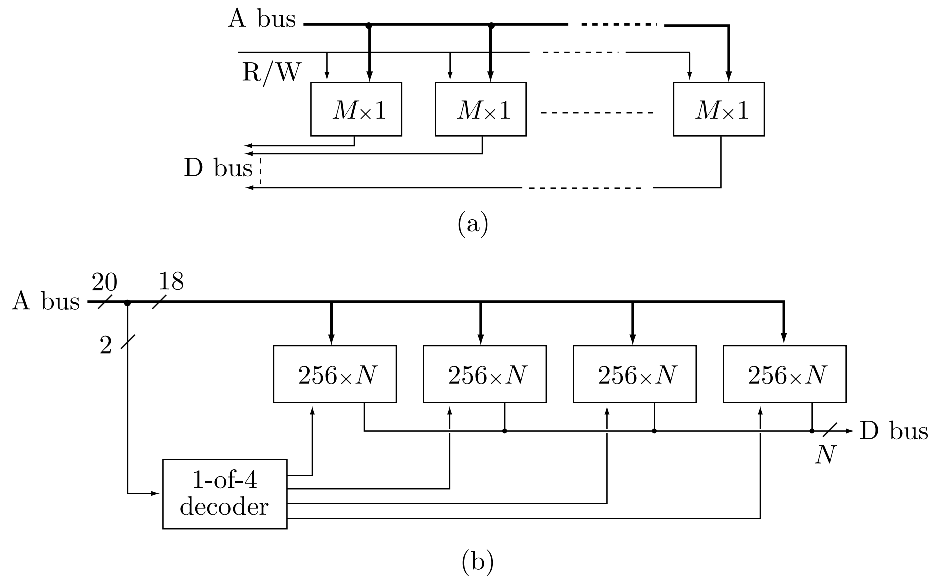



**Figure 8.3:** Increasing $N$ (a) and increasing $M$ (b)

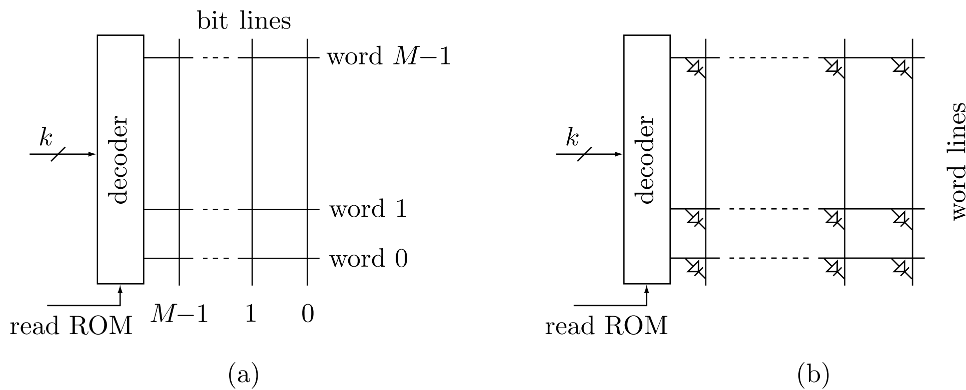



**Figure 8.4:** ROM (a) and PROM (b)

done by releasing the charges, either by exposing the device to ultra violet light or by applying high voltage to it in a direction opposite that of the charges.

▸ **Exercise 8.4:** Suggest a typical application where the content of a ROM has to be changed several times before it stabilizes.

## 8.4 PLA

The term PLA stands for *programmable logic array*. This device is similar to a ROM. An address is sent to it, and out comes a piece of data. However, in a PLA, only those locations that are actually needed are fabricated. In a conventional $M \times N$ ROM or PROM, all $2^M$ locations are fabricated and exist on the memory chip. A 1K PROM has all 1024 locations, even though any particular application may use just a small number of locations. A PLA, on the other hand, may have 10 address lines but only 75 actual locations. Most of the 1024 possible addresses may not correspond to any actual words, while several addresses may correspond to the same word.

An interesting application of a PLA is conversion from the old, obsolete punched cards character codes to ASCII codes. The standard punched cards that were so popular with computers in the past had 12 rows and 80 columns where holes could be punched. Each column corresponded to one character, so each character had a 12-bit code. However, only 96 characters were actually used. An application that reads

punched cards and converts the character codes to ASCII may benefit from a PLA with 12-bit addresses and 96 8-bit locations. Each of the 96 locations is set to an ASCII code, and a character is converted by simply using its original, 12-bit code as an address to the PLA. The content of that address is the required ASCII code.

▸ **Exercise 8.5:** Can this problem be solved with a ROM?

A PLA consists of an array of AND gates followed by an array of OR gates. Every input is connected to all the AND gates, and the output of every AND gate is connected to all the OR gates. The resulting device has two grids, one between the inputs and the AND gates and the other between the AND gates and the OR gates. The device is made such that there are no electrical connections at the grid points. The last step in the fabrication process is to program the device by connecting the two wires at certain grid points. Figure 8.5a shows a PLA with four inputs $a_3a_2a_1a_0$, three AND gates, and two OR gates. With four input (i.e., address) lines, there are 16 possible addresses. However, the connections shown allow only for the seven addresses 1010 (the left AND gate), 0x01 (the center gate) and 00xx (the right one). The bottom OR gate (output bit $d_0$) outputs a 1 for addresses 1010 and 0x01, and the top OR gate outputs a 1 for addresses 1010 and 00xx.

▸ **Exercise 8.6:** List the 2-bit content of each of the seven addresses.

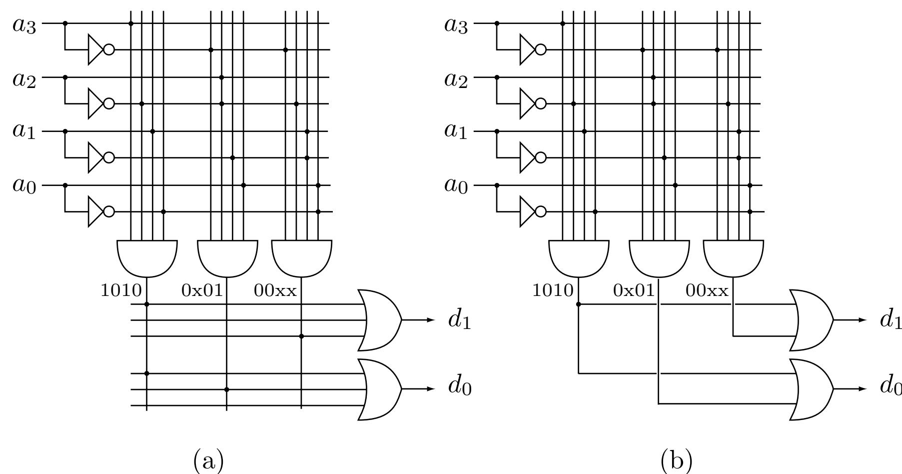



**Figure 8.5:** PLA (a) and PAL (b)

A relative of the PLA is the FPLA (field programmable logic array). This device is manufactured as a PLA with connections at all the grid points and with fuses at every input to a logic gate. A special instrument is used to burn some fuses selectively, thereby disconnecting their inputs from the gates. This effectively programs the FPLA.

A third member of the PLA family is the PAL (programmable array logic). This useful digital device is similar to a PLA with the difference that the array of OR gates is not programmable (i.e., there is no complete grid between the AND gates and the OR gates). Figure 8.5b shows an example.

My father possessed an extraordinary memory, especially for dates,
so that he knew, when he was very old, the day of the birth,
marriage, and death of a multitude of persons in Shropshire

— Charles Darwin, *Life and Letters of Charles Darwin, Volume I*

# 9 The ALU

The ALU of a modern computer is a complex device that can execute the many operations specified by the machine instructions. These operations, as mentioned in Section 2.11, can be classified into three categories, namely arithmetic, logic, and shifts. In addition, the ALU also performs comparisons. This chapter is a survey of some of the execution circuits found in a modern ALU. The aim is to convince the reader that all the ALU operations can be performed by logic gates. We start with the arithmetic operations.

## 9.1 Integer Addition and Subtraction

The simplest arithmetic operations are adding and subtracting two bits $A$ and $B$. Table 9.1 lists the sum $S$, carry $C$, difference $D$, and borrow $R$ resulting from these operations.

| | | $A+B$ | | $A-B$ | |
|---|---|---|---|---|---|
| $A$ | $B$ | $S$ | $C$ | $D$ | $R$ |
| 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 | 1 | 1 |
| 1 | 0 | 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 | 0 |

**Table 9.1:** Adding and subtracting two bits

It is clear from the table that $S = D = A \oplus B$, $C = A \cdot B$, and $R = A' \cdot B$ (where $\oplus$ is the XOR operation, Table 2.6). Thus, adding two bits requires an XOR and AND gates, and subtracting them requires the same gates plus an inverter. These two simple operations are similar, so they can be combined in one device with a control line specifying which operation is required. Such a device is called a half-adder-subtractor (HAS) because it cannot add or subtract entire numbers.

A full adder (FA) and full subtractor (FS) operate on entire numbers, not just a single pair of bits, by propagating the carry or borrow. A FA should be able to add two bits and an input carry, and produce a sum and an output carry. A FS should be able to subtract two bits, subtract an input borrow from the result, and produce a difference and an output borrow. Table 9.2 lists the eight cases that have to be handled by a full-adder-subtractor. The input carry and input borrow are denoted by $IC$ and $IB$, respectively.

It is easy to see that the four outputs can be produced by

$$\begin{aligned}
S &= A \oplus B \oplus IC,\\
C &= A' \cdot B \cdot IC + [A \cdot B' \cdot IC + A \cdot B \cdot IC' + A \cdot B \cdot IC] = A' \cdot B \cdot IC + A(B + IC),\\
D &= A \oplus B \oplus IB,\\
R &= [A' \cdot B' \cdot IB + A' \cdot B \cdot IB' + A' \cdot B \cdot IB] + A \cdot B \cdot IB = A \cdot B \cdot IB + A'(B + IB).
\end{aligned}$$

| | | $A+B+IC$ | | | $A-B-IB$ | | |
|---|---|---|---|---|---|---|---|
| $A$ | $B$ | $IC$ | $S$ | $C$ | $IB$ | $D$ | $R$ |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 | 1 | 1 | 1 |
| 0 | 1 | 0 | 1 | 0 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 |
| 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

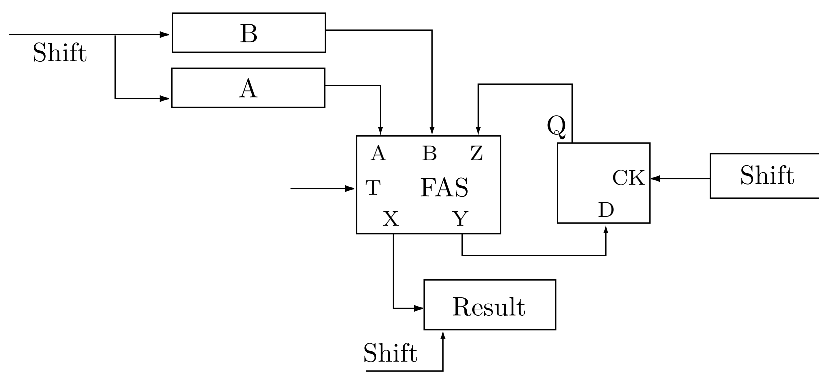
**Table 9.2:** Adding and subtracting three bits

As a result, a full-adder-subtractor can be designed with four inputs $A$, $B$, $Z$ ($IC$ or $IB$), and $T$ (high for add and low for subtract) and two outputs $X$ ($S$ or $D$) and $Y$ ($C$ or $R$). This device is defined by

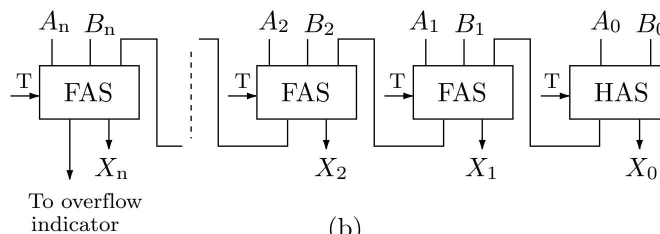
$$\begin{aligned}
X &= A \oplus B \oplus Z,\\
Y &= C \cdot T + R \cdot T' = A' \cdot B \cdot Z \cdot T + A(B+Z)T + A \cdot B \cdot Z \cdot T' + A'(B+Z)T'\\
&= B \cdot Z(A'T + AT') + (B+Z)(A \cdot T + A' \cdot T')\\
&= B \cdot Z(A \oplus T) + (B+Z)(A \oplus T)'.
\end{aligned}$$

▸ **Exercise 9.1:** Use logic gates to design an FAS.

Such a FAS, combined with a latch, can be used as a *serial* adder/subtractor (Figure 9.3a). This circuit is triggered by the clock pulse, and it adds one pair of bits for each pulse. The numbers to be added/subtracted are held in two shift registers whose least-significant bits are fed to the $A$ and $B$ inputs of the FAS. The $X$ output of the FAS is fed into the result register (also a shift register), and the $Y$ output is sent to the latch, where it is saved until the next clock pulse, when it is sent to the $Z$ input.




(a)




(b)

**Figure 9.3:** An adder/subtractor, (a) serial and (b) parallel

Figure 9.3b shows how the FAS can serve as a building block of a *parallel* adder/subtractor. Each FAS receives a pair of bits $A_i$ and $B_i$. The carry/borrow is sent to the next most-significant FAS, where it is added to the next pair of bits. The least significant bit position does not have any input carry/borrow, so it needs only a HAS, not a FAS. This is a simple design, but it is not completely parallel. All the stages get their inputs in parallel and produce their outputs simultaneously, but the two outputs of stage $i$ are initially wrong, since the $Z$ input to this stage is initially wrong. Stage $i$ receives the correct $Z$ input from stage $i-1$ only when that stage produces correct outputs, and this happens only after stage $i-1$ has received the correct $Z$ input from stage $i-2$. All the stages operate continuously, but there is a ripple effect that propagates from the least-significant to the most-significant stages. The most-significant stage produces correct outputs after a delay proportional to the number of stages, so this device is not truly parallel and deserves the name ripple-carry adder/subtractor.

In practice, the computer uses two's complement numbers (Section 2.20.3), so the ALU does not need a subtractor, but it needs a complementor. More sophisticated methods for adding integers are described below, but first here is a short discussion of two's complementors.

A parallel two's complementor is shown in Figure 9.4a. The "complement" control signal transfers the contents of register $A$ to a counter while inverting every bit. After a short delay, to guarantee that all the bits have been transferred, the control signal arrives at the counter and increments its content by 1, to form the two's complement.

There are two special cases. The case $A = 0$ generates overflow, since the counter is first filled up with 1's, then incremented. The case $A = 10\ldots0$ is also problematic, since the counter is first filled up with $01\ldots1$, then incremented to produce the original value $10\ldots0$. The conclusion is that this number (the smallest negative integer) cannot be 2's complemented. This case can easily be detected, though, by comparing the final sign of the counter to the original sign of $A$. They should be different and the case above is the only one where they will be identical.

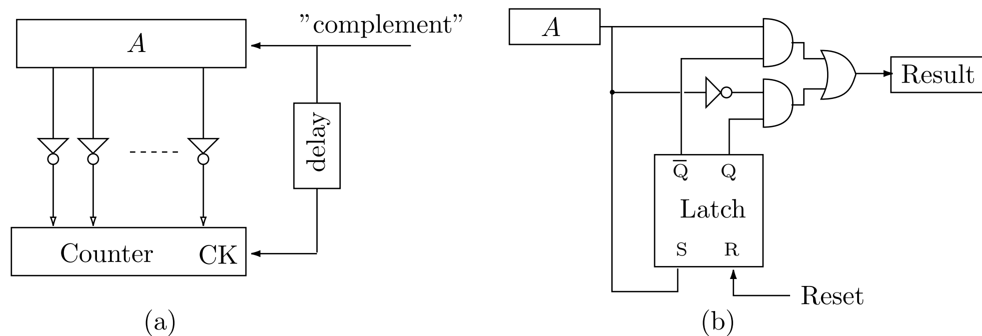



**Figure 9.4:** A two's complementor, (a) parallel and (b) serial

It is also possible to generate the two's complement of an integer serially, bit by bit. This is based on the observation that the 2's complement of the integer $x\ldots x10\ldots0$ is $\bar{x}\bar{x}\ldots\bar{x}10\ldots0$, i.e., the least-significant zeros and the first bit of 1 preceding them are preserved in the process. All the bits to the left of that 1 should be flipped.

▸ **Exercise 9.2:** Prove this observation.

The serial two's complementor shown in Figure 9.4b uses an SR latch to detect that 1. The latch is initially reset. Bits are moved from the right end of $A$ to the left end of the result register (both registers are shifted to the right). As long as these bits are zeros, the latch stays reset, so the upper AND gate is open. The first 1 sets the latch (and also passes to the result register unchanged), so the lower AND gate opens and all the remaining bits from $A$ are complemented before they pass to the result register.

▸ **Exercise 9.3:** Show that the remaining bits from $A$ do not affect the state of the latch regardless of their values.

Once the ALU knows how to produce the two's complement of an integer, only an adder is needed in order to implement both addition and subtraction. The next section describes an efficient adding technique known as *carry look-ahead*.

### 9.1.1 Carry Look-Ahead Addition

The performance of a ripple-carry adder is limited because of the time it takes to propagate the carry from stage to stage. The carry look-ahead (CLA) method described here generates simultaneously all the carries and sends them to all the adder stages. Such an adder is (at least theoretically) truly parallel. In practice, however, fan-in limitations restrict the performance of such a device to less than truly parallel.

For each bit position, the CLA adder has a (modified) full adder (FA), except the least-significant position, where there is a half adder (HA). Each of these devices receives a pair of bits $A_i$ and $B_i$ to be added and outputs certain information to a CLA circuit common to all the stages. The circuit uses this information to calculate all the carry bits $C_i$, which are then sent to the individual stages. When a full-adder receives a carry bit, it generates the correct sum bit $S_i$.

To understand how the carries can be computed in parallel, we observe from Table 9.2 that the carry $C$ output from a stage depends on the input bits $A$ and $B$ and on the input carry $IC$ as follows: $A = B = 1 \to C = 1$, $A = B = 0 \to C = 0$, $A \neq B \to C = IC$. If the two input bits are zeros, the stage does not generate a carry. If those bits are 1's, a carry is always generated, regardless of the input carry. If the two input bits are different, the input carry is propagated. The process can be summarized by

$$C_i = A_iB_i + (\bar{A}_iB_i + A_i\bar{B}_i)IC_i = G_i + P_iC_{i-1},$$

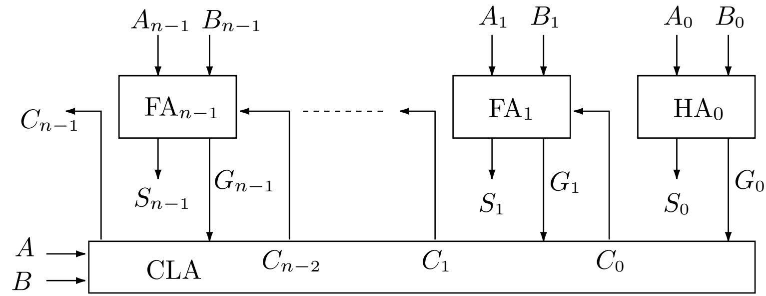
where $G_i = A_iB_i$ and $P_i = \bar{A}_iB_i + A_i\bar{B}_i = A_i \oplus B_i$. This is a recursive expression for $C_i$ that can be written explicitly

$$\begin{aligned}
&C_0 = G_0,\\
&C_1 = G_1 + P_1C_0,\\
&C_2 = G_2 + P_2C_1 = G_2 + P_2(G_1 + P_1C_0) = G_2 + P_2G_1 + P_2P_1C_0,\\
&C_3 = G_3 + P_3C_2 = G_3 + P_3(G_2 + P_2G_1 + P_2P_1C_0) = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1C_0,\\
&\quad\vdots\\
&C_n = G_n + P_nC_{n-1} = G_n + P_nG_{n-1} + P_nP_{n-1}G_{n-2} + \cdots + P_nP_{n-1}P_{n-2}\ldots P_1C_0.
\end{aligned}$$

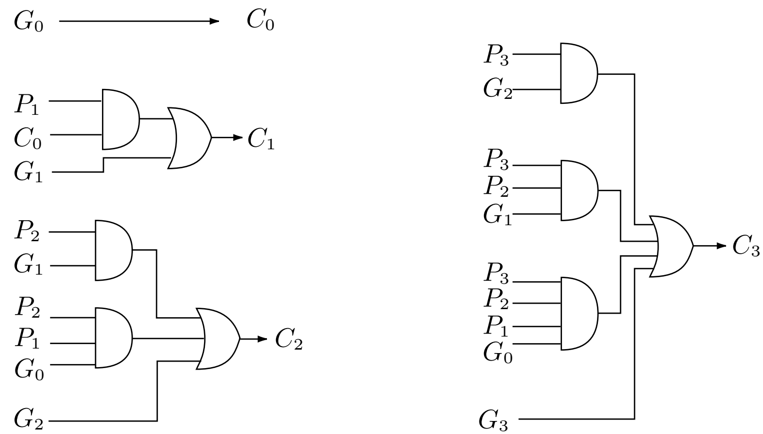
Thus, the CLA circuit receives the $G_i$'s and $P_i$'s from the individual modified full adders (except the single half adder, which sends $C_0$), and generates all the $C_i$'s (from $C_1$ to $C_n$) in parallel. Figure 9.5a is a schematic diagram of the FAs and the CLA circuit. Figure 9.5b shows the details of generating $C_1$, $C_2$, and $C_3$. It is clear that fan-in becomes a problem. Even $C_3$ requires gates with a fan-in of 4! Therefore, a practical CLA adder must be designed in several parts. Each part waits for a carry bit from its predecessor, adds its input bits using the CLA method, then sends its most-significant carry to the next part. Figure 9.5c shows a 12-bit, three-part adder, where each part adds four bits. Such an adder is a cross between a ripple-carry adder and a true CLA adder.

▸ **Exercise 9.4:** What fan-in is required for $C_4$?

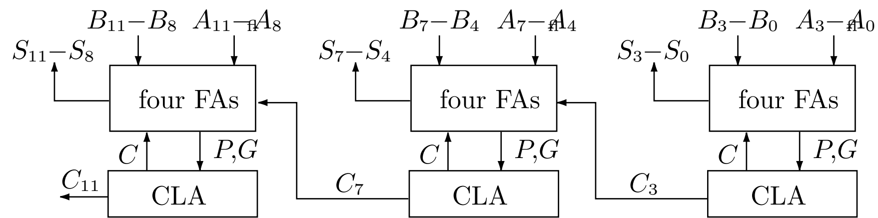
▸ **Exercise 9.5:** The computation of $C_3$ requires the expression $P_3P_2P_1G_0$, which is one reason for the fan-in of 4. However, the subexpression $P_2P_1G_0$ is available as part of the calculation of $C_2$, and so can be used to reduce the fan-in required for $C_3$. What's wrong with this argument?




(a)




(b)




(c)

**Figure 9.5:** Carry look-ahead adder

## 9.2 The Accumulator

A design for a general-purpose register is presented in this section. This register can perform shifts, complements, logic operations, and addition and subtraction. It is also used by the multiplication and division circuits of Sections 9.3 and 9.4, where it is assisted by another general-purpose register, called the MQ. Old, second and third-generation computers many times had one such register, called *accumulator*. Modern computers generally have a block of general-purpose registers.

The accumulator design described here is based on a pair of registers AL, AU (for lower and upper). Four types of control signals are used to operate on this register pair. They are shift signals, logic signals, add/subtract signals, and a complement signal.

The four shifts are Up, Down, Left, and Right. They are summarized in Figure 9.6a with the details shown in Figure 9.6b. The extreme bits of AL are either lost or can be shifted into other computer components, such as the carry (C) flag. Note that the left and right shifts are different from those of a regular shift register because they also move the shifted number from AL up to AU.

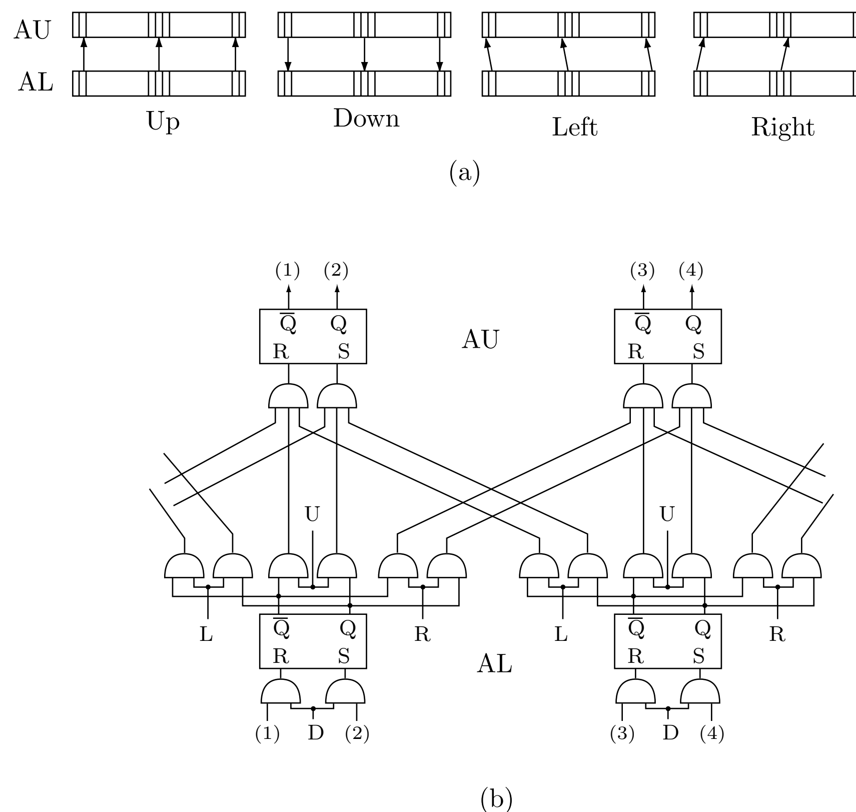



**Figure 9.6:** Shifts in the accumulator

Adding and subtracting requires an FAS for each bit position (except the least-significant one, where only an HAS is needed). Any register S can be transferred to the row of FASs, added to or subtracted from AU, with the result sent to AL. Thus, the two control signals ADD and SUB perform AL $\leftarrow$ AU + S and AL $\leftarrow$ AU − S, respectively. Figure 9.7a,b shows the general setup and details.

The 1's complement operation in our accumulator is AL $\leftarrow \overline{\text{AU}}$. This is done by transferring the $Q$ output of each stage AU$_i$ to the $R$ input of stage AL$_i$ and transferring the $\bar{Q}$ output of AU$_i$ to the $S$ input of AL$_i$. The logical AND operation is AL $\leftarrow$ (AL and AU). This is done by feeding the $Q$ outputs of AL$_i$ and AU$_i$ to an AND gate, and sending the gate's output to the $S$ input of AL$_i$ while the inverse of the gate's output is sent to the $R$ input of AL$_i$.

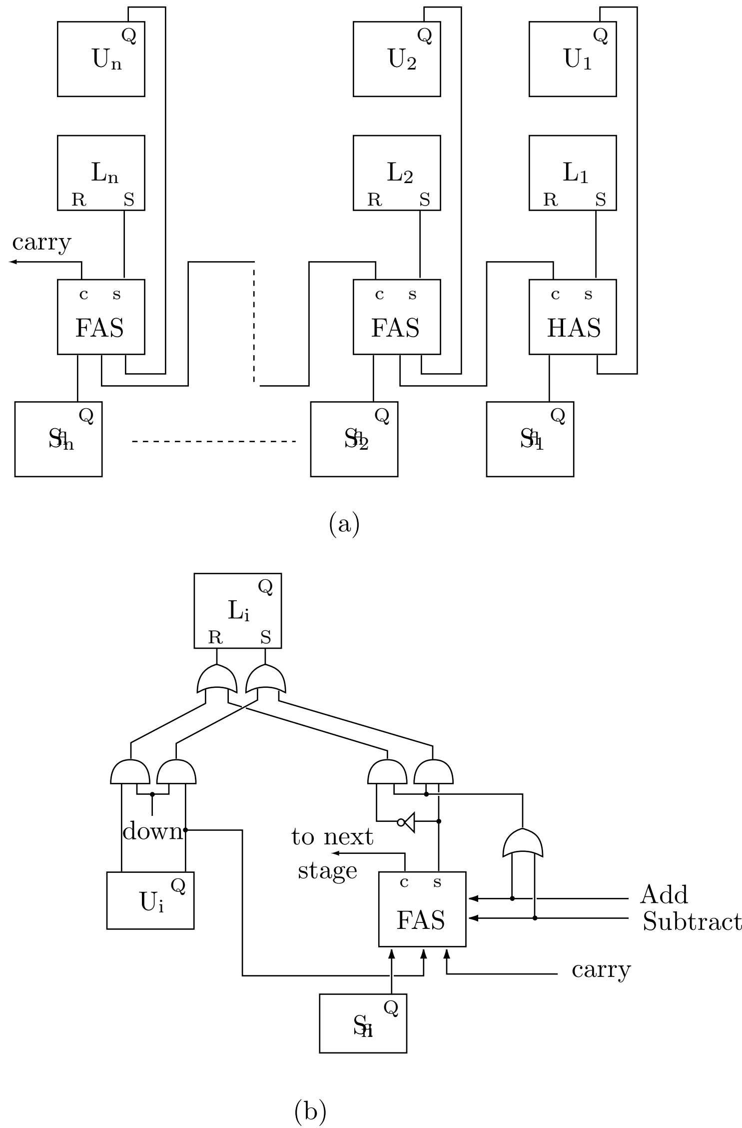



**Figure 9.7:** Addition/Subtraction in the accumulator

## 9.3 Integer Multiplication

Multiplying binary integers turns out to be a simple operation, since it can be done by repeated shifting and adding. The reason for this is that the binary multiplication table is simple. Multiplying two bits results in a zero, except when both bits are 1's. Thus, the binary multiplication table is identical to the truth table of the AND gate, but this fact is used only indirectly in the design of the actual circuits. Table 9.8 is an example of multiplying two 4-bit integers. It shows how this operation is expressed naturally in terms of shifts and additions, and why the size of the product is the sum of the sizes of the individual numbers.

Studying this example leads us to the following multiplication rule: Set the product to zero. Go over the bits of the multiplier (= 0110) from right (least-significant) to left. If the current multiplier bit is 1, generate a partial product that's a copy of the multiplicand (= 1001); otherwise, generate a partial product of all zeros. Shift the partial product to the left and add it to the product so far.

The implementation described here uses the following three registers:

1. AC (accumulator). This will contain the most-significant part of the product.

$$
\begin{array}{rl}
\times \; 1001 & \text{multiplicand} \\
0110 & \text{multiplier} \\
\hline
0000 & \\
1001\phantom{0} & \\
1001\phantom{00} & \\
0000\phantom{000} & \\
\hline
0110110 &
\end{array}
$$

**Table 9.8:** Multiplying two 4-bit integers

2. MQ (multiplier-quotient). This is originally set to the multiplier. At the end of the operation, the MQ contains the least-significant part of the product.

3. S (source). This contain the multiplicand (it can be a register or a word in memory).

The algorithm is based on shifts and additions, but is different from the rule above. It generates the partial product only when this should be the multiplicand. A partial product of zeros is never generated. Instead of shifting the partial product to the left, the sum-so-far (which is contained in the pair AC, MQ) is shifted to the right and nonzero partial products are added to it.

1. $\mathrm{AC} \leftarrow 0$.
2. If $\mathrm{MQ}_0 = 1$, then $\mathrm{AC} = \mathrm{AC} + \mathrm{S}$.
3. right shift AC and MQ as one unit.
4. Repeat steps 2 and 3 $n$ times.

In order to simplify our implementation, we assume that both the AC and MQ are double registers, i.e., there are two registers AU and AL (for *upper* and *lower*) and two registers MU and ML. We assume that each can be shifted up and down (i.e., AU ← AL and AL ← AU) and also left and right. The control signals for the vertical shifts are denoted by SAU, SAD, SMU, and SMD. The algorithm now becomes

1. $\mathrm{AU} \leftarrow 0$.
2. SMD. If $\mathrm{MU}_0 = 1$, then $\mathrm{AL} = \mathrm{AU} + \mathrm{S}$, else SAD.
3. right shift AC and MQ as one unit.
4. Repeat steps 2 and 3 $n$ times.

Figure 9.10a is a schematic diagram of the integer multiplication circuit of the ALU. It shows that bit 0 of the MU register is fed to this circuit. In addition, the circuit also receives a "start" input from the control unit, and the clock pulse $\phi$. When done, the multiplication circuit sends a "done" signal back to the control unit. The ADD output is a control signal to perform the addition $\mathrm{AL} = \mathrm{AU} + \mathrm{S}$. The SR output is a control signal that shifts and Ac and MQ to the right. The circuit contains a modulo $2n+2$ counter (Figure 9.10b) that's reset by the "start" signal and is clocked by the clock pulse. The counter has $L+1$ output lines $C_0$, $C_1, \ldots, C_L$ that generate the correct sequence of shift signals sent to the AC and MQ. This sequence is listed in Table 9.9.

| $C_0$ | $C_1$ | $C_2$ | $\ldots$ | $C_L$ | output |
|---|---|---|---|---|---|
| 0 | 0 | 0 | | 0 | CR |
| 1 | 0 | 0 | | 0 | SMD and either ADD or SAD |
| 0 | 1 | 0 | | 0 | SR (1st time) |
| 1 | 1 | 0 | | 0 | SMD and either ADD or SAD |
| 0 | 0 | 1 | | 0 | SR (2nd time) |
| ⋮ | ⋮ | ⋮ | | ⋮ | |
| 0 | 1 | 1 | | 1 | SR ($n$th time) |
| 1 | 1 | 1 | | 1 | done |

**Table 9.9:** Outputs of the multiplication circuit

Using the table, it is not hard to figure out how each of the outputs of the multiplication circuit is generated by the $L+1$ counter outputs.

$$\begin{aligned}
\mathrm{CR} &= \bar{\mathrm{C}}_0\bar{\mathrm{C}}_1\ldots\bar{\mathrm{C}}_\mathrm{L} = \bar{\mathrm{C}}_0(\bar{\mathrm{C}}_1\bar{\mathrm{C}}_2\ldots\bar{\mathrm{C}}_\mathrm{L}),\\
\mathrm{SMD} &= \mathrm{C}_0(\overline{\mathrm{C}_1\mathrm{C}_2\ldots\mathrm{C}_\mathrm{L}}),\\
\mathrm{ADD} &= \mathrm{MU}_0\mathrm{C}_0(\overline{\mathrm{C}_1\mathrm{C}_2\ldots\mathrm{C}_\mathrm{L}}),\\
\mathrm{SAD} &= \overline{\mathrm{MU}}_0\mathrm{C}_0(\overline{\mathrm{C}_1\mathrm{C}_2\ldots\mathrm{C}_\mathrm{L}}),\\
\mathrm{SR} &= \bar{C}_0(C_1+C_2+\cdots+C_L) = \bar{C}_0(\overline{\bar{C}_1\bar{C}_2\ldots\bar{C}_L}),\\
\mathrm{done} &= C_0C_1\ldots C_L = C_0(C_1C_2\ldots C_L).
\end{aligned}$$

Therefore, only the two expressions $\overline{C_1C_2\ldots C_L}$ and $\overline{\bar{C}_1\bar{C}_2\ldots\bar{C}_L}$ have to be generated. This part of the multiplication circuit is shown in Figure 9.10c.

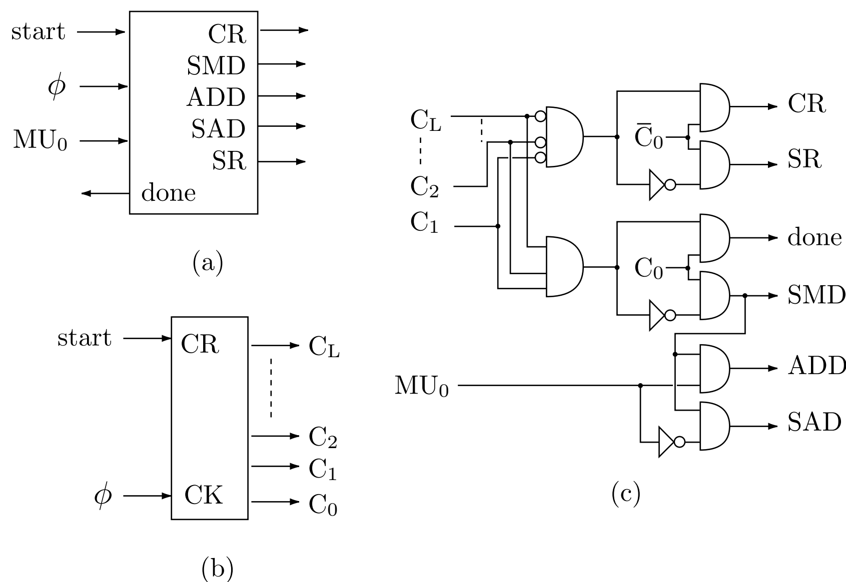



**Figure 9.10:** Integer multiplication (a) general, (b) counter, and (c) details

▸ **Exercise 9.6:** There is a problem with the circuit as shown in Figure 9.10b. When the multiplication is over, the clock pulse continues to clock the counter, thereby generating SMD, ADD, and other signals that may corrupt the content of the accumulator and MQ registers. Show how to stop clocking the counter as soon as the "done" signal is generated.

This simple method can multiply only unsigned numbers. When 2's complement numbers are used, any negative numbers should be complemented before the multiplication starts, and the sign of the product should be determined by comparing the signs of the original numbers. It is possible, however, to design a multiplication circuit that multiplies signed numbers directly. Such a circuit must be able to handle two cases as follows:

1. The multiplicand ($A$ in $A\times B$) is negative. The partial products (which are either $A$ or zero) are negative. When shifting AC and MQ to the right, a 1 should be inserted on the left end, instead of a zero. In general, the sign bit of $A$ should be inserted.

2. The multiplier $B$ is negative. Here we observe that if an integer $N$ is represented by $n$ bits, then the 2's complement of $N$ equals $2^n-N$. Therefore, a negative $B$ has an unsigned value of $2^n-B$ and multiplying $A$ by $B$ results in $A(2^n-B)=2^nA-A\cdot B$. The result should therefore be corrected by subtracting $2^nA$.

▸ **Exercise 9.7:** Illustrate these concepts by multiplying $12\times(-6)$ and $(-6)\times 12$.

### 9.3.1 Simultaneous Multiplication

The simultaneous multiplication circuit described here is similar to the ripple-carry adder/subtractor. It is not entirely parallel, since each stage produces wrong results until it receives the correct carry from its predecessor. The following example of multiplying two 4-bit numbers makes this process clear

$$\begin{array}{r} A_4A_3A_2A_1 \\ B_4B_3B_2B_1 \\ \hline R_8R_7R_6R_5R_4R_3R_2R_1. \end{array}$$

Four partial products are generated by ANDing $A$ with each of the four bits of $B$

$$\begin{array}{rrrrrrr} & & & A_4B_1 & A_3B_1 & A_2B_1 & A_1B_1, \\ & & A_4B_2 & A_3B_2 & A_2B_2 & A_1B_2, & \\ & A_4B_3 & A_3B_3 & A_2B_3 & A_1B_3, & & \\ A_4B_4 & A_3B_4 & A_2B_4 & A_1B_4. & & & \end{array}$$

The eight bits of the result are produced by adding the seven columns of these partial products as follows (where the notation $A_iB_j$ means a logical AND but a "+" means arithmetic addition):

$$\begin{aligned}
R_1 &= A_1B_1, \\
R_2 &= A_2B_1 + A_1B_2, \\
R_3 &= A_3B_1 + A_2B_2 + A_1B_3 + \text{carry from } R_2, \\
R_4 &= A_4B_1 + A_3B_2 + A_2B_3 + A_1B_4 + \text{carry from } R_3, \\
R_5 &= A_4B_2 + A_3B_3 + A_2B_4 + \text{carry from } R_4, \\
R_6 &= A_4B_3 + A_3B_4 + \text{carry from } R_5, \\
R_7 &= A_4B_4 + \text{carry from } R_6, \\
R_8 &= \text{carry from } R_7.
\end{aligned}$$

Figure 9.11 shows an implementation of this design.

### 9.3.2 The Booth Algorithm

The performance of the multiplication method described in Section 9.3 depends on the number of 1's in the multiplier. For each 1 in the multiplier, the multiplicand has to be added to the product-so-far. The Booth method utilizes the properties of binary numbers to try to reduce the number of 1's in the multiplier. In many cases, this method speeds up the multiplication of 2's complement integers substantially. For some multipliers, however, the method actually slows down the multiplication.

The Booth method depends on the observation that an integer that consists of a single run of 1 bits can always be written as the difference of two integers, each with a single 1 bit instead of the run. This is also true when the integer has several runs of 1's. Here are some examples of unsigned numbers

$$\begin{aligned}
0011110 &= 0100000 - 0000010 = 01000\bar{1}0, \\
001111001110 &= 010000010000 - 000001000010 = 01000\bar{1}0100\bar{1}0, \\
001111 &= 010000 - 000001 = 01000\bar{1}, \\
11110 &= 100000 - 00010 = 1000\bar{1}0.
\end{aligned}$$

We denote each example by $a = b - c = d$. The integer $d$ following the second equal sign in each of the examples combines the bits of $b$ and $c$, with a bar added above the bits of $c$. This integer is the *recoded* version of the multiplier. Using this principle, the (unsigned) multiplication

$$\times \begin{array}{r} 11101010 \\ 11110 \end{array} \quad \text{can be written in the form} \quad \times \begin{array}{r} 11101010 \\ 1000\bar{1}0 \end{array}.$$

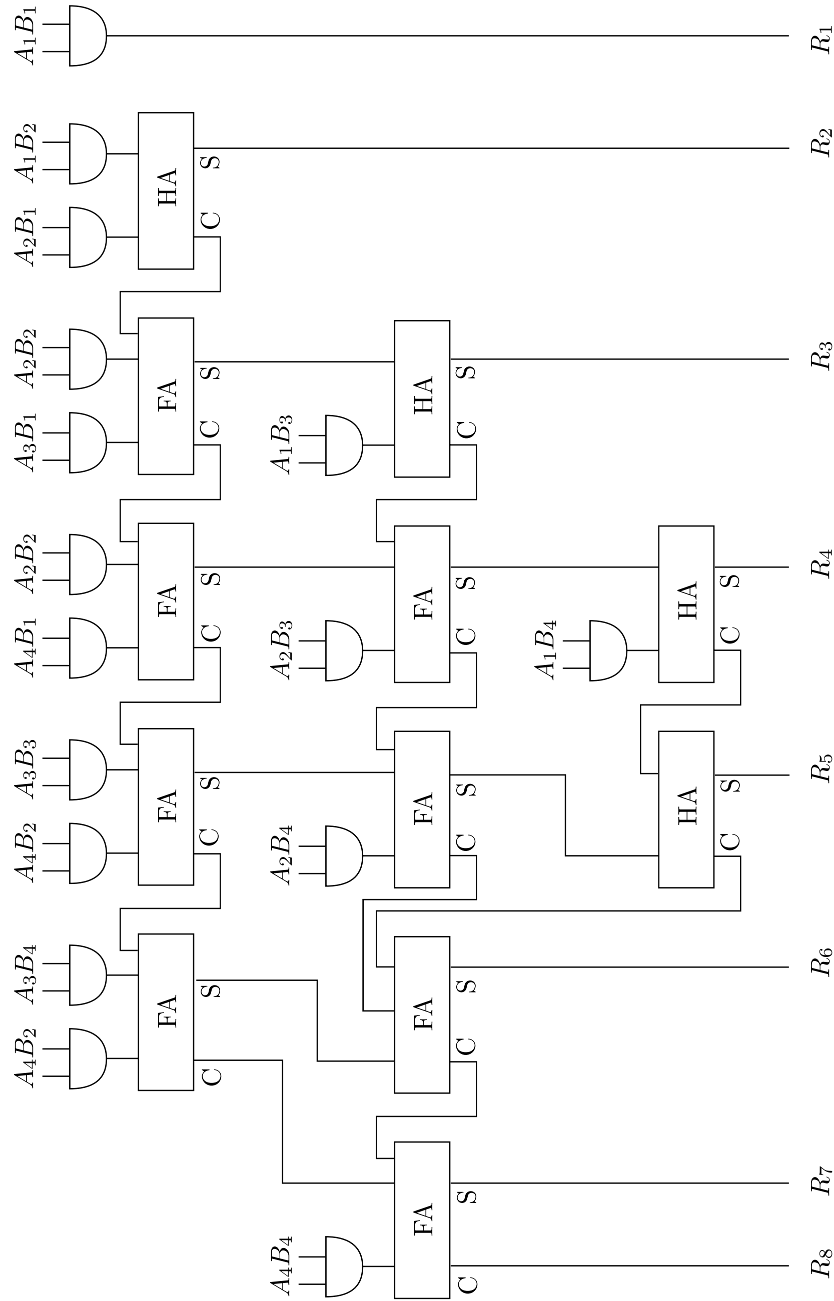



**Figure 9.11:** Simultaneous multiplication of 4-bit integers

Originally, four copies of the multiplicand (shifted relative to each other) had to be added. In the Booth form, only two copies (one of them two's complemented) need be added.

Recoding any (signed) $n$-bit multiplier to the compact form used by the Booth algorithm is done in three simple steps as follows:

Step 1. Denote the bits of the multiplier by $b_{n-1}\dots b_1 b_0$, where $b_{n-1}$ is the sign bit. Add a bit $b_{-1} = 0$ on the right side of the multiplier. For example, 111001111 becomes 111001111|0.

Step 2. Scan the multiplier from $b_0$ on the right to $b_{n-1}$ on the left. Since each $b_i$ (including $b_0$) now has a predecessor, Table 9.12 can be used to determine the bit (0, 1, or $-1$) that recodes $b_i$.

Step 3. Remove the extra bit on the right.

The numbers can now be multiplied by preparing a partial product for each nonzero bit of the multiplier. For a bit of 1, the partial product is a copy of the multiplicand. For a bit of $-1$, the partial product is the two's complement of the multiplicand. The partial products are shifted and added in the usual way.

| Multiplier bit$_i$ | bit$_{i-1}$ | Recoded bit$_i$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | $-1$ |
| 1 | 1 | 0 |

**Table 9.12:** Booth compact multiplier

Here are some examples of recoding signed integers in this way

| Original (signed) | Recoded (unsigned) | Decimal equivalent |
|---|---|---|
| 10000000 → | $\bar{1}$0000000 | $-128 \to -128$, |
| 11110000 → | 000$\bar{1}$0000 | $-16 \to -16$, |
| 01110000 → | 100$\bar{1}$0000 | $112 \to 128 - 16$, |
| 01101110 → | 10$\bar{1}$100$\bar{1}$0 | $110 \to 128 - 32 + 16 - 2$. |

We illustrate this method with two examples of multiplying signed integers. The first example is the two 7-bit positive integers $46 \times 30 = 0101110_2 \times 0011110_2 = 10101100100_2 = 1380$. Table 9.13a shows how these numbers are multiplied "normally." The Booth method is shown in Table 9.13b (notice that there is a carry that should be ignored).

| (a) | (b) | (c) |
|---|---|---|
| 0101110 | 0101110 | |
| 0011110 | 01000$\bar{1}$0 | |
| 0000000 | 00000000000000 | 01100 |
| 0101110 | 1111111010010 (2's compl.) | 0$\bar{1}$1$\bar{1}$0 |
| 0101110 | 000000000000 | 0000000000 |
| 0101110 | 00000000000 | 111110100 |
| 0101110 | 0000000000 | 00001100 |
| 0000000 | 000101110 | 1110100 |
| 0000000 | 00000000 | 000000 |
| 0010101100100 | 1) 00010101100100 | 11110111000 |

**Table 9.13:** Booth multiplication examples

The second example is of 5-bit integers where the multiplier is negative $12\times(-6) = 01100_2\times 11010_2 = 110111000_2 = -72$. The multiplier is recoded as $0\bar{1}1\bar{1}0$ and the process is summarized in Table 9.13c.

It is clear that the Booth algorithm is data dependent. Certain multipliers are better than others and result in fast multiplication. The best Booth multiplier is a number with just one long run of 1's, such as $0011\ldots100\ldots0$. Such a multiplier recodes into just two nonzero bits and therefore takes full advantage of the algorithm.

▸ **Exercise 9.8:** What is the worst multiplier for the Booth algorithm?

The two advantages of the Booth algorithm are (1) both positive and negative multipliers are recoded to the compact representation in the same way and (2) it normally speeds up the multiplication by skipping over 1's. On the other hand, the hardware implementation of this algorithm is complex, so it is normally implemented in microcode.

### 9.3.3 Three-Bit Shift

The principle of the basic integer multiplication method of Section 9.3 is to multiply the multiplicand by each bit of the multiplier to generate the partial products. This results in a simple algorithm, since each partial product is either zero or is the multiplicand. The tradeoff is slow execution, because the algorithm performs an iteration for each bit of the multiplier. The three-bit shift method performs an iteration for each segment of three bits of the multiplier. Since a group of three bits can have values in the range $[0,7]$, each partial product has the form $i\times$multiplicand where $0\le i\le 7$.

The three-bit method starts by preparing the eight quantities $i\times$multiplicand for $i=0,1,\ldots,7$ in a buffer. The multiplier is segmented into groups of three bits, denoted by $\ldots, s_3, s_2, s_1, s_0$. The algorithm then goes into a loop where in iteration $i$ it uses segment $s_i$ as a pointer to the buffer. The number pointed to becomes the $i$th partial product. It is shifted to the left by $2^{3i}$ and is added to the product-so-far. The advantage of this method is a short loop (one-third the number of iterations, compared to the basic algorithm of Section 9.3). The downside is the overhead needed to prepare the eight multiples of the multiplicand. Also, since the eight words of the buffer contain up to $7\times$multiplicand, they have to be three bits wider than the multiplicand. If the multiplicand occupies an $n$-bit word, the buffer must consist of special $n+3$-bit registers.

It is also possible to have a four-bit shift multiplication, and this method can have two versions. The simple version is a straightforward extension of the three-bit method. It uses a buffer where the 16 multiples $i\times$multiplicand for $i=0,1,\ldots,15$ are prestored. Each iteration uses the next four bits of the multiplier as a pointer to the buffer.

The sophisticated version uses just a 9-word buffer where the nine multiples $i\times$multiplicand for $i=0,1,\ldots,8$ are prestored. Each group of four multiplier bits is converted into a partial product that is shifted by $2^{4i}$ and is either added to or subtracted from the product-so-far. If the current segment $s_i$ of four multiplier bits is in the range $[0,7]$, it is used as a pointer to the 9-word buffer and the word pointed to is added (after being shifted) as a partial product. If $s_i$ is in the range $[8,15]$, then the difference $16-s_i$ is first computed. This quantity is in the range $[8,1]$. It is used as a pointer to the buffer, and the word pointed to is shifted and then *subtracted* from the product-so-far. This means that the product-so-far is smaller by $16\times$multiplicand (shifted) from what it should be. To compensate, the next multiplier segment $s_{i+1}$ is incremented by 1. The following (unsigned) example illustrates this version.

We select an arbitrary multiplicand $m$ and the integer

$$\underbrace{11}_{s_4}\ \underbrace{1010}_{s_3}\ \underbrace{1111}_{s_2}\ \underbrace{1100}_{s_1}\ \underbrace{0100}_{s_0} = 241{,}604$$

as our multiplier. The first segment, $s_0$ is 4, so it is used as a pointer to the buffer, and the multiple $4\cdot m$ is added (shifted to the left by $2^0$) to the product-so-far (that's still zero). The second segment $s_1$ is 12, so $16-12=4$ is used as a pointer, the multiple $4\cdot m$ is subtracted (shifted to the left by $2^4$) from the product-so-far, and segment $s_2$ is incremented by 1. This segment is originally 15, so incrementing it causes it to overflow. Its value is wrapped around to become zero, and the next segment, $s_3$ is incremented from 10 to 11. Segment $s_2$ does not change the product-so-far, and segment $s_3$ subtracts $16-11=5$ multiples of the multiplicand from it and increments segment $s_4$ to 4. The last step is to use segment $s_4$ as a pointer

and to increment the product-so-far by $4 \cdot m$. The final value of the product-so-far is

$$(4 \cdot 2^0 - 4 \cdot 2^4 + 0 \cdot 2^8 - 5 \cdot 2^{12} + 4 \cdot 2^{16})m = (4 - 64 + 0 - 20{,}480 + 262{,}144)m = 241{,}604 \cdot m.$$

▸ **Exercise 9.9:** Is it possible to have a two-bit shift multiplication?

## 9.4 Integer Division

Much as subtraction is the opposite of addition, division is the opposite of multiplication. Multiplication is performed by a loop of shifts and additions; division is similarly done by repeated shifts (in the opposite direction) and subtractions. There are, however, three differences between multiplication and division as follows:

1. Division is defined only if the divisor is nonzero.
2. The result of dividing two integers is a quotient and a remainder.
3. The quotient may be too large to fit in a single word or register.

An integer division is denoted by

$$\frac{\text{dividend}}{\text{divisor}} = \text{quotient, remainder} \quad \text{or} \quad \text{dividend} \div \text{divisor} = \text{quotient, remainder}.$$

The algorithm described here assumes that the dividend is stored in the register pair AC and MQ and the divisor is in S (a register or a memory location). When the process is over, the MQ contains the quotient and the AC contains the remainder. The original divisor in S is unchanged. The basic algorithm is

1. If $S = 0$ or $S \le AC$, set error flag and abort.
2. Shift AC and MQ to the left, creating a zero in $MQ_0$.
3. If $S \le AC$, then $AC \leftarrow AC - S$ and $MQ_0 \leftarrow 1$.
4. Repeat steps 2 and 3 $n$ times.

As an example, we assume 4-bit registers and divide the 8-bit integer $66 = 0100\,0010_2$ by $5 = 0101_2$ to obtain a quotient of $13 = 1101_2$ and a remainder of 1. Table 9.14 lists the individual steps. Note that the process stops when steps 2 and 3 of the above algorithm have been repeated four times.

| AC | MQ | S | Description |
|---|---|---|---|
| 0100 | 0010 | 0101 | Initial. $S \neq 0$ and $S > AC$, so proceed. |
| 1000 | 0100 | | 1st shift. $S < AC$, so subtract. |
| 0011 | 0101 | | Set $MQ_0 \leftarrow 1$. |
| 0110 | 1010 | | 2nd shift. $S < AC$, so subtract. |
| 0001 | 1011 | | Set $MQ_0 \leftarrow 1$. |
| 0011 | 0110 | | 3rd shift. $S > AC$. |
| 0110 | 1100 | | 4th shift. $S < AC$, so subtract. |
| 0001 | 1101 | | Set $MQ_0 \leftarrow 1$. |

**Table 9.14:** Dividing two integers

We would like to use our design of double accumulator and MQ registers, and this design has no gates to compare AC to S, so we change our basic method and replace the comparisons by subtractions. The algorithm is modified to become

0. If $S = 0$, abort.
1. $AC \leftarrow AC - S$
2. If $AC \ge 0$, abort.
3. $AC \leftarrow AC + S$. Restore, cancel step 1.
4. Left shift AC and MQ, creating a zero in $MQ_0$.
5. $AC \leftarrow AC - S$.
6. If $AC \ge 0$, set $MQ_0 \leftarrow 1$, else restore $AC \leftarrow AC + S$.

7. `Repeat steps 4 through 6` $n$ `times.`

Step 0 can be omitted because we assume an unsigned dividend. If step 0 is omitted and S is zero, then step 1 would be AC $\leftarrow$ AC $-$ 0, and in step 2 the AC would be nonnegative and the algorithm would abort.

In our design for the accumulator, subtraction is done by the two signals AL $\leftarrow$ AU $+\bar{\text{S}}$ and UP. Therefore, steps 1 and 5 in the algorithm above become AL $\leftarrow$ AU $+\bar{\text{S}}$ and steps 3 and 6 (AC $\leftarrow$ AC $+$ S) can be eliminated. There is no need to restore, since the original AU is not affected. If the subtraction is to become permanent (i.e., if the "then" part of step 6 is executed), then an UP shift would set AU to the result of the subtraction. When these changes are incorporated into the algorithm above, we realize that S is no longer added to anything, just subtracted from the AC. We therefore prepare the 2's complement of S in a temporary register T. Thus, the final version of our algorithm is

1. $\text{AL} \leftarrow \overline{\text{S}}$ `(2's complement).`
2. $\text{T} \leftarrow \text{AL}$.
3. $\text{AL} \leftarrow \text{AU} + \text{T}$. `(replaces` $\text{AC} \leftarrow \text{AC} - \text{S}$`).`
4. `If` $\text{AL}_\text{n} = 0$`, abort (replaces previous step 2).`
5. `Shift AC and MQ down (replaces previous step 4).`
6. `Shift AC and MQ left and up (also replaces previous step 4).`
7. $\text{AL} \leftarrow \text{AU} + \text{T}$ `(replaces previous step 5).`
8. `If` $\text{AL}_\text{n} = 0$`, shift AC up and set` $\text{MQ}_0 \leftarrow 1$ `(replaces previous step 6).`
9. `Repeat steps 5 through 8` $n$ `times.`

Based on this algorithm, an integer division circuit can be designed with the inputs and outputs shown in Figure 9.15a. Steps 1–4 are executed once. Steps 5–8 are executed $n$ times each, and one more step is needed, to send a "done" signal at the end. The total number of steps needed for the division is therefore $4 + 4n + 1 = 4n + 5$, an odd number. Instead of constructing a counter modulo an odd number, we use an $L$-stage counter, where $L$ is determined by $2^{L-1} < 4n + 5 < 2^L$. An AND gate is connected to the outputs of those stages that go high after $4n + 5$ counts, and the output of this gate becomes the "done" signal. As an example, consider $n = 4$. The value of $4n + 5$ is 21, implying $L = 5$. After 21 counts, the 5-stage counter outputs 20, or $10100_2$, so the AND gate should be connected as shown in Figure 9.15b. Table 9.16 is the truth table of the general integer division circuit.

We already know how the "done" signal is generated by the division circuit. An examination of Table 9.16 tells how the remaining eight output signals depend on the counter outputs $C_i$. We first define the auxiliary quantity $X = C_3 + C_4 + \cdots + C_L$. The eight output signals are:

$$
\begin{aligned}
\text{COMPL} &= \bar{C}_1\bar{C}_2\ldots\bar{C}_L = \bar{C}_1\bar{C}_2(\overline{C_3 + \cdots + C_L}) = \bar{C}_1\bar{C}_2\bar{X},\\
\text{TAL} &= C_1\bar{C}_2\ldots\bar{C}_L = C_1\bar{C}_2\bar{X},\\
\text{ADD} &= \bar{C}_1C_2 \text{ and any } C_3\ldots C_L = \bar{C}_1C_2,\\
\text{abort} &= C_1C_2\bar{C}_3\ldots\bar{C}_L\overline{\text{AL}}_n = C_1C_2\bar{X}\overline{\text{AL}}_n,\\
\text{SD} &= \bar{C}_1\bar{C}_2(C_3 + C_4 + \cdots + C_L)\overline{\text{done}} = \bar{C}_1\bar{C}_2X\,\overline{\text{done}},\\
\text{SL} &= C_1\bar{C}_2(C_3 + \cdots + C_L) = C_1\bar{C}_2X,\\
\text{SU}, \text{MU01} &= C_1C_2\overline{\text{AL}}_n.
\end{aligned}
$$

The last point to be mentioned is how the counter is stopped. This is shown in Figure 9.15c. The counter is clocked by the clock pulse $\phi$, and is stopped when either "done" or "abort" are generated by the division circuit. These signals are sent back to the control unit, to indicate the state of the division, but they are also used to stop the counter when the division is complete.

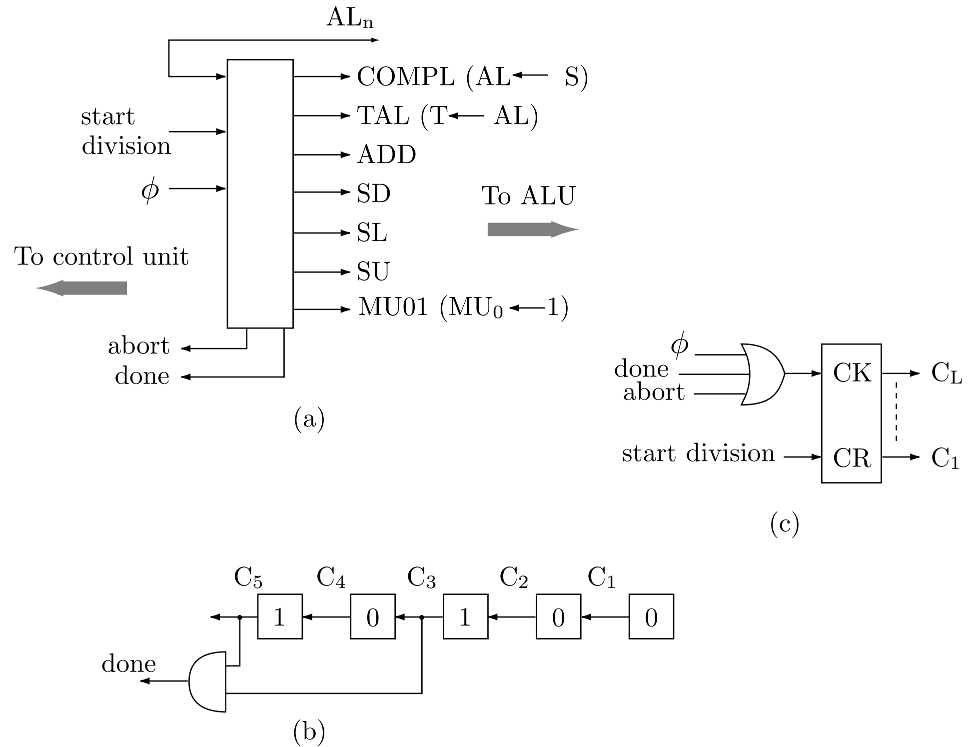



**Figure 9.15:** Integer division

| $C_1$ | $C_2$ | $C_3$ | $C_4$ | $\dots$ | $C_L$ | output signal |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | $\cdots$ | 0 | COMPL |
| 1 | 0 | 0 | 0 | | 0 | TAL |
| 0 | 1 | 0 | 0 | | 0 | ADD |
| 1 | 1 | 0 | 0 | | 0 | if $AL_n = 0$, abort |
| | | | | | | 1st iteration |
| 0 | 0 | 1 | 0 | | 0 | SD |
| 1 | 0 | 1 | 0 | | 0 | SL |
| 0 | 1 | 1 | 0 | | 0 | ADD |
| 1 | 1 | 1 | 0 | | 0 | if $AL_n = 0$, then SU and MU01 |
| | | | | | | 2nd iteration |
| 0 | 0 | 0 | 1 | | 0 | SD |
| 1 | 0 | 0 | 1 | | 0 | SL |
| 0 | 1 | 0 | 1 | | 0 | ADD |
| 1 | 1 | 0 | 1 | | 0 | if $AL_n = 0$, then SU and MU01 |
| | | | | | | $n$th iteration |
| 0 | 0 | $x$ | $x$ | | $x$ | SD |
| 1 | 0 | $x$ | $x$ | | $x$ | SL |
| 0 | 1 | $x$ | $x$ | | $x$ | ADD |
| 1 | 1 | $x$ | $x$ | | $x$ | if $AL_n = 0$, then SU and MU01 |
| 0 | 0 | $x$ | $x$ | $\cdots$ | $x$ | done |

**Table 9.16:** Outputs of division circuit

## 9.5 Shifts

The shift operation, with its various modes and applications, is discussed in Section 2.13.3. The ALU may have one or more shift registers in order to implement the different types of shifts. This section shows two designs for general shift registers, one using D flipflops and the other using JK flipflops. A shift register may be unidirectional or bidirectional. It may have serial or parallel inputs (or both). In a shift register with serial input, the bits to be shifted are input one by one. Each time a bit is shifted into one end, the register shifts its content, which moves one bit out of the other end. Such a register should be cleared before it is used. In contrast, parallel input is fed into the shift register in one step, following which the register is shifted by the desired amount. A shift register may also have serial or parallel outputs and it may also have both.

Figure 9.17a shows a serial-serial (i.e., serial input and serial output) shift register based on D flipflops. This register shifts because the output Q of a D flipflop is set to the input D on a clock pulse (in the diagram, they are set on the trailing edge of the clock pulse). The register has serial output, but parallel output can be obtained "for free" just by connecting lines to the various Q outputs. The figure also shows how parallel input can be added to this shift register by using flipflops with a "preset" input.

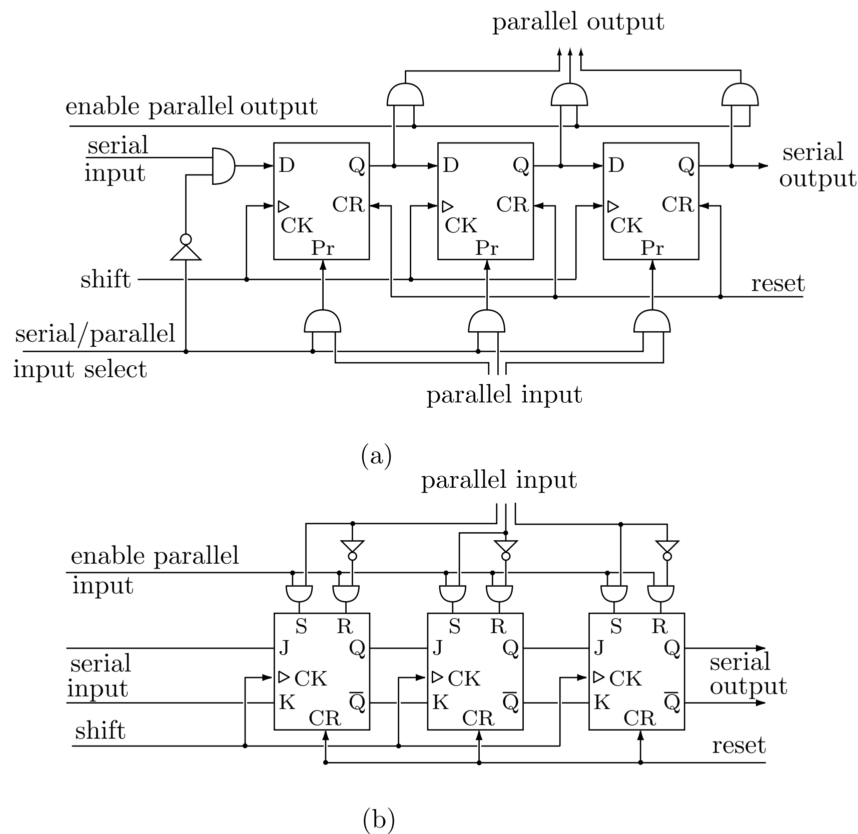
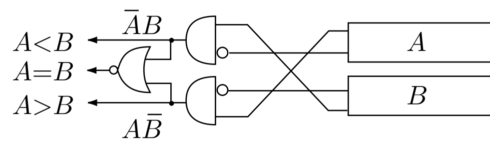


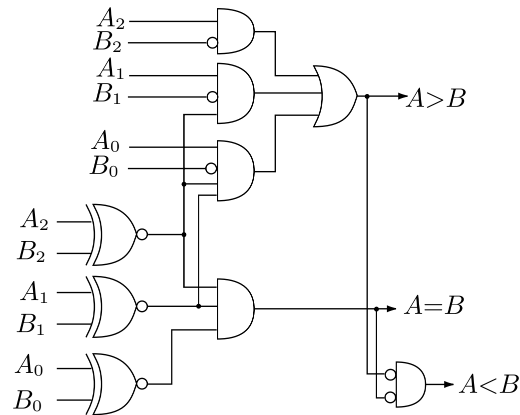

**Figure 9.17:** Shift registers

Figure 9.17b illustrates a similar shift register based on JK flipflops connected in a master-slave configuration. These flipflops have synchronous JK inputs and asynchronous SR inputs. The former perform the shifts and the latter are used for the parallel input.

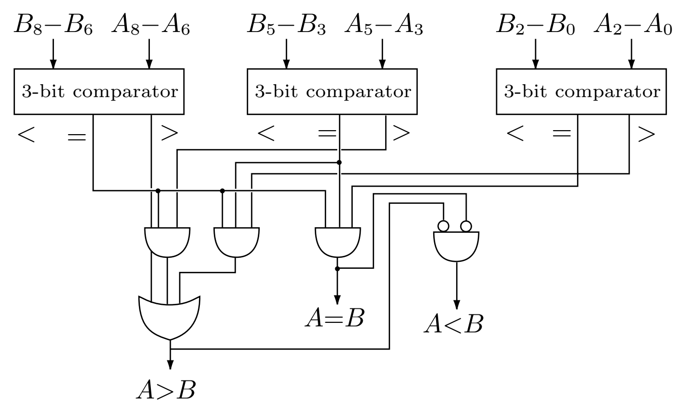
These diagrams also suggest that a typical shift register can easily be extended to perform circular shifts by feeding the serial output into the serial input of the register.




(a)




(b)




(c)

**Figure 9.18:** Comparing integers

## 9.6 Comparisons

Section 2.11 discusses integer comparisons. Two integers can be compared either by subtracting them and checking the sign of the difference (but subtraction can generate overflow) or by comparing pairs of bits from left to right, looking for the first pair that differs. The result of a comparison is one of the three relations ">", "=", and "<". This section describes simple designs for a serial and a parallel comparators.

Figure 9.18a shows a serial comparator. The two numbers to be compared are first moved to the temporary registers $A$ and $B$. These registers are rotated to the left bit by bit, and pairs of bits are sent to the comparator from the left ends of $A$ and $B$. Eventually, one of the three output lines goes high. This line should set the correct status flags and also stop the comparison.

A parallel comparator is more complex, since it requires enough gates to compare all bit pairs in parallel. It also uses high fan-in gates, which makes it impractical for large numbers. Figure 9.18b shows a parallel comparator for 3-bit integers. The design is straightforward. Comparing larger numbers can be done in two stages. The first stage uses several parallel comparators, each comparing 3-4 consecutive bit pairs. The second stage uses the results of the first stage to decide on the relation between the numbers. Figure 9.18c shows such a comparator for 9-bit integers. The first stage consists of three 3-bit comparators (like the one shown in Figure 9.18b) and the second stage has gates with fan-in of up to three.

### 9.6.1 Summary

This short chapter discusses a few basic ALU operations. In addition to those operations, the ALU of a modern computer has circuits to perform floating-point arithmetic, BCD arithmetic, certain logical operations, and several types of shifts. The floating-point arithmetic operations are described in Section 2.21. They are by far the most complex operations implemented in the ALU. In fact, until the 1990s, most ALUs did not include circuits for the floating-point operations and a computer user had to either obtain a special coprocessor (so called "math chip") for these operations or implement them in software.

When a child, I should have thought it almost as much of a
disgrace to spell a word wrong, or make a mistake in the
multiplication table, as to break one of the Ten Commandments.

— Lucy Larcom, *A New England Girlhood*

# 10 Assemblers

Assemblers are not very popular nowadays, since higher-level languages are so much easier to use, and result in fast code. Still, a person interested in computers should know something about assemblers because of the following reasons:

- The concepts used by assemblers (translation, execution of directives, macros) are general and are commonly used by other software systems.

- Since assemblers deal with machine instructions, knowledge of assemblers leads to understanding of machine instructions, and thus to an understanding of computer organization.

## 10.1 Introduction

A work on assemblers should naturally start with a definition. However, computer science is not as precise a field as mathematics, so most definitions are not rigorous. The definition this author likes best is:

> An assembler is a translator that translates source instructions (in symbolic language) into target instructions (in machine language), on a one to one basis.

This means that each source instruction is translated into *exactly* one target instruction.

This definition has the advantage of clearly describing the translation process of an assembler. It is not a precise definition, however, because an assembler can do (and usually does) much more than just translation. It offers a lot of help to the programmer in many aspects of writing the program. The many types of help offered by the assembler are grouped under the general term *directives* (or *pseudo-instructions*).

Another good definition of assemblers is:

> An assembler is a *translator* that translates a machine-oriented language into machine language.

This definition distinguishes between assemblers and compilers. Compilers being translators of problem-oriented languages or of machine-independent languages. This definition, however, says nothing about the one-to-one nature of the translation, and thus ignores a most important operating feature of an assembler.

One reason for studying assemblers is that the operation of an assembler reflects the architecture of the computer. The assembler language depends heavily on the internal organization of the computer. Architectural features such as memory word size, number formats, internal character codes, index registers, and general purpose registers, affect the way assembler instructions are written and the way the assembler handles instructions and directives. This fact explains why there is an interest in assemblers today and why a course on assembler language is still required for many, perhaps even most, computer science degrees.

The first assemblers were simple *assemble-go* systems. All they could do was to assemble code directly in memory and start execution. It was quickly realized, however, that *linking* is an important feature, required even by simple programs. The pioneers of programming have developed the concept of the *routine library* very early, and they needed assemblers that could locate library routines, load them into memory, and link them to the main program. It is from this task of locating, loading, and linking—of assembling a single working program from individual pieces—that the name *assembler* originated. Today, assemblers are translators and they work on one program at a time. The tasks of locating, loading, and linking (as well as many other tasks) are performed by a loader.

A modern assembler has two inputs and two outputs. The first input is short, typically a single line typed at a keyboard. It activates the assembler and specifies the name of a source file (the file containing the source code to be assembled). It may contain other information that the assembler should have before it starts. This includes commands and specifications such as:

- The names of the object file and listing file.
- Display (or do not display) the listing on the screen while it is being generated.
- Display all error messages but do not stop for any error.
- Save the listing file and do not print it (see below).
- This program does not use macros.
- The symbol table is larger (or smaller) than usual and needs a certain amount of memory.

All these terms are explained elsewhere. An example is the command line that invokes MACRO, the VAX assembler. The line

```
MACRO /SHOW=MEB /LIST /DEBUG ABC
```

activates the assembler, tells it that the source program name is `abc.mar` (the `.mar` extension is implied), that binary lines in macro expansions should be listed (shown), that a listing file should be created, and that the debugger should be included in the assembly.

Another typical example is the following command line that invokes the Microsoft Macro assembler (MASM) for the 80x86 microprocessors.

```
MASM /d /Dopt=5 /MU /V
```

It tells the assembler to create a pass 1 listing (`/D`), to create a variable `opt` and set its value to 5, to convert all letters read from the source file to upper case (`MU`), and to include certain information in the listing file (the `V`, or verbose, option).

The second input is the *source file*. It includes the symbolic instructions and directives. The assembler translates each symbolic instruction into one machine instruction. The directives, however, are not translated. The directives are our way of asking the assembler for help. The assembler provides the help by executing (rather than translating) the directives. A modern assembler can support as many as a hundred directives. They range from `ORG`, which is very simple to execute, to `MACRO`, which can be very complex.

The first and most important output of the assembler is the *object file*. It contains the assembled instructions (the machine language program) to be loaded later into memory and executed. The object file is an important component of the assembler-loader system. It makes it possible to assemble a program once, and later load and run it often. It also provides a natural place for the assembler to leave information to the loader, instructing the loader in several aspects of loading the program. This information is called *loader directives*. Note, however, that the object file is optional. The user may specify no object file, in which case the assembler generates only a listing.

The second output of the assembler is the *listing file*. For each line in the source file, a line is created in the listing file, containing:

- The Location Counter (page 275).
- The source line itself.
- The machine instruction (if the source line is an instruction), or some other relevant information (if the source line is a directive).

The listing file is generated by the assembler, sent to the printer, gets printed, and is then discarded. The user, however, can specify either not to generate a listing file or not to print it. There are also directives that control the listing. They can be used to suppress parts of the listing, to print page headers, or to control the printing of macro expansions.

The cross-reference information is normally a part of the listing file, although the MASM assembler creates it in a separate file and uses a special utility to print it. The cross-reference is a list of all symbols used in the program. For each symbol, the point where it is defined and all the places where it is used, are listed.

▸ **Exercise 10.1:** Why would anyone want to suppress the listing file or not to print it?

As mentioned earlier, the first assemblers were assemble-go type systems. They did not generate any object file. Their main output was machine instructions loaded directly into memory. Their secondary output was a listing. Such assemblers are also in use today (for reasons explained in Section 10.6) and are called one-pass assemblers. In principle, a one pass assembler can produce an object file, but such a file would be absolute and its use is limited.

Most current assemblers are of the *two-pass* variety. They generate an object file that is relocatable and can be linked and loaded by a loader.

A loader, as the name implies, is a program that loads programs into memory. Modern loaders, however, do much more than that. Their main tasks are loading, relocating, linking and starting the program. In a typical run, a modern linking-loader can read several object files, load them one by one into memory, relocating each as it is being loaded, link all the separate object files into one executable module, and start execution at the right point. Using such a loader has several advantages (Section 10.7), the most important being the ability to write and assemble a program in several, separate, parts.

Writing a large program in several parts is advantageous, for reasons that will be briefly mentioned but not fully discussed here. The individual parts can be written by different programmers (or teams of programmers), each concentrating on his own part. The different parts can be written in different languages. It is common to write the main program in a higher-level language and the procedures in assembler language. The individual parts are assembled (or compiled) separately, and separate object files are produced. The assembler or compiler can only see one part at a time and does not see the whole picture. It is only the loader that loads the separate parts and combines them into a single program. Thus when a program is assembled, the assembler does not know whether this is a complete program or just a part of a larger program. It therefore assumes that the program will start at address zero and assembles it based on that assumption. Before the loader loads the program, it determines its true start address, based on the memory areas available at that moment and on the previously loaded object files. The loader then loads the program, making sure that all instructions fit properly in their memory locations. This process involves adjusting memory addresses in the program, and is called *relocation*.

Since the assembler works on one program at a time, it cannot link individual programs. When it assembles a source file containing a main program, the assembler knows nothing about the existence of any other source files containing, perhaps, procedures called by the main program. As a result, the assembler may not be able to properly assemble a procedure call instruction (to an external procedure) in the main program. The object file of the main program will, in such a case, have missing parts (holes or gaps) that the assembler cannot fill. The loader has access to all the object files that make up the entire program. It can see the whole picture, and one of its tasks is to fill up any missing parts in the object files. This task is called *linking*.

The task of preparing a source program for execution includes translation (assembling or compiling), loading, relocating, and linking. It is divided between the assembler (or compiler) and the loader, and dual assembler-loader systems are very common. The main exception to this arrangement is *interpretation*. Interpretive languages such as `BASIC` or `APL` use the services of one program, the interpreter, for their execution, and do not require an assembler or a loader. It should be clear from the earlier discussion that the main reason for keeping the assembler and loader separate is the need to develop programs (especially large ones) in separate parts. The detailed reasons for this will not be discussed here. We will, however, point out the advantages of having a dual assembler-loader system. They are listed below, in order of importance.

- It makes it possible to write programs in separate parts that may also be in different languages.

- It keeps the assembler small. This is an important advantage. The size of the assembler depends on the size of its internal tables (especially the symbol table and the macro definition table). An assembler designed to assemble large programs is large because of its large tables. Separate assembly makes it possible to assemble very large programs with a small assembler.

- When a change is made in the source code, only the modified program needs to be reassembled. This property is a benefit if one assumes that assembly is slow and loading is fast. Often, however, loading is slower than assembling, and this property is just a feature, not an advantage, of a dual assembler-loader system.

- The loader automatically loads routines from a library. This is considered by some an advantage of a dual assembler-loader system but, actually, it is not. It could easily be done in a single assembler-loader program. In such a program, the library would have to contain the source code of the routines, but this is typically not larger than the object code.

## 10.2 A Short History of Assemblers

One of the first stored program computers was the EDSAC (Electronic Delay Storage Automatic Calculator) developed at Cambridge University in 1949 by Maurice Wilkes and W. Renwick. From its very first days the EDSAC had an assembler, called *Initial Orders*. It was implemented in a read-only memory formed from a set of rotary telephone selectors, and it accepted symbolic instructions. Each instruction consisted of a one letter mnemonic, a decimal address, and a third field that was a letter. The third field caused one of 12 constants preset by the programmer to be added to the address at assembly time.

It is interesting to note that Wilkes was also the first to propose the use of labels (which he called *floating addresses*), the first to use an early form of macros (which he called *synthetic orders*), and the first to develop a subroutine library.

The IBM 650 computer was delivered around 1953 and had an assembler very similar to present day assemblers. SOAP (Symbolic Optimizer and Assembly Program) did symbolic assembly in the conventional way, and was perhaps the first assembler to do so. However, its main feature was the optimized calculation of the address of the next instruction. The IBM 650 (a decimal computer, incidentally), was based on a magnetic drum memory and the program was stored in that memory. Each instruction had to be fetched from the drum and had to contain the address of its successor. For maximum speed, an instruction had to be placed on the drum in a location that would be under the read head as soon as its predecessor was completed. SOAP calculated those addresses, based on the execution times of the individual instructions.

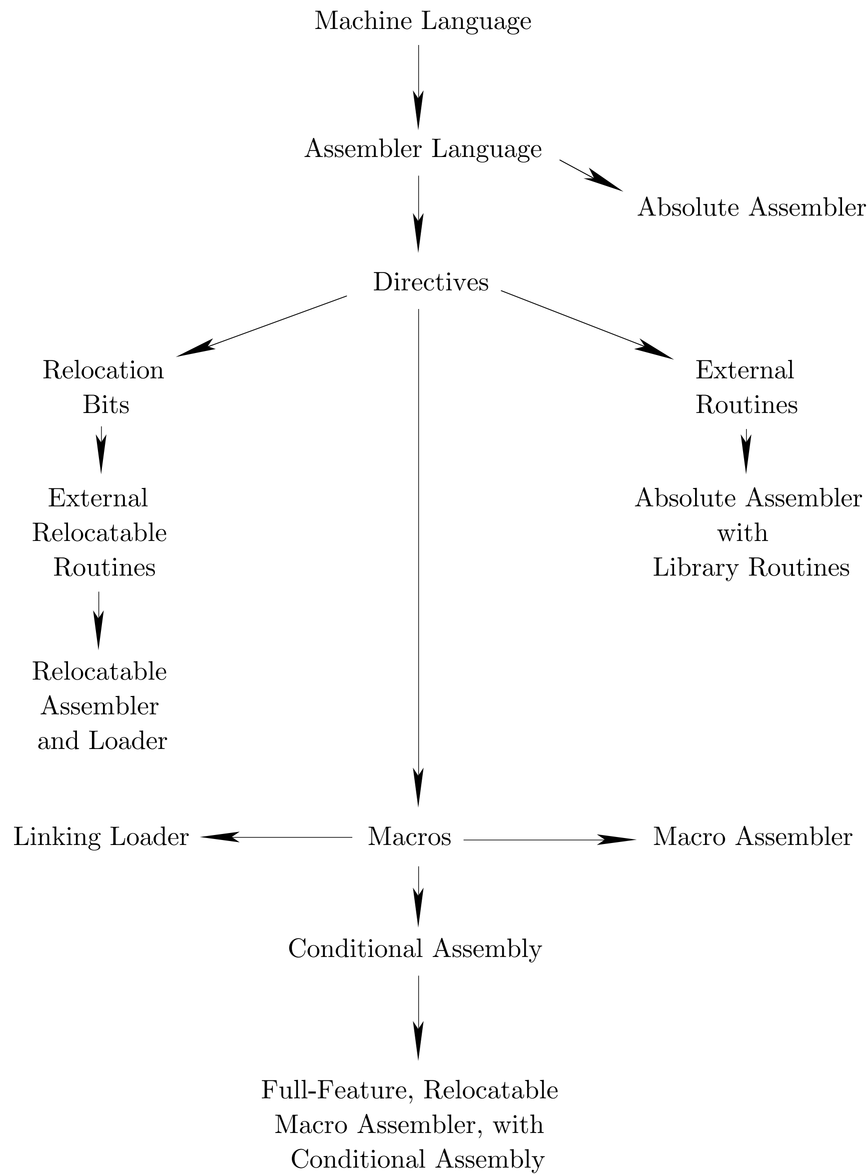
One of the first commercially successful computers was the IBM 704. It had features such as floating-point hardware and index registers. It was first delivered in 1956 and its first assembler, the UASAP-1, was written in the same year by Roy Nutt of United Aircraft Corp. (hence the name UASAP—United Aircraft Symbolic Assembly Program). It was a simple binary assembler, did practically nothing but one-to-one translation, and left the programmer in complete control over the program. SHARE, the IBM users' organization, adopted a later version of that assembler and distributed it to its members together with routines produced and contributed by members. UASAP has pointed the way to early assembler writers, and many of its design principles are used by assemblers to this day. The UASAP was later modified to support macros.

In the same year another assembler, the IBM Autocoder was developed by R. Goldfinger for use on the IBM 702/705 computers. This assembler (actually several different Autocoder assemblers) was apparently the first to use macros. The Autocoder assemblers were used extensively and were eventually developed into large systems with large macro libraries used by many installations.

Another pioneering early assembler was the UNISAP, for the UNIVAC I & II computers, developed in 1958 by M. E. Conway. It was a one-and-a-half pass assembler, and was the first one to use local labels (Section 10.9).

By the late fifties, IBM had released the 7000 series of computers. These came with a macro assembler, SCAT, that had all the features of modern assemblers. It had many directives (*pseudo instructions* in the IBM terminology), an extensive macro facility, and it generated relocatable object files.

The SCAT assembler (Symbolic Coder And Translator) was originally written for the IBM 709 and was modified to work on the IBM 7090. The GAS (Generalized Assembly System) assembler was another powerful 7090 assembler.




**Phases in the historical development of assemblers and loaders.**

The idea of macros originated with several people. M. D. McIlroy was probably the first to propose the modern form of macros and the idea of conditional assembly. He implemented these ideas in the GAS assembler mentioned earlier.

One of the first full-feature loaders, the linking loader for the IBM 704–709–7090 computers, is an example of an early loader supporting both relocation and linking.

The earliest discussion of meta-assemblers seems to be by Ferguson. The idea of high-level assemblers originated with Wirth and had been extended, a few years later, by an anonymous software designer at NCR, who proposed the main ideas of the NEAT/3 language.

The diagram summarizes the main phases in the historical development of assemblers and loaders.

## 10.3 Types of Assemblers and Loaders

- A One-pass Assembler: One that performs all its functions by reading the source file once.

- A Two-Pass Assembler: One that reads the source file twice.

- A Resident Assembler: One that is permanently loaded in memory. Typically such an assembler resides in ROM, is very simple (supports only a few directives and no macros), and is a one-pass assembler. The above assemblers are described below.

- A Macro-Assembler: One that supports macros (Chapter 11).

- A Cross-Assembler: An assembler that runs on one computer and assembles programs for another. Many cross-assemblers are written in a higher-level language to make them portable. They run on a large machine and produce object code for a small machine.

- A Meta-Assembler: One that can handle many different instruction sets.

- A Disassembler: This, in a sense, is the opposite of an assembler. It translates machine code into a source program in assembler language.

- A high-level assembler. This is a translator for a language combining the features of a higher-level language with some features of assembler language. Such a language can also be considered a machine dependent higher-level language.

- A Micro-Assembler: Used to assemble microinstructions. It is not different in principle from an assembler. Note that microinstructions have nothing to do with programming microcomputers.

Combinations of those types are common. An assembler can be a Macro Cross-Assembler or a Micro Resident one.

- A Bootstrap Loader: It uses its first few instructions to either load the rest of itself, or load another loader, into memory. It is typically stored in ROM.

- An Absolute Loader: Can only load absolute object files, i.e., can only load a program starting from a certain, fixed location in memory.

- A Relocating Loader: Can load relocatable object files and thus can load the same program starting at any location.

- A Linking Loader: Can link programs that were assembled separately, and load them as a single module.

- A Linkage Editor: Links programs and does some relocation. Produces a load module that can later be loaded by a simple relocating loader.

## 10.4 Assembler Operation

The basic principles of assembler operation are simple, involving just one problem, *unresolved references*. This is a simple problem that has two simple solutions. The problem is important, however, since its two solutions introduce, in a natural way, the two main types of assemblers namely, the *one-pass* and the *two-pass*.

### 10.4.1 The Source Line

A typical source line has four fields, a label (or a location), a mnemonic (or operation), an operand, and a comment.

example `LOOP ADD R1,ABC PRODUCING THE SUM`

In this example, `LOOP` is a label, `ADD` is a mnemonic meaning to add, `R1` stands for register 1, and `ABC` is the label of another source line. `R1` and `ABC` are two operands that make up the operand field. The example above is, therefore, a double-operand instruction. When a label is used as an operand, we call it a symbol. Thus, in our case, `ABC` is a symbol.

The comment is for the programmer's use only. It is read by the assembler, it is listed in the listing file, and is otherwise ignored.

The label field is only necessary if the instruction is referred to from some other point in the program. It may be referred to by another instruction in the same program, by another instruction in a different program (the two programs should eventually be linked), or by itself.

The word *mnemonic* comes from the Greek $\mu\nu\epsilon\mu o\nu\iota\kappa o\sigma$, meaning *pertaining to memory*; it is a memory aid. The mnemonic is always necessary. It is the operation. It tells the assembler what instruction needs to be assembled or what directive to execute (but see the comment below about blank lines).

The operand depends on the mnemonic. Instructions can have zero, one, or two operands (very few computers have also three operand instructions). Directives also have operands. The operands supply information to the assembler about the source line.

As a result, only the mnemonic is mandatory, but there are even exceptions to this rule. One exception is blank lines. Many assemblers allow blank lines—in which all fields, including the mnemonic, are missing—in the source file. They make the listing more readable but are otherwise ignored.

Another exception is comment lines. A line that starts with a special symbol (typically a semicolon, sometimes an asterisk and, in a few cases, a slash) is considered a comment line and, of course, has no mnemonic. Many modern assemblers support a `COMMENT` directive with the following form:

`COMMENT` *delimiter* text *delimiter*

Where the text between the delimiters is a comment. This way the programmer can enter a long comment, spread over many lines, without having to start each line with the special comment symbol. Example:

```
 COMMENT =This is a long
comment that ...
  .
  .
... sufficient to describe what you want=
```

Old assemblers were developed on punched-card based computers. They required the instructions to be punched on cards such that each field of the source line was punched in a certain field on the card. The following is an example from the IBMAP (Macro Assembler Program) assembler for the IBM 7040. A source line in this assembler has to be punched on a card with the format:

| Columns | Field |
|---|---|
| 1-6 | Label |
| 7 | Blank |
| 8- | Mnemonic |

and also obey the following rules:

- The operand must be separated from the mnemonic by at least one blank, and must start on or before column 16.

- The comment must be separated from the operand by at least one blank. If there is no operand, the comment may not start before column 17.

- The comment extends through column 80 but columns 73–80 are normally used for sequencing and identification.

It is obviously very hard to enter such source lines from a keyboard. Modern assemblers are thus more flexible and do not require any special format. If a label exists, it must end with a ':'. Otherwise, the individual fields should be separated by at least one space (or by a tab character), and subfields should be separated by either a comma or parentheses. This rule makes it convenient to enter source lines from a keyboard, but is ambiguous in the case of a source line that has a comment but no operand.

example: `EI   ;ENABLE ALL INTERRUPTS`

The semicolon guarantees that the word `ENABLE` will not be considered an operand by the assembler. This is why many assemblers require that comments start with a semicolon.

▸ **Exercise 10.2:** Why a semicolon and not some other character such as '$' or '@' ?

Many modern assemblers allow labels without an identifying ':'. They simply have to work harder in order to identify labels.

The instruction sets of some computers are designed such that the mnemonic specifies more than just the operation. It may also contain part of the operand. The Signetics 2650 microprocessor, for example, has many mnemonics that include one of the operands. A 'Store Relative' instruction on the 2650 may be written '`STRR,R0 SAV`'; the mnemonic field includes `R0` (the register to be stored in location `SAV`), which is an operand.

On other computers, the operation may partly be specified in the operand field. The instruction '`IX7 X2+X5`', on the CDC Cyber computers means: "add register `X2` and register `X5` as integers, and store the sum in register `X7`." The operation appears partly in the operation field ('I') and partly in the operand field ('+'), whereas `X7` (an operand) appears in the mnemonic. This makes it harder for the assembler to identify the operation and the operands and, as a result, such instruction formats are not common.

▸ **Exercise 10.3:** What is the meaning of the Cyber instruction `FX7 X2+X5`?

To translate an instruction, the assembler uses the opcode table, which is a static data structure. The two important columns in the table are the mnemonic and opcode. Table 10.1 is an example of a simple opcode table. It is part of the IBM 360 opcode table and it includes other information.

| mnemonic | opcode | type | length |
|---|---|---|---|
| `A` | 5A | RX | 4 |
| `AD` | 6A | RX | 4 |
| `ADR` | 2A | RR | 2 |
| `AER` | 3A | RR | 2 |
| `AE` | 1A | RR | 2 |

**Table 10.1**

The mnemonics are from one to four letters long (in many assemblers they may include digits). The opcodes are two hexadecimal digits (8 bits) long, and the types (which are irrelevant for now) provide more information to the assembler.

The opcode table should allow for a quick search. For each source line input, the assembler has to search the opcode table. If it finds the mnemonic, it uses the opcode to start assembling the instruction. It also uses the other information in the opcode table to complete the assembly. If it does not find the mnemonic in the table, it assumes that the mnemonic is that of a directive and proceeds accordingly.

The opcode table thus provides for an easy first step of assembling an instruction. The next step is using the operand to complete the assembly. The opcode table should contain information about the number and types of operands for each instruction. In Table 10.1 above, the *type* column provides this information. Type `RR` means a Register-Register instruction. This is an instruction with two operands, both registers. The assembler expects two operands, both numbers between 0 and 15 (the IBM 360 has 16 general-purpose registers). Each register number is assembled as a 4-bit field.

▸ **Exercise 10.4:** Why does the IBM 360 have 16 general purpose registers and not a round number such as 15 or 20?

**Example:** The instruction `AR 4,6` means: add register 6 (the source) to register 4 (the destination operand). It is assembled as the 16-bit machine instruction `1A46`, in which `1A` is the opcode and `46`, the two operands.

Type `RX` stands for Register-indeX. In these instructions the operand consists of a register followed by an address.

**Example:** `BAL 5,14`. This instruction calls a procedure at location 14, and saves the return address in register 5 (`BAL` stands for Branch And Link). It is assembled as the 32-bit machine instruction `4550000E` in which `00E` is a 12-bit address field (`E` is hexadecimal 14), `45` is the opcode, `5` is register 5, and the two zeros in the middle are irrelevant to our discussion. (A note to readers familiar with the IBM 360—This example ignores base registers as they do not contribute anything to our discussion of assemblers.)

▸ **Exercise 10.5:** What are the two zeros in the middle of the instruction used for?

This example is atypical. Numeric addresses are rarely used in assembler programming, since keeping track of their values is a tedious task better left to the assembler. In practice, symbols are used instead of numeric addresses. Thus the above example is likely to be written as `BAL 5,XYZ`, where `XYZ` is a symbol whose value is an address. Symbol `XYZ` should be the label of some source line. Typically the program will contain the two lines

```
XYZ   A     4,ABC   ;THE SUBROUTINE STARTS HERE
      .
      .
      BAL   5,XYZ   ;THE SUBROUTINE IS CALLED
```

Besides the basic task of assembling instructions, the assembler offers many services to the user, the most important of which is handling symbols. This task consists of two different parts, defining symbols, and using them. A symbol is defined by writing it as a label. The symbol is used by writing it in the operand field of a source line. A symbol can only be defined once but it can be used any number of times. To understand how a value is assigned to a symbol, consider the example above. The 'add' instruction `A` is assembled and is eventually loaded into memory as part of the program. The value of symbol `XYZ` is the memory address of that instruction. This means that the assembler has to keep track of the addresses where instructions are loaded, since some of them will become values of symbols. To do this, the assembler uses two tools, the *location counter* (LC), and the *symbol table*.

The LC is a variable, maintained by the assembler, that contains the address into which the current instruction will eventually be loaded. When the assembler starts, it clears the LC, assuming that the first instruction will go into location 0. After each instruction is assembled, the assembler increments the LC by the size of the instruction (the size in words, not in bits). Thus the LC always contains the current address. Note that the assembler does not load the instructions into memory. It writes them on the object file, to be eventually loaded into memory by the loader. The LC, therefore, does not point to the current instruction. It just shows where the instruction will eventually be loaded. When the source line has a label (a newly defined symbol), the label is assigned the current value of the LC as its value. Both the label and its value (plus some other information) are then placed in the symbol table.

The symbol table is an internal, dynamic table that is generated, maintained, and used by the assembler. Each entry in the table contains the definition of a symbol and has fields for the name, value, and type of the symbol. Some symbol tables contain other information about the symbols. The symbol table starts empty, labels are entered into it as their definitions are found in the source, and the table is also searched frequently to find the values and types of symbols whose names are known. Various ways to implement symbol tables are discussed in Section 10.14.

In the above example, when the assembler encounters the line

```
XYZ A 5,ABC ;THE SUBROUTINE STARTS HERE
```

it performs two independent operations. It stores symbol `XYZ` and its value (the current value of the LC) in the symbol table, and it assembles the instruction. These two operations have nothing to do with each

other. Handling the symbol definition and assembling the instruction are done by two different parts of the assembler. Often, they are performed in different phases of the assembly.

If the LC happens to have the value 260, then the entry

| name | value | type |
|---|---|---|
| XYZ | 0104 | REL |

will be added to the symbol table (104 is the hex value of decimal 260, and the type REL will be explained later).

When the assembler encounters the line

```
BAL 5,XYZ
```

it assembles the instruction but, in order to assemble the operand, the assembler needs to search the symbol table, find symbol `XYZ`, fetch its value and make it part of the assembled instruction. The instruction is, therefore, assembled as `45500104`.

▸ **Exercise 10.6:** The address in our example, 104, is a relatively small number. Often, instructions have a 12-bit field for the address, allowing addresses up to $2^{12}-1=4095$. What if the value of a certain symbol exceeds that number?

This is, in a very general way, what the assembler has to do in order to assemble instructions and handle symbols. It is a simple process and it involves only one problem which is illustrated by the following example.

```
     BAL  5,XYZ   ;CALL THE SUBROUTINE
     .
     .
XYZ  A    4,ABC   ;THE SUBROUTINE STARTS HERE
```

In this case the value of symbol `XYZ` is needed *before* label `XYZ` is defined. When the assembler gets to the first line (the `BAL` instruction), it searches the symbol table for `XYZ` and, of course, does not find it. This situation is called the *future symbol problem* or the problem of *unresolved references*. The `XYZ` in our example is a future symbol or an unresolved reference.

Obviously, future symbols are not an error and their use should not be prohibited. The programmer should be able to refer to source lines which either precede or follow the current line. Thus the future symbol problem has to be solved. It turns out to be a simple problem and there are two solutions, a *one-pass assembler* and a *two-pass assembler*. They represent not just different solutions to the future symbol problem but *two different approaches to assembler design and operation*. The one-pass assembler, as the name implies, solves the future symbol problem by reading the source file *once*. Its most important feature, however, is that it does not generate a relocatable object file but rather loads the object code (the machine language program) directly into memory. Similarly, the most important feature of the two-pass assembler is that it generates a relocatable object file, that is later loaded into memory by a loader. It also solves the future symbol problem by performing two passes over the source file. It should be noted at this point that a one-pass assembler can generate an object file. Such a file, however, would be *absolute*, rather than relocatable, and its use is limited. Absolute and relocatable object files are discussed later in this chapter. Figure 10.2 is a summary of the most important components and operations of an assembler.

## 10.5 The Two-Pass Assembler

A two-pass assembler is easier to understand and will be discussed first. Such an assembler performs two passes over the source file. In the first pass it reads the entire source file, looking only for label definitions. All labels are collected, assigned values, and placed in the symbol table in this pass. No instructions are assembled and, at the end of the pass, the symbol table should contain all the labels defined in the program. In the second pass, the instructions are again read and are assembled, using the symbol table.

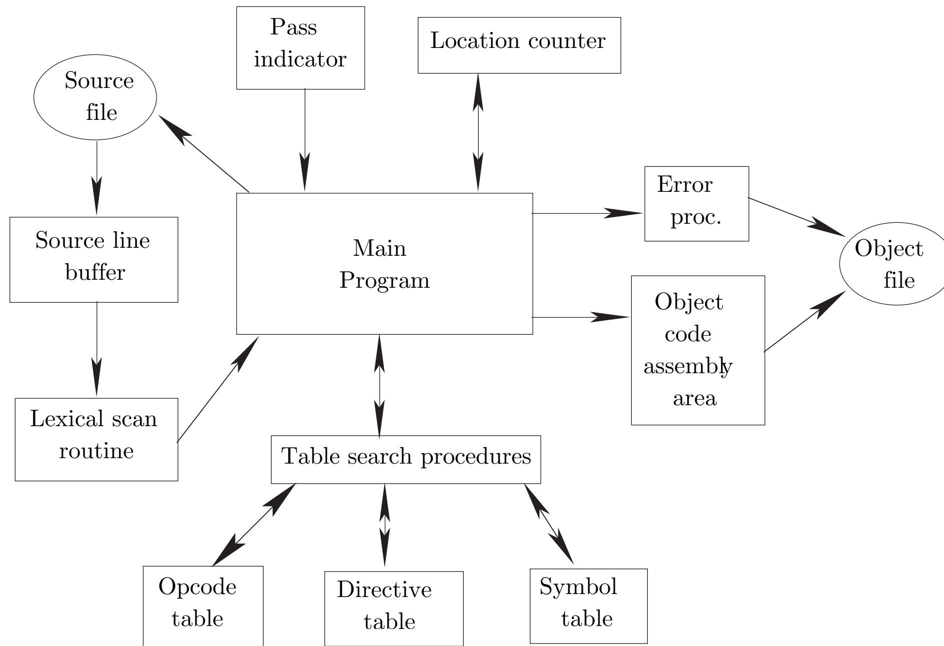



**Figure 10.2**. The main components and operations of an assembler

▸ **Exercise 10.7:** What if a certain symbol is needed in pass 2, to assemble an instruction, and is not found in the symbol table?

To assign values to labels in pass 1, the assembler has to maintain the LC. This in turn means that the assembler has to determine the size of each instruction (in words), even though the instructions themselves are not assembled.

In many cases it is easy to figure out the size of an instruction. On the IBM 360, the mnemonic determines the size uniquely. An assembler for this machine keeps the size of each instruction in the opcode table together with the mnemonic and the opcode (Table 10.1). On the DEC PDP-11 the size is determined both by the type of the instruction and by the addressing mode(s) that it uses. Most instructions are one word (16-bits) long. However, if they use either the *index* or *index deferred* modes, one more word is added to the instruction. If the instruction has two operands (source and destination) both using those modes, its size will be 3 words. On most modern microprocessors, instructions are between 1 and 4 bytes long and the size is determined by the opcode and the specific operands used.

This means that, in many cases, the assembler has to work hard in the first pass just to determine the size of an instruction. It has to look at the mnemonic and, sometimes, at the operands and the modes, even though it does not assemble the instruction in the first pass. All the information about the mnemonic and the operand collected by the assembler in the first pass is extremely useful in the second pass, when instructions are assembled. This is why many assemblers save all the information collected during the first pass and transmit it to the second pass through an *intermediate file.* Each record on the intermediate file contains a copy of a source line plus all the information that has been collected about that line in the first pass. At the end of the first pass the original source file is closed and is no longer used. The intermediate file is reopened and is read by the second pass as its input file.

A record in a typical intermediate file contains:

- The record type. It can be an instruction, a directive, a comment, or an invalid line.
- The LC value for the line.
- A pointer to a specific entry in the opcode table or the directive table. The second pass uses this pointer to locate the information necessary to assemble or execute the line.
- A copy of the source line. Notice that a label, if any, is not use by pass 2 but must be included in the

intermediate file since it is needed in the final listing.

Figure 10.3 is a flow chart summarizing the operations in the two passes.

There can be two problems with labels in the first pass; *multiply-defined labels* and *invalid labels*. Before a label is inserted into the symbol table, the table has to be searched for that label. If the label is already in the table, it is doubly or even multiply defined. The assembler should treat this label as an error and the best way of doing this is by inserting a special code in the *type* field in the symbol table. Thus, a situation such as

```
AB   ADD 5,X
     .
     .
AB   SUB 6,Y
     .
     .
     JMP AB
```

will generate the entry:

| name | value | type |
|---|---|---|
| AB | — | MTDF |

in the symbol table.

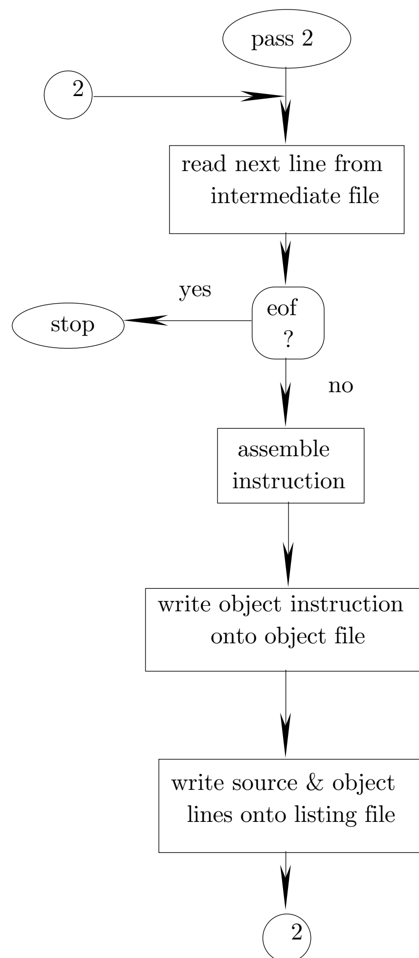



**Figure 10.3**. The operations of the two-pass assembler (part 2)

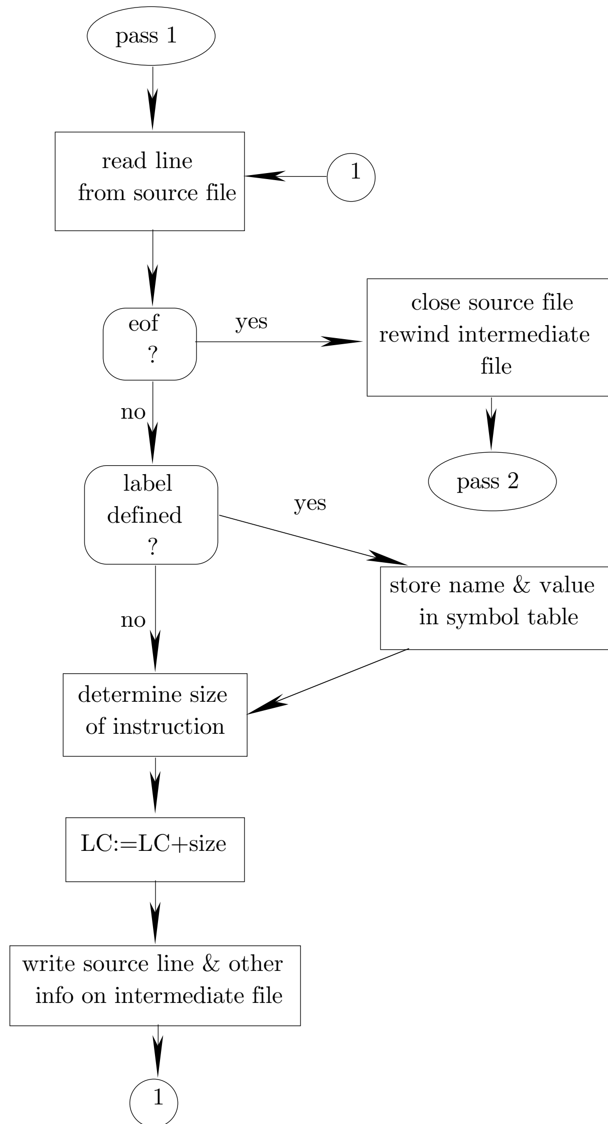



**Figure 10.3**. The operations of the two-pass assembler (part 1)

Labels normally have a maximum size (typically 6 or 8 characters), must start with a letter, and may only consist of letters, digits, and a few other characters. Labels that do not conform to these rules are invalid labels and are normally considered a fatal error. However, some assemblers will truncate a long label to the maximum size and will issue just a warning, not an error, in such a case.

▸ **Exercise 10.8:** What is the advantage of allowing characters other than letters and digits in a label?

The only problem with symbols in the second pass is *bad symbols*. These are either multiply-defined or undefined symbols. When a source line uses a symbol in the operand field, the assembler looks it up in the symbol table. If the symbol is found but has a type of `MTDF`, or if the symbol is not found in the symbol table (i.e., it has not been defined), the assembler responds as follows.

- It flags the instruction in the listing file.
- It assembles the instruction as far as possible, and writes it on the object file.
- It flags the entire object file. The flag instructs the loader not to start execution of the program. The object file is still generated and the loader will read and load it, but not start it. Loading such a file may be

useful if the user wants to see a memory map.

The `JMP AB` instruction above is an example of a bad symbol in the operand field. This instruction cannot be fully assembled, and thus constitutes our first example of a fatal error detected and issued by the assembler.

The last important point regarding a two-pass assembler is the box, in the flow chart above, that says *write object instruction onto the object file.* The point is that when the two-pass assembler writes the machine instruction on the object file, it has access to the source instruction. This does not seem to be an important point but, in fact, it constitutes the main difference between the one-pass and the two-pass assemblers. This point is the reason why a one-pass assembler can only produce an absolute object file (which has only limited use), whereas a two-pass assembler can produce a relocatable object file, which is much more general. This important topic is explained later in this chapter.

## 10.6 The One-Pass Assembler

The operation of a one-pass assembler is different. As its name implies, this assembler reads the source file once. During that single pass, the assembler handles both label definitions and assembly. The only problem is future symbols and, to understand the solution, let's consider the example in Figure 10.4.

```
LC
36        BEQ      AB  ;BRANCH ON EQUAL
          .
          .
67        BNE      AB  ;BRANCH ON NOT EQUAL
          .
          .
89        JMP      AB  ;UNCONDITIONALLY
          .
          .
126  AB   anything
```

**Figure 10.4**. Example of future symbols

Symbol `AB` is used three times as a future symbol. On the first reference, when the LC happens to stand at 36, the assembler searches the symbol table for `AB`, does not find it, and therefore assumes that it is a future symbol. It then inserts `AB` into the symbol table but, since `AB` has no value yet, it gets a special type. Its type is `U` (undefined). Even though it is still undefined, it now occupies an entry in the symbol table, an entry that will be used to keep track of `AB` as long as it is a future symbol. The next step is to set the 'value' field of that entry to 36 (the current value of the LC). This means that the symbol table entry for `AB` is now pointing to the instruction in which `AB` is needed. The 'value' field is an ideal place for the pointer since it is the right size, it is currently empty, and it is associated with `AB`. The `BEQ` instruction itself is only partly assembled and is stored, incomplete, in memory location 36. The field in the instruction were the value of `AB` should be stored (the address field), remains empty.

When the assembler gets to the `BNE` instruction (at which point the LC stands at 67), it searches the symbol table for `AB`, and finds it. However, `AB` has a type of `U`, which means that it is a future symbol and thus its 'value' field (=36) is not a value but a *pointer*. It should be noted that, at this point, a type of `U` does not necessarily mean an undefined symbol. While the assembler is performing its single pass, any undefined symbols must be considered future symbols. Only at the end of the pass can the assembler identify undefined symbols (see below). The assembler handles the `BNE` instruction by:

- Partly assembling it and storing it in memory location 67.

- Copying the pointer 36 from the symbol table to the partly assembled instruction in location 67. The instruction has an empty field (where the value of `AB` should have been), where the pointer is now stored. There may be cases where this field in the instruction is too small to store a pointer. In such a case the assembler must resort to other methods, one of which is discussed below.

- Copying the LC (=67) into the 'value' field of the symbol table entry for `AB`, rewriting the 36.

When the assembler reaches the `JMP AB` instruction, it repeats the three steps above. The situation at those three points is summarized below.

| memory | | symbol table | | | memory | | symbol table | | | memory | | symbol table | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| loc | contents | n | v | t | loc | contents | n | v | t | loc | contents | n | v | t |
| 36 | `BEQ -` | | . | | 36 | `BEQ -` | | . | | 36 | `BEQ -` | | . | |
| | . | | . | | | . | | . | | | . | | . | |
| | . | `AB` | 36 | `U` | | . | `AB` | 67 | `U` | | . | `AB` | 89 | `U` |
| | | | . | | 67 | `BNE 36` | | . | | 67 | `BNE 36` | | . | |
| | | | . | | | | | . | | | . | | . | |
| | | | | | | | | | | | . | | | |
| | | | | | | | | | | 89 | `JMP 67` | | | |

It is obvious that an indefinite number of instructions can refer to `AB` as a future symbol. The result will be a *linked list* linking all these instructions. When the definition of `AB` is finally found (the LC will be 126 at that point), the assembler searches the symbol table for `AB` and finds it. The 'type' field is still `U` which tells the assembler that `AB` has been used as a future symbol. The assembler then follows the linked list of instructions using the pointers found in the instructions. It starts from the pointer found in the symbol table and, for each instruction in the list, the assembler:

- saves the value of the pointer found in the address field of the instruction. The pointer is saved in a register or a memory location ('temp' in the figure below), and is later used to find the next incomplete instruction.

- Stores the value of `AB` (=126) in the address field of the instruction, thereby completing it.

The last step is to store the value 126 in the 'value' field of `AB` in the symbol table, and to change the type to `D`. The individual steps taken by the assembler in our example are shown in the table below.

| Address | Contents | Contents | Contents |
|---|---|---|---|
| 36 | `BEQ -` | `BEQ -` | `BEQ 126` |
| | . | | |
| | . | | |
| 67 | `BNE 36` | `BNE 126` | `BNE 126` |
| | . | | |
| | . | | |
| 89 | `JMP 126` | `JMP 126` | `JMP 126` |
| | `temp=67` | `temp=36` | `temp=/` |
| | Step 1 | Step 2 | Step 3 |

It therefore follows that at the end of the single pass, the symbol table should only contain symbols with a type of `D`. At the end of the pass, the assembler scans the symbol table for undefined symbols. If it finds any symbols with a type of `U`, it issues an error message and will not start the program.

Figure 10.5 is a flow chart of a one-pass assembler.

The one-pass assembler loads the machine instructions in memory and thus has no trouble in going back and completing instructions. However, the listing generated by such an assembler is incomplete since it cannot backspace the listing file to complete lines previously printed. Therefore, when an incomplete instruction (one that uses a future symbol) is loaded in memory, it also goes into the listing file as incomplete. In the example above, the three lines using symbol `AB` will be printed with asterisks '*' or question marks '?', instead of the value of `AB`.

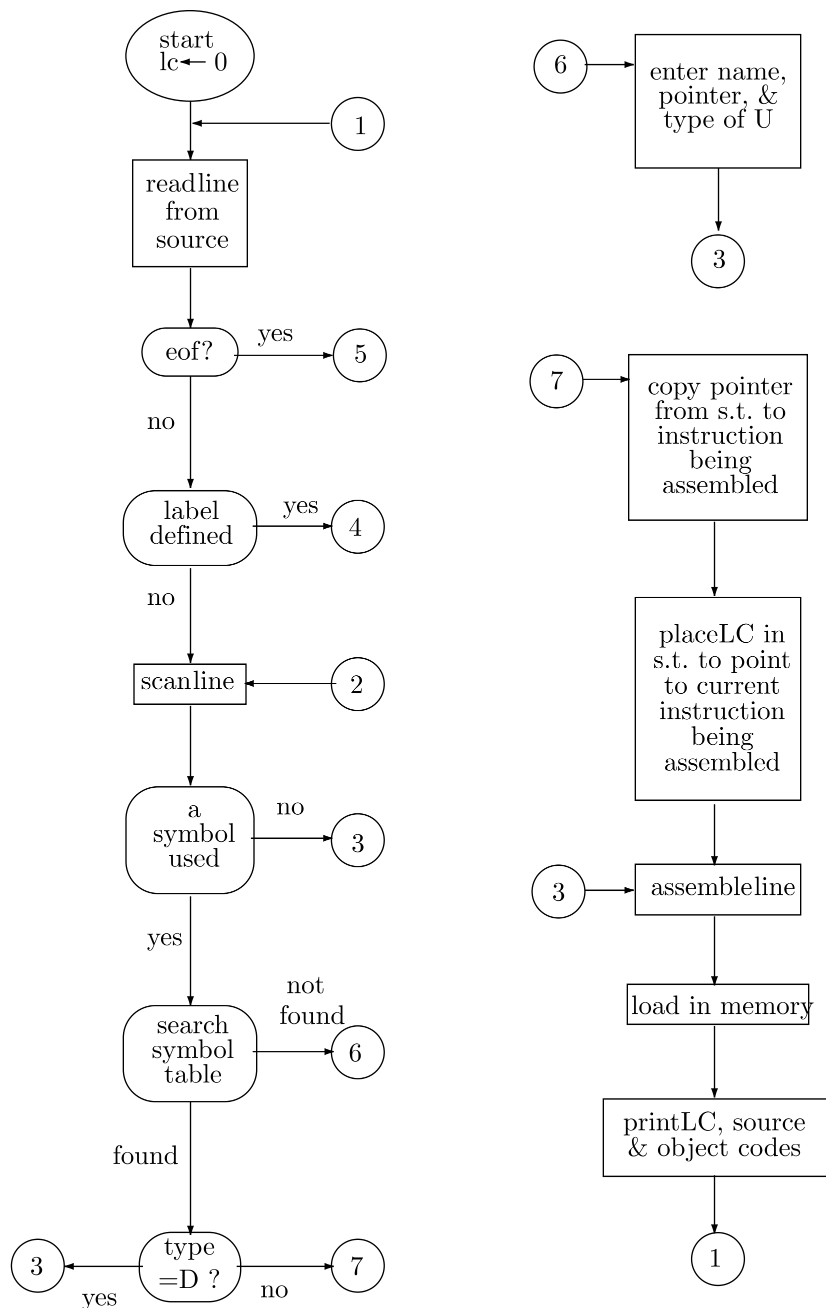



**Figure 10.5**. The operations of the one-pass assembler (part 1)

The key to the operation of a one-pass assembler is the fact that it loads the object code directly in memory and does not generate an object file. This makes it possible for the assembler to go back and complete instructions in memory at any time during assembly.

The one-pass assembler can, in principle, generate an object file by simply writing the object program from memory to a file. Such an object file, however, would be absolute. Absolute and relocatable object files are discussed in Section 10.7.

One more point needs to be mentioned here. It is the case where the address field in the instruction is too small for a pointer. This is a common case, since machine instructions are designed to be short and

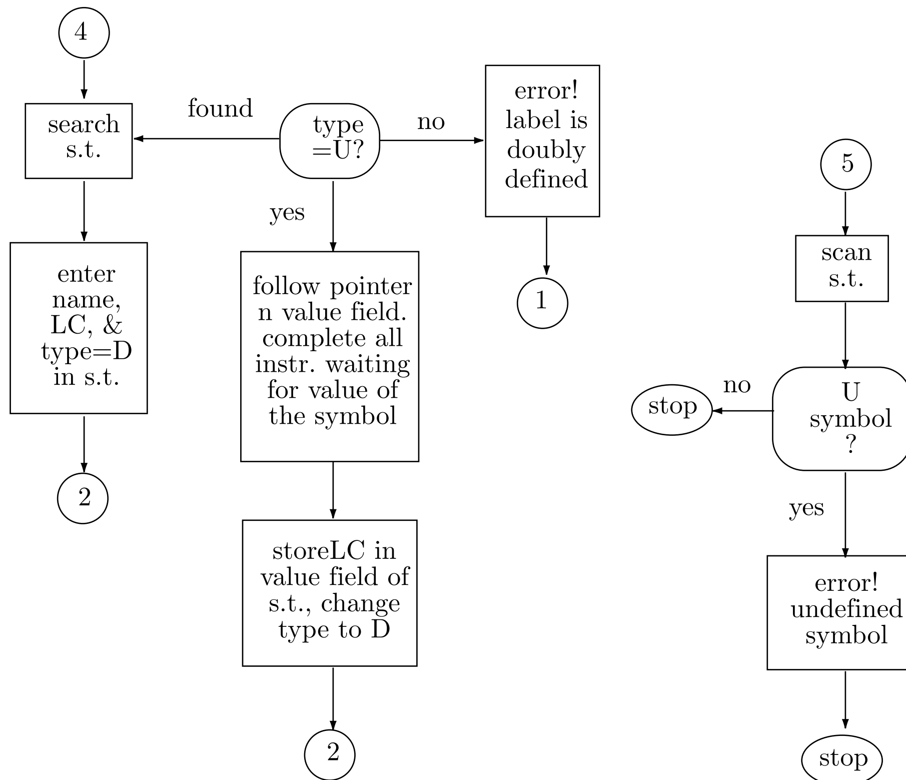



**Figure 10.5**. The operations of the one-pass assembler (part 2)

normally do not contain a full address. Instead of a full address, a typical machine instruction contains two fields, mode and displacement (or *offset*), such that the mode tells the computer how to obtain the full address from the displacement (Section 2.3). The displacement field is small (typically 8–12 bits) and has no room for a full address.

To handle this situation, the one-pass assembler has an additional data structure, a collection of linked lists, each corresponding to a future symbol. Each linked list contains, in its nodes, pointers to instructions that are waiting to be completed. The list for symbol `AB` is shown in Figure 10.6 in three successive stages of its construction.

When symbol `AB` is found, the assembler uses the information in the list to complete all incomplete instructions. It then returns the entire list to the pool of available memory.

An easy way to maintain such a collection of lists is to house them in an array. Figure 10.7 shows our list, occupying positions 5,9,3 of such an array. Each position has two locations, the first being the data item stored (a pointer to an incomplete instruction) and the second, the array index of the next node of the list.

symbol table

| n | v | t |
|---|---|---|
| AB | 5 | U |

| 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|
| 36 | | 89 | | | | 67 |
| / | | 9 | | | | 3 |

**Figure 10.7**. Housing a linked list in an array

▸ **Exercise 10.9:** What would be good Pascal declarations for such a future symbol list:
a. Using absolute pointers.
b. Housed in an array.

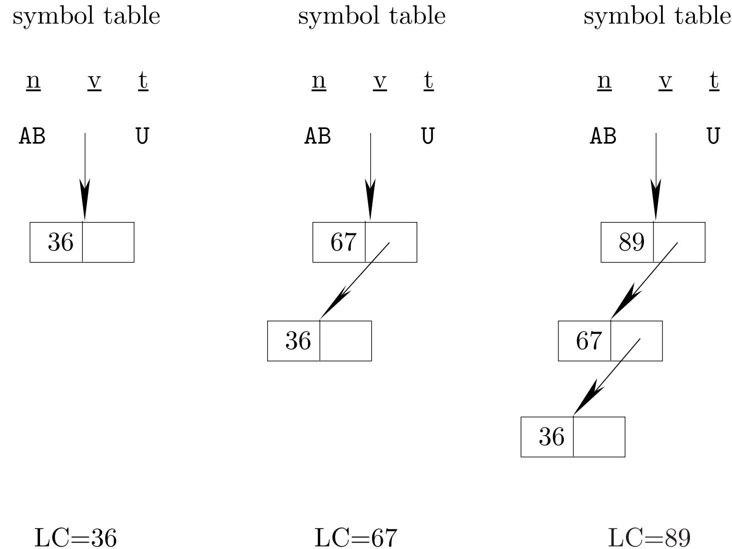



**Figure 10.6**. A linked list for symbol `AB`

## 10.7 Absolute and Relocatable Object Files

To illustrate the concept of absolute and relocatable object files, the following example is used, assuming a two-pass assembler.

```
LC
86              JMP TO
                .
                .
104     TO      ADD 1,2
```

The `JMP` instruction is assembled as `JMP 104` and is written onto the object file. When the object file is loaded starting at address 0, the `JMP` instruction is loaded at location 86 and the `ADD` instruction, at location 104. When the `JMP` is executed, it will cause a branch to 104, i.e., to the `ADD` instruction.

On subsequent loads, however, the loader may decide to load the program starting at a different address. On large computers, it is common to have several programs loaded in memory at the same time, each occupying a different area. Assuming that our program is loaded starting at location 500, the `JMP` instruction will go into location 586 and the `ADD`, into location 604. The `JMP` should branch to location 604 but, since it has not been modified, it will still branch to location 104 which is not only the wrong location, but is even outside the memory area of the program.

### 10.7.1 Relocation Bits

In a relocatable object file, this situation is taken care of by cooperation between the assembler and the loader. The assembler identifies and flags each item on the object file as either absolute or relocatable. The `JMP` instruction above would be relocatable since it uses a symbol (`TO`). The `ADD` instruction, on the other hand, would be absolute. It is assembled as `ADD 12` and will always add registers 1 and 2 regardless of the start address of the program.

In its simplest form, flagging each item is done by adding an extra bit, a *relocation bit*, to it. The relocation bit is set by the assembler to 0 if the item is absolute, and to 1, if it is relocatable. The loader, when reading the object file and loading instructions, reads the relocation bits. If an object instruction has a relocation bit of 0, the loader simply loads it into memory. If it has a relocation bit of 1, the loader relocates it by adding the start address to it. It is then loaded into memory in the usual way. In our example, the `JMP TO` instruction will be relocated by adding 500 to it. It will thus be loaded as `JMP 604` and, when executed, will branch to location 604 i.e., to the `ADD` instruction.

The relocation bits themselves are not loaded into memory since memory should contain only the object code. When the computer executes the program, it expects to find just instructions and data in memory. Any relocation bits in memory would be interpreted by the computer as either instructions or data.

This explains why a one-pass assembler cannot generate a relocatable object file. The type of the instruction (absolute or relocatable) can be determined only by examining the original source instruction. The one-pass assembler loads the machine instructions directly in memory. Once in memory, the instruction is just a number. By looking at a machine instruction in memory, it is impossible to tell whether the original instruction was absolute or relocatable. Writing the machine instructions from memory to a file will create an object file without any relocation bits, i.e., an absolute object file. Such an object file is useful on computers were the program is always loaded at the same place. In general, however, such files have limited value.

Some readers are tempted, at this point, to find ways to allow a one-pass assembler to generate relocation bits. Such ways exist, and two of them will be described here. The point is, however, that the one-pass assembler is a simple, fast, assemble-load-go program. Any modifications may result in a slow, complex assembler, thereby losing the main advantages of one-pass assembly. It is preferable to keep the one-pass assembler simple and, if a relocatable object file is necessary, to use a two-pass assembler.

Another point to realize is that a relocatable object file contains more than relocation bits. It contains loader directives and linking information. All this is easy for a two-pass assembler to generate but hard for a one-pass one.

### 10.7.2 One-Pass, Relocatable Object Files

Two ways are discussed below to modify the one-pass assembler to generate a relocatable object file.

**1.** A common approach to modify the basic one-pass assembler is to have it generate a relocation bit each time an instruction is assembled. The instruction is then loaded into memory and the relocation bit may be stored in a special, packed array outside the program area. When the object code is finally written on the object file, the relocation bits may be read from the special array and attached each to its instruction.

Such a method may work, but is cumbersome, especially because of future symbols. In the case of a future symbol, the assembler does not know the type (absolute or relocatable) of the missing symbol. It thus cannot generate the relocation bit, resulting in a hole in the special array. When the symbol definition is finally found, the assembler should complete all the instructions that use this symbol, and also generate the relocation bit and store it in the special array (a process involving bit operations).

**2.** Another possible modification to the one-pass assembler will be briefly outlined. The assembler can write each machine instruction on the object file as soon as it is generated and loaded in memory. At that point the source instruction is available and can be examined, so a relocation bit can be prepared and written on the object file with the instruction. The only problem is, as before, instructions using future symbols. They must go on the object file incomplete and without relocation bits. At the end of the single pass, the assembler writes the entire symbol table on the object file.

The task of completing those instructions is left to the loader. The loader initially skips the first part of the object file and reads the symbol table. It then rereads the file, and loads instructions in memory. Each time it comes across an incomplete instruction, it uses the symbol table to complete it and, if necessary, to relocate it as well.

The trouble with this method is that it shifts assembler tasks to the loader, forcing the loader to do what is essentially a two-pass job.

None of these modifications is satisfactory. The lesson to learn from these attempts is that, traditionally, the one-pass and two-pass assemblers have been developed as two different types of assemblers. The first is fast and simple; the second, a general purpose program which can support many features.

### 10.7.3 The Task of Relocating

The role of the loader is not as simple as may seem from the above discussion. Relocating an instruction is not always as simple as adding a start address to it. On the IBM 7090/7094 computers, for example, many instructions have the format:

| Field | opcode | Decrement | Tag | Address |
|---|---|---|---|---|
| Size (in bits) | 3 | 15 | 3 | 15 |

The exact meaning of the fields is irrelevant except that the *Address* and *Decrement* fields may both contain addresses. The assembler must determine the types of both fields (either can be absolute or relocatable), and prepare two relocation bits. The loader has to read the two bits and should be able to relocate either field. Relocating the Decrement field means adding the start address just to that field and not to the entire instruction.

▸ **Exercise 10.10:** How can the loader add something to a field in the middle of an instruction ?

Those familiar with separate assembly (the `EXTRN` and `ENTRY` directives) know that each field can in fact have three different types, Absolute, Relocatable, and *special relocation*. Thus the assembler generally has to generate two relocation bits for each field which, in the case of the IBM 7090/7094 (or similar computers), implies a total of four relocation bits. The loader uses those pairs of relocation bits as identification bits, identifying each line in the relocatable object file as one of four types: an absolute instruction, a relocatable instruction, an instruction requiring special relocation, or as a loader directives.

On the PC, an absolute object file has the extension `.COM`, and a relocatable object file has the extension `.EXE`.

### 10.7.4 Relocating Packed Instructions

An interesting problem is, how does the assembler handle relocation bits when several instructions are packed in one word?

In a computer such as the CDC Cyber, only 30-bit and 60-bit instructions may contain addresses. There are only six ways of combining instructions in a 60-bit word, as Figure 10.8 shows.

<table>
<tr><td colspan="4">60</td></tr>
<tr><td colspan="2">30</td><td colspan="2">30</td></tr>
<tr><td colspan="2">30</td><td>15</td><td>15</td></tr>
<tr><td>15</td><td colspan="2">30</td><td>15</td></tr>
<tr><td>15</td><td>15</td><td colspan="2">30</td></tr>
<tr><td>15</td><td>15</td><td>15</td><td>15</td></tr>
</table>

**Figure 10.8**. Packing long instructions

The assembler has to generate one of the values 0–5 as a 3-bit relocation field attached to each word as it is written on the object file. The loader reads this field and uses it to perform the actual relocation (Figure 10.9).

## 10.8 Absolute and Rel. Address Expressions

Most assemblers can handle address expressions. Generally, an address expression may be written instead of just a symbol. Thus the instruction `LOD R1,AB+1` loads register 1 from the memory location following `AB`; the instruction `ADD R1,AB+5` similarly operates on the memory location whose address is 5 greater than the address `AB`. More complex expressions can be used, and the following two points should be observed:

▪ Many assemblers (almost all the old ones and some of the newer ones) do not recognize operator precedence. They evaluate any expression strictly from left to right and do not even allow parentheses. Thus the expression `A+B*C` will be evaluated by the assembler as `(A+B)*C` and not, as might be expected, as `A+(B*C)`. The reason is that complex address expressions are rarely necessary in assembler programs and it is therefore pointless to add parsing routines to the assembler.

▪ When an instruction using an expression is assembled, the assembler should generate a relocation bit based on the expression. Every expression should therefore have a well defined type. It should be either

<table>
<tr><td>0</td><td colspan="4">60</td></tr>
<tr><td>1</td><td colspan="2">30</td><td colspan="2">30</td></tr>
<tr><td>2</td><td colspan="2">30</td><td>15</td><td>15</td></tr>
<tr><td>3</td><td>15</td><td colspan="2">30</td><td>15</td></tr>
<tr><td>4</td><td>15</td><td>15</td><td colspan="2">30</td></tr>
<tr><td>5</td><td>15</td><td>15</td><td>15</td><td>15</td></tr>
</table>

**Figure 10.9**. Adding relocation information

absolute or relative. As a result, certain expressions are considered invalid by the assembler. Examples: `AB+1` has the same type as `AB`. Typically `AB` is relative, but it is possible to define absolute symbols (by means of `EQU` and `SET`). In general, an expression of the form *rel+abs, rel-abs*, are relative, and expressions of the form $abs \pm abs$ are absolute.

An expression of the form $rel - rel$ is especially interesting. Consider the case

```
LC
16   A LOD
       .
       .
27   B STO
```

The value of `A` is 16 and its type is relative (meaning `A` is a regular label, defined by writing it to the left of an instruction). Thus `A` represents address 16 from the start of the program. Similarly `B` is address 27 from the start of the program. It is thus reasonable to define the expression `B-A` as having a value of $27 - 16 = 11$ and a type of absolute. It represents the distance between the two locations, and that distance is 11, regardless of where the program starts.

▸ **Exercise 10.11:** What about the expression `A-B`? is it valid? If yes, what are its value and type?

On the other hand, an expression of the form $rel + rel$ has no well-defined type and is, therefore, invalid. Both `A` and `B` above are relative and represent certain addresses. The sum `A+B`, however, does not represent any address. In a similar way $abs * abs$ is *abs*, $rel * abs$ is *rel* but $rel * rel$ is invalid. $abs/abs$ is *abs*, $rel/abs$ is *rel* but $rel/rel$ is invalid. All expressions are evaluated at the last possible moment. Expressions in any pass 0 directives (see Chapter 11 for a discussion of pass 0) are evaluated when the directive is executed, in pass 0. Expressions in any pass 1 directives are, similarly, evaluated in pass 1. All other expressions (in instructions or in pass 2 directives) are evaluated in pass 2.

An extreme example of an address expression is `A-B+C-D+E` where all the symbols involved are relative. It is executed from left to right `(((A-B)+C)-D)+E`, generating the intermediate types: $(((rel - rel) + rel) - rel) + rel \to ((abs + rel) - rel) + rel \to (rel - rel) + rel \to abs + rel \to rel$. A valid expression.

Generally, expressions of the type `X+A-B+C-D+`$\cdots$`+M-N+Y` are valid when `X,Y` are absolute and `A,B,...,N` are relative. The relative symbols must come in pairs like `A-B` except the last one `M-N`, where `N` may be missing. If `N` is missing, the entire expression is relative, otherwise, it is absolute.

▸ **Exercise 10.12:** How does the assembler handle an expression such as `A-B+K-L` in which all the symbols are relative but `K,L` are external?

### 10.8.1 Summary

The two-pass assembler generates the machine instructions in pass two, where it has access to the source instructions. It checks each source instruction and generates a relocation bit according to:

- If the instruction uses a relative symbol, then it is relocatable and the relocation bit is 1.
- If the instruction uses an absolute symbol (such as `EQU`) or uses no symbols at all, then the instruction is absolute and the relocation bit is 0.
- An instruction in the relative mode contains an offset, not the full address, and is therefore absolute (see Section 2.5 for the ralative mode).

The one-pass assembler generates the object file at the end of its single pass, by dumping all the machine instructions from memory to the file. It has no access to the source at that point and therefore cannot generate relocation bits.

As a result, those two types of assemblers have evolved along different lines, and represent two different approaches to the overall assembler design, not just to the problem of resolving future symbols.

## 10.9 Local Labels

In principle, a label may have any name that obeys the simple syntax rules of the assembler. In practice, though, label names should be descriptive. Names such as `DATE, MORE, LOSS, RED` are preferable to `A001, A002,...`

There are exceptions, however. The use of the non-descriptive label `A1` in the following example:

```
       .
       JMP A1
DDCT   DS 12   reserve 12 locations for array DDCT
A1     .
       .
```

is justified since it is only used to jump over the array `DDCT`. (Note that the array's name is descriptive, possibly meaning *deductions* or *double-dictionary*) The `DS` directive is used to reserve memory for an array. We say that `A1` is used only locally, to serve a limited purpose.

As a result, many assemblers support a feature called local labels. It is due to M. E. Conway who used it in the early UNISAP assembler for the UNIVAC I computer. The main idea is that if a label is used locally and does not require a descriptive name, why not give it a name that will signify this fact. Conway used names such as `1H`, `2H` for the local labels. The name of a local label in our examples is a single decimal digit. When such a label is referred to (in the operand field), the digit is followed by either `B` or `F` (for Backward or Forward).

```
LC
            .
            .
13    1:    ...
            .
            .
17         JMP 1F         jump to 24
            .
            .
24    1:   LOD R2,1B      1B here means address 13
            .
            .
31    1:   ADD R1,2F      2F is address 102
            .
            .
102   2:   DC 1206,-17
            .
            .
115        SUB R3,2B-1    102-1=101
```

**Example. Local labels.**

The example shows that local labels is a simple, useful, concept that is easy to implement. In a two-pass assembler, each local label is entered into the symbol table as any other symbol, in pass 1. Thus, the symbol table in our example contains

| n | v |
|---|---|
| 1 | 13 |
| 1 | 24 |
| 1 | 31 |
| 2 | 102 |

**Symbol Table**

The order of the labels in the symbol table is important. If the symbol table is sorted between the two passes, all occurrences of each local label should remain sorted by value. In pass 2, when an instruction uses a local label such as `1F`, the assembler identifies the specific occurrence of label 1 by comparing all local labels 1 to the current value of the LC. The first such instruction in our example is the `JMP 1F` at LC=17. Clearly, the assembler should look for a local label with the name '1' and a value $\geq 17$. The smallest such label has value 24. In the second case, LC=24 and the assembler is looking for a `1B`. It needs the label with name '1' and a value which is the largest among all values $< 24$. It therefore identifies the label as the '1' at 13.

▸ **Exercise 10.13:** If we modify the instruction at 24 above to read `1:  LOD R2,1F` would the `1F` refer to address 31 or 24?

In a one-pass assembler, again the labels are recognized and put into the symbol table in the single pass. An instruction using a local label `iB` is no problem, since is needs the most recent occurrence of the local label '1' in the table. An instruction using an `iF` is handled like any other future symbol case. An entry is opened in the symbol table with the name `iF`, a type of `U`, and a value which is a pointer to the instruction.

In the example above, a snapshot of the symbol table at LC=32 is

| n | v | t | |
|---|---|---|---|
| 1 | 13 | D | |
| 1 | 24 | D | |
| 1 | 31 | D | 31 is the value of the third 1 |
| 2 | 31 | U | 31 is a pointer to the ADD instruction |

**Symbol Table**

An advantage of this feature is that the local labels are easy to identify as such, since their names start with a digit. Most assemblers require regular label names to start with a letter.

In modern assemblers, local labels sometimes use a syntax different from the one shown here.

### 10.9.1 The LC as a Local Symbol

Virtually all assemblers allow a notation such as `BPL *+6` where '*' stands for the current value of the LC. The operand in this case is located at a point 6 locations following the `BPL` instruction.

The LC symbol can be part of any address expression and is, of course, relocatable. Thus `*+A` is valid if `A` is absolute, while `*-A` is always okay (and is absolute if `A` is relative, relative if `A` is absolute). This feature is easy to implement. The address expression involving the '*' is calculated, using the current value of the LC, and the value is used to assemble the instruction, or execute the directive, on the current source line. Nothing is stored in the symbol table.

Some assemblers use the asterisk for multiplication, and may designate the period '.' or the '$' for the LC symbol.

On the PDP-11 the notation `X: .=.+8` is used to increment the LC by 8, and thus to reserve eight locations (compare this to the DS directive).

▸ **Exercise 10.14:** What is the meaning of `JMP *`, `JMP *-*`?

## 10.10 Multiple Location Counters

This feature makes it possible to write the source program in a certain way, convenient to the programmer, and load it in memory in a different way, suitable for execution. It happens often that, while writing a program, a new piece of data, say an array, is needed at a certain point. It is handy to write the declaration of the array immediately following the first instruction that uses it, like:

```
    ADD D,...
D DS 12
```

However, at run time, the hardware, after executing the `ADD` instruction, would try to execute the first element of array `D` as an instruction. Obviously, instructions and data have to be separated, and normally all the arrays and constants are declared at the end of the program, following the last executable instruction (`HLT`).

### 10.10.1 The USE Directive

Multiple location counters make it possible to enjoy the best of both worlds. The data can be declared when first used, and can be loaded at the end of the program or anywhere else the programmer wishes. This feature uses several directives, the most important of which will be described here. It is based on the principle that new location counters can be declared and given names, at assembly time, at any point in the source code. The example above can be handled by declaring a location counter with a name (such as `DATA`) instructing the assembler to assemble the `DS` directive under that LC, and to switch back to the main LC—which now must have a name—like any other LC. Its name is ‘ ’ (a space).

This is done by the special directive `USE`:

```
   ADD D, ...
   USE DATA
D  DS 12
   USE *
   .
   .
```

This directive directs the assembler to start using (or to resume the use of) a new location counter. The name is specified in the operand field, so an empty operand means the main LC. The asterisk ‘*’ implies the previous LC, the one that was used before the last `USE`.

▸ **Exercise 10.15:** The previous section discusses the use of asterisk as the LC value. When executing a `USE *`, how does the assembler know that the asterisk is not the LC value?

The `USE` directives divide the program into several sections, which are loaded, by the loader, into separate memory areas. The sections are loaded in the order in which their names appear in the source. Figure 1–8 is a good example:

At load time, the sections would be loaded in the order `MAIN`, `DATA`, `BETA`, `GAMMA` or 1,3,6,2,5,4,7. Such a load involves an additional loader pass.

▸ **Exercise 10.16:** Can we start a program with a `USE ABC`? in other words, can the first section be other than the main section?

Another example of the same feature is procedures. In assembler language, a procedure can be written as part of the main program. However, the procedure should only be executed when called from the main program. Therefore, it should be separated from the main instruction stream, since otherwise the hardware

```
      .
      .   (1)
      .
USE   DATA
      .
      .   (2)
      .
USE   *
      .
      .   (3)
      .
USE   BETA
      .
      .   (4)
      .
USE   DATA
      .
      .   (5)
      .
USE   <space>
      .
      .   (6)
      .
USE   GAMMA
      .
      .   (7)
      .
      END
```

**Figure 1–8. Dividing a program into sections.**

would execute it when it runs into the first instruction of the procedure. So something like:

```
           .
           .
  0       LOD ...
           .
           .
 15       SUB ...
 16       CALL P
 17   P   ADD R5,N
           .
           .
 45       RET
 46       CLR ...
           .
           .
104       END
```

is wrong. The procedure is defined on lines 17–45 and is called on line 16. This makes the source program more readable, since the procedure is written next to its call. However, the hardware would run into the procedure and start executing it right after it executes line 16, i.e., right after it has been called. The solution is to use a new LC—named, perhaps, `PROC`—by placing a `USE PROC` between lines 16, 17 and a `USE *` between lines 45, 46.

### 10.10.2 COMMON Blocks

Fortran programmers are familiar with the COMMON statement. This is a block of memory reserved in a common area, accessible to the main program and to all its procedures. It is allocated by the loader high in memory, overlaying the loader itself. The common area cannot be preloaded with constants, since it uses the same memory area occupied by the loader. In many assemblers, designed to interface with Fortran programs, there is a preassigned LC, called //, such that all data declared under it end up being loaded in the common area. The concept of labeled common in Fortran also has its equivalent in assembler language. The Fortran statement COMMON/NAM/A(12),B(5) can be written in assembler language as:

```
   .
   USE /NAM/
A  DS 12
B  DS 5
   USE DAT
C  DC 56,-90
   USE
   .
   .
```

The two arrays A, B would be loaded in the labeled common /NAM/, while the constants labeled C would end up as part of section DAT.

The IBM 360 assembler has a CSECT directive, declaring the start of a *control section*. However, a control section on the 360 is a general feature. It can be used to declare sections like those described here, or to declare sections that are considered separate programs and are assembled separately. They are later loaded together, by the loader, to form one executable program. The different control sections are linked by global symbols, declared either as external or as entry points.

The VAX MACRO assembler has a .PSECT directive similar to CSECT, and it does not support multiple LCs. A typical VAX example is:

```
   .TITLE CALCULATE PI
   .PSECT DATA, NOEXE,WRT
A=2000
B: .WORD  6
C: .LONG  8
   .PSECT CODE, EXE,NOWRT
   .ENTRY PI,0
   .
   .
   <instructions>
   .
   .
   $EXIT
   .PSECT CONS, NOEXE,NOWRT
K: .WORD  1230
   .END   PI
```

Each .PSECT includes the name of the section, followed by attributes such as EXE, NOEXE, WRT, NOWRT.

The memory on the 80x86 microprocessors is organized in 64K (highly overlapping) segments. The microprocessor can only generate 16-bit addresses, i.e., it can only specify an address within a segment. A physical address is created by combining the 16-bit processor generated address with the contents of one of the *segment registers* in a special way. There are four such registers: The DS (data segment), CS (code segment), SS (stack segment) and ES (extra segment).

When an instruction is fetched, the PC is combined with the CS register and the result is used as the address of the next instruction (in the code segment). When an instruction specifies the address of a piece

of data, that address is combined with the DS register, to obtain a full address in the data segment. The extra segment is normally used for string operations, and the stack segment, for stack-oriented instructions (PUSH, POP or any instructions that use the SP or BP registers).

The choice of segment register is done automatically, depending on what the computer is doing at the moment. However, there are directives that allow the user to override this choice, when needed.

As a result of this organization, there is no need for multiple LCs on those microprocessors.

## 10.11 Literals

Many instructions require their operands to be addresses. The `ADD` instruction is typically written `ADD AB,R3` or `ADD R3,AB` where `AB` is a symbol and the instruction adds the contents of location `AB` to register 3. Sometimes, however, the programmer wants to add to register 3, not the contents of any memory location but a certain constant, say the number $-7$. Modern computers support the immediate mode which allows the programmer to write `ADD #-7,R3`. The number sign '#' indicates the *immediate mode* and it implies that the instruction contains the operand itself, not the address of the operand. Most old computers, however, do not support this mode; their instructions have to contain addresses, not the operands themselves. Also, in many computers, an immediate operand must be a small number.

To help the programmer in such cases, some assemblers support *literals.*A notable example is the MPW assembler for the Macintosh computer. A literal is a constant preceded by an equal sign '='. Using literals, the programmer can write `ADD =-7,R3` and the assembler handles this by:

- Preloading the constant $-7$ in the first memory location in the literal table. The literal table is loaded in memory immediately following the program.

- Assembling the instruction as `ADD TMP,R3` where `TMP` is the address where the constant was loaded

Such assemblers may also support octal (=O377 or =377B), hex (=HFF0A), real (=1.37 or =12E-5) or other literals.

### 10.11.1 The Literal Table

To handle literals, the assembler maintains a table, the literal table, similar to the symbol table. It has columns for the name, value, address and type of each literal. In pass 1, when the assembler finds an instruction that uses a literal, such as $-7$, it stores the name ($-7$) in the first available entry in the literal table, together with the value ($1\ldots11001_2$) and the type (decimal). The instruction itself is treated as any other instruction with a future symbol. At the end of pass 1, the assembler uses the LC to assign addresses to the literals in the table. In pass 2, the table is used, in much the same way as the symbol table, to assemble instructions using literals. At the end of pass 2, every entry in the literal table is treated as a `DC` directive and is written on the object file in the usual way. There are three points to consider when implementing literals.

- Two literals with the same name are considered identical; only one entry is generated in the literal table. On the other hand, literals with different names are treated as different even if they have identical values, such as =12.5 and =12.50.

- All literals are loaded following the end of the program. If the programmer wants certain literals to be loaded elsewhere, the `LITORG` directive can be used. The following example clarifies a point that should be mentioned here.

```
  .
ADD =-7,R3
  .
LITORG
  .
SUB =-7,R4
  .
```

The first $-7$ is loaded, perhaps with other literals, at the point in the program where the `LITORG` is specified. The second $-7$, even though identical to the first, is loaded separately, together with all the literals used since the `LITORG`, at the end of the program.

The `LITORG` directive is commonly used to make sure that a literal is loaded in memory close to the instruction using it. This may be important in case the relative mode is used.

▪ The LC can be used as a literal '= ∗'. This is an example of a literal whose name is always the same, but whose value is different for each use.

▸ **Exercise 10.17:** What is the meaning of `JMP =*`?

### 10.11.2 Examples

As has been mentioned before, some assemblers support literals even though the computer may have an immediate mode, because an immediate operand is normally limited in size. However, more and more modern computers, such as the 68000 and the VAX, support immediate operands of several sizes. Their assemblers do not have to support any literals. Some interesting VAX examples are:

1. `MOVL #7,R6` is assembled into `D0 07 56`. `D0` is the opcode, `07` is a byte with two mode bits and six bits of operand. The two mode bits (`00`) specify the short literal mode. This is really a short immediate mode. Even though the word 'literal' is used, it is not a use of literal but rather an immediate mode. The difference is that, in the immediate mode, the operand is part of the instruction whereas, when a literal is used, the instruction contains the address of the operand, not the operand itself. The third byte (`56`) specifies the use of register 6 in mode 5 (register mode). The assembler has generated a three-byte `MOVL` instruction in the short literal mode. This mode is automatically selected by the assembler if the operand fits in six bits.
2. `MOVW I^#7,R6` is assembled into `B0 8F 0007 56`. Here the user has forced the assembler to use the immediate mode by specifying `I^`. The immediate operand becomes a word (2 bytes or 16 bits) and the instruction is now 5 bytes long. The second byte specifies register `F` (which happens to be the PC on the VAX) in mode 8 (autoincrement). This combination is equivalent to the immediate mode, where the immediate operand is stored in the third byte of the instruction. The last byte (`56`) is as before.
3. Again, a `MOVL` instruction but in a different context.

```
LC
              MOVL #DATA,R6   assembled into D0 8F 00000037' 56
               .
               .
0037   DATA   .BYTE ...
               .
               .
```

   Even though the operand is small (0037) and fits in six bits, the assembler has automatically selected the immediate mode (`8F`) and has generated the constant as a long word (32 bits). The reason is that the source instruction uses a future symbol (`DATA`). The assembler has to determine the instruction size in pass 1 and, since `DATA` is a future symbol, the assembler does not have its value and has to assume the largest possible value. The result is a seven byte instruction instead of the three bytes in the first example!
   Incidentally, the quote in (`00000047'`) indicates that the constant is relocatable.

## 10.12 Attributes of Symbols

The value of a symbol is just one of several possible attributes of the symbol, stored, together with the symbol name, in the symbol table. Other attributes may be the type, LC name, and length. The LC name is important for relocation. So far the meaning of relocation has been to add the start address of the program. With multiple LCs, the meaning of 'to relocate' is to add the start address of the current LC section. When a relocatable instruction is written on the object file, it is no longer enough to assign it a relocation bit of 1. The assembler also has to write the name of the LC under which the instruction should be relocated. Actually, a code number is written instead of the name.

Not every assembler supports the length attribute and, when supported, this attribute is defined in different ways by different assemblers. The length of a symbol is defined as the length of the associated

instruction. Thus `A LOD R1,54` assigns to label `A` the size of the `LOD` instruction (in words). However the directive `C DS 3`may assign to label `C` either length 3 (the array size) or length 1 (the size of each array element). Also, a directive such as `D DC 1.786,'STRNG',9` may assign to `D` either length 3 (the number of constants) or the size of the first constant, in words.

The most important attribute of a symbol is its value, such as in `SUB R1,XY`. However, any attribute supported by the assembler should be accessible to the programmer. Thus things such as `T'A, L'B` specify the type and length of symbols and can be used throughout the program. Examples such as:

`H DC L'X` the length of X (in words) is preloaded in location `H`

`G DS L'X` array `G` has `L'X` elements

`AIF (T'X=ABS).Z` a conditional assembly directive, (Chapter 11).

are possible, even though not common.

## 10.13 Assembly-Time Errors

Many errors can be detected at assembly time, both in pass 1 and pass 2. Chapter 11 discusses pass 0, in connection with macros and, that pass, of course, can have its own errors.

Assembler errors can be classified in two ways, by their severity, and by their location on the source line. The first classification has three classes: Warnings, errors, and fatal errors. A warning is issued when the assembler finds something suspicious, but there is still a chance that the program can be assembled and run successfully. An example is an ambiguous instruction that can be assembled in several ways. The assembler decides how to assemble it, and the warning tells the user to take a careful look at the particular instruction. A fatal erroris issued when the assembler cannot continue and has to abort the assembly. Examples are a bad source file or a symbol table overflow.

If the error is neither a warning nor fatal, the assembler issues a message and continues, trying to find as many errors as possible in one run. No object file is created, but the listing created is as complete as possible, to help the user to quickly identify all errors.

The second classification method is concerned with the field of the source instruction where the error was detected. Wrong comments cannot be detected by the assembler, which leaves four classes of errors, label, operation, operand, and general.

1. Label Errors. A label can either be invalid (syntactically wrong), undefined, or multiply-defined. Since labels are handled in pass 1, all label errors are detected in that pass. (although undefined errors are detected at the end of pass 1).
2. Operation errors. The mnemonic may be unknown to the assembler. This is a pass 1 (or even pass 0) error since the mnemonic is necessary to determine the size of the instruction.
3. Operand errors. Once the mnemonic has been determined, the assembler knows what operands to expect. On many computers, a `LOD` instruction requires a register followed by an address operand. A `MOV` instruction may require two address operands, and a `RET` instruction, no operands. An error 'wrong operand(s)' is issued if the right type of operand is not found.

Even if the operands are of the right type, their values may be out of range. In a seemingly innocent instruction such as `LOD R16,#70000`, either operand, or even both, may be invalid. If the computer has 16 registers, `R0-R15`, then `R16` is out of range. If the computer supports 16-bit integers, then the number 70000 is too large.

Even if the operands are valid, there may still be errors such as a bad addressing mode. Certain instructions can only use certain modes. A specific mode can only use addresses in a certain range

4. General errors do not pertain to any individual line and have to do with the general status of the assembler. Examples are 'out of memory', 'cannot read/write file xxx', 'illegal character read from source file', 'table xxx overflow' 'phase error between passes'.

The last example is particularly interesting and will be described in some detail. It is issued when pass 1 makes an assumption that turns out, in pass 2, to be wrong. This is a severe error that requires a reassembly. Phase errors require a computer with sophisticated instructions and complex memory management; they don't exist on computers with simple architectures. The Intel 80x86 microprocessors—with variable-size instructions, several offset sizes, and segmented memory management—are a good example of computer architecture where phase errors may easily occur.

Here are two examples of phase errors on those microprocessors.

- An instruction in a certain code segment refers to a variable declared in a data segment *following* the code segment. In pass 1, the assembler assumes that the variable is declared in the same segment as the instruction, and is a future symbol. The instruction is determined accordingly. In pass 2, when the time comes to assemble the instruction, all the variables are known, and the assembler discovers that the variable in question is `far`. A longer instruction is necessary, the pass-1 assumption turns out to be wrong, and pass 2 cannot assemble the instruction. This error is illustrated below.

```
CODE_S        SEGMENT PUBLIC
       ..
       ..
       MOV    AL,ABC
       ..
       ..
CODE_S ENDS
DATA_S        SEGMENT PUBLIC
ABC    DB     123
       ..
DATA_S ENDS
       END    START
```

- an instruction in the relative mode has a field for the relative address (the offset). Several possible offset sizes are possible on the 80x86, depending on the distance between the instruction and its operand. If the operand is a future symbol, even in the same segment as the instruction, the assembler has to guess a size for the offset. In pass 2 the operand is known and, if it too far from the instruction, the offset size guessed in pass 1 may turn out to be too small.

The Microsoft macro assembler (MASM), a typical modern assembler for the 80x86 microprocessors, features a list of about 100 error messages. Even an early assembler such as IBMAP for the IBM 7090 had a list of 125 error messages, divided into four classes according to the severity of the error.

## 10.14 The Symbol Table

The organization of the symbol table is the key to fast assembly. Even when working on a small program, the assembler may use the symbol table hundreds of times and, consequently, an efficient implementation of the table can cut the assembly time significantly even for short programs.

The symbol table is a dynamic structure. It starts empty and should support two operations, insertion and search. In a two-pass assembler, insertions are done only in the first pass and searches, only in the second. In a one-pass assembler, both insertions and searches occur in the single pass. The symbol table does not have to support deletions, and this fact affects the choice of data structure for implementing the table. A symbol table can be implemented in many different ways but the following methods are almost always used, and will be discussed here:

- A linear array.
- A sorted array with binary search.
- Buckets with linked lists.
- A binary search tree.
- A hash table.

### 10.14.1 A Linear Array

The symbols are stored in the first $N$ consecutive entries of an array, and a new symbol is inserted into the table by storing it in the first available entry (entry $N+1$) of the array. A typical Pascal code for such an

array would be:

```
var symtab:  record
N: 0..lim;
tabl:  array[0..lim] of record
          name:  string;
          valu:  integer;
          type:  char;
     end;
end;
```

Where lim is some suitable constant. The variable $N$ is initially set to zero, and it always points to the last entry in the array. An insertion is done by:

- Testing to make sure that $N <$ lim (the symbol table is not full).
- Incrementing $N$ by 1.
- Inserting the name, value, and type into the three fields, using $N$ as an index.

The insertion takes fixed time, independent of the number of symbols in the table.

To search, the array of names is scanned entry by entry. The number of steps involved varies from a minimum of 1 to a maximum of $N$. Every search for a non-existent symbol involves $N$ steps, thus a program with many undefined symbols will be slow to assemble because the average search time will be high. Assuming a program with only a few undefined symbols, the average search time is $N/2$. In a two-pass assembler, insertions are only done in the first pass so, at the end of that pass, $N$ is fixed. All searches in the second pass are performed in a fixed table. In a one-pass assembler, $N$ grows during the pass, and thus each search takes an average of $N/2$ steps, but the values of $N$ are different.

Advantages. Fast insertion. Simple operations.

Disadvantages. Slow search, specially for large values of $N$. Fixed size.

### 10.14.2 A Sorted Array

The same as a linear array, but the array (actually, the three arrays) is sorted, by name, after the first pass is completed. This, of course, can only be done in a two-pass assembler. To find a symbol in such a table, binary search is used, which takes an average of $\log_2 N$ steps. The difference between $N$ and $\log_2 N$ is small when $N$ is small but, for large values of $N$, the difference can get large enough to justify the additional time spent on sorting the table.

Advantages. Fast insertion and fast search. Since the table is already sorted, the preparation of a cross-reference listing.

Disadvantages. The sort takes time, which makes this method useful only for a large number of symbols (at least a few hundred).

### 10.14.3 Buckets with Linked Lists

An array of 26 entries is declared, to serve as the start of the buckets. Each entry points to a bucket that is a linked list of all those symbols that start with the same letter. Thus all the symbols that start with a 'C' are linked together in a list that can be reached by following the pointer in the third entry of the array. Initially all the buckets are empty (all pointers in the array are null). As symbols are inserted, each bucket is kept sorted by symbol name. Notice that there is no need to actually sort the buckets. The buckets are kept in sorted order by carefully inserting each new symbol into its proper place in the bucket. When a new symbol is presented, to be inserted in a bucket, the bucket is first located by using the first character in the symbol's name (one step). The symbol is then compared to the first symbol in the bucket (the symbol names are compared). If the new symbol is less (in lexicographic order) than the first, the new one becomes the first in the bucket. Otherwise, the new symbol is compared to the second symbol in the bucket, and so on. Assuming an even distribution of names over the alphabet, each bucket contains an average of $N/26$ symbols, and the average insertion time is thus $1 + (N/26)/2 = 1 + N/52$. For a typical program with a few hundred symbols, the average insertion requires just a few steps.

A search is done by first locating the bucket (one step), and then performing the same comparisons as in the insertion process shown earlier. The average search thus also takes $1 + N/52$ steps.

Such a symbol table has a variable size. More nodes can be allocated and added to the buckets, and the table can, in principle, use the entire available memory.
Advantages. Fast operations. Flexible table size.
Disadvantages. Although the number of steps is small, each step involves the use of a pointer and is therefore slower than a step in the previous methods (that use arrays). Also, some programmers always tend to assign names that start with an `A`. In such a case all the symbols will go into the first bucket, and the table will behave essentially as a linear array.

Such an implementation is recommended only if the assembler is designed to assemble large programs, and the operating system makes it convenient to allocate storage for list nodes.

▸ **Exercise 10.18:** What if symbol names can start with a character other than a letter? Can this data structure still be used? If yes, how?

### 10.14.4 A Binary Search Tree

This is a general data structure used not just for symbol tables, and is quite efficient. It can be used by either a one pass or two pass assembler with the same efficiency.

The table starts as an empty binary tree, and the first symbol inserted into the table becomes the root of the tree. Every subsequent symbol is inserted into the table by (lexicographically) comparing it with the root. If the new symbol is less than the root, the program moves to the left son of the root and compares the new symbol with that son. If the new symbol is greater than the root, the program moves to the right son of the root and compares as above. If the new symbol turns out to be equal to any of the existing tree nodes, then it is a doubly-defined symbol. Otherwise, the comparisons continue until a node is reached that does not have a son. The new symbol becomes the (left or right) son of that node.

**Example:** Assuming that the following symbols are defined, in this order, in a program.

`BGH J12 MED CC ON TOM A345 ZIP QUE PETS`

Symbol `BGH` becomes the root of the tree, and the final binary search tree is shown in Figure 10.10.

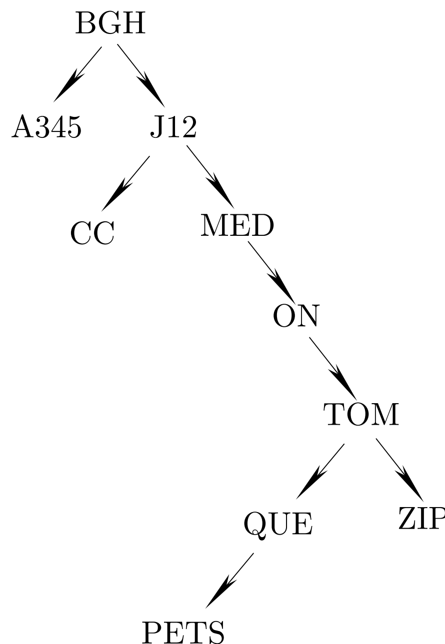



**Figure 10.10**. A binary search tree

Most texts on Data Structures discuss binary search trees. The minimum number of steps for insertion or search is obviously 1. The maximum number of steps depends on the height of the tree. The tree in

Figure 10.10 above has a height of 7, so the next insertion will require from one to seven steps. The height of a binary tree with $N$ nodes varies between $\log_2 N$ (which is the height of a fully balanced tree), and $N$ (the height of a skewed tree). It can be proved that an average binary tree is closer to a balanced tree than to a skewed tree, and this implies that the average time for insertion or search in a binary search tree is of the order of $\log_2 N$.

Advantages. Efficient operation (as measured by the average number of steps). Flexible size.

Disadvantages. Each step is more complex than in an array-based symbol table.

The recommendations for use are the same as for the previous method.

### 10.14.5 A Hash Table

This method comes in two varieties, open hash, which uses pointers and has a variable size, and closed hash, which is a fixed-size array.

### 10.14.6 Closed Hashing

A closed hash table is an array (actually three arrays, for the *name*, *value*, and *type*), normally of size $2^N$, where each symbol is stored in an entry. To insert a new symbol, it is necessary to obtain an index to the entry where the symbol will be stored. This is done by performing an operation on the name of the symbol, an operation that results in an $N$-bit number. An $N$-bit number has a value between 0 and $2^N - 1$ and can thus serve as an index to the array. The operation is called hashing and is done by hashing, or scrambling, the bits that constitute the name of the symbol. For example, consider 6-character names, such as `abcdef`. Each character is stored in memory as an 8-bit ASCII code. The name is divided into three groups of two characters (16-bits) each, `ab cd ef`. The three groups are added, producing an 18-bit sum. The sum is split into two 9-bit halves which are then multiplied to give an 18-bit product. Finally $N$ bits are extracted from the middle of the product to serve as the hash index. The hashing operations are meaningless since they operate on codes of characters, not on numbers. However, they produce an $N$-bit number that depends on all the bits of the original name. A good hash function should have the following two properties:

- It should consider all the bits in the original name. Thus when two names that are slightly different are hashed, there should be a good chance of producing different hash indexes.

- For a group of names that are uniformly distributed over the alphabet, the function should produce indexes uniformly distributed over the range $0 \ldots 2^N - 1$.

Once the hash index is produced, it is used to insert the symbol into the array. Searching for symbols is done in an identical way. The given name is hashed, and the hashed index is used to retrieve the value and the type from the array.

Ideally, a hash table requires fixed time for insert and search, and can be an excellent choice for a large symbol table. There are, however, two problems associated with this method namely, collisions and overflow, that make hash tables less than ideal.

Collisions involve the case where two entirely different symbol names are hashed into identical indexes. Names such as `SYMB` and `ZWYG6` can be hashed into the same value, say, 54. If `SYMB` is encountered first in the program, it will be inserted into entry 54 of the hash table. When `ZWYG6` is found, it will be hashed, and the assembler should discover that entry 54 is already taken. The collision problem cannot be avoided just by designing a better hash function. The problem stems from the fact that the set of all possible symbols is very large, but any given program uses a small part of it. Typically, symbol names start with a letter, and consist of letters and digits only. If such a name is limited to six characters, then there are $26 \times 36^5$ ($\approx 1.572$ billion) possible names. A typical program rarely contains more than, say, 500 names, and a hash table of size 512 ($= 2^9$) may be sufficient. When 1.572 billion names are mapped into 512 positions, more than 3 million names will map into each position. Thus even the best hash function will generate the same index for many different names, and a good solution to the collision problem is the key to an efficient hash table.

The simplest solution involves a linear search. All entries in the symbol table are originally marked as vacant. When the symbol `SYMB` is inserted into entry 54, that entry is marked occupied. If symbol `ZWYG6` should be inserted into entry 54 and that entry is occupied, the assembler tries entries 55, 56 and so on. This implies that, in the case of a collision, the hash table degrades to a linear table.

Another solution involves trying entry $54 + P$ where $P$ and the table size are relative primes. In either case, the assembler tries until a vacant entry is found or until the entire table is searched and found to be all occupied.

It can be shown that the average number of steps to insert (or search for a) symbol is $1/(1 - p)$ where $p$ is the percent-full of the table. The value $p = 0$ corresponds to an empty table, $p = 0.5$ means a half-full table, etc. The following table gives the average number of steps for a few values of $p$.

| p | number of steps |
|---|---|
| 0 | 1 |
| .4 | 1.66 |
| .5 | 2 |
| .6 | 2.5 |
| .7 | 3.33 |
| .8 | 5 |
| .9 | 10 |
| .95 | 20 |

It is clear that when the hash table gets more than 50%–60% full, performance suffers, no matter how good the hashing function is. Thus a good hash table design makes sure that the table never gets more than 60% occupied. At that point the table is considered overflowed.

The problem of hash table overflow can be handled in a number of ways. Traditionally, a new, larger table is opened and the original table is moved to the new one by rehashing each element. The space taken by the original table is then released. A better solution, though, is to use open hashing.

### 10.14.7 Open Hashing

An open hash table is a structure consisting of buckets, each of which is the start of a linked list of symbols. It is very similar to the buckets with linked lists discussed earlier. The principle of open hashing is to hash the name of the symbol and use the hash index to select a bucket. This is better than using the first character in the name, since a good hash function can evenly distribute the names over the buckets, even in cases where many symbols start with the same letter.

The shortage of a single kind of bolt
would hold up the entire assembly. . .

—Henry Ford, 1926, *Today and Tomorrow*

# 11
# Macros

Webster defines the word macro (derived from the greek $\mu\alpha\kappa\rho o\sigma$) as meaning long, great, excessive or large. The word is used as a prefix in many compound technical terms, e.g., Macroeconomics, Macrograph, Macronesia. We will see that a single macro directive can result in many source lines being generated, which justifies the use of the word *macro* in assemblers. As mentioned in the introduction, macros were introduced into assemblers very early in the history of computing, in the 1950s.

## 11.1 Introduction

Strictly speaking, macros are directives, but since they are so commonly used (and also not easy for the assembler to execute), most assemblers consider them *features*, rather than directives. The concept of a macro is not limited to assemblers. It is useful in many applications and has been used in many software systems.

A macro is similar to a subroutine (or a procedure), but there are important differences between them. A subroutine is a section of the program that is written once, and can be used often by simply calling it from any point in the program. Similarly, a macro is a section of code that the programmer writes (defines) once, and then can use often. The main difference between a subroutine and a macro is that the former is stored in memory once (just one copy), whereas the latter is duplicated as many times as necessary.

A subroutine call is specified by an instruction that is executed by the hardware. The hardware saves the return address and causes a branch to the start of the subroutine. Macros, however, are handled in a completely different way. A macro call is specified by a directive that is executed by the assembler. The assembler generates a copy of the macro and places it in the program, in place of the directive. This process is called macro expansion, and each expansion causes another copy of the macro to be generated and placed in the program, thus increasing the size of the object code.

As a result, subroutines are completely handled by the hardware, at run time; macros are completely handled by the assembler, at assembly time. The assembler knows nothing about subroutines; the hardware knows nothing about macros. A subroutine call instruction is assembled in the usual way and is treated by the assembler as any other instruction. Macro definition and macro expansion, however, are executed by the assembler, so it has to know all the features, options and exceptions associated with them. The hardware, on the other hand, executes the subroutine call instruction, so it has to know how to save the return address and how to branch to the subroutine. The hardware, however, gets to execute the object code after it has been assembled, with the macros expanded in place. By looking at an instruction in the object code, it is impossible to tell whether it came from the main program or from an expanded macro. The hardware thus executes the program in total ignorance of macros.

Figure 11.1 illustrates the differences between subroutines and macros:

When program `A` in Figure 11.1a is executed, execution starts at label `K` and each `CALL N` instruction causes the subroutine (section 1) to be executed. The order of execution will thus be 2,1,3,1,4. Even

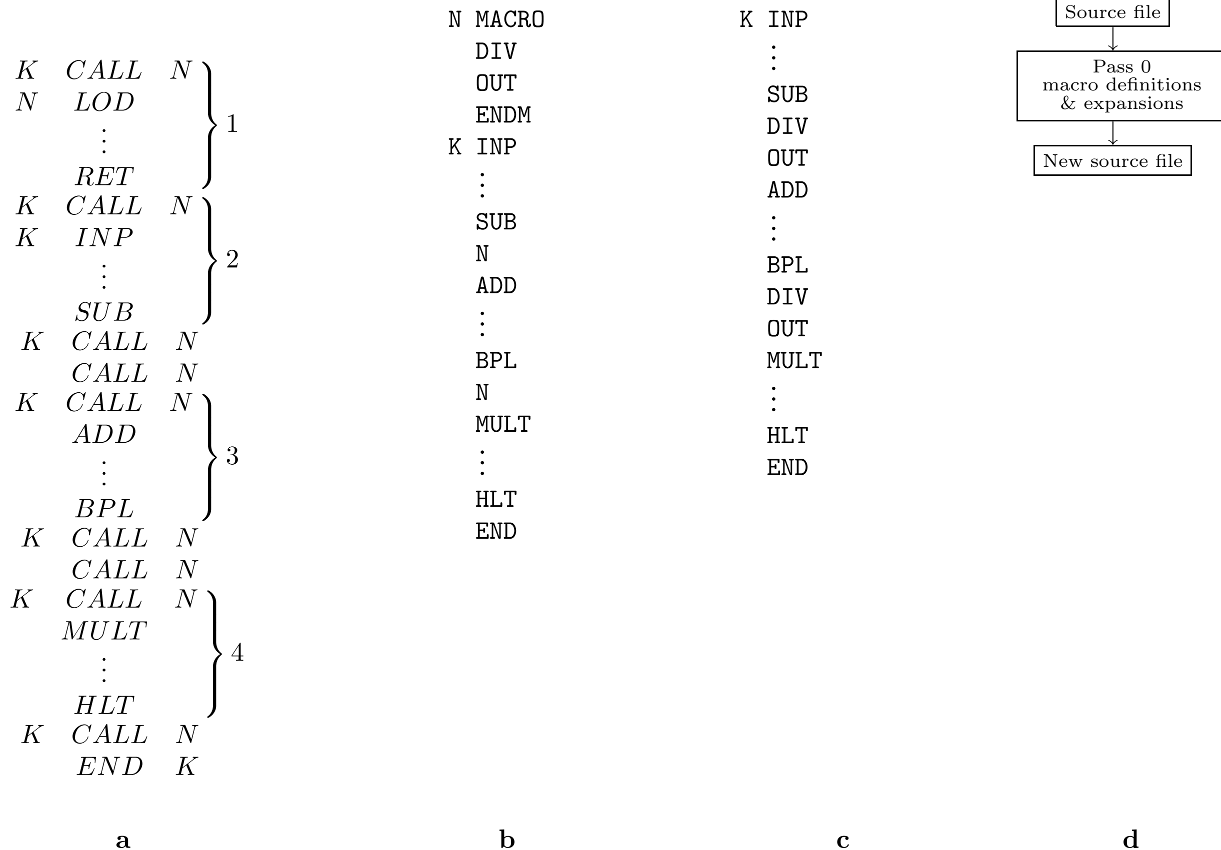



**Figure 11.1**. The difference between macros and subroutines
(a) A program with a subroutine N. (b) A program with a macro N.
(c) Same program after the macro expansion. (d) A summary of pass 0.

though section 1 is the first in the program, it is not the first to be executed since it constitutes the definition of a subroutine. When the program is assembled, the assembler reads and assembles the source file straight through. It does not change the positions of the different sections, and does not treat section 1 (the subroutine) in any special way. The order of execution has therefore to do only with the way the `CALL` instruction works. This is why subroutines are a hardware feature.

Program `B` (Figure 11.1b) is handled in a different way. The assembler reads the `MACRO`, `ENDM` directives and thus recognizes the two instructions `DIV`, `OUT` as the body of macro `N`. It then places a copy of that body wherever it finds a source line with `N` in the operation field. The output is the same program, with the macro definition removed, and with all the expansions in place. This output (Figure 11.1c) is ready to be assembled, in the usual way, by passes 1 and 2. This is why macros are an assembler feature and handling them is done by a special pass, pass 0, where a new source file is generated (Figure 11.1d) to be read by pass 1 as its source file.

Having a separate pass 0 simplifies the design of the assembler, since it divides the entire assembly job in a logical way between the three passes. The user, of course, has to pay a price in the form of increased assembly time, but this is a reasonable price to pay for the added power of the assembler. It is possible to combine passes 0 and 1 into one pass, which speeds up the assembler. However, this results in a very complex pass 1, which takes more time to write and debug, and reduces assembler reliability.

The task of pass 0 is thus to read the source file, handle all macro definitions and expansions, and generate a new source file that is identical to the original file, except that it does not have the macro definitions, and it has all the macro expansions in place. In principle, the new source file should have no mention of macros; in practice, it needs to have some macro information which eventually is transferred to

pass 2, to be written on the listing file. This point is further discussed below.

### 11.1.1 Macro Definition And Expansion

To define a macro, the `MACRO`, `ENDM` directives are used.

```
NU  MACRO    the header, or prototype of the macro.
    LOD A    the body (or model
    ADD B    statements) of the
    STO C    defined macro.
    ENDM     the trailer of the definition
```

Those two directives always come in pairs. The `MACRO` directive defines the start of the macro definition, and should have the macro name in the label field. The `ENDM` directive specifies the end of the definition.

Some assemblers use different syntax to define a macro. The IBM 360 assembler uses the following syntax:

```
      MACRO
&p1   name   &p2,&p3,...
      .
      .
      MEND   instead of ENDM
```

where `&p1, &p2` are parameters (Section 11.2), each starting with an ampersand '&'.

To expand a macro, the name of the macro is placed in the operation field, and no special directives are necessary.

```
.
.
COMP ..
NU
SUB ..
.
.
```

The assembler recognizes `NU` as the name of a macro, and expands the macro by placing a copy of the macro definition between the `COMP` and `SUB` instructions. The object code generated will contain the codes of the five instructions:

```
COMP ..
LOD A
ADD B
STO C
SUB ..
```

Handling macros involves two separate phases. Handling the definition and handling the expansions. A macro can only be defined once (see the discussion of nested macros in Section 11.6 for exceptions, however), but it can be expanded many times. Handling the definition is a relatively simple process. The assembler reads the definition from the source file and saves it in a special table, the *Macro Definition Table* (MDT). The assembler does not try to check the definition for errors, to assemble it, execute it, or do anything else with it. It just saves the definition as it is (again, there is an exception, mentioned on page 305, that has to do with identifying parameters). On encountering the `MACRO` directive, the assembler switches from the normal mode of operation to a special macro-definition mode in which it:

- locates available space in the MDT
- reads source lines and saves them in the MDT until an `ENDM` is read.

Upon reading `ENDM` from the source file, the assembler switches back to the normal mode. If the `ENDM` is missing, the assembler stays in the macro definition mode and saves source lines in the MDT until an obvious error is found, such as another `MACRO`, or the `END` of the entire program. In such a case, the assembler issues an error (run away definition) and aborts the assembly.

Handling a macro expansion starts when the assembler reads a source line that is not any instruction or directive. The assembler searches the MDT for a macro with that name and, on locating it, switches from the normal mode of operation to a special macro-expansion mode in which it:

- Reads a source line from the MDT.
- Writes it on the new source file, unless it is a pass 0 directive, in which case it is immediately executed.
- Repeats the two steps until the end of the macro is located in the MDT.

The following example illustrates this process. The macro definition contains an error and a label.

```
BAD  MACRO
     ADD #1,R4
     A$D R5       wrong mnemonic
LAN  CMP R3,R5
     ENDM
```

The definition is stored in the MDT with the error (`A$D`) and the label. Since the assembler copies the macro definition verbatim, it does not recognize `LAN` as a label at this point. The macro may later be expanded several times, causing several copies to be written onto the new source file. Pass 0 does not check these copies in any way and, as a result, does not issue any error messages (note that pass 0 does not handle labels and does not maintain the symbol table). When pass 1 reads the new source file, it discovers the multiple definitions of `LAN` and issues an error on the second and subsequent definitions. When pass 2 assembles the instructions, it discovers the bad `A$D` instructions and flags each of them.

▸ **Exercise 11.1:** In such a case, how can we ever define a macro with a label?

This does not sound like a good way to implement macros. It would seem better to assemble the macro when it is first encountered, i.e., when its definition is found, and to store the assembled version in the MDT. The reason why assemblers do not do that but rather treat macros as described earlier, is because of the use of parameters.

## 11.2 Macro Parameters

The use of parameters is the most important feature of macros. It is similar to the use of parameters!in subroutines, but there are important differences. The following examples illustrate the use of parameters in a simple macro. They show that parameters can be used in all the fields of an instruction, not just in the operation field.

```
      1                 2                   3                   4

MG1  MACRO        MG2  MACRO A,B,C     MG3  MACRO A,B,C     MG4  MACRO P
     LOD G             LOD A                A G                  LOD G
     ADD H             ADD B                B H             P    ADD H
     STO I             STO C                C I                  STO I
     ENDM              ENDM                 ENDM                 ENDM
```

Example 1 is a simple, three-line macro without parameters. Every time it is expanded, the same source lines are generated. They add the contents of memory locations `G` and `H`, and store the result in location I. Example 2 uses three parameters `A,B,C` for the three locations involved. The macro still does the same but, each time it is expanded, different locations are added, and the result is stored in a different location. When such a macro is expanded, the user should specify values (actual arguments) for the three parameters. Thus the expansion `MG2 X,Y,Z` would generate:

```
LOD X
ADD Y
STO Z
```

For the assembler to be able to assemble those instructions, the arguments `X,Y,Z` must be valid symbols defined in the program, i.e., the program should contain:

```
X DS 4
Y DC 44
Z EQU $FF00
```

or some similar lines on which labels `X,Y,Z` are defined.

This example shows why the assembler cannot assemble a macro at the time the definition is found in the source file. The macro lines may depend on parameters that are assigned values only when the macro is expanded. Thus, in general, macro lines can only be assembled or executed when the macro is expanded.

The process of assigning the value of an actual argument to a formal parameter is called *binding*. Thus the formal parameter `A` is bound to the actual argument `X`. The process of placing the actual arguments in place of the formal parameters when expanding a macro, is called *parameter substitution*.

▸ **Exercise 11.2:** Consider the case of an actual argument that happens to be identical to a formal parameter. If the macro of example 2 above is expanded as `MG2 B,X,Y`, we would end up with the expansion

```
LOD B
ADD X
STO Y
```

However, `B` is the name of the second parameter. Would the assembler perform double substitution, to end up with `LOD X`?

Example 3 is even more striking. Here the parameters are used in the operation field. The operands are always the same. When such a macro is expanded, the user should specify three arguments which are valid mnemonics. The expansion `MG3 LOD,SUB,STO` would generate:

```
LOD G
SUB H
STO I
```

whereas the expansion `MG3 LOD,CMP,JNE` would generate

```
LOD G
CMP H
JNE I
```

which is a very different macro. It is obvious now that such a macro cannot be assembled when it is defined.

In example 4, the parameter is in the label field. Each expansion of this macro will have to specify an argument for the parameter, and that argument will become a new label. Thus `MG4 NON` generates

```
     LOD G
NON  ADD H
     STO I
```

and `MG4 BON` generates

```
     LOD G
BON  ADD H
     STO I
```

Each expansion involves the creation of a new label that will be added to the symbol table in pass 1. To avoid multiply-defined labels, each expansion should use a different argument. The argument could also be null (see below), which would generate no label. It is important to realize, however, that the label becomes known only when the macro is expanded and not before. Macro expansions, therefore, must be done early in the assembly process. They cannot be done in the second pass because the symbol table must be stable during that pass. All macro expansions are done in pass 0 (except in assemblers that combine passes 0 and 1).

▸ **Exercise 11.3:** How many parameters can a macro have?

Another, more important, reason why macros must be expanded early is the need to maintain the LC during the first pass. The LC is used, during that pass, to assign values to symbols. It is important, therefore, to execute each expansion and to increment the LC while individual lines are expanded. If macros are handled in pass 0, then pass 1 is not concerned about them and it proceeds normally.

The main advantage of having a separate pass 0 is simplicity. Pass 0 only handles macros and pass 1 remains unchanged. Another advantage is that the memory area used for the MDT in pass 0 can be used for other purposes—like storing the symbol table—in the other passes. The main disadvantage of pass 0 is the additional time it takes, but there is a simple way to speed it up. It is only necessary to require that all macro definitions appear at the very beginning of the source file. When pass 0 starts, it examines the first source line. If it is not a `MACRO` directive, pass 0 assumes that there are no macro definitions (and, consequently, no macro expansions) in the program, and it immediately starts pass 1, directing it to use the original, instead of the new, source file.

Another point that should be noted here is the listing information that relates to macros. In principle, the new source file, generated by pass 0, contains no macro information. Pass 0 completely handles all macro definitions and expansions, and pass 1 needs to know nothing about macros. In practice, though, the user wants to see a listing of the macro definitions and, sometimes, also listings of all the expansions. This information can only be generated by pass 0 and should be transferred, first to pass 1 through the new source file, and then to pass 2—where the listing is created—through the intermediate file. This listing information cannot be transferred by storing it in memory since, with many definitions and expansions, it may be too large.

A related problem is the listing of macro expansions. The definition of a macro should obviously be listed, but a macro may be expanded often, and the user may want to suppress the listing of all or some of the expansions. Special directives that tell the assembler how to handle listing of macro expansions are discussed in the literature.

### 11.2.1 Attributes of Macro Parameters

The last example of the use of parameters is a macro whose arguments may be compound.

```
C  MACRO L1,L2,L3,L4,L5,L6
   ADD  L1,L2(2)    L2 is assumed compound and its 2nd component used
   L3
   B'L4 DEST
   C'L5'D L6
   .
   .
   ENDM
```

An expansion of such a macro may look like this

```
C SUM,(D,T,U),(SUB R1,L1),Z,,SN
```

The second argument is compound and has three components. The third argument is also compound, since it has a space and a comma in it, and therefore must be parenthesized. The parentheses of a compound argument are stripped off during expansion. The fifth argument is null.

The above expansion generates:

```
ADD␣␣SUM,T     the symbol '␣' stands for a space
SUB␣R1,SUM
BZ␣DEST
CD␣SN
.
.
```

Which illustrates the following points about handling arguments in a macro expansion:

1. There are two spaces between the `ADD` and the `SUM` on the first line. This is because the macro definition has two spaces between the `ADD` and the `L1`. In the second line, though, there is only one space between the `SUB` and the `R1`. This is because the argument in the expansion has one space in it. The assembler expands a macro by copying the arguments, as strings, into the original macro line without any editing (except that when the argument is compound, its parentheses are stripped off). In the third line, the parameter occupies three positions (`'L4`) but the argument `Z` only takes one position. When `Z` is

substituted for `‘L4`, the source line becomes two positions shorter, which means that the rest of the line is moved two positions to the left. The assembler also preserves the original space between `L4` and `DEST`. As a result, the expanded line has one space between `BZ` and `DEST`.

2. The second line of the definition simply reads `L3`. The assembler replaces `L3` by the corresponding argument `SUB R1,L1` and, before trying to assemble it, scans the line again, looking for more occurrences of parameters. In our example it finds that the newly generated line has an occurrence of `L1` in it. This is immediately replaced by the argument corresponding to `L1` (`SUM`) and the line is then completely expanded. The process of macro expansion turns out to be a little more complex than originally described.
3. In line three of the definition the quote (’) separates the character `B` from the parameter `L4`. On expanding this line, the assembler treats the quote as a separator, removes it, and concatenates the argument `Z` to the character `B`, thus forming `BZ`. If `BZ` is a valid mnemonic, the line can be assembled. This is another example of a macro line that has no meaning before being completely expanded.
4. The argument corresponding to parameter `L5` is null. The result is the string `CD` with nothing between the `C` and the `D`. Again, if `CD` is a valid mnemonic (or directive), the line can eventually be assembled (or executed). Otherwise, the assembler flags it as an error.

Note that there is a difference between a null argument and an argument that is blank. In the expansion `C SUM,(D,T,U),(SUB␣R1,L1),Z,␣,SN`, the fifth argument is a blank space, which ends up being inserted between the `C` and the `D` in the expanded source line. The final result is `C␣D␣SN` which is not the same as `CD␣SN`. It could even be interpreted by the assembler as `C`=label, `D`=mnemonic, `SN`=operand. If a mnemonic `D` exists, the instruction would be assembled and `C` would be placed in the symbol table, without any error messages or warnings.

▸ **Exercise 11.4:** What if the last argument of an expansion is null? How can the assembler distinguish between a missing last argument and a null one?

A little thinking shows that the precise names of the formal parameters are not important. The parameters are only used when the macro is expanded. No parameter names are used after pass 0 (except that the original names should appear in the listing). As a result, the assembler replaces the parameter names with serial numbers when the macro is placed in the MDT. This makes it easier to locate occurrences of the parameters when the macro is later expanded. Thus in one of the examples above:

```
D  MACRO A,B,C
   LOD A
   ADD B
   STO C
   ENDM
```

macro `D` is stored in the MDT as: `D|3|␣␣LOD␣#1|␣␣ADD␣#2|␣␣STO␣#3▌`

The vertical bar | is used to separate source lines in the MDT. The ‘3’ in the second field indicates that the macro has three parameters, and #1, #2 etc., refer to the parameters. The bold vertical bar ▌ signals the end of the macro in the MDT. This editing slows down the assembler when handling a macro definition but it speeds up every expansion. Since a macro is defined once but may be expanded often, the advantage is clear.

It is now clear that three special characters are needed when macros are placed in the MDT. In general, the assembler should use characters that cannot appear in the original definition of the macro, and each assembler has a list of characters that are invalid inside macros, or are even prohibited anywhere in the source file.

The example above is simple since each parameter is a single letter. In the general case, a parameter may be a string of characters, called a *token*. The editing process mentioned earlier is done by breaking up each source line into tokens. A token is an indivisible unit of the line and is made up of a string of letters and digits, or a single special character. Each token is compared to all parameter names and, in case of a match, the token is replaced by its serial number. The example above may now be rewritten with the parameter

names changed to more than one letter.

```
D  MACRO AD,BCD,ST7
   LOD AD
   ADD BCD
   STO ST7
   ENDM
```

Let's follow the editing of the second line. It is broken up into the two tokens `ADD` and `BCD`, and each token is compared with all three parameter names. The first token '`ADD`' almost matches the first parameter '`AD`'. The second one '`BCD`' exactly matches the second parameter. That token is replaced by the serial number '#2' of the parameter, and the entire line—including the serial number—is stored in the MDT. Our new example will be stored in the MDT in exactly the same way as before, illustrating the fact that parameter names can be changed without affecting the meaning of the macro.

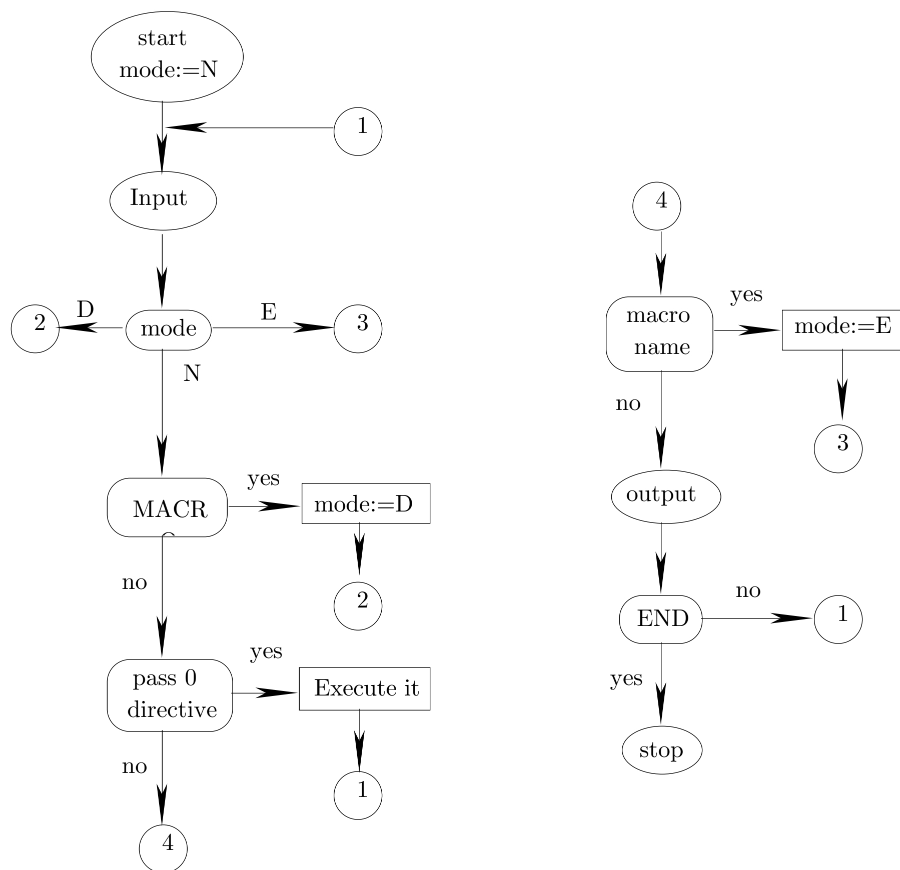



**Figure 11.2**a. Summary of Pass 0 (part I).

### 11.3 Pass 0

Pass 0 is devoted to handling macros. Macro definitions and expansions are fully done in pass 0, and the output of that pass, the new source file, contains no trace of any macros (except the listing information mentioned earlier). The file contains only instructions and directives, and is ready for pass 1. Pass 0 does not handle label definitions, does not use the symbol table, does not maintain the LC, and does not calculate the size of instructions. It should, however, execute certain directives that pertain to macros, and it has to handle two types of special symbols. The directives are called pass 0 directives, they have to do with

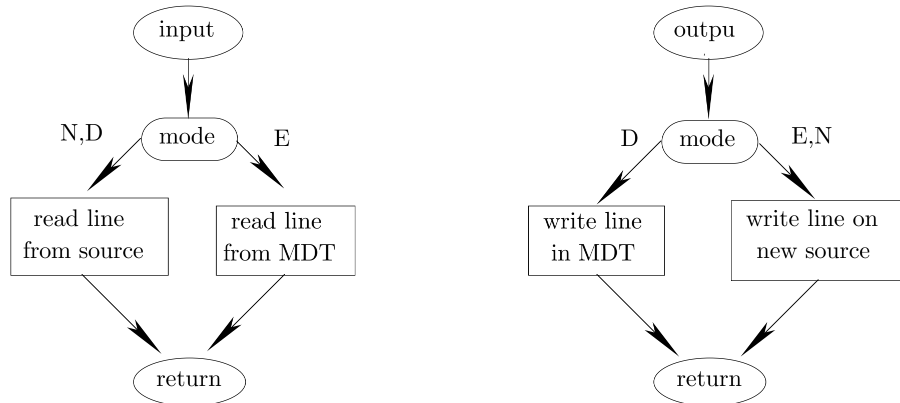



**Figure 11.2**b. Summary of Pass 0 (part II).

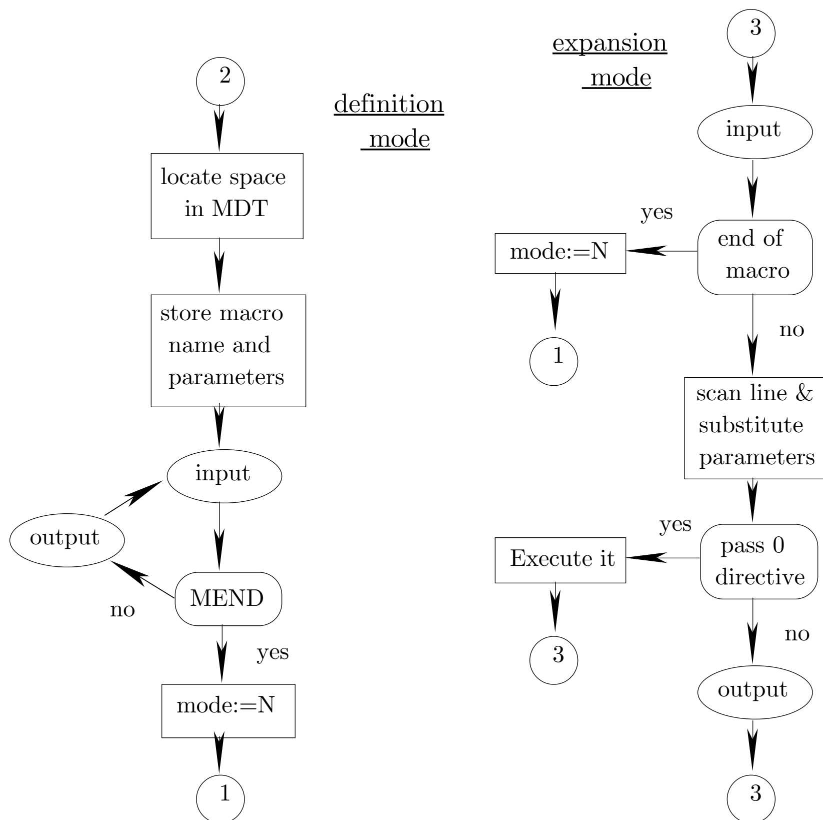



**Figure 11.2**c. Summary of Pass 0 (part III).

conditional assembly, and are explained in Section 11.8. The special symbols involved are `SET` symbols and sequence symbols. They are fully handled by pass 0, are stored in a temporary symbol table, and are discarded at the end of the pass. The following rules (and Figure 11.2) summarize the operations of pass 0.

1. Read the next source line.
2. If it is `MACRO`, read the entire macro definition and store in MDT. Goto 1.
3. If it is a pass 0 directive, execute it. Goto 1. Such a directive is written on the new source file but in a special way, not as a normal directive, since it is only needed for the listing in pass 2.

4. If it is a macro name, expand it by bringing in lines from the MDT, substituting parameters, and writing each line on the new source file (or executing it if it a pass 0 directive). Goto 1.
5. In any other case, write the line on the new source file. Goto 1
6. If current line was the `END` directive, stop (end of pass 0).

To implement those rules, the concept of an operating mode is introduced. The assembler can be in one of three modes: the normal mode (N)—in which it reads lines from the source file and writes them to the new source file; the macro definition mode (D)—in which it reads lines from the source file and writes them into the MDT; and the macro expansion mode (E)—in which it reads lines from the MDT, substitutes parameters, and writes the lines on the new source file. A fourth mode (DE) is introduced in Section 11.6 in connection with nested macros.

▸ **Exercise 11.5:** Normally, the definition of a macro must precede any expansions of it. If we eliminate that restriction, what modifications do we have to make to the assembler?

## 11.4 MDT Organization

Most computers have operating systems (OS) that provide supervision and offer services to the users. The date and time are two examples of such services. To use such a service (e.g., to ask the OS for the date) the user has to place a request to the OS. Since the typical user cannot be expected to be familiar with the OS, the OS provides built-in macros, called *system macros*. To get the date, for example, the user should write a source line such as `DATE D` where `DATE` is a system macro, and `D` is an array in which `DATE` stores the date. As a result, the MDT does not start empty. When pass 0 starts, the MDT contains all the system macros. As more macro definitions are added to the MDT, it grows larger and larger and the problem of efficient search becomes more and more important. The MDT should be organized to allow for efficient search, and most MDTs are organized in one of two ways: as a *chained* MDT, or as an array where each macro is pointed to from a Macro Name Table (MNT).

A chained MDT is a long array containing all the macro definitions, linked with backwards pointers. Each definition points to its predecessor and each is made up of individual fields separated by a special character that we will denote by |. A typical definition contains:

| ··· | ❙ name | pointer | # of params | first line | ··· | last line ❙ | name | pointer | ··· |
|---|---|---|---|---|---|---|---|---|---|

where the last separator ❙ is immediately followed by the name of the next macro. Such an MDT is easy to search by following the pointers and comparing names. Since the pointers point backwards, the table is searched from the end to the beginning; an important feature. It guarantees that when a multiply-defined macro is expanded, the back search will always find the last definition of the macro. Multiply-defined macros are a result of nested macro definitions(Section 11.9), or of macros that users write to supersede system macros. In either case, it is reasonable to require that the most recent definition always be used. The advantage of this organization is its flexibility. It is only limited by one parameter, the size of the array. The total size of all the definitions cannot exceed this parameter. Thus we can define a few long macros or many short ones.

The other way to organize the MDT is to store the macros in an MDT array with separators as described before, but without pointers. An additional array, called the MNT, contains pairs `<macro name, pointer>` where the pointers point to the start of a definition in the MDT array. The advantage of this organization is that the MNT has fixed size entries, allowing for a faster search of names. However, the total amount of macros that can be stored in such an MDT is limited by two parameters. The size of the MDT array—which limits the total size of all the macros—and the size of the MNT—which limits the number of macros that can be defined. Figure 11.3 is an example of such an MDT organization. It shows an MDT array with 3 macros. The first has 3 parameters, the second, 4, and the third, 2. The MNT array has fixed-size entries.

### 11.4.1 The `REMOVE` Directive

Regardless of the way the MDT is organized, if the assembler supports system macros, it should also support a directive to remove a macro from the MDT. A user writing a macro to redefine an existing system macro may want the redefinition to be temporary. They define their macro, expand it as many times as necessary,

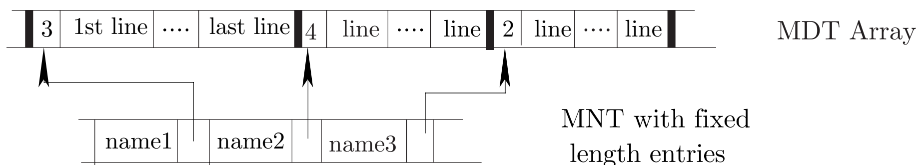



**Figure 11.3**. MNT–MDT structure

and then remove it such that the original system macro can be used again. Such a directive is typically called `REMOVE`.

In old assemblers this directive is executed by removing the pointer that points to the macro, not the macro itself. Removing the macro itself and freeing the space in the MDT is done by many new assemblers (Chapter 10). It is done by changing the macro definition to a null string, and storing another (smaller) macro in the space thus freed. After defining and removing many macros, the MDT becomes fragmented; it can be defragmented by moving macros around and changing pointers in the MNT.

▸ **Exercise 11.6:** What could be a practical example justifying the actual removal of a macro from the MDT?

### 11.4.2 Searching the MDT

Pass 0 has to identify each source line as either a macro name, a pass 0 directive, or something else. To achieve quick identification, the pass 0!directives should be stored in the MDT with a special flag, identifying them as directives. This way, only the MDT has to be searched in pass 0.

An assembler combining passes 0 and 1 has to do more searching. Each source line has to be identified as either an instruction (its size should be determined), as a pass 0 or pass 1 directive (to be executed), as a pass 2 directive (to be written on the intermediate file), or as a macro name (to be expanded).

One approach is to search the opcode table first (it should contain all the mnemonics and directives) and the MDT next, the idea being that most source lines are either instructions or directives, macro expansions are relatively rare.

The alternative approach is to search the MDT first and the opcode table next. This way, the programmer can define macros which redefine existing instructions or directives. If this approach is used, the MDT must be designed to allow for quick search.

## 11.5 Other Features of Macros

### 11.5.1 Macro Parameters And Arguments

Associating macro parameters with their actual arguments can be done in three ways. By position, by name, and by numeric position in the argument list. The first method is the simplest and most common. If the macro is defined as `M MACRO P1,P2,P3` then the expansion `M ADA,JON,YON` obviously associates the actual value `ADA` with the first parameter `P1`, the actual value `JON` with the second parameter `P2`, etc. The expansion `M ADA,,NON` has an empty second argument, and the second parameter `P2` is therefore bound to a null value.

Associating by name, however, is different. Using the macro definition above, we can have an expansion `M P2=DON,P1=SON,P3=YON`. Here each of the three actual arguments `SON, DON, YON` is explicitly associated with one of the parameters.

▸ **Exercise 11.7:** What is the meaning, if any, of the expansion
`M P2=DON,P1=DON,P3=DON`?

It is possible to combine the two methods, such as in `M P3=MAN,DAN,P1=JAN`. Here the second argument has no name association and it therefore corresponds to the second parameter. This, of course, implies that an expansion such as `M P2=MAN,DAN,P1=JAN`, is wrong.

▸ **Exercise 11.8:** There is, however, a way to assign such an expansion a unique meaning. What is it?

The third method uses a special parameter named `SYSLIST` such that `SYSLIST(i)` refers to the ith

argument of the current macro expansion. A possible definition is

```
M  MACRO    no parameters are declared.
   LOD SYSLIST(2)
   STO SYSLIST(1)
   ENDM
```

The expansion `M X,Y` will generate

```
LOD Y
STO X
```

### 11.5.2 Delimiting Macro Parameters

In all the examples shown earlier, macro parameters were delimited by a comma ','. It is possible, however, to use other characters as delimiters, or to use a scheme where parameters can be delimited in a general way. Such a scheme is used by the TEX typesetting system that has many features of a programming language, although its main task is to set type.

In TEX, a macro can be defined by `\def\xyz#1.#2.␣{...}`. This defines a macro `xyz` with two parameters. The first is delimited by a period '.', and the second, by a period and a space '`.␣`'. In the expansion
`\xyz-12.45,=a98.62.␣abc...`
the first parameter would be bound to '`-12`', and the second, to '`45,=a98.62`'. Note that the comma is part of the second actual argument, not a delimiter. Also, the period in '`98.62`' is considered part of the second argument, not a delimiter.

▸ **Exercise 11.9:** What happens if the user forgets the period-space?

### 11.5.3 Numeric Values of Arguments

5. Macro arguments are normally treated as strings. However, MACRO, the VAX assembler, can optionally use the value, rather than the name, of an argument. This is specified by means of a '\'. A simple example is:

```
.MACRO  CLEER ARG
CLRL    R'ARG
.ENDM
```

After defining this macro, we assign `CONS=5`, and expand `CLEER` twice. The expansion `CLEER CONS` generates `CLRL RCONS` (which is probably wrong), whereas the expansion `CLEER \CONS` generates `CLRL R5`.

### 11.5.4 Attributes of Macro Arguments

Each argument in a macro expansion has attributes that can be used to make decisions—inside the macro definition—each time the macro is expanded. At the time the macro definition is written, the arguments are unknown. They only become known when the macro is expanded, and may have different attributes each time the macro is expanded.

▸ **Exercise 11.10:** Chapter 10 mentions attributes of symbols. What is the difference between attributes of symbols and of macro arguments?

We will discuss the six attributes supported by the IBM 360 assembler and will use, as an example, the simple macro definition:

```
M  MACRO P1
   P1
   ENDM
```

followed by the three expansions:

```
M FIRST
M SEC
M (X,Y,Z)    the argument is compound
```

We assume that symbols `FIRST`, `SEC` are defined by:

- FIRST DC P'+1.25'. DC is the Define Code directive, and symbol FIRST is the name of a packed decimal constant.

- SEC ADD 5,OP. Symbol SEC is the label of an ADD instruction

The example illustrates the following attributes:

- The count attribute, K, is the length of the actual argument. Thus K'P1 is 5 in the first expansion, 3 in the second one, and 7, in the third.

- The type attribute, T, is the type of the actual argument. In the first expansion it is 'P' (for Packed decimal) and in the second, 'I' (since the argument is an Instruction). In the third expansion the type is 'N' (a self-defined term).

- The length attribute, L, is the number of bytes occupied by the argument in memory; 2 in the first case, since the packed decimal constant 1.25 occupies two bytes, and 4 in the second case, since the ADD instruction takes 4 bytes. The compound argument of the third expansion has no length attribute.

- The integer attribute, I, is the number of decimal digits in the integer part of the argument; 1 in the first expansion, 0 in the second.

- The scaling attribute, S, is the number of decimal digits in the fractional part of the argument; 2 in the first example and 0 in the second one.

- The number attribute, N only has a meaning if the argument is compound. It specifies the number of elements in the compound argument. In the third example above N'P1 is 3.

The attributes can be used in the macro itself and the most common examples involve conditional assembly (Section 11.8).

### 11.5.5 Directives Related to Arguments

MACRO, the VAX assembler, supports several interesting arguments that make it easy to write sophisticated macros. Here are a few:

- The .NARG directive provides the number of actual arguments. There may be fewer arguments than parameters. Its format is .NARG symbol and it return the number of arguments in the symbol. Thus macro Block

```
.MACRO  Block A,B,C,D
.NARG   Num
 ...
.BLKW   Num
.ENDM
```

creates a block of 1–4 words each time it is expanded, depends on the number of arguments in the expansion.

- The directive .NCHR returns the size of a character string. Its format is .NCHR symbol,<string>. Thus after defining:

```
    .MACRO  Exmpl L,S
    .NCHR   Size,S
     ...
L:  .BLKW   Size
    .ENDM
```

the expansion Exmpl M,<Yours T> will generate M: .BLKW 7

- The IF-ELSE-ENDIF directive can be used to determine whether an argument is BLANK or NOT_BLANK, or whether two arguments are IDENTICAL or DIFFERENT. It can also be used outside macros to compare any quantities known in pass 0, and to determine if a pass 0 quantity is defined or not.

### 11.5.6 Default Arguments

Some assemblers allow a definition such as N MACRO M1,M2=Z,M3, meaning that if the actual argument binding M2 is null in a certain expansion, then M2 will be bound to Z by default.

▸ **Exercise 11.11:** What is the meaning of N MACRO M1,M2=,M3?

### 11.5.7 Automatic Label Generation

When the definition of a macro contains a label, successive expansions will result in the label being multiply-defined.

```
L    MACRO
     P1 ..
     BNZ G12    branch on non-zero to label G12
A    LOD ..     just a line with a label
     .
     .
G12  STO ..
     ENDM
```

Each time this macro is expanded, symbols `A`, `G12` will be defined. As a result, the second and subsequent expansions will cause assembler errors.

Most assemblers offer help in the form of automatically generated unique names. They are called local labels or automatic labels. Here are two examples, the first one from the MPW assembler for the Macintosh computer. The two lines with labels in the above definition can be written as:

```
A    SYSNDX LOD ...
G12  SYSNDX STO ...
```

and every time the macro is expanded, the assembler will append a suffix of the form `00001`, `00002`,... to any label generated. This way all labels generated are unique. In our example they will be `A00001`, `G1200002`, `A00003`, ...

The second example is from MACRO, the VAX assembler. When a local label is needed in a macro, a parameters should be added preceded by a '?'. Thus in:

```
       .MACRO  ABS A,B,?NEG,?DONE
       TSTL    A
       BLSS    NEG
       MOVL    A,B
       BRB     DONE
NEG:   MNEGL   A,B
DONE:  .ENDM
```

### 11.5.8 The IRP Directive

A pair of `IRP` directives, placed inside a macro, define a sequence of lines. They direct the assembler to repeatedly duplicate and assemble the sequence a number of times determined by a compound parameter. Here is an example from the MPW assembler for the Macintosh computer:

```
MAC  MACRO P1,P2   P1 should be a compound parameter.
     IRP P1
     ADD P1        this will be repeated for every component of P1
     IRP
     .
     .
     ENDM
```

The sequence to be duplicated consists of the single instruction `ADD`. Each time it is duplicated, one of the components of the compound parameter `P1` is selected. The expansion:

```
MAC (A,B,#3),H   will generate
ADD A
ADD B
ADD #3
.
.
```

Here is another IRP example, from MACRO, the VAX assembler.

```
.MACRO  CallProc Name,A,B,C,D
.NARG   Num
.IRP    Temp,<D,C,B,A>
.IIF    NOT_BLANK,Temp, PUSHL Temp
.ENDR
CALLS   #<Num-1>,Name
.ENDM
```

The IRP directive loops 4 times, assigning the actual arguments of D, C, B, A to Temp. The .IIF directive generates a PUSHL instruction, to push the argument's address on the stack, for any non-blank argument. Finally, a CALLS instruction is generated, to preform the actual procedure call. This is a handy macro that can be used to call procedures with up to 4 parameters.

IRP stands for Indefinite RePeat.

### 11.5.9 The PRINT Directive

When a program contains many long macros that are expanded often, the programmer may not want to see all the expansions listed in the printed output. The PRINT directive may be used to suppress listing of macro expansions (PRINT NOGEN) or to turn on such listings (PRINT GEN). This directive does not affect the listing of the macro definitions or of the body of the program. Those listings are controlled by the LIST, NOLIST directives.

▸ **Exercise 11.12:** Is the PRINT NOGEN itself listed?

MACRO, the VAX assembler, supports the similar directives .SHOW and .NOSHOW. Thus one can write, for example, .NOSHOW ME to suppress listings of all macro expansions (ME stands for Macro Expansion), or .SHOW MEB (where MEB stands for Macro Expansion Binary) to list just lines that actually create binary code generated during macro expansions.

### 11.5.10 Comment Lines in Macros

If a macro definition contains comment lines such as in:

```
  MACRO A,B,C
* WATCH OUT, PARAMETER B IS SPECIAL
  .
  .
  C R1
* THE PREVIOUS LINE CHANGES ITS MEANING
  .
  .
```

The comments should be printed, together with the definition of the macro, in the listing file, but should they also be printed with each expansion? The most general answer is: It depends. Some comments refer to the lines in the body of the macro and should be printed each time an expansion is printed (as mentioned elsewhere, the printing of macro expansions is optional). Other comments refer to the formal parameters of the macro, and should be printed only when the macro definition is printed. The decision should be made by the programmer, which means that the assembler should have two types of comment lines, the regular type, which is indicated by an asterisk, and the special type, indicated by another character, such as a '!', for comments that should be printed only as part of a macro definition.

## 11.6 Nested Macros

Another useful feature of macros is the ability to nest them. This can be done in two ways: Nested macro definition and nested macro expansion. We will discuss the latter first.

### 11.6.1 Nested Macro Expansion

This is the case where the expansion of one macro causes another macro (or even more than one macro) to be expanded. Example:

```
C  MACRO
   COMP
   JMP
   ENDM
A  MACRO
   ADD
   C
   SUB
   ENDM
B  MACRO
   LOD
   A
   STO
   ENDM
```

There is nothing unusual about macro `C`. An expansion of macro `A`, however, is special since it involves an expansion of `C`. An expansion of `B` is even more involved. Most assemblers support nested macro expansion since it is useful and also easy to implement. They allow it up to a certain maximum depth. In our example, the expansion of `B` turns out to be nested to a depth of 2. The expansion of `C` is nested to a depth of 0.

▸ **Exercise 11.13:** Why is nested macro expansion useful?

To understand how this feature is implemented, we expand macro `B` in the example above. Keep in mind that this is done in pass 0. The assembler locates `B` in the MDT and switches to the macro expansion mode. In that mode, it fetches the source lines from the MDT instead of from the source file. Otherwise, that mode is very similar to the normal mode. Each line is identified as either the name of a macro, a pass 1 directive, or something else. If the line is the name of a macro, the assembler suspends the expansion of `B`, fully expands the new macro, and then returns to complete `B`'s expansion. If the line is a pass 0 directive, the assembler executes it. If the line is something else, the assembler scans it, substitutes parameters, and writes the line on the new source file. During this process the assembler maintains a pointer that always points to the current line in the MDT.

When the expansion of `B` starts, the macro is located in the MDT, and a pointer is set to point to its first line. That line (`LOD`) is fetched and identified as an instruction. It is written on the new source file, and the pointer is updated to point to the second line (`A`). The second line is then fetched and is identified as a macro name. The assembler then suspends the expansion of `B` and starts expanding `A` by performing the following steps:

- It locates macro `A` in the macro definition table
- It sets a new pointer to the first line of macro `A`.
- It saves the values of the actual arguments of macro `B`.

From then on, macro `A` is expanded in the usual way until its second line (`C`) is fetched. At that point the new pointer points to that line. The assembler suspends the expansion of `A` and starts the expansion of macro `C` by performing three steps as above. While expanding macro `C`, the assembler uses a third pointer and, since `C` is not nested, the expansion terminates normally and the third pointer is discarded. At that point the assembler returns to the expansion of macro `A` and resumes using the second pointer (which should be incremented to point to the next waiting line of `A`). When the expansion of `A` is completed, the assembler

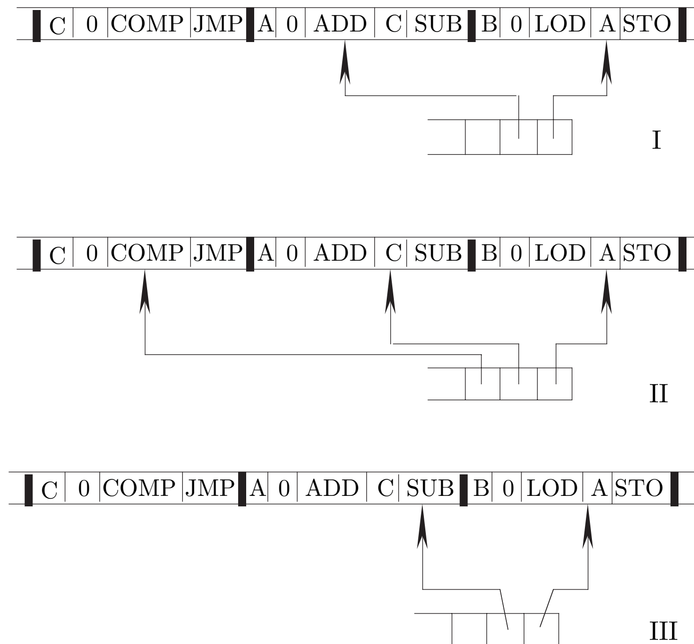



**Figure 11.4**. Three typical steps in a nested macro expansion

discards the second pointer and switches to the first one—which means resuming the expansion of the original macro `B`. Three typical steps in this process are shown in Figure 11.4.

In part I, the second line of `B` has been fetched and the (first) pointer points to that line. The expansion of macro `A` has just started and the second pointer points to the first line of `A`.

In part II, the second pointer points to the second line of `A`. This means that the line being processed is the second line of `A`. The expansion of `C` has just started.

In part III, the expansion of `C` has been completed, the third pointer discarded, and the assembler is in the process of fetching the third line of `A`.

The rules for nested macro expansion therefore are:

▪ In the macro expansion mode, when encountering the name of a macro, find it in the MDT, set up a new pointer to point to the first line, save the arguments of the current macro, and continue expanding, using the new pointer.

▪ After fetching and expanding the last source line of a macro, discard the current pointer and start using the previous one (and the previous set of arguments).

▪ If there is no previous pointer, the (nested) macro expansion is over.

From this discussion it is clear that the pointers are used in a Last In First Out (LIFO) order, and should thus be stored in a *stack*. This stack is called the *macro expansion stack* (MES), and its size determines the number of possible pointers and thus the maximum depth of nesting.

Implementing nested macro expansions is, therefore, done by declaring a stack and using it while expanding a macro. All other details of the expansion remain the same as in a normal expansion (however, see conditional assembly, Section 11.8).

The following is a set of rules and a flow chart (Figure 11.5, a generalized subset of Figure 11.2) which illustrate the main operations of a pass 0 supporting nested macro expansions.

1. Input line from MDT (since mode=E).
2. If it is a pass 0 directive, execute it. Goto 1.
3. If is is a macro name, start a new expansion. Goto 1.
4. If it is an end-of-macro character, stop current expansion and look back in MES.
   - If MES empty, change mode to N. Goto main input.

- Else—start using previous pointer in MES, remain in E mode. Goto 1.

5. Line is something else, substitute parameters and write line on new source file. Goto 1.

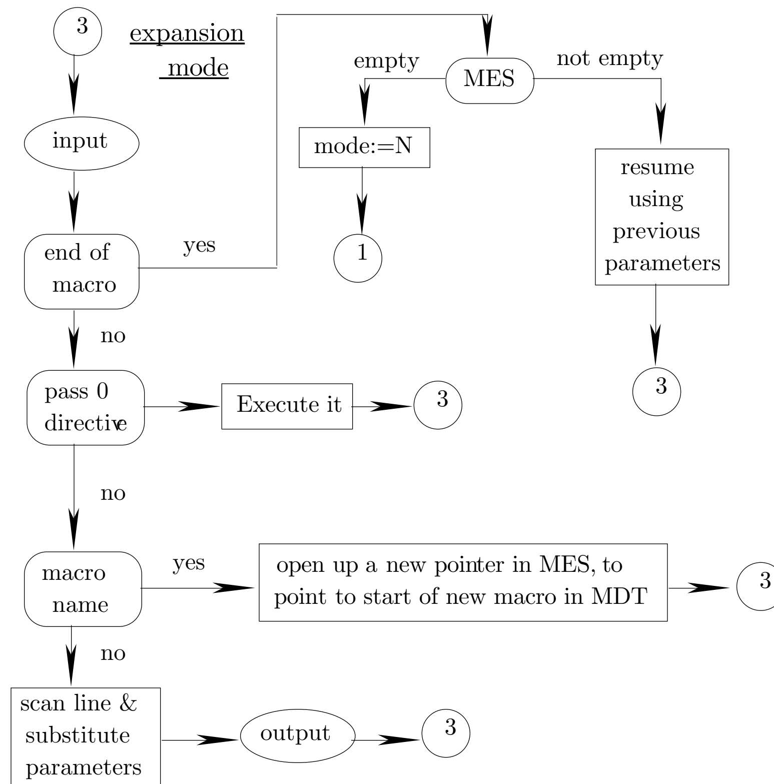



**Figure 11.5**. A Summary of pass 0 with nested macro expansions.

## 11.7 Recursive Macros

A very important special case of nested macro expansion is the case where a macro expands itself. Macro `NOGOOD` below is such an example but, as its name implies, not a good one.

```
NOGOOD  MACRO
        INST1
        NOGOOD
        INST2
        ENDM
```

An attempt to expand this macro will generate `INST1` and will then start the inner expansion. The inner expansion will do the same. It will generate `INST1` and start a third expansion. There is nothing in this macro to stop any of the inner expansions and go back to complete the outer ones. Such an expansion will very quickly overflow the MES, no matter how large it is.

It turns out, though, that such macros, called recursive macros, are very useful. To implement such a macro, a mechanism is necessary that will stop the recursion (the self expansion) at some point. Such a mechanism is supported by many assemblers and is called *conditional assembly*.

## 11.8 Conditional Assembly

This is a general feature that is not limited to macros. If an assembler supports this feature, then it can be used throughout the program. Its main use, however, happens to be inside macros. In the case of a recursive macro, conditional assembly is mandatory. In order to stop a recursive expansion, a quantity is needed that will have different values for each inner expansion. By updating this quantity each time an expansion starts, the programmer can stop the recursive expansion when that quantity satisfies a certain condition.

Such a quantity must have two attributes. It must have a value in pass 0 at assembly time (and thus it cannot be the contents of any register or of memory; they only get defined at run time) and the assembler should be able to change its value (therefore it cannot be a constant or a regular symbol; they have fixed values). Such a quantity must therefore be either an actual argument of a macro or a new feature, and that feature is called a `SET` symbol.

`SET` symbols are special symbols that can be defined and redefined by the `SET` directive, which is very similar to the `EQU` directive. The differences between `SET` and `EQU` symbols are:

- `SET` symbols are only used in pass 0 to implement conditional assembly; `EQU` symbols are defined in pass 1 and used in pass 2. The `SET` directive is thus a pass 0 directive.

- `SET` symbols can be redefined. Thus it is permissible to have `Q SET 0` followed by `Q SET 1` or by `Q SET Q+1`. `Q` is a `SET` symbol and its value can be changed and used in pass 0. The `SET` directive is the only way to redefine symbols (there is, however, a `MICCNT` directive used by some assemblers) but it should be emphasized that `SET` symbols are different from other symbols. They are stored in a temporary symbol table (the main symbol table does not exist in pass 0) and are discarded at the end of pass 0. The recursive macro above can now be written:

```
NOGOOD  MACRO
        INST1
Q       SET Q+1
        NOGOOD
        INST2
        ENDM
```

Before such a macro is expanded, `Q` needs to be assigned a value with a `SET` directive. Assigning a value to `Q` in any other way would make it a fixed symbol. The two source lines:

```
Q  SET 1
   NOGOOD
```

will start a recursive expansion where, in each step, the value of `Q` (in the temporary symbol table) will be incremented by 1. This expansion is still an infinite one and, in order to stop it after `N` steps, a test is necessary, to compare the value of `Q` to `N`. This test is another pass 0 directive, typically called `AIF` (Assembler IF). Its general form is `AIF exp.symbol`, where `exp` is a boolean expression containing only quantities known to the assembler. The assembler evaluates the expression and, if its value is true, the assembler goes to the line labeled `.symbol`.

The next version of the same macro is now:

```
GOOD  MACRO
      INST1
Q     SET Q+1
      AIF Q=N.F    if Q equals N then go to line labeled .F
      GOOD
.F    INST2
      ENDM
```

An expansion such as

```
N EQU 2
Q SET 0
  GOOD
```

will generate the following:

```
1.     INST1     generated for the first time
2.  Q  SET Q+1   sets Q to 1 in the temp. symbol table
3.     AIF Q=N.F Q not equal N so do not go to .F
4.     GOOD      start expanding next level
5.     INST1     generated for the second time
6.  Q  SET Q+1   sets Q to 2 in the temp. symbol table
7.     AIF Q=N.F Q equals 2 so go to .F
8.     INST2     generated for the first time. Notice that .F
9.     INST2     does not appear since it is not a regular symbol
```

Of the 9 lines expanded, lines 2, 3, 4, 6, 7 are pass 0 directives. They are executed and do not appear in the final program. Lines 1, 5, 8, 9 are instructions, and are the only ones actually expanded and written onto the new source file.

The macro has been recursively expanded to a depth of 2 because of the way symbols `Q`, `N` have been defined. It is also possible to say `AIF Q=2.F`, in which case the depth of recursion will depend only on the initial value of `Q`.

▸ **Exercise 11.14:** Is it valid to write `AIF Q>N.F`?

An important question that should be raised at this point is: what can `N` be? Clearly it can be anything known in pass 0, such as another (non-future) `SET` symbol, a constant, or an actual argument of the current macro. This, however, offers only a limited choice and some assemblers allow `N` to be an absolute symbol, defined by an `EQU`. An `EQU`, however, is a pass 1 directive, so such an assembler should look at each `EQU` directive in pass 0 and—if the `EQU` defines an absolute symbol—execute it, store the symbol in the temporary symbol table, and write the `EQU` on the new source file for a second execution in pass 1. In pass 1, all `EQU` directives are executed and all `EQU` symbols, absolute and relative, end up in the permanent symbol table.

Some assemblers go one more step and combine pass 0 with pass 1. This makes for a very complex pass but, on the other hand, it means that conditional assembly directives can use any quantities known in pass 1. They can use the values of any symbols in the symbol table (i.e., any non-future symbols), the value of the LC, and other things known in pass 1 such as the size of the last instruction read. The following is an example along these lines.

```
   .
   .
P  DS 10    P is the start address of an array of length 10
N  DS 3     N is the start address of an array of length 3 immediately
   .        following array P. Thus N=P+10
   .
Q  SET P    The value of Q is an address
   GOOD     depth of recursion will be 10 since
   .        it takes 10 steps to increment the
   .        value of Q from address P to address N.
```

The next example is more practical. It is a recursive macro `FACT` that calculates a factorial. Calculating a factorial is a common process and is done by computers all the time. In our example, however, it is done at *assembly time*, not at run time.

```
FACT  MACRO N
S     SET S+1
K     SET K*S
      AIF S=N.DON
      FACT N
.DON  ENDM
```

The expansion:

```
S  SET 0
K  SET 1
   FACT 4
```

will calculate 4! (=24) and store it in the temporary symbol table as the value of the SET symbol K. The result can only be used at assembly time, since the SET symbols are wiped out at the end of pass 0. Symbol K could be used, for example, in an array declaration such as: FACT4 DS K which declares FACT4 as an array of length 4!. This, of course, can only be done if the assembler combines passes 0 and 1.

The FACT macro is rewritten below using the IIF directive. IIF stands for *Immediate IF*. It has the form IIF condition,source line. If the condition is true, the source line is expanded, otherwise, it is ignored.

```
FACT  MACRO N
S     SET S+1
K     SET K*S
      IIF S≠N,(FACT N)
      ENDM
```

Some assemblers support directives such as:

- IFT. The general form is IFT cond. If the condition is true, the assembler skips the next source line.
- IFF. The general form is IFF cond. This is the opposite of IFT.
- EXIT. This terminates the present expansion. It is used inside an IF to terminate the expansion early if a certain condition is true.
- IF1, IF2, ENDC. These exist on the MASM assembler for the IBM PC. Anything between an IF1 and an ENDC will be done (assembled or executed) in pass 1. Similarly for IF2. These directives are supported by the MASM assembler.
- IF-ELSE-ENDIF. These can only appear together and they extend the simple IF into an IF-THEN-ELSE construct. Example:

```
IF X=2
 line 1
ELSE
 line 2
 line 3
ENDIF
```

If X=2, then line 1 will be expanded, otherwise, lines 2 and 3 will be expanded.

▸ **Exercise 11.15:** What can X be?

The IBM 360, 370 assemblers, were the first ones to offer an extensive conditional assembly facility, and similar facilities are featured by most modern assemblers. It is interesting to note that the MPW assembler for the Macintosh computer supports conditional assembly directives that are almost identical to those of the old 360. Some of the conditional assembly features supported by the 360, 370 assemblers will be discussed here, with examples. Notice that those assemblers require all macro parameters and all SET symbols to start with an ampersand '&'.

The AIF directive on those assemblers has the format AIF (exp)SeqSymbol. The expression may contain SET symbols, absolute EQU symbols, constants, attributes of arguments, arithmetic operators, the six relationals (EQ NE GT LT GE LE), the logical operators AND OR NOT, and parentheses. SeqSymbol (sequence symbol) is a symbol that starts with a period. Such a symbol is a special one, it has no value and is not stored in any symbol table. It is used only for conditional assembly and thus only exists in pass 0. When the assembler executes an AIF and decides to go to, say, symbol .F, it searches, in the MDT, for a line labeled .F, sets the current MES pointer to point to that line, and continues the macro expansion. If a line fetched from the MDT has such a label, the line goes on the new source file without the label, which guarantees that only pass 0 will see the sequence symbols. In contrast, regular symbols (address symbols and absolute symbols) do not participate in pass 0. They are defined in pass 1, and their values used in pass 2.

**Examples:**

- AIF (&A(&I) EQ 'ABC').TGT where &A is a compound parameter and &I is a SET symbol used to select one component of &A.

▪ AIF (T'&X EQ O).KL if the type attribute of argument &X is O, meaning a null argument, then go to symbol .KL.

▪ AIF (&J GE 8).HJ where &J could be either a parameter or a SET symbol.

▪ AIF (&X AND &B EQ '(' ).LL where &X is a B type SET symbol (see below) and &B is either a C type SET symbol or a parameter.

The AGO Directive: The general format is 'AGO SeqSymbol'. It directs the assembler to the line labeled by the symbol. (This is an unconditional goto at assembly time, not run time.)

The ANOP (Assembler No OPeration) directive. The assembler does nothing in response to this directive, and its only use is to provide a line that can be labeled with a sequence symbol. This directive is used in one of the examples at the end of this chapter.

SET symbols. They can be of three types. A (arithmetic), B (boolean) or C (character). An A type SET symbol has an integer value. B type have boolean values of true/false or 1/0. The value of a C type SET symbol is a string.

Any SET symbol has to be declared as one of the three types and its type cannot be changed. Thus:

```
LCLA &A,&B
LCLB &C,&D
LCLC &E,&F
```

declare the six symbols as local SET symbols of the appropriate types. A local SET symbol is only known inside the macro in which it is declared (or inside a *control section*, but those will not be discussed here). There are also three directives to assign values to the different types of SET symbols.

```
&A SETA   1
&A SETA   &A+1
&B SETA   1+(B'1011'*X'FF1'-15)/&A-N'SYSLIST(&A)  where B'1011' is a binary constant,
            X'FF1' is a hex constant, and N' is the number attribute
            (the number of components) of the second argument of the current
            expansion (the second, because &A=2).
&C SETB   (&A LT 5)
&D SETB   1  means 'true'
&E SETC   ''  the null string
&E SETC   '12'  the string '12'
&F SETC   '34'
&F SETC   '0&E&F.5'  the string '012345'. The dot separates the value of &F from the 5
&E SETC   'ABCDEF'(2,3)  the string 'BCD'. Substring notation is allowed.
```

### 11.8.1 Global SET Symbols

The three directives GBLA, GBLB, GBLC declare global SET symbols. This feature is very similar to the COMMON statement in Fortran. Once a symbol has been declared global in one macro definition, it can be declared

global in other macro definitions that follow, and all those declarations will use the same symbol.

```
N1  MACRO
    GBLA &S
&S  SETA 1
    AR &S,2
    N2
    ENDM
N2  MACRO
    LCLA &S
&S  SETA 2
    SR &S,2  the local symbol is used
    N3
    ENDM
N3  MACRO
    GBLA &S
    CR &S,2  the global symbol is used
    ENDM
```

The expansion `N1` will generate

```
AR 1,2
SR 2,2
CR 1,2
```

### 11.8.2 Array SET Symbols

Declarations such as `LCLA &A(7)` are allowed, and generate a `SET` symbol which is an array. Such symbols can be very useful in sophisticated applications.

The following examples summarize many of the features of conditional assembly discussed here.

```
SUM   MACRO &L,&P1,&P2,&P3
      LCLA &SS
&SS   SETA 1
&L    ST&P1 5,TEMP            save register 5 in temp
      L&P1 5,&P2(1)           load first component of P2
.B    AIF (&SS GE N'&P2).F    if done go to .F
&SS   SETA &SS+1              use &SS as a loop index
      A&P1 5,&P2(&SS)         add next component of P2
      AGO .B                  loop
.F    ST&P1 5,&P3             store sum in P3
      L&P1 5,TEMP             restore register 5
      MEND
```

The expansion `SUM LAB,D,(A,B,C),DEST` will generate

```
LAB  STD 5,TEMP
     LD 5,A
     AD 5,B
     AD 5,C
     STD 5,DEST
     LD 5,TEMP
```

This macro uses the conditional assembly directives to loop and generate several `AD` (Add Double) instructions. The number of instructions generated equals the number of components (the N attribute) of the third argument.

A more sophisticated version of this macro lets the user specify, in argument `®`, which register to use. If argument `®` is omitted (its T attribute equals O) the macro selects register 5.

```
SUM   MACRO &L,&P1,&P2,&P3,&REG
      LCLC &CC
      LCLC &RR
      LCLA &AA
&AA   SETA 1
&CC   SETC '&L'
&RR   SETC '&REG'
      AIF (T'&REG NE 'O').Q      is argument &REG a null?
&RR   SETC '5'                   yes, use register 5
&CC   ST&P1 &RR,TEMP             and label this instruction
&CC   SETC ' '
.Q    ANOP                       a source line just to have a label
&CC   L&P1 &RR,&P2(1)            this inst. is labeled if &REG is not null
.B    AIF (&AA GE N'&P2).F       if done, go to .F
&AA   SETA &AA+1
      A&P1 &RR,&P2(&AA)          the main ADD instr. to be generated
      AGO .B                     loop
.F    ST&P1 &RR,&P3
      AIF (T'&REG NE 'O').H      same text as above
      L&P1 &RR,TEMP              restore the register
.H    ENDM
```

The expansion `SUM LAB,D,(A,B,C),DEST,` generates code identical to the one generated for the previous example. The expansion `SUM LAB,D,(A,B,C),DEST,3` will generate similar code, the only difference being that register 3 will be used instead of register 5.

## 11.9 Nested Macro Definition

This is the case where the definition of one macro contains the definition of another. Example:

```
X  MACRO
   MULT    the body of X starts here...
Y  MACRO
   ADD
   JMP
   ENDM
   DIV     and ends here. It is 6 lines long
   ENDM
```

The definition of macro `Y` is nested inside the definition of `X`. This feature is not as useful as nested macro expansion but many modern assemblers support it (older assemblers typically did not). In this section we will see what this feature means, and look at two different ways of implementing it.

The first thing that needs to be modified, in order to implement this feature, is the macro definition mode. In this mode, the assembler reads source lines and stores them in the MDT until it encounters an `ENDM`. The first modification has to do with the `MACRO` directive itself. When the assembler reads a line with a `MACRO` directive while it is in the macro definition mode, it treats it as any other line (i.e., it stores it in the MDT) but it also increments a special counter, the *definition level counter*, by 1. This counter starts at 1 when entering definition mode, and is updated to reflect the current level of nested definitions. When the assembler encounters an `ENDM` directive, it decrements the counter by 1 and tests it. If the counter is positive, the assembler treats the `ENDM` as any other line. If the counter is zero, the assembler knows that the `ENDM` signals the end of the entire nested definition, and it switches back to the normal mode. This process is described in more detail below.

In our example, `X` will end up in the MDT as the 6-line macro

| X | 0 | MULT | Y MACRO | ADD | JMP | ENDM | DIV | |
|---|---|---|---|---|---|---|---|---|

(We ignore any pointers.) Macro `Y` will not be recognized as a macro but will be stored in the MDT as part of the definition of macro `X`.

Another method for matching MACRO-ENDM pairs while reading-in a macro definition is to require each ENDM to contain the name of the macro it terminates. Thus the above example should be written:

```
X  MACRO
   MULT
Y  MACRO
   ADD
   JMP
   ENDM Y
   DIV
   ENDM X
```

This requires more work on the part of the programmer but, on the other hand, makes the program easier to read.

▸ **Exercise 11.16:** In the case where both macros X, Y end at the same point

```
X  MACRO
   -
   -
Y  MACRO
   -
   -
   ENDM Y
   ENDM X
```

do we still need two ENDM lines?

The next thing to modify is the macro expansion mode. While expanding macro X, the assembler will encounter the inner MACRO directive. This implies that the assembler should be able to switch to the macro definition mode from the macro expansion mode, and not only from the normal mode. This feature does not add any special difficulties and is straightforward to implement. While in the macro expansion mode, if the assembler encounters a MACRO directive, it switches to the macro definition mode, allocates an available area in the MDT for the new definition and creates that definition. Specifically, the assembler:

- fetches the next source line from the MDT, not from the source file.
- stores the line in the macro definition table, as part of the definition of the new macro.
- repeats until it encounters an ENDM line (more precisely, until it encounters an ENDM line that matches the current MACRO line). At this point the assembler switches back to the macro expansion mode, and continues with the normal expansion.

Now may be a good point to mention that such nesting can be done in one direction only. A macro expansion may cause the assembler to temporarily switch to the macro definition mode; a macro definition, however, cannot cause the assembler to switch to the expansion mode. When the assembler is in a macro expansion mode, it may expand a macro with nested definition, so it has to enter the macro definition mode from within the macro expansion mode. However, if the assembler is in the macro definition mode and the macro being defined specifies the expansion of an inner macro, the assembler does not start the nested expansion while in the process of defining the outer macro. It therefore does not enter the macro expansion mode from within the macro definition mode. The reason being that when a macro is defined, its definition is stored in the MDT without any attempt to assemble, execute, or expand anything. As a result, the assembler can only be in one of the four following modes: normal, definition, expansion, and definition inside expansion. They will be numbered 1–4 and denoted by N,D,E,and DE, respectively.

In the example above, when macro X is defined, macro Y becomes part of the body of X and is not recognized as an independent macro. When X is expanded, though, macro Y is defined and, from that point on, Y can also be expanded. The only reason to implement and use this feature is that macro X can be expanded several times, each expansion of X creating a definition of Y in the MDT, and those definitions—because of the use of parameters—do not have to be the same.

Each time `Y` is expanded, the assembler should, of course, expand the most recent definition of `Y` (otherwise nested macro definition would be a completely useless feature). Expanding the most recent definition of `Y` is simple. All that the assembler has to do is to search the MDT *in reverse order*; start from the new macros and continue with the older ones. This feature of backward search has already been mentioned, in connection with macros that redefine existing instructions.

The above example can be written, with the inclusion of parameters, and with some slight changes, to make it more realistic.

```
X  MACRO A,B,C,D
   MULT A
Y  MACRO C
   C
   ADD DIR
   JMP B
   ENDM Y
   DIV C
   Y D
   ENDM X
```

The body of `X` now contains a definition of `Y` and also an expansion of it. An expansion of `X` will generate:

- A `MULT` instruction.
- A definition of `Y` in the MDT.
- A `DIV` instruction.
- An expansion of `Y`, consisting of three lines, the last two of which are `ADD`, `JMP`.

The expansion `X SEC,TOR,DIC,BPL` will generate:

```
   MULT SEC    first line
Y  MACRO C     macro Y gets stored
   C           in the macro definition
   ADD DIR     table with the B parameter
   JMP TOR     changed to TOR but with
   ENDM Y      the C parameter unchanged
   DIV DIC     second line
   Y BPL       a line which is expanded to give -
   BPL         third line. BPL is substituted for the C parameter
   ADD DIR     fourth line
   JMP TOR     fifth line
```

The only thing that may be a surprise in this example is the fact that macro `Y` is stored in the MDT without `C` being substituted. In other words. `Y` is defined as `Y MACRO C` and not as `Y MACRO DIC`. The rule in such a case is the same as in a block structured language. Parameters of the outer macro are global and are known inside the inner macro unless they are redefined by that macro. Thus parameter `B` is replaced by its value `TOR` when `Y` is defined, but parameter `C` is not replaced by `DIC`.

Since we are interested in how things are done by the assembler, the implementation of this feature will be discussed in detail. In fact, we will describe in detail two ways to implement nested macro definitions. One is the traditional way, described below. The other, due to G. Revesz is more recent and more elegant; it is described in Section 11.9.2.

### 11.9.1 The Traditional Method

Normally, when a macro definition is entered into the MDT, each parameter is replaced with a serial number #1, #2, ... To support nested macro definition, the assembler replaces each parameter, not with a single serial number, but with a pair of numbers (definition level, serial number). To determine those pairs, a stack, called the *macro definition stack* (MDS), is used.

When the assembler starts pass 0, it clears the stack and initializes a special counter (the Dlevel counter mentioned earlier) to 0. Every time the assembler encounters a `MACRO` line, it increments the level counter by 1 and pushes the names of the parameters of that level into the stack, each with a pair (level counter, $i$) attached, where $i$ is the serial number of the parameter. The assembler then starts copying the definition into the MDT, comparing every token on every line with the stack entries (starting with the most recent stack entry). If a token in one of the macro lines matches a stack entry, the assembler considers it to be a parameter (of the current level or any outer one). It fetches the pair (l,$i$) from the stack entry that matched, and stores #(l,$i$) in the MDT instead of storing the token itself. If the token does not match any stack entry, it is considered a stand-alone token and is copied into the MDT as part of the source line.

When an `ENDM` is encountered, the stack entries for the current level are popped out and Dlevel is decremented by 1. After the last `ENDM` in a nested definition is encountered, the stack is left empty and Dlevel should be 0.

The example below shows three nested definitions and the contents of the MDT. It also shows the macro definition stack when the third, innermost, definition is processed (the stack is at its maximum length at this point).

The following points should be noted about this example:

- Lines 3,5,8, and 10 in the MDT show that the assembler did not treat the inner macros `Q`, `R` as independent ones. They are just part of the body of macro `P`.
- On line 4, the #(2,1) in the MDT means parameter 1 (`A`) of level 2 (`Q`), while the #(1,3) means parameter 3 (`C`) of level 1 (`P`).
- On line 7, #(3,3) is parameter 3 (`E`) of level 3 (`R`) and not that of level 2 (`Q`). The `H` is not found in the stack and is therefore considered a stand-alone symbol, not a parameter.
- On line 11, the assembler is back to level 1 where none of the symbols is a parameter. The stack at this point only contains the four bottom lines, and symbols `E,F,G,H` are all considered stand-alone.

| # | | source line | MDT |
|---|---|---|---|
| 1 | P | MACRO A,B,C,D | P\|4\| |
| 2 | | A,B,C,D | #(1,1),#(1,2),#(1,3),#(1,4)\| |
| 3 | Q | MACRO A,B,E,F | Q MACRO #(2,1),#(2,2),#(2,3),#(2,4)\| |
| 4 | | A,B,C,D | #(2,1),#(2,2),#(1,3),#(1,4)\| |
| 5 | R | MACRO A,C,E,G | R MACRO #(3,1),#(3,2),#(3,3),#(3,4)\| |
| 6 | | A,B,C,D | #(3,1),#(2,2),#(3,2),#(1,4)\| |
| 7 | | E,F,G,H | #(3,3),#(2,4),#(3,4),H\| |
| 8 | | ENDM R | ENDM R\| |
| 9 | | E,F,G,H | #(2,3),#(2,4),G,H\| |
| 10 | | ENDM Q | ENDM Q\| |
| 11 | | E,F,G,H | E,F,G,H\| |
| 12 | | ENDM P | ▌ |

Figure 11.6 is a flow chart (a generalized subset of Figure 11.2) summarizing the operations described earlier.

### 11.9.2 Revesz's Method

There is another, newer and more elegant method—due to G. Revesz—for implementing nested macro definitions. It uses a single serial number—instead of a pair—to *tag* each macro parameter, and also has the advantage that it allows for an easy implementation of nested macro expansions and nested macro definitions within a macro expansion.

The method is based on the following observation: When a macro `A` is being defined, we are only concerned with the definition of `A` (the level 1 definition) and not with any inner, nested definitions, since `A`

<u>stack</u>

```
G (3,4)    top
E (3,3)
C (3,2)
A (3,1)
F (2,4)
E (2,3)
B (2,2)
A (2,1)
D (1,4)
C (1,3)
B (1,2)
A (1,1)    bottom
```

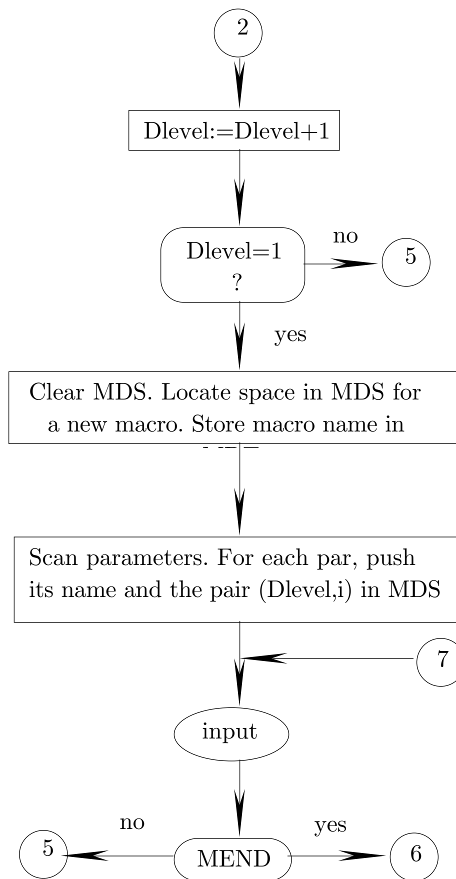



**Figure 11.6**. The classical method for nested macro definitions (part I)

is the only one that is stored in the MDT as a separate macro. In such a case why be concerned about the parameters of all the inner, higher level, nested definitions inside `A`? Any inner definitions are only handled when `A` is expanded, so why not determine their actual arguments at that point?

This is an example of an algorithm where laziness pays. We put off any work to the latest possible point in time, and the result is simple, elegant, and correct.

The method uses the two level counters Dlevel and Elevel as before. There are also three stacks, one for

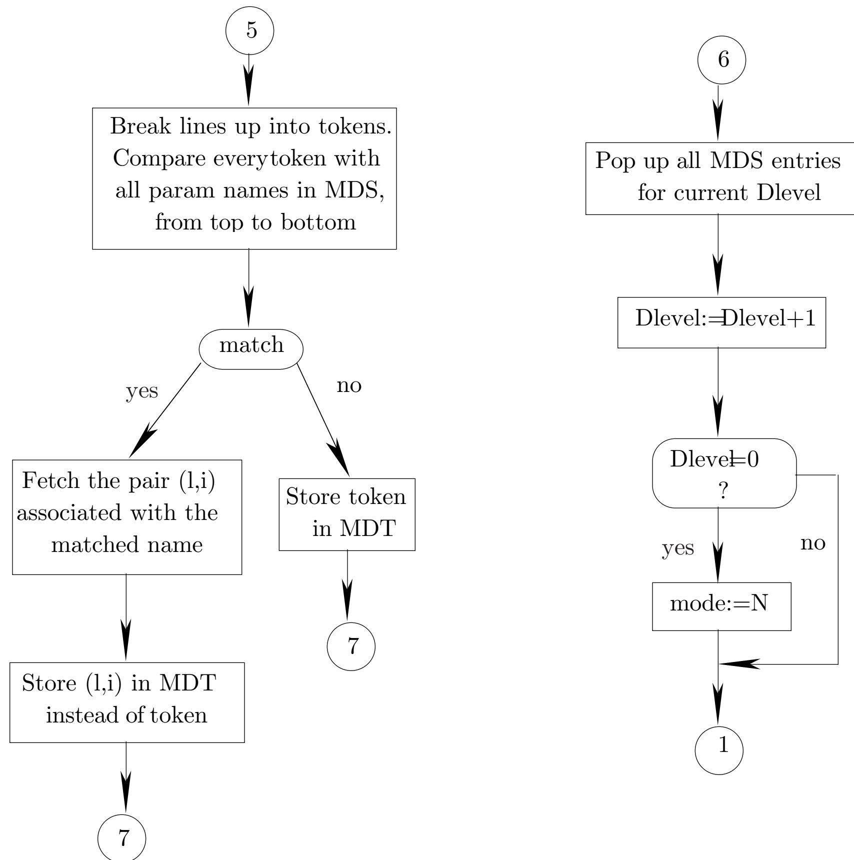



**Figure 11.6**. The classical method for nested macro definitions (part II)

formal parameters (P), the second for actual arguments (A), and the third, (MES), for the nested expansion pointers. A non-empty formal parameter stack implies that the assembler is in the definition mode (D), and a non-empty argument stack, that it is in the expansion mode (E). By examining the state (empty/non empty) of those two stacks, we can easily tell in which of the 4 modes the assembler currently is.

Following are the details of the method in the case of a nested definition. When the assembler is in mode 1 or 3, and it finds a `MACRO` line, it switches to mode 2 or 4 respectively, where it:

1. Stores the names of the parameters in stack P, each with a serial number attached. Since stack P should be originally empty, this is level 1 in P.
2. Opens up an area in the MDT for the new macro.
3. Brings in source lines to be stored in the MDT, as part of the new definition. Each line is first checked to see if it is a `MACRO` or a `MEND`.
4. If the current source line is `MACRO`, the line itself goes into the MDT and the assembler enters the names of the parameters of the inner macro in stack P as a new, higher level. Just the names are entered, with no serial numbers.

This, again, is an important principle of the method. It distinguishes between the parameters of level 1 and those of the higher levels, but it does not distinguish between the parameters of the different higher levels. Again the idea is that, at macro definition time we only handle the level 1, outer macro, so why bother to resolve parameter conflicts on higher levels at this time. Such conflicts will be resolved by the same method anytime an inner macro is defined (becomes level 1).

5. If the current line is a `MEND`, the line itself again goes into the MDT and the assembler removes the highest level of parameters from P. Thus after finding the last `MEND`, stack P should be empty again, signifying a non-macro definition mode. The assembler should then switch back to the original mode

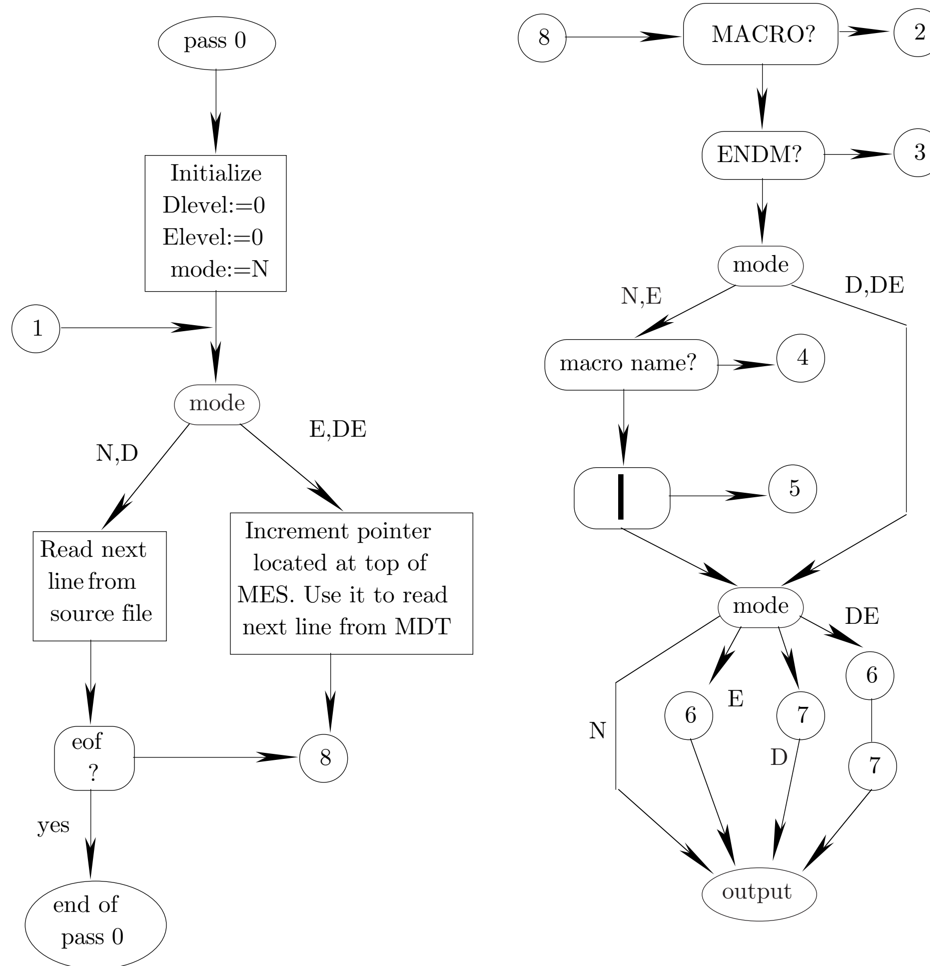



**Figure 11.7**. Revesz's method for nested macro definitions (part I)

(either 1 or 3).

6. If the source line is neither of the above, it is scanned, token by token, to determine what tokens are parameters. Each token is compared with all elements on stack P, from top to bottom. There are three cases:
   a: No match. The token is not a parameter of any macro and is not replaced by anything.
   b: The token matches a name in P that does not have a serial number attached. The token is thus a parameter of an inner macro, and can be ignored for now. We simply leave it alone, knowing that it will be replaced by some serial number when the inner macro is eventually defined (that will happen when some outer macro is eventually expanded).
   c: The token matches a name in P that has a serial number attached. The token is thus a parameter of the currently defined macro (the level 1 macro), and is replaced by the serial number.

After comparing all the tokens of a source line, the line is stored in the MDT. It is a part of the currently defined macro.

The important point is that the formal parameters are replaced by serial numbers only for level 1, i.e., only for the outermost macro definition. For all other nested definitions, only the names of the parameters are placed on the stack, reflecting the fact that only level 1 is processed in macro definition mode. That level ends up being stored in the MDT with each of its parameters being replaced by a serial number.

On recognizing the name of a macro, which can happen either in mode 1 (normal) or 3 (expansion), the assembler enters mode 3, where it:

7. Loads stack A with a new level containing the actual arguments. If the new macro has no arguments,

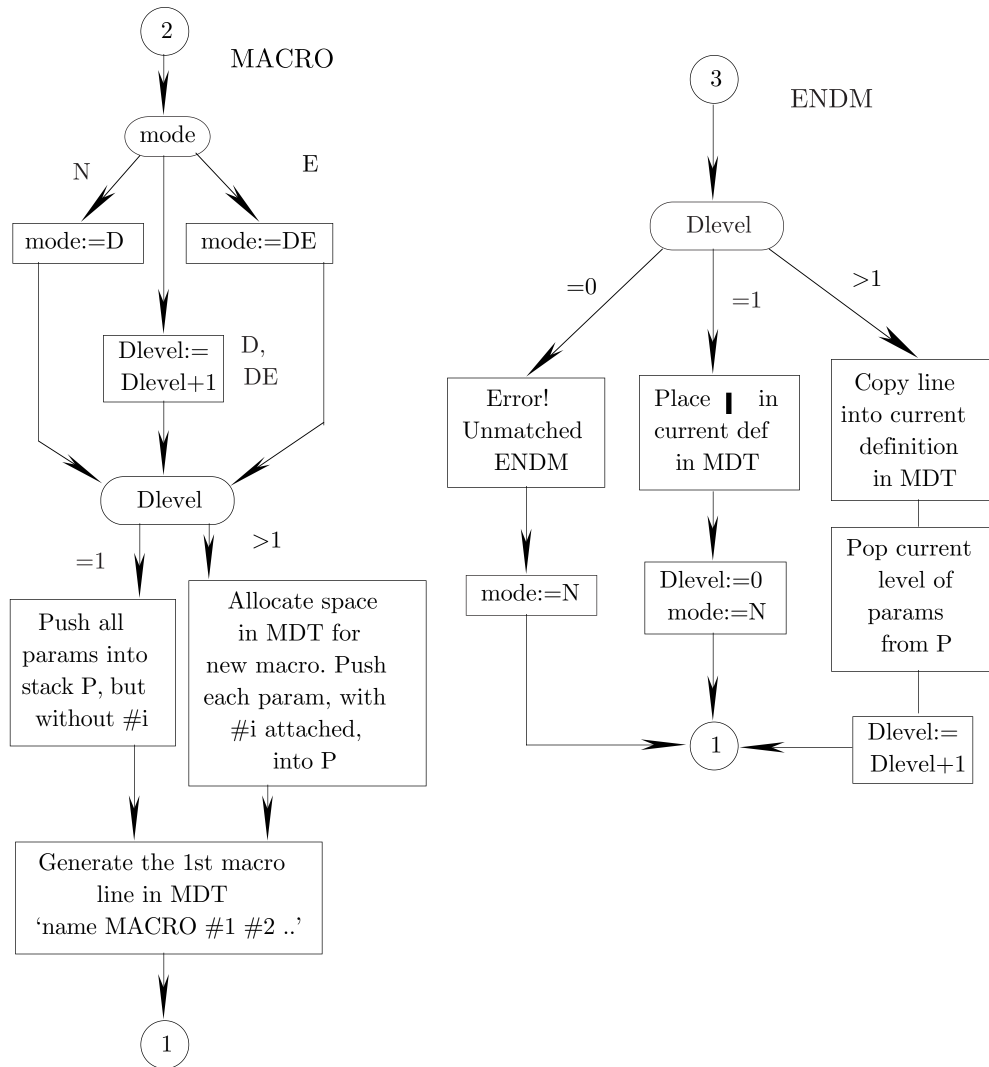



**Figure 11.7**. Revesz's method for nested macro definitions (part II)

the new level is empty, but it should always be in stack A as a level.

8. Inserts a new pointer in the MES, pointing to the first line—in the MDT—of the macro to be expanded.
9. Starts expanding the macro by bringing in individual lines from the MDT. Each line is scanned and all serial numbers are replaced by arguments from stack A. The line is then checked to distinguish three cases:
   - d: The line is `MACRO`, the assembler enters mode 4 (definition mode within expansion mode) and follows rules 1–6 above.
   - e: The current line is `MEND`, the assembler again follows the rules above and may decide, if this is the last `MEND`, to return from mode 4 back to mode 3.
   - f: For any other line, the line is written on the new source file as explained earlier.

   On recognizing the end of the macro (no more lines to expand), the assembler:
10. Removes the highest level from stack A.
11. Deletes the top pointer from the MES.
12. Checks the MES and stack A. There are three cases:
    - g: Neither is empty. There is an outer level of expansion, the assembler stays in mode 3 and resumes the expansion as above.

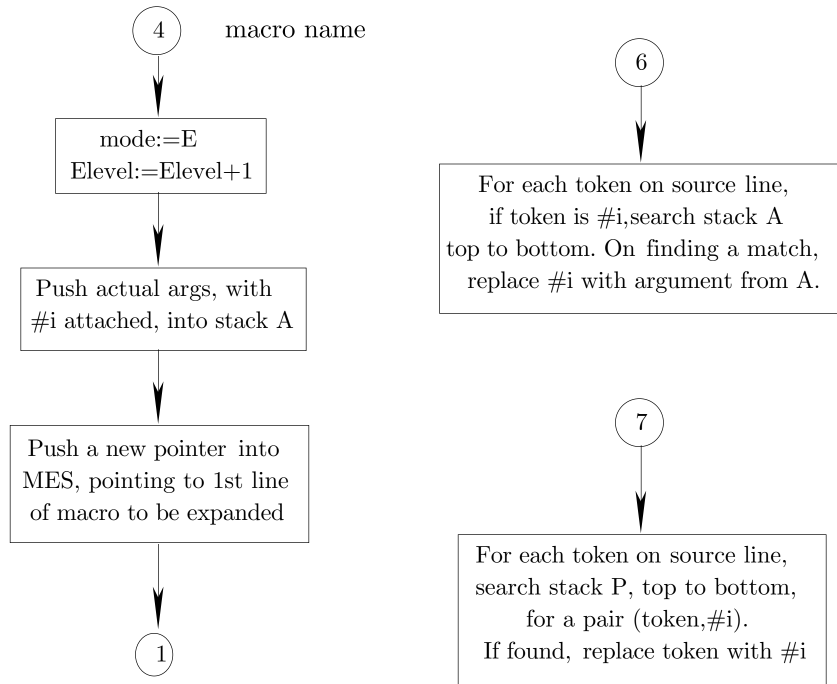



**Figure 11.7**. Revesz's method for nested macro definitions (part III)

h: Only one is empty. This should never happen and can only result from a bug in the assembler itself. The assembler should issue a message like 'impossible error, inform a systems programmer'.

i: Both are empty. This is the end of the expansion. The assembler switches to mode 1.

The following example illustrates a macro `L1` whose definition is nested by `L2` which, in turn, is nested by a level 3 macro, `L3`.

```
1   L1 MACRO A,B,C,D  L1   #1#2#3#4
2      MOV   A,B      MOV  #1#2
3      CMP   C,D      CMP  #3#4
4   L2 MACRO E,F,D,C  L2   MACRO E,F,D,C  L2   #1#2#3#4
5      MOV   E,C      MOV  E,C            MOV  #1,#4
6      CMP   A,X      CMP  #1,X           CMP  M,X
7   L3 MACRO A,E,C,G  L3   MACRO A,E,C,G  L3   MACRO A,E,C,G  L3   MACRO #1#2#3#4
8      MOV   C,E      MOV  C,E            MOV  C,#1           MOV #3,W
9      CMP   A,G      CMP  A,G            CMP  A,G            CMP #1,#4
10     MEND  L3       MEND L3             MEND L3             |    (end of macro)
11     ADD   A,F      ADD  #1,F           ADD  M,#2
12     SUB   C,B      SUB  C,#2           SUB  #4,N
13     MEND  L2       MEND L2             |    (end of macro)
14     ADD   E,F      ADD  E,F
15     SUB   G,B      SUB  G,#2
16     MEND  L1       |    (end of macro)

    Source definition   Definition of L1   Definition of L2   Definition of L3
```

When macro `L1` is defined, in pass 0, the result is a 14-line definition (lines 2–15) in the MDT. During

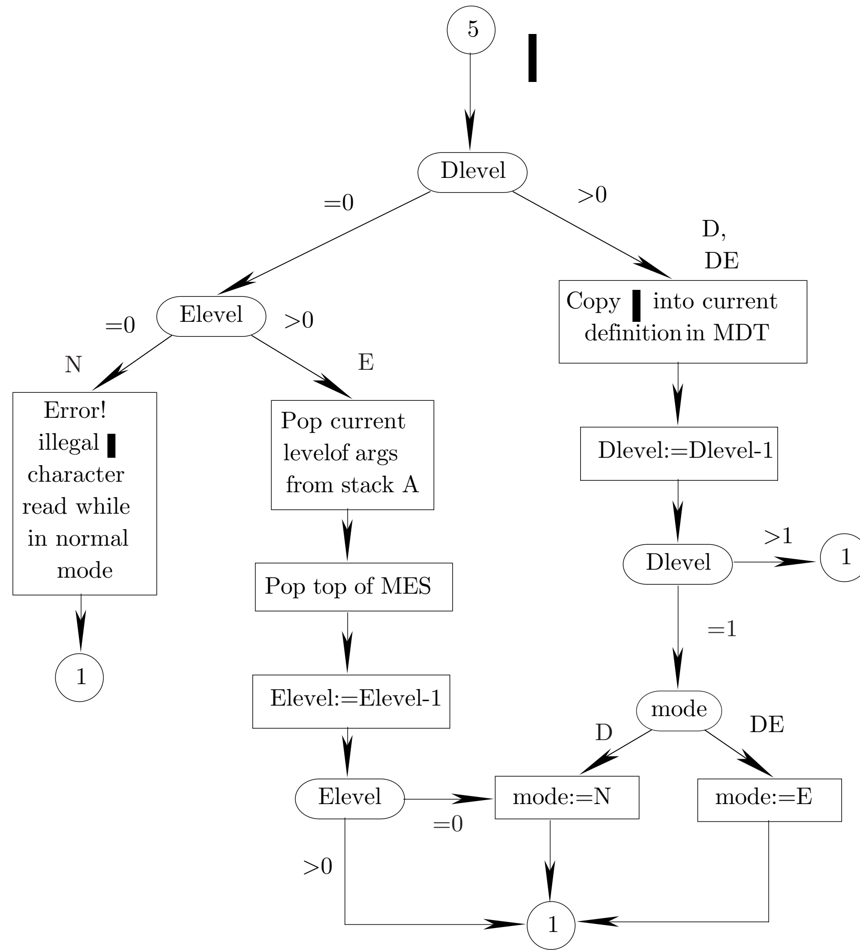



**Figure 11.7**. Revesz's method for nested macro definitions (part IV)

the definition, stack P goes through several levels as follows:

| after line | **1** | **4** | **7** | **10** | **13** | **16** |
|---|---|---|---|---|---|---|
| | `D #4` | `C` | `G` | `C` | `D #4` | |
| | `C #3` | `D` | `C` | `D` | `C #3` | empty |
| | `B #2` | `F` | `E` | `E` | `B #2` | |
| | `A #1` | `E` | `A` | `F` | `A #1` | |
| | | `D #4` | `C` | `D #4` | | |
| | | `C #3` | `D` | `C #3` | | |
| | | `B #2` | `F` | `B #2` | | |
| | | `A #1` | `E` | `A #1` | | |
| | | | `D #4` | | | |
| | | | `C #3` | | | |
| | | | `B #2` | | | |
| | | | `A #1` | | | |

When `L1` is first expanded with, say, arguments `M,N,P,Q`, it results in:

1. The 14 lines brought from the MDT and examined. Lines 2,3,14,15 are eventually written on the new source file, as: `MOV M,N` `CMP P,Q` `ADD E,F` `SUB G,N`.

2. Lines 4–13 become the 8-line definition of `L2` in the MDT as shown earlier. Later, when `L2` is expanded, with arguments `W,X,Y,Z`, it results in:
   1. Lines 5,6,11,12 written on the new source file as:
      `MOV W,Z CMP M,X ADD M,Y SUB Z,M`.
   2. Lines 7–10 stored in the MDT as the 2-line definition of `L3`. From that point on, `L3` can also be expanded.

Figure 4–7 is a summary of this method, showing how to handle nested macro definitions and expansions, as well as definition mode nested inside expansion mode.

### 11.9.3 A Note

The TEX typesetting system, mentioned earlier, supports a powerful macro facility that also allows for nested macro definitions. It is worth mentioning here because it uses an unconventional notation. To define a macro `a` with two parameters, the user should say '`\def\a#1#2{...#1...#2...}`'. Each `#p` refers to one of the parameters. To nest the definition of macro `b` inside `a`, a double `##` notation is used. Thus in the definition '`\def\a#1{..#1..\def\b##1{..##1..#1}}` ' the notation `##1` refers to the parameter of `b`, and `#1`, to that of macro `a`.

The rule is that each pair `##` is reduced to one `#`. This way, macro definitions can be nested to any depth.

Example:

```
\def\a#1{'#1'\def\b##1{[#1,##1] \def\x####1{(#1,##1,####1)}\x X}\b Y}
```

The definition of macro `a` consists of:

- Printing `a`'s parameter in quotes.
- Defining macro `b`.
- Expanding `b` with the argument `Y`.

Similarly, the definition of macro `b` includes:

- Printing `a`'s parameter and `b`'s parameter in square brackets.
- Defining macro `x` with one parameter.
- Expanding `x` with the argument `X`.

Macro `x` simply prints all three arguments, `a`'s, `b`'s and its own, in parentheses.

▸ **Exercise 11.17:** What is the result of the expansion `\a A`?

The reader should convince himself that the rule above really allows for unlimited nesting of macro definitions.

## 11.10 Summary of Pass 0

In order to concentrate all macro operations in one place, many assemblers perform an additional pass, pass 0, mentioned before, in which all macros are defined and expanded. The result of pass 0 is a new source file that serves as the source input for pass 1. To simplify the handling of macros, many assemblers require all macros to be defined at the beginning of the program. The last `ENDM` in that block of macro definitions is followed by a source line that is not a `MACRO`, and this signals to the assembler the end of macro definitions. From that point on, the assembler only expects macro expansions and will flag any macro definition as an

error. Example :

```
     BEGIN EX
A    MACRO P,Q
     .
     .
     ENDM
M    MACRO C,II
     .
     .
     ENDM
X    DS 12   an array declaration
N    MACRO
     .
     .
     ENDM
     .
     .
```

The definition of macro `N` will be flagged, since it occurs too late in the source.

With this separation of macro definition and expansion, pass 0 is easy to implement. In the normal mode such an assembler simply copies the source file into the new source. Macro definitions only affect the MDT, while macro expansions are written onto the new source file.

Such an assembler, however, cannot support nested macro definitions. They generate new macro definitions too late in pass 0.

An assembler that supports nested macro definitions cannot impose the restriction above, and can only require the user to define each macro before it is expanded.

Having a separate pass 0 simplifies pass 1. The flow charts in his chapter prove that pass 0 is at least as complicated as pass 1, so keeping them separate reduces the total size and the complexity of the assembler. Combining pass 0 and pass 1, however, has the following advantages:

- No new source file needed, saving both file space and assembly time.

- Procedures—such as input procedures or procedures for checking the type of a symbol—that are needed in both passes, only need to be implemented once.

- All pass 1 facilities (such as `EQU` and other directives) are available during macro expansion. This simplifies the handling of conditional assembly.

... Then there are young men who dance around and get paid by the women, they're called 'macros' and aren't much use to anyone ...

—Noël Coward (1939), *To Step Aside*

# A
# History of Computers

This short chapter presents an outline of the major developments in the history of computers, concentrating on topics that are not well known. Because of the importance and popularity of computers, there are many books on this topic ([Augarten 84], [Burks 88], [Goldstine 72], [Randell 82], and [Singh 99] to cite a few examples), and historical resources on the web (see [IowaState] or search under `computers` and `history`).

## A.1 A Brief History of Computers

The old history of computing machines starts with the French mathematician Blaise Pascal, although some sources, such as [Shallit 00] mention much older devices for computations. Strictly speaking, the first mechanical calculator was built by Wilhelm Schickard, a professor at the university of Tübingen in Germany. His machine, however, remained practically unknown in his time and did not have any influence on subsequent historical developments. This is why today we consider Pascal the father of calculating machines. In 1642, at age nineteen, he designed and built an 8-digit, 2-register mechanical calculator (termed the Pascaline) that could add and subtract. His calculator had two sets of wheels, one to enter the next number and the other, to hold the sum/difference. He was the first to realize that decimal digits could be arranged on wheels in such a way that each wheel—when passing from 9 to 0—would drag its neighbor on the left one-tenth of a revolution, thereby propagating the carry mechanically. This method of carry propagation has remained the basis of all the mechanical and electromechanical calculators up to the present.

> "I submit to the public a small machine of my own invention by means of which alone you may, without effort, perform all the operations of arithmetic, and may be relieved of the work which has often times fatigued your spirit."
>
> Blaise pascal.

It is interesting to note that Pascal was also the father of the modern science of probability and also the inventor of the game of roulette. He also created, in his last year, the first public transportation system, an omnibus service in Paris. Legend has it that he developed his calculator after his father, a provincial tax collector in Rouen, asked him to help with that year's tax rolls.

The next step occurred thirty years later, when Gottfried Wilhelm von Leibniz improved on Pascal's design by inventing the stepped wheel, which made it possible to do multiplication and division, as well as addition and subtraction. His machine had four registers, the two additional ones for the multiplier and the

multiplicand. It used chains and pulleys, and is described in [Smith 29]. Surprisingly, his invention found little use in his time.

In the nineteenth century, the British mathematician and philosopher Charles Babbage devoted the better part of his life to the development of calculating devices (engines). Babbage, who was also the inventor of the speedometer and the cowcatcher, and who was also the first to develop reliable life expectancy tables, kept designing better calculating engines. His first important design, the difference engine (1823), used the method of finite differences for calculating a large number of useful functions. His main project, however, was the analytical engine (1834), which was to be a general-purpose calculating machine, similar to present day computers. It had data input (from punched cards), automatic sequencing (a program on punched cards), a control unit, a memory unit (consisting of 1000 words of 50 decimal digits each, equivalent to about 170.9K bits), and an ALU (mill) that could perform arithmetic and logic operations. A number could be read from memory, sent to the mill, operated on, and stored back in memory. An important feature was the program store, a memory unit where the program was stored in the form of machine instructions with opcodes and addresses, similar to those found in modern computers. The idea of using punched cards was adopted from the Jacquard loom, which used them to control the weaving of a pattern. For more information on this original and complex design see [Augarten 84], pages 63–64. A detailed biography of Babbage and his work is [Hyman 82].

Babbage ended up spending most of his $100,000 inheritance, and another $1500 in government grants, without completing any machines. It is commonly believed today that his engines could not be built because of the state of the art of mechanical engineering at his time. We now know, however, that during Babbage's lifetime, some of his designs had been implemented by George and Edvard Scheutz, a Swedish father and son team. This implies that lack of technology was just one of Babbage's problems. His main problem was that he lost interest in any work in progress as soon as he came up with a new, more powerful design. He was a perfectionist and a difficult person to work with.

Seventy years after Babbage, In the 1930s and 1940s there were three, roughly parallel efforts directed by Konrad Zuse, Alan Turing, and Howard Aiken, to build larger calculating machines.

In Berlin, the young German engineer Konrad Zuse, motivated by what he called "those awful calculations required of civil engineers," built three electromechanical calculating machines that worked successfully. The Z-1, his first machine, was completely mechanical. It was built from erector parts in his parents living room and was used mainly to help him do stress analysis calculations

The Z-2, Zuse's second attempt (again at home), used electromechanical relays (the type used in telephones in his day) and could solve simultaneous algebraic equations. It worked as planned, and it encouraged his employers to help him build the Z-3.

The Z-3, completed in late 1941, was electromechanical, consisted of 2600 relays, and contained most of the components of a modern, general-purpose digital computer. It had a $64 \times 22$ memory and an ALU that could perform the four arithmetic operations. It was slow, taking 3–5 seconds for a single multiplication. The program was read from punched tape. Zuse received a German patent (#Z23624, filed April 11, 1936) on the Z-3. It seems that this machine deserves the title "the world's first operational, programmable computer."

Encouraged by the performance of the Z-3, Zuse started on the Z-4, and founded a company. The end of the war, however, put a stop to his efforts. It is also interesting to note that Zuse was the first to develop a higher-level programming language, the Plankalkul. [Roja 00] is an article on Zuse and his work. A translation of Zuse's own article appears in [Randell 82], pages 159–166. Arnold Fast, a blind mathematician hired by Zuse in 1941 to program the Z-3, probably deserves the title "the world's first programmer of an *operational* programmable computer."

Most of Zuse's machines were destroyed during the war, and information about his work did not reach the world until the mid-1960s. It seems that Zuse was, for a while, way ahead of other computer developers but, perhaps fortunately for the world, the German government did not understand his work, gave him little support, and even drafted him as a simple soldier in world war II. Luckily for him, his boss at the Heinkel aircraft manufacturing company needed him, and secured his discharge.

In Bletchley Park, a manor house in Hertfordshire, England, a team that included Alan Turing spent the war years trying to break the Enigma, the German code used before and during the war (see [Singh 99] for the Enigma code and its cracking).

Part of the British effort to crack the Enigma code involved the construction, in 1940, of machines—

primitive by today's standards—called 'bombes.' These were electromechanical devices that could perform just one type of computation. (Another name for the bombes was the "Robinson machine," after a popular cartoonist who drew Rube Goldberg machines.) It seems that the bombes deserve the title "the world's first operational computer" since Babbage's machine were never completed. Later in the war, the British discovered that the Germans used another cipher, dubbed the Lorenz, that was far more complicated than the enigma. Breaking the Lorenz code required a machine much more sophisticated than the bombes, a machine that could perform statistical analysis, data searching, and string matching, and that could easily be reconfigured to perform different operations when necessary. Max Newman, one of the Bletchley mathematicians, came up with a design for such a machine, but his superiors were convinced that constructing it, especially during the war, was beyond their capabilities.

Fortunately, an engineer by the name of Tommy Flowers heard about the idea and believed that it was workable. He worked for the British post office in North London, were he managed to convert Newman's design into a working machine in ten months. He finished the construction and delivered his machine to Bletchley park in December 1943. It was called Colossus, and it had two important features: it was completely electronic, using 1500 vacuum tubes and no mechanical relays, and it was programmable. It used paper tapes for input and output. Today, the Colossus is one of the few candidates for the title "the first modern electronic computer," but for a long time it was kept secret.

After the war, Colossus was dismantled, and its original blueprints destroyed. This is why for many years, others were credited with the invention of the modern computer.

After the war, Turing designed the ACE computer, an ambitious project for that time. It was supposed to be a large, stored-program machine, driven at a million steps per second. A scaled down version was eventually completed in 1950.

Both Zuse and Turing, independently of each other, decided to use *base-two* (binary) numbers in their machines, thus marking a departure from the traditional base-ten numbers, and opening the way to efficient arithmetic circuits and algorithms.

At about the same time, Howard Aiken, a physicist at Harvard University, approached IBM with a proposal to build a large electromechanical calculator. The result was the Mark I computer, completed in January, 1943. It used mechanical relays and sounded, as an observer once noted "like a roomful of ladies knitting." The Mark I lasted 16 years, during which it was used mostly for calculating mathematical tables. Like all early computers, the Mark I had a very small internal memory and had to read its instructions from paper tape. It seems that the Mark I deserves the title "the first programmable computer built by an American." See [Augarten 84], pages 103–107, for more information on this machine.

An independent development was the Atanasoff-Berry computer (ABC), developed during 1937–42 in Ames, Iowa, USA, at Iowa State University (then Iowa State college), by John V. Atanasoff, with the help of graduate student Clifford Berry. Today it seems that this machine is the best candidate for the title "the world's first *electronic* computer" (however, it was nonprogrammable, see [Burks 88], [IowaState 00], and [Mollenhoff 88]).

Atanasoff and Berry later founded the Atanasoff-Berry Computer company, that pioneered several innovations in computing, including logic devices, parallel processing, regenerative memory, and a separation of memory and computing functions.

On October 19, 1973, US Federal Judge Earl R. Larson made, following a lengthy trial, a decision that declared the ENIAC patent of Mauchly and Eckert invalid and named Atanasoff the inventor of the electronic digital computer. Details about this historically important machine are available in [Mollenhoff 88] and [Burks 88].

In recognition of his achievement, Atanasoff was awarded the National Medal of Technology by President George Bush at the White house on November 13, 1990.

"I have always taken the position that there is enough credit for everyone in the invention and development of the electronic computer."

—John Vincent Atanasoff

Table A.1 summarizes the most important developments in calculating devices that were either mechanical, electromechanical, or electronic analog.

| Date | Inventor/Developer | Features | Innovations |
|---|---|---|---|
| 1623 | W. Schickard | Add/Subtract | Carry propagation |
| 1642 | B. Pascal | Add/Subtract | Carry propagation & complement numbers |
| 1671 | W. Leibniz | Four operations | "Stepped Reckoner" |
| 1801 | J. Jacquard | Control of weaving loom | Controlled by punched cards |
| 1822 | C. Babbage | Difference Engine | Polynomial evaluation by finite differences |
| 1834 | C. Babbage | Analytical Engine | Automatic sequencing |
| 1930 | V. Bush | Integration | Analog mechanical |
| 1940 | G. Stibitz | General Purpose | Relays |
| 1940 | J. Atanasoff | Nonprogrammable | Electronic |
| 1940 | A. Turing | Robinson (nonprogrammable) | Relays |
| 1941 | K. Zuse | General Purpose | Relays |
| 1941 | A. Turing | Code Breaking | Vacuum tubes |
| 1942 | V. Bush | High-precision Integration | Analog electromechanical |
| 1944 | H. Aiken | General Purpose | Mark I, Relays |
| 1946 | Eckert Mauchly | Manually progr. | ENIAC |
| 1952 | von Neumann | Stored program | IAS computer |

**Table A.1**. Important Developments in Calculating Devices

## A.2 First Generation. Vacuum Tubes

It is generally agreed that the era of the modern digital electronic computer started on February 16, 1946, the day the ENIAC computer was unveiled. It seems that this machine is the best candidate for the long title "the world's first fully-electronic, general-purpose (i.e., programmable) digital computer." Designed and built at The Moore School of Electrical Engineering in Philadelphia, by John W. Mauchly and J. Presper Eckert, it was financed by no other than the US army. Here is this true story.

During WWII, the Computing Laboratory at the University of Pennsylvania was responsible for computing ballistic firing tables for US Army artillery. Because of the tremendous amount of calculations needed, they were getting behind all the time. This was why Mauchly and Eckert proposed, as a radical solution, the development of an automatic calculating machine. Fortunately for world history, the army approved this costly and uncertain project. The US government thus ended up paying $486,804 for the computer. This was extremely fortunate, as no other organization in the world was able or willing to sponsors the development of such a machine at that time.

The ENIAC [Scott 99] was a monstrously huge machine. Eighty foot long, weighing about 30 tons, and containing 17468 tubes, 6000 switches, 10000 capacitors and 4000 knobs. According to legend, the lights of Philadelphia dimmed each time it was turned on. It was a thousand times faster than the Mark I, could add in 2 milliseconds, multiply in 2.8 ms and divide in 24 ms. Its main feature, however, was that it did not have enough storage for the program, and could only be programmed manually, by means of switches and cables. From 1945 until January 1947, the ENIAC ran at the Moore School at the University of Pennsylvania, where it was designed and built. Then the machine was moved to Aberdeen Proving Ground, where it operated from July 1947 until it was dismantled in October 1955. A pictorial reference for John W. Mauchly and the ENIAC is [ENIAC 00].

Although ENIAC operated for nine years, it was a unique, experimental device, not a production

computer. Parts were custom made for this machine and procedures were developed specifically for it. Working with ENIAC was a slow and tedious process. A mathematical problem had to be understood and its solution written in a way that allowed rewiring of the machine to solve the problem. Obviously, such a process was worth the effort only for the problems that interested the sponsors (i.e., the army).

The modern approach to the design and programming of computers was developed at the Institute for Advanced Study in Princeton by the mathematician John von Neumann and his collaborators. Their main ideas were summarized in the now famous report that was published in 1945 [Burks et al. 45]. In that report they called their design "The IAS computer" (IAS is the Institute for Advanced Study, in Princeton, New Jersey). To the world in general their design is known as the "von Neumann machine." The first machine built according to this design was the EDVAC (Electronic Discrete Variable Computer), completed in 1952. It contained 4000 tubes and 10000 diodes, and remained operative until 1962.

**John von Neumann**

Perhaps the greatest mathematician of the twentieth century, Janos Neumann was born in Budapest, on December 28, 1903. His father, a banker, was well enough to buy the Hungarian title "Margattai," which his son later changed to "von." The young Janos showed signs of genius from an early age, becoming the best mathematics student of the Lutheran gymnasium in Budapest, which he entered in 1911. In 1926, at age 22, he earned his doctorate from the university of Budapest. He spent 1926 at the university of Götingen, and became a lecturer at the university of Berlin in 1927. Von Neumann moved to the United States in 1930, joining Princeton university and, in 1933, the Institute for Advanced Study.

Stories and anecdotes on von Neumann abound. He had a prodigious memory, could do complex calculations in his head, was always very elegantly dressed, and liked fast cars, sending about one car a year to the junkyard. He also liked gadgets, which partly explains his interest in computers.

His interest in calculating machines started when he did consulting for the Manhattan project, in the early 1940s, at Los Alamos. In 1944 he heard about the ENIAC, came to Philadelphia to see the machine, became very interested, and in June 1945 produced the *First Draft of a Report on the EDVAC*. This report, together with the EDVAC itself, has secured von Neumann's place in history as the creator of the "von Neumann machine."

After the EDVAC, von Neumann moved to other pursuits, He did important work in game theory, cybernetics, and cellular automata. He died in 1957, at age 54, from cancer.

Von Neumann and his collaborators realized that controlling a computer can and should be done *internally*. They were also the first to understand that a computer performed *logical* operations and that the arithmetic aspects of the computer were secondary to the logical ones. In 1944 this way of thinking was a major step forward and opened up the way to the design of the modern digital computer. The main contribution of this small group to the field of computer design is the idea of the *stored program*. The program is stored inside the computer, in the same memory with the data, and the computer uses logic circuits—of the same type used to perform arithmetic operations—to fetch instructions from memory, decode and execute them.

To summarize, the main idea in a von-Neumann machine is to have a storage unit (the memory) where *both* the program and the data are stored. Instructions are fetched from memory one by one and are executed by the control unit. Data items are also fetched from or written into the same memory. This idea has proved so powerful and useful that even today most computers are von-Neumann machines and relatively few computers operate according to different principles.

Two good references on the early history of computers and computing machines are [Goldstine 72] and [Randall 82]. In addition, the Computer Science Encyclopedia [Ralston 85] contains several articles on the history of computers and computing machines, and on the main participants.

In 1947 Eckert and Mauchly formed the Eckert-Mauchly computer corporation. They started making the UNIVAC line of computers and were subsequently acquired by the Sperry-Rand corporation to become its UNIVAC division.

The IBM corporation, long a maker of punched card equipment and business machines, supported the Mark I computer, the last of the electro-mechanical relay computers. After the success of the first UNIVAC computers, IBM decided to enter the computer business and delivered its first commercial computer, the 701, in 1953.

Thus, the early 1950s saw two main computer makers, Sperry and IBM. Sperry developed its UNIVAC line and IBM, its 700 series.

## A.3 Second Generation: Transistors

The invention of the transistor in 1947 was the cause of the first major revolution in computer design. Suddenly, it became possible to build computers that were reasonably sized, did not consume huge quantities of electrical power, and were reliable. The first two manufacturers to take advantage of the new technology were NCR and RCA. IBM came in late but had a number of very good entries in the form of the 7000 series of computers. The 7000 series computers were the first ones with an instruction back-up register (to buffer the next instruction), I/O channels, and a *multiplexor*. The multiplexor was the forerunner of today's bus controllers.

The invention of core memories, by J. Forrester in 1953, provided computers with new powers. Almost overnight it became possible to build memories with hundreds, or even thousands, of words. Core memories were faster, smaller, cheaper, and much more reliable than any of the older storage methods.

## A.4 Third Generation: ICs

In the early 1950s, with just a few years' experience with transistors, engineers have pinpointed their next problem—*interconnections*. With transistors, it became possible to build dense circuits, containing many components on a small circuit board. However, the individual components still had to be connected by wires, which put a limit to the density of a circuit. The problem was solved in October, 1958 by Jack Kilby of Texas Instruments, and in April 1959, by Robert Noyce, then at Fairchild Camera and Instrument. They invented the *integrated circuit*, popularly known as the *chip*.

Computer manufacturers immediately realized the importance of this development and came up, in the early 1960's, with computers made with ICs. The two most important early ones were the IBM/360 and the PDP-8. The former has already been mentioned in connection with the definition of computer architecture. The latter was the first true minicomputer, and a very successful one! It has changed the way computers were used. The PDP-8 was first used as a stand-alone inexpensive computer until people realized that it was small enough to be incorporated into other devices and instruments. This turned out to be a very successful concept and it started what is known today as the OEM market (Original Equipment Manufacturers). An OEM is a manufacturer who buys a computer and includes it as part of a larger system or device.

## A.5 Later Developments

There is no general agreement as to what constitutes the fourth or fifth generations of computers, or even whether they exist at all. Many authors view the advent of VLSI as the start of the fourth computer generation. Some experts suggest that advances in artificial intelligence will soon start the fifth generation of very intelligent computers.

The term VLSI (Very Large Scale Integration) refers to chips with many components. To qualify as VLSI, a chip should have a minimum of 10,000 or 100,000 components, depending on who you listen to.

Many people regard the microprocessor as the start of the fourth generation. It should be noted that a microprocessor is not a computer on a chip. A microprocessor is a processor on a chip (see Section 1.1 for the definition of the term processor) and is not a complete computer. There are relatively few computers on a chip (single chip computers) but they are computationally weak and are mostly special-purpose.

The story of the microprocessor starts on November 15,1971, when Intel, a large electronic manufacturer, announced the 4004 microprocessor in an ad in *Electronic News*, a trade magazine. This microprocessor consisted of about 2300 transistors, and was designed by Ted Hoff, an engineer with Intel. His problem was to design a new calculator with as few parts as possible, and he realized that one microprocessor could replace many of the parts traditionally used in a calculator. The 4004 was an immediate success, so a year

later, Intel introduced the 8008, the first 8-bit microprocessor. It was organized around a set of six 8-bit registers, and a set of 45 instructions. It could address 16Kb of memory. The first competition came, also in 1972, from Fairchild semiconductor and from Rockwell, who came up with their microprocessors.

In 1973, Intel announced the 8080 microprocessor, consisting of about 5000 transistors. It had 74 instructions and could address 64Kb of memory—a huge memory space at that time. The 8080 offered about 10 times the performance of the 8008. In 1974, Motorola announced the 6800 microprocessor, a direct competitor to the 8080. By the end of 1974, there were about 20 different microprocessors on the market.

The Altair computer was advertised in the January, 1975 issue of *Electronics* magazine. This was a personal computer based on the 8080, which was sold as a kit. It became an immediate success, and it encouraged other manufacturers to come up with other microprocessors and personal computers. By the end of 1976, more than 55 microprocessors were available, including the Zilog Z-80, a powerful 8-bit machine with a set of 158 instructions.

In 1978, Intel came up with the 8086 microprocessor, the first successful 16-bit microprocessor. It consisted of 29,000 transistors, and could address 1Mb of memory. Immediate competitors were the Zilog Z-8000, Motorola 68000 and National Semiconductor 16000. The first 32-bit microprocessors took longer to develop. They appeared on the scene in 1985–86 and included the DEC VAX, the Motorola 68020, National Semiconductor NS32032, and the Intel iAPX 432. The latter consisted of about 200,000 transistors, could address a main memory of 16Mb, and could execute two million instructions/second.

In the 1990s, microprocessors such as the Intel Pentium and the Motorola PowerPC have been approaching the performance of supercomputers. If current trends continue, supercomputers may eventually disappear, and only microprocessors (and networks of microprocessors) will be used in the future.

The reader should notice that this short history discusses just computers. It does not include details on the development of programming languages, operating systems, and networks. The history of these topics is, of course, well known and widely available. Section 3.27 discusses the Internet, Section 4.2 is a short history of microprogramming, and Section 6.2 is a short history of RISC.

Where a calculator on the ENIAC is equipped with 18,000 vacuum tubes and weighs 30 tons, computers in the future may have only 1,000 vacuum tubes and weigh only 1 1/2 tons.

March 1949, *Popular Mechanics*

# B
# Unicode

This material (and much more) is available at the Unicode organization, `http://www.unicode.org`.

The Unicode Worldwide Character Standard is a character coding system designed to support the interchange, processing, and display of the written texts of the diverse languages of the modern world. In addition, it supports classical and historical texts of many written languages.

In its current version (3.0), the Unicode standard contains 49,194 distinct coded characters derived from the Supported Scripts. These characters cover the principal written languages of the Americas, Europe, the Middle East, Africa, India, Asia, and Pacifica.

Some modern written languages are not yet supported or only partially supported due to a need for further research into the encoding needs of certain scripts. See Unsupported Scripts for a detailed list.

*The Unicode Standard*, Version 2.0, is published by Addison-Wesley, 1996, ISBN 0-201-48345-9.

NOTE: While a lot of information is available online, without access to the book an implementer misses important information about conformance requirements and other explanatory information.

**The Unicode Standard: A Technical Introduction**

NOTE: This is intended as a concise source of information about the Unicode Standard. It is neither a comprehensive definition of, nor a technical guide to the Unicode Standard. The authoritative source of information is *The Unicode Standard*, Version 2.0, Addison Wesley Publisher, 1996.

The Unicode Standard is the international standard used to encode text for computer processing. It is a subset of the International Standard ISO/IEC 10646-1,1993, whose 32-bit values can represent 4,294,967,296 characters. Its design is based on the simplicity and consistency of ASCII, but goes far beyond ASCII's limited ability to encode only the Latin alphabet. The Unicode Standard provides the capacity to encode all of the characters used for the major written languages of the world.

To accommodate the many thousands of characters used in international text, the Unicode Standard adopts a 16-bit codeset, thereby providing codes for 65,536 characters, enough to represent all the characters/symbols used by all of the world's major languages. To keep character coding simple and efficient, the Unicode Standard assigns each character a unique 16-bit value, and does not use complex modes or escape codes.

Unicode provides a consistent coding format for international character sets and brings order to a chaotic state of affairs that has made it difficult to exchange text files across language borders. Computer users who deal with multilingual text—international business people, linguists, researchers, scientists, and others—will find that the Unicode Standard greatly simplifies their work. Mathematicians and technicians, who regularly use mathematical symbols and other technical characters, will also find the Unicode Standard valuable.

**Unicode and ISO/IEC 10646**

The Unicode Standard, Version 2.0 is code value for code value equivalent with the Universal Character Set, two-octet form (UCS-2) of ISO/IEC 10646. As the UCS-2 subset of ISO 10646, the Unicode Standard's 65,536 code values are the first 65,536 code values of ISO 10646. These code values contain all of the

characters currently defined by ISO 10646. All other ISO 10646 code values are reserved for future expansion. ISO 10646's full codeset is called Universal Character Set, four octets form (UCS-4).

What Characters Does the Unicode Standard Include?

The Unicode Standard defines codes for characters used in every major language written today. Scripts include Latin, Greek, Cyrillic, Armenian, Hebrew, Arabic, Devanagari, Bengali, Gurmukhi, Gujarati, Oriya, Tamil, Telugu, Kannada, Malayalam, Thai, Lao, Georgian, Tibetan, Japanese kana, the complete set of modern Korean hangul, and a unified set of Chinese/Japanese/Korean (CJK) ideographs.

The Unicode Standard also includes punctuation marks, diacritics, mathematical symbols, technical symbols, arrows, dingbats, etc. It provides codes for diacritics, which are modifying character marks such as the tilde (˜), that are used in conjunction with base characters to encode accented or vocalized letters (ñ, for example). In all, the Unicode Standard provides codes for nearly 39,000 characters from the world's alphabets, ideograph sets, and symbol collections.

The Unicode Standard has 18,000 unused code values for future expansion. The standard also reserves over 6,000 code numbers for private use, which software and hardware developers can assign internally for their own characters and symbols.

**Design Basis**

To make it possible for a character set to successfully encode, process, and interpret text, the character set must:

- define the smallest useful elements of text to be encoded;
- assign a unique code to each element; and,
- provide basic rules for encoding and interpreting text so that programs can successfully read and process text.

These requirements are the basis for the design of the Unicode Standard.

**Defining Elements of Text**

When spoken language is written, it is represented by textual elements that are used to create words and sentences. These elements may be letters such as "w" or "M"; characters such as those used in Japanese hiragana to represent syllables; or ideographs such as those used in Chinese to represent full words or concepts.

Each writing system defines its own text elements, a definition that often changes depending on the process handling the text. For example, in historic Spanish language sorting, "ll" counts as a single text element. When Spanish words are typed, however, "ll" is two separate text elements: "l" and "l".

To avoid deciding what is and is not a text element in different processes, the Unicode Standard breaks up a writing system into code elements (commonly called "characters" when writing about the Unicode Standard). A code element is a part of a writing system defined by the Unicode Standard to be fundamental and useful for computer text processing. For the most part, code elements correspond to the most commonly used text elements. For ex-ample, each upper- and lowercase letter in the English alphabet is a single code element.

The Unicode Standard, in cases that lack universal agreement over what constitutes a text element, defines a unique code element using the text element definition that is most useful for encoding text with a computer. For example, instead of defining "ll" as a single code element for Spanish text, the Unicode Standard defines each "l" as a separate code element. The task of combining two "l"s together for alphabetic sorting is left to the software processing the text.

The Unicode Standard defines a few codes for the presentation of text. Some control the direction of text—left-to-right or right-to-left—for rare cases where text must change directions within a single run of text.

The Unicode Standard defines explicit characters for line and paragraph endings. With ASCII, the Line Feed and Carriage Return characters are often used ambiguously. The Unicode Standard eliminates that ambiguity.

**Text Processing**

Almost all computer text processes encode and interpret text as they process it. For example, consider a word processor user typing text at a keyboard. The computer's system software receives a message that the user pressed a key combination for "T", which it encodes as U+0054. The word processor stores the

number in memory, and also passes it on to the display software responsible for putting the character on the screen. The display software, which may be a window manager or part of the word processor itself, uses the number as an index to find an image of a "T", which it draws on the monitor screen. The process continues as the user types in more characters.

The Unicode Standard directly addresses only encoding and interpreting text and not any other actions performed on the text. For example, the word processor may check the typist's input after it has been encoded to look for misspelled words, and then beep if it finds any. Or it may insert line breaks when it counts a certain number of characters entered since the last line break. It is an important principle of the Unicode Standard that it does not specify how to carry out these processes as long as the character encoding and decoding is performed properly.

**Interpreting Characters and Rendering Glyphs**

The difference between interpreting a character (identifying a code value) and rendering it on screen or paper is crucial to understanding the Unicode Standard's role in text processing. The character identified by a Unicode code value is an abstract entity, such as: "latin character capital a" or "bengali digit 5." The mark made on screen or paper—called a glyph—is a visual representation of the character.

The Unicode Standard does not define glyph images. Glyph images are not defined by the Unicode Standard, which defines only how characters are interpreted, not how glyphs are rendered. The software or hardware-rendering engine of a computer is responsible for the appearance of the characters on the screen. The Unicode Standard does not specify the size, shape, nor orientation of onscreen characters; it specifies only the code values assigned to those characters.

**Creating Composite Characters**

Textual elements may be encoded as composed character sequences; in presentation, the multiple characters are rendered together. For example, "â" is a composite character created by rendering "a" and "ˆ" together. A composed character sequence is typically made up of a base letter, which occupies a single space, and one or more non-spacing marks, which are rendered in the same space as the base letter.

The Unicode Standard specifies the order of characters used to create a composite character. The base character comes first, followed by one or more non-spacing marks. If a textual element is encoded with more than one non-spacing mark, the order in which the non-spacing marks are stored isn't important if the marks don't interact typographically. If they do, order is important. The Unicode Standard specifies how competing non-spacing characters are applied to a base character.

The Unicode Standard offers precomposed characters to retain compatibility with established standards. The Unicode Standard offers another option for some composite characters—precomposed characters. Each precomposed character is represented by a single code value rather than two or more code values, which may combine during rendering and thus create a "prefabricated" composite character. For example, the character "ü" can be encoded as the single code value U+00FC "ü" or as the base character U+0075 "u" followed by the non-spacing character U+0308 "¨". The Unicode Standard offers precomposed characters to retain compatibility with established standards such as Latin1, which includes many precomposed characters such as "ü" and "ñ".

The Unicode Standard defines decompositions for all precomposed characters. Precomposed characters may be decomposed for consistency or analysis. For example, a word processor importing a text file containing the precomposed character "ü" may decompose that character into a "u" followed by the non-spacing character "¨". Once the character has been decomposed, it may be easier for the word processor to work with the character because the word processor can now easily recognize the character as a "u" with modifications. This allows easy alphabetical sorting for languages where character modifiers do not affect alphabetical order.

**Principles of the Unicode Standard**

The Unicode Standard was created by a team of computer professionals, linguists, and scholars to become a worldwide character standard, one easily used for text encoding everywhere. To that end, the Unicode Standard follows a set of fundamental principles: * Sixteen-bit characters * Logical order * Full Encoding * Unification * Characters, not glyphs * Dynamic composition * Semantics * Equivalent Sequence * Plain Text * Convertibility

The character sets of many existing standards are incorporated within the Unicode Standard. For example, Latin-1 character set is its first 256 characters. The Unicode Standard includes the repertoire of

characters from many other international, national and corporate standards as well.

Duplicate encoding of characters is avoided by unifying characters within scripts across languages; characters that are equivalent in form are given a single code. Chinese/Japanese/Korean (CJK) consolidation is achieved by assigning a single code for each ideograph that is common to more than one of these languages. It does this instead of providing a separate code for the ideograph each time it appears in a different language. (These three languages share many thousands of identical characters because their ideograph sets evolved from the same source.)

Dynamic character composition allows marked character creation. Each character and diacritic or vowel mark is encoded separately, which allows the characters to be combined to create a marked character (such as é). The Unicode Standard also provides single codes for marked characters in order to provide consistency with many preexisting character set standards.

Characters are stored in logical order. The Unicode Standard includes characters to specify changes in direction when scripts of different directionality are mixed. For all scripts Unicode text is in logical order within the memory representation, corresponding to the order in which text is typed on the keyboard. The Unicode Standard specifies an algorithm for the presentation of text of opposite directionality, for example, Arabic and English; as well as, occurrences of mixed directionality text within.

**Assigning Character Codes**

A single 16-bit number is assigned to each code element defined by the Unicode Standard. Each of these 16-bit numbers is called a code value and, when referred to in text, is listed in hexadecimal form following the prefix "U+". For example, the code value U+0041 is the hexadecimal number 0041 (equal to the decimal number 65). It represents the character "A" in the Unicode Standard.

Each character is also assigned a unique name that specifies it and no other. For example, U+0041 is assigned the character name "latin capital letter a". U+0A1B is assigned the character name "gurmukhi letter cha". These Unicode names are identical to the ISO/IEC 10646 names for the same characters.

The Unicode Standard groups characters together by scripts in code blocks. A script is any system of related characters. The standard retains the order of characters in a source set where possible. When the characters of a script are traditionally arranged in a certain order— alphabetic order, for example—the Unicode Standard arranges them in its code space using the same order whenever possible. Code blocks vary greatly in size. For example, the Cyrillic code block doesn't exceed 256 code values, while the CJK code block has a range of thousands of code values.

Code elements are grouped logically throughout the range of code values (called the codespace). The coding starts at U+0000 with the standard ASCII characters, and continues with Greek, Cyrillic, Hebrew, Arabic, Indic and other scripts, followed by symbols and punctuation. The code space continues with hiragana, katakana, and bopomofo. The unified Han ideographs are followed by the complete set of modern hangul. The surrogate range of code values is reserved for future expansion. Towards the end of the codespace is a range of code values reserved for private use, followed by the compatibility zone; the compatibility zone contains character variants that are encoded only to transcoding to earlier standards and old implementations. The last two code values are reserved—U+FFFE and U+FFFF.

A range of code values are reserved as user space. Those code values have no universal meaning, and may be used for characters specific to a program or by a group of users for their own purposes. For example, a group of choreographers may design a set of characters for dance notation and encode the characters using code values in user space. A set of page-layout programs may use those same code values as control codes to position text on the page. The main point of user space is that the Unicode Standard assigns no meaning to these code values, and reserves them as user space, promising never to assign them meaning in the future.

**Conformance to the Unicode Standard**

The Unicode Standard specifies unambiguous requirements for conformance in terms of the principles and encoding architecture it embodies. A conforming implementation has the following characteristics, as a minimum requirement:

- characters are 16-bit units;
- characters are interpreted with Unicode semantics;
- unassigned codes are not used; and,
- unknown characters are not corrupted.

The full conformance requirements are available within The Unicode Standard, Version 2.0, Addison Wesley, 1996.

**For Further Information**

The fundamental source of information about Unicode is *The Unicode Standard*, Version 2.0, published by Addison Wesley, 1996. This book comes with a CD-ROM that contains character names and properties, as well as tables for mapping and transcoding. The Unicode Standard, Version 2.0, as well as Proceedings of the International Unicode Conferences may be ordered from the Unicode Consortium by using the Publications Order Form. Updates and Errata of the Unicode Standard are posted on this web site.

**Chapter 1—Introduction**

This excerpt from the book *The Unicode Standard*, Version 2.0 has been reformatted and edited for use on the web.

The Unicode Standard is a fixed-width, uniform encoding scheme for written characters and text. The repertoire of this international character code for information processing includes characters for the major scripts of the world, as well as technical symbols in common use. The Unicode character encoding treats alphabetic characters, ideographic characters, and symbols identically, which means that they can be used in any mixture and with equal facility. The Unicode Standard is modeled on the ASCII character set, but uses a 16-bit encoding to support fall multilingual text. No escape sequence or control code is required to specify any character in any language.

The Unicode Standard specifies a numerical value and a name for each of its characters; in this respect, it is similar to other character encoding standards from ASCII onwards (see Figure 1-1). The Unicode Standard is code-for-code identical with International Standard ISO/IEC 10646-1:1993. Information Technology—Universal Multiple-Octet Coded Character Set (UCS)-Part 1: Architecture and Basic Multilingual Plane.

As well as assigning character codes and names, the Unicode Standard provides other information not found in conventional character set standards, but crucial for using character encoding in implementations. The Unicode Standard defines properties for characters and includes application data such as case mapping tables and mappings to the repertoires of international, national, and industry character sets. The Unicode Consortium provides this additional information to ensure consistency in interchange of Unicode data.

**1.1 Design Goals**

The primary goal of the development effort for the Unicode Standard was to remedy two serious problems common to most multilingual computer programs: overloading of the font mechanism when encoding characters, and use of multiple, inconsistent character codes due to conflicting national and industry character standards. The ASCII 7-bit code space and its 8-bit extensions, although used in most computing systems, are limited to 128 and 256 code positions, respectively. These 7- and 8-bit code spaces are inadequate in the global computing environment.

when the Unicode project began in 1988, groups most affected by the lack of a consistent international character standard included the publishers of scientific and mathematical software, newspaper and book publishers, bibliographic information services, and academic researchers. More recently, the computer industry has adopted an increasingly global outlook, building international software that can be easily adapted to meet the needs of particular locations and cultures. The explosive growth of the Internet has added to the demand for a character set standard that can be used all over the world.

The designers of the Unicode Standard envisioned a uniform method of character identification which would be more efficient and flexible than previous encoding systems. The new system would be complete enough to satisfy the needs of technical and multilingual computing and would encode a broad range of characters for professional quality typesetting and desktop publishing worldwide.

The original design goals of the Unicode Standard were established as:

- Universal The repertoire must be large enough to encompass all characters that were likely to be used in general text interchange, including those in major international, national, and industry character sets.

- Efficient Plain text, composed of a sequence of fixed-width characters. provides an extremely useful model because it is simple to parse: software does not have to maintain state, look for special escape sequences, or search forward or backward through text to identify characters.

- Uniform A fixed character code allows efficient sorting. searching, display. and editing of text.

- Unambiguous. Any given 16-bit value always represents the same character

Figure 1-2 demonstrates some of these features, contrasting Unicode encoding to mixtures of single-byte character sets, with escape sequences to shift the meanings of bytes.

**1.2 Coverage**

The Unicode Standard, Version 2.0 contains 38,885 characters from the world's scripts. These characters are more than sufficient not only for modem communication, but also for

the classical forms of many languages. Languages that can be encoded include Russian3 Arabic, Anglo-Saxon, Greek) Hebrew, Thai, and Sanskrit. The unified Han subset contains 20,902 ideographic characters defined by national and industry standards of China, Japan, Korea) and Taiwan. In addition, the Unicode Standard includes mathematical operators and technical symbols) geometric shapes) and dingbats. Overall character allocation and the code ranges are detailed in Chapter 2, General Structure.

Included in the Unicode Standard are characters from all major international standards approved and published before December 31, 1990, in particular, the ISO International Register of Character Sets, the ISO/lEC 6937 and ISO/IEC 8859 families of standards, as well as ISO/IEC 8879 (SGML). Other primary sources included bibliographic standards used in libraries (such as ISO/lEC 5426 and ANSI Z39.64), the most prominent national standards, and various industry standards in very common use (including code pages and character sets from Adobe, Apple, Fujitsu. Hewlett-Packard, IBM, Lotus, Microsoft, NEC, WordPerfect, and Xerox). The complete Hangul repertoire of Korean National Standard KS C 5601 was added in Version 2.0. For a complete list of ISO and national standards used as sources, see the bibliography.

The Unicode Standard does not encode idiosyncratic, personal. novel, rarely exchanged, or private-use characters, nor does it encode logos or graphics. Artificial entities, whose sole function is to serve transiently in the input of text, are excluded. Graphologies unrelated to text, such as musical and dance notations, are outside the scope of the Unicode Standard. Font variants are explicitly not encoded. The Unicode Standard includes a Private Use Area, which may be used to assign codes to characters not included in the repertoire of the Unicode Standard.

The Unicode Consortium (see Section 1.4, The Unicode Consortium) periodically develops proposals for new scripts. The Consortium welcomes the submission of new characters for possible inclusion in the Unicode Standard. (For instructions on how to submit characters to the Unicode Consortium, see Appendix B, Submitting New Characters.)

**1.3 About this book**

This book defines Version 2.0 of the Unicode Standard. The general principles and architecture of the Unicode Standard, requirements for conformance, and guidelines for implementers precede the actual coding information. The accompanying CD-ROM carries tables of use to implementers.

Chapter 2 sets forth the fundamental principles underlying the Unicode Standard and covers specific topics such as text processes, overall character properties, and the use of non-spacing marks.

Chapters constitutes the formal statement of conformance. It opens with the conformance clauses themselves, which are followed by sections that define more precisely terms used in the clauses. The remainder of this chapter presents the normative algorithms for three processes: the canonical ordering of combining marks, the encoding of Korean Hangul syllables by conjoining jamo, and the formatting of bidirectional text.

Chapter 4 describes character properties, both normative (required) and informative. Since code charts alone are not sufficient for implementation, the Unicode Standard also specifies character properties, some of which are required for conformance.

Chapter 5 discusses implementation issues, including compression, strategies for dealing with unknown and missing characters, and transcoding to other standards.

Chapter 6 contains character block descriptions, part of the coding information in the Unicode Standard. A character block generally contains characters from a single script (for example. Tibetan) or is a collection of a particular type of character (for example, Mathematical Operators). A character block description gives basic information about the script or collection and may discuss specific characters.

Chapter 7 presents the individual characters, arranged by character block An overview of a particular character block is given by means of a code chart. With the exception of the blocks for East Asian ideographs and Korean hangul syllables, the individual characters of a block are identified in the accompanying names list

Chapter 8 provides a radical/stroke index to East Asian ideographs.

The major table on the CD-ROM is the Unicode Character Database, which gives character codes, char-

acter names (with Version 1.0 name if different), character properties, and decompositions for decomposable or compatibility characters. The CD-ROM also includes property-based mapping tables (for example, tables for case) and transcoding tables for international, national, and industry character sets (including the Han cross-reference table). (For the complete contents of the CD-ROM, see its READ ME file.)

Notational Conventions

Throughout this book, certain typographic conventions are used. In running text, an individual Unicode value is expressed as U+nnnn, where nnnn is a four digit number in hexadecimal notation, using the digits 0–9 and the letters A–F (for 10 through 15 respectively). In tables, the U+ may be omitted for brevity

- U+0416 is the Unicode value for the character named CYRILLIC CAPITAL LETTER ZHE.

A range of Unicode values is expressed as U+xxxr–U+yyyy or U+xxxx–U+yyyy, where xxxx and yyyy are the first and last Unicode values in the range, and the arrow or long dash indicates a contiguous range.

- The range U+0900–U+097F contains 128 character values.

Unicode characters have unique names, which are identical to those of the English language version of International Standard ISO/IEC 10646. Unicode character names contain only uppercase Latin letters A through Z, space, and hyphen-minus; this convention makes it easy to generate computer language identifiers automatically from the names. Unified East Asian ideographs are named CJK UNIFIED IDEOGRAPH-X, where X is replaced with the hexadecimal Unicode value; for example, CJK UNIFIED IDEOGRAPH-4E00. The names of Hangul syllables are generated algorithmically; for details, see Hangul Syllable Names in Section 3.10, Combining Jamo Behavior

In running text, a formal Unicode name is shown in small capitals (for example, GREEK SMALL LETTER MU), and alternative names (aliases) appear in italics (for example, umlaut).

Italics are also used to refer to a text element that is not explicitly encoded (for example. pasekh alef), or to set off a foreign word (for example, the Welsh word ynghyd).

The symbols used in the character names list are described at the beginning of Chapter 7, Code Charts.

In the text of this book, the word "Unicode" if used alone as a noun refers to the Unicode Standard or a Unicode character value.

**1.4 The Unicode Consortium**

The Unicode Consortium was incorporated in January 1991, under the name Unicode, Inc., to promote the Unicode Standard as an international encoding system for information interchange, to aid in its implementation, and to maintain quality control over future revisions. The Unicode Consortium is the central focus and contact point for conducting these activities.

To further these goals, the Unicode Consortium cooperates with the International Organization for Standardization (ISO). The Consortium holds a Class C liaison membership with ISO/IEC JTCI/SC2; it participates both in the work of JTC1/SC2/WG2 (the working group within ISO responsible for computer character sets) and in the work of the Ideographic Rapporteur Group of WG2. The Consortium is a member company of ANSI Subcommittee X3U. In addition, member representatives in many countries also work with their national standards bodies.

A number of standards organizations are Liaison Members of the Unicode Consortium. ECMA (a European-based organization for standardizing information and communication systems), Association of Common Chinese Code of the Center for Computer & Information Development Research (China), and the Technical Committee on Information Technology of the Viet Nam General Department for Standardization, Metrology; and Quality Control CTCVN/TC), and the WG2 standards committee of Korea.

Membership in the Unicode Consortium is open to organizations and individuals anywhere in the world who support the Unicode Standard and who would like to assist in its extension and widespread implementation. Full and Associate Members represent a broad spectrum of corporations and organizations in the computer and information processing industry. The Consortium is supported through the volunteer efforts of employees of member companies and individual members, and financially through membership dues.

The Unicode Technical Committee

The Unicode Technical Committee (UTC) is the working group within the Consortium responsible for the creation, maintenance, and quality of the Unicode Standard. The UTC controls all technical input to the standard and makes associated content decisions. UTC members represent the companies that are Pull and Associate Members of the Consortium. Observers are welcome to attend UTC meetings and may participate

in the discussions, since the intent of the UTC is to act as an open forum for the free exchange of technical ideas.

**1.5 The Unicode Standard and ISO/IEC 10646**

During 1991, the Unicode Consortium and the International Organization for Standardization (ISO) recognized that a single, universal character code was highly desirable. Mutually acceptable changes were made to Version 1.0 of the Unicode Standard and to the first ISO/TEC Draft International Standard DIS 10646.1, and their repertoires were merged into a single character encoding in January 1992. After international ballot and editorial changes to accommodate comments, the final ISO standard was published in May 1993 as ISO/IEC 10646-1:1993, Information Technology-Universal Multiple-Octet Coded Character Set (UCS)-Part 1: Architecture and Basic Multilingual Plane.

In accord with the merger agreement, a revision of the Unicode Standard was published in 1993 as Unicode Technical Report #4, with the title: The Unicode Standard, Version 1.1, Prepublication Edition. Version 1.1 of the Unicode Standard specified a repertoire and set of code assignments identical to those of the new ISO/IEC standard.

After the initial release of ISO/IEC 10646 and the Unicode Standard Ver. 1.1, both ISO JTC1/SC2/WG2 (the ISO working group responsible for ISO/IEC 10646) and the Unicode Technical Committee continued to develop the merged standard. These developments lead to Version 2.0 of the Unicode Standard, incorporating the first seven amendments made to or proposed for ISO/IEC 10646. (For details, see Appendix C. Relationship to ISO/IEC 10646. and Appendix D, Cumulative Changes)

**Supported Scripts**

The Unicode Character Standard primarily encodes scripts rather than languages. That is, where more than one language shares a set of symbols that have a historically related derivation, the union of the set of symbols of each such language is unified into a single collection identified as a single script. These collections of symbols (i.e., scripts) then serve as inventories of symbols which are drawn upon to write particular languages. In many cases, a single single script may serve to write tens or even hundreds of languages (e.g., the Latin script). In other cases only one language employs a particular script (e.g., Hangul).

The primary scripts currently supported by Unicode 2.0 are:

* Arabic * Gurmkhi * Lao * Armenian * Han * Malayalam * Bengali * Hangul * Oriya * Bopomofo * Hebrew * Phonetic * Cyrillic * Hiragana * Tamil * Devanagari * Kannada * Telugu * Georgian * Katakana * Thai * Greek * Latin * Tibetan * Gujarati

In addition to the above primary scripts, a number of other collections of symbols are also encoded by Unicode. These collections, which may be referred to as secondary scripts (or as pseudo-scripts), consist of the following:

* Numbers * General Diacritics * General Punctuation * General Symbols * Mathematical Symbols * Technical Symbols * Dingbats * Arrows, Blocks, Box Drawing Forms, and Geometric Shapes * Miscellaneous Symbols * Presentation Forms

Unicode provides a unique number for every character,
no matter what the platform,
no matter what the program,
no matter what the language.
—The Unicode Organization

# C
# The Compact Disc

The CD is a relatively new storage medium. Because of its great success, both for sound and for digital recording, and because of its complexity, it deserves special mention.

## C.1 Capacity

The compact disc (CD) was developed by Philips and Sony, starting around 1974. In June, 1980, the two companies agreed on a common CD standard. In 1981 this standard was accepted by the Digital Audio Disc committee, and has been being used ever since. The standard includes details of the signal format, disc material, and error correcting code. Notice the spelling 'disc', as opposed to a magnetic 'disk'.

The immediate success of the audio CD motivated the development of the CD-ROM (for digital information) in 1985, the CD-V (for video), and the CD-I (interactive), both in 1987.

To understand the capacity requirements of the CD, let's consider the way sound is recorded digitally. We can consider sound as a function $s(t)$ of the time $t$. The function goes up and down according to the intensity of the sound, so its frequency may change all the time. However, at each point in time $s(t)$ has a value, representing the intensity (amplitude) of the sound. The principle of digital recording is to look at $s(t)$ periodically, and to record its value as a binary number. This process is called *sampling*.

Clearly, if we don't sample a sound often enough, we don't get enough information about it, and thus cannot reconstruct it (play it back) correctly. On the other hand, we don't want to sample too often. This is where Nyquist theorem (developed in 1928) comes to the rescue. It declares: "If the sampling is done at a rate equal to twice the highest frequency of the sound, then the sound can always be fully reconstructed."

The human ear can normally hear sounds up to a frequency of about 20–22KHz, which means that sound recording should use a sample rate of about 40–44KHz. The CD standard specifies a rate of 44.1KHz. To record an hour of music (The maximum capacity of a CD is 74 minutes, 33 seconds) thus requires a sample of $44100 \times 3600 = 158760000$ numbers. To get high quality sound, the CD standard specifies 16-bit samples, bringing the total number of bits to 2,540,160,000. Stereo sound requires two numbers per sample, resulting in 5,080,320,000 bits. This is equivalent to 635,040,000 bytes. Synchronization and modulation information, and parity bits for error correction (see below) bring this number even higher. The CD thus has impressive storage capacity.

(We say that stereo music requires two channels, while mono requires just one. The CD standard specifies three ways of recording sound: (1) stereo samples, recorded on the two channels; (2) mono samples recorded twice, on both channels; (3) mono samples on one channel with the other channel left empty. The standard does not allow for two different mono samples to be recorded on both channels.)

The fundamental unit of data on a CD-ROM is the sector. Every CD-ROM is composed of a given amount of sectors. The amount of user data contained in each sector depends on the mode type of the sector. Mode 1 sectors (the most common) contain 2048 bytes of user data. Most PC, Macintosh, and UNIX CD-ROMs are mode 1. Mode 2 sectors contain 2336 bytes of user data. Examples of mode 2 CD-ROMs are

XA, Photo-CD, and CD-I. Older recordable CDs (CD-R) were labeled "74 min., 650 Mbytes." Such a CD had 336,075 sectors, which translates to 74 min., 43 sec. playing time. The capacity was $336{,}075 \times 2048 = 688{,}281{,}600$ bytes or 656.396484375 Mbytes (where mega is $2^{20} = 1{,}048{,}576$). However, it is possible to record more sectors on a CD, thereby increasing its capacity. This is somewhat risky, since the extra sectors approach the edge of the CD, where they can easily get scratched or dirty. New CD-Rs are labeled as 700 Mbyte or 80 minutes. They have 359,849 blocks for a total capacity of $359{,}849 \times 2048 = 736{,}970{,}752$ bytes or 702.83 Mbytes.

In order to read 702.83 Mbytes in 80 minutes, a CD drive has to read 150Kbytes per second. This speed is designated 1X and was typical for CD players made before 1993. Currect CD drives (year 2002) can read CDs at speeds of up to 56X, where 700 Mbytes are read in just under 86 seconds!

▸ **Exercise C.1:** What is the capacity of a CD-ROM with 345,000 sectors in mode 1 and in mode 2?

## C.2 Description

Physically, the CD is a disc, 1.2 millimeters thick, with a 120mm diameter. The hole, at the center, is 15mm in diameter. The distance between the inner and outer circumferences is thus $(120 - 15)/2 = 52.5$ mm. Of this, only 35mm are actually used, leaving a safety margin both inside and outside. The information is recorded on a metallic layer (typically aluminum, silver or gold), that's .5–1$\mu$ thick (where $\mu$ or micron, is $10^{-6}$ meter). Above this layer there is a protective lacquer coating (10–30$\mu$ thick), with the printed label. Below the metal layer there is the disc substrate, normally made of transparent polycarbonate. It occupies almost the entire thickness of the disc. Since the protective layer is so thin, any scratches on the label can directly damage the metallic layer. Even pen marks can bleed through and cause permanent damage. On the other hand, scratches on the substrate are usually handled by the error correcting code (see below).

The digital information is recorded in pits arranged in a spiral track that runs from the inner circumference to the outer one. The pits are extremely small. Each is .5$\mu$ wide and .11$\mu$ deep. Pit lengths range from .833$\mu$ to 3.56$\mu$. The track areas between pits are called *land*. The distance between successive laps of the track is 1.6$\mu$. As a result, the track makes 22,188 revolutions in the 35 mm recording area. Its total length is about 3.5 miles. The information is recorded such that any edge of a pit corresponds to binary 1, and the area in the pits and in the lands (between pits) corresponds to consecutive zeros. To reduce fabrication problems, the pits should not be too short or too long, which means that the number of binary ones recorded should be carefully controlled (see below).

To read the disc, a laser beam is focused on the track through the disc substrate, and its reflection measured. When the beam enters a pit, the reflection drops to almost zero, because of interference. When it leaves the pit, the reflection goes back to high intensity. Each change in the reflection is read as binary one. To read the zeros, the length of a pit, and between pits, must be measured accurately. The disc must thus rotate with a constant linear velocity (CLV). This implies that, when the reading laser moves from the inner parts of the track to the outer ones, the rotational speed has to decrease (from 500 RPM to about 200 RPM). The track contains synchronization information used by the CD player to adjust the speed. The aim of the player is to read 4.3218 million bits per second, which translates to a CLV of 1.2–1.4 meter/sec.

The CD was made possible by two advances, a technological one (high precision manufacturing), and a scientific one (error-correcting codes). Here are some numbers illustrating the two points:

1. Remember the old LP vinyl records? Information is stored on such a record in a narrow spiral groove whose width is about the diameter of a human hair (about 0.05 mm). More than 30 laps of the CD track would fit in such a groove!

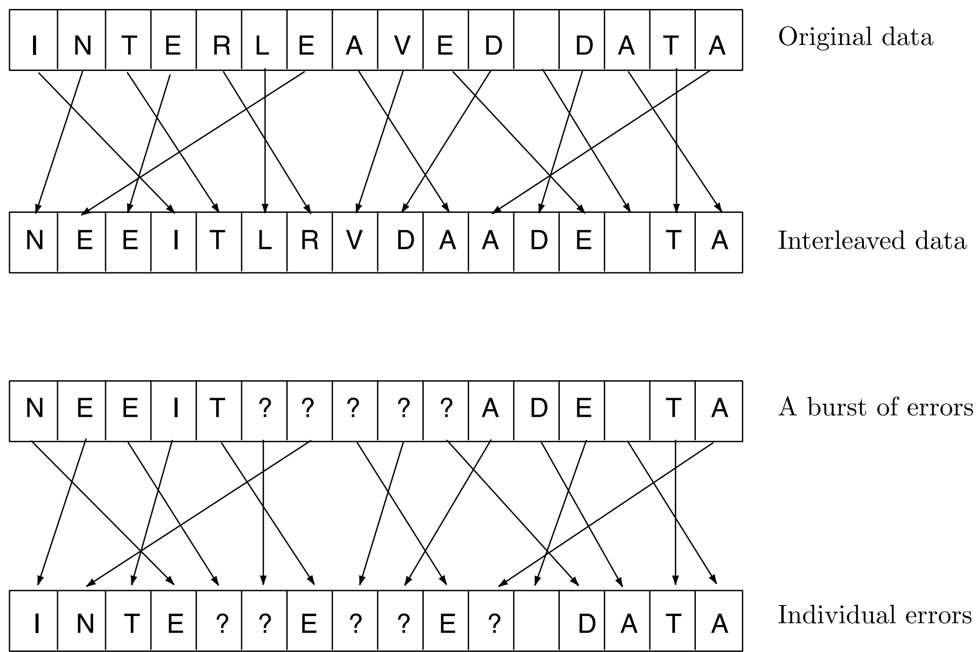
2. A CD is read at a rate of 4.3218 million bits per second. The reading is done by measuring the reflection of a laser beam from the track, and is thus sensitive to surface scratches, to imperfections in the track, and to surface contamination because of fingerprints, dust particles, etc. All these factors introduce errors in reading, but the digital nature of the recording allows for error correction.

(A note about cleaning. Clean a CD only if you must, since cleaning may create many invisible scratches and cause more harm than good. Use a soft, moistened cloth, and work radially, from the center to the rim. Cleaning with a circular motion might create a long scratch paralleling a track, thus introducing a long burst of errors.)

## C.3 Error-Correction

It is obvious that reading a CD-ROM must be error free, but error correction is also important in an audio CD, because one bad bit can cause a big difference in the note played. Consider the two 16-bit numbers 0000000000000000 and 1000000000000000. The first represents silence and the second, a loud click. Yet they differ by one bit only! The size of a typical dust particle is $40\mu$, enough to cover more than 20 laps of the track, and cause several bursts of errors (Figure C.1b). Without extensive error correction, the music would sound like one big scratch.

Any error correction method used must be very sophisticated, since the errors may come in bursts, or may be individual. The use of parity bits makes it possible to correct individual errors, but not a burst of consecutive errors. This is why *interleaving* is used, in addition to parity bits. The principle of interleaving is to rearrange the samples before recording them on the CD, and to reconstruct them after they have been read. This way a burst of errors during the read is translated to individual errors (Figure C.1a), that can then be corrected by their parity bits.



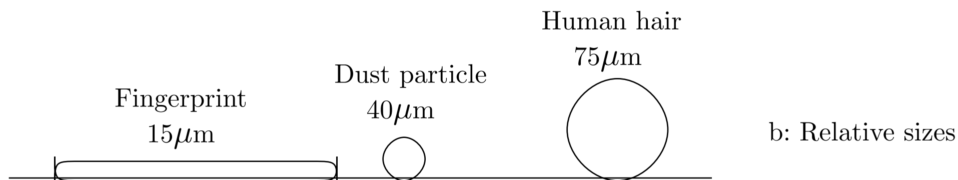




**Figure C.1**. Properties of a CD

The actual code used in CDs is called the Cross Interleaved Reed-Solomon Code (CIRC). It was developed by Irving S. Reed and Gustave Solomon at Bell labs in 1960 and is a powerful code. One version of

this code can correct up to 4000 consecutive bit errors, which means that even a scratch as long as three millimeters can be tolerated on a CD. The principle of CIRC is to use a geometric pattern that is so familiar that it can be reconstructed even if large parts of it are missing. It's like being able to recognize the shape of a rectangular chunk of cheese after a mouse has nibbled away large parts of it.

Suppose that the data consists of the two numbers 3.6 and 5.9. We consider them to be the $y$ coordinates of two-dimensional points and we assign them $x$ coordinates of 1 and 2. We thus end up with the points $(1, 3.6)$ and $(2, 5.9)$. We consider those points the endpoints of a line and we calculate four more points on this line, with $x$ coordinates of 3, 4, 5, and 6. They are $(3, 8.2)$, $(4, 10.5)$, $(5, 12.8)$, and $(6, 15.1)$. Since the $x$ coordinates are so regular, we only need to store the $y$ coordinates of these points. We thus store (or write on the CD) the six numbers 3.6, 5.9, 8.2, 10.5, 12.8, and 15.1.

Now suppose that two errors occur among those six numbers. When the new sequence of six numbers is checked for the straight line property, the remaining four numbers can be identified as being collinear and can still be used to reconstruct the line. Once this is done, the two bad numbers can be corrected since their $x$ coordinates are known. Even three bad numbers out of those six can be corrected since the remaining three numbers would still be enough to identify the original straight line.

It is even more reliable to start with three numbers $a$, $b$, and $c$, to convert them to the points $(1, a)$, $(2, b)$, and $(3, c)$, and to calculate the (unique) parabola that passes through these points. Four more points, with $x$ coordinates of 4, 5, 6, and 7, can then be calculated on this parabola. Once the seven points are known they provide a strong pattern. Even if four of the seven get corrupted the remaining three can be used to reconstruct the parabola and correct the four bad ones. It may happen that three numbers will get corrupted in such a way that they will lie on a new parabola, but this is extremely rare.

## C.4 Encoding

The coding process starts with a chunk of 24 data bytes (twelve 16-bit samples, or six 32-bit stereo samples), and ends with 32 bytes of scrambled data and parity which are written on the CD (after EFM modulation, see below) as a frame. For each frame the hardware generates another byte called the subcode (see below). The overhead is thus $8 + 1$ bytes for every 24 bytes, or 37.5%. It can be shown that even a burst of about 4000 bad data bits can be completely corrected by this code, which justifies the overhead.

Before writing a 33-byte frame on the CD, the recording hardware performs Eight-to-Fourteen Modulation (EFM). Each byte is used as a pointer to a table with 14-bit entries, and it is the table entry which is finally written on the CD. The idea is to control the length of the pits by controlling the number of consecutive zeros. It is easier to fabricate the pits if they are not too short and not too long. EFM modulation produces 14-bit numbers that have at least two binary zeros and at most ten zeros between successive binary ones. There are, of course, $2^{14} = 16384$ 14-bit numbers, and 267 of them satisfy the condition above. Of those, 256 were selected and placed in the table. (Two more are used to modulate the subcodes.) Here are the first ten entries of the table:

| 8-bit pointer | 14-bit pattern |
|---|---|
| 00000000 | 01001000100000 |
| 00000001 | 10000100000000 |
| 00000010 | 10010000100000 |
| 00000011 | 10001000100000 |
| 00000100 | 01000100000000 |
| 00000101 | 00000100010000 |
| 00000110 | 00100001000000 |
| 00000111 | 00100100000000 |
| 00001000 | 01001001000000 |
| 00001001 | 10000001000000 |
| 00001010 | 10010001000000 |

There is still the possibility of a 14-bit number ending with a binary 1, followed by another 14-bit number starting with a 1. To avoid that, three more *merging bits* are recorded on the track after each 14-bit number. Two of them are zeros and the third one is selected to suppress the signal's low frequency component. This process results in about 3 billion pits being fabricated on the track.

To summarize the encoding process, it starts with a group of 24 bytes (192 bits), which are encoded into $24 + 8 + 1 = 33$ bytes. These are translated into 33 14-bit numbers, which are recorded, each with 3 merging bits, on the track. A synchronization pattern of 24 bits (plus 3 merging bits) is also recorded at the start of the track. The original 192 bits thus require $24 + 3 + 33 \times (14 + 3) = 588$ bits to be recorded, which corresponds to an overhead of $(588 - 192)/192 = 396/192 \approx 206\%$. To record 635,040,000 data bytes, we thus need to record about 1,309,770,000 bytes (about 10.5 billion bits)! The following summarizes the format of a frame.

| sync. 24 | merg. bits 3 | byte 0 14 | merg. bits 3 | byte 1 14 | merg. bits 3 | ··· ··· | byte 32 14 | merg. bits 3 |
|---|---|---|---|---|---|---|---|---|

The CD player has to read the information at the rate of 44,100 samples per second (corresponding to a 44.1KHz sampling rate). Each (stereo) sample consists of two 16-bit numbers, so there are six samples per frame. This is why $44100/6 = 7350$ frames have to be read each second, which translates to a bit rate of $7350 \times 588 = 4321800$bits/sec.

## C.5 The Subcode

As stated earlier, each frame contains an additional byte called the subcode. The eight bits of the subcode are labeled PQRSTUVW. In an audio CD, only the P & Q bits are used. When the CD is read, the subcode bytes from 98 consecutive frames are read and assembled into a *subcode frame*. Recall that the frames are read at a rate of 7350 per second. Since $7350/98 = 75$, we get that 75 subcode frames are assembled each second by the CD player! A subcode frame gets the two 14-bit sync patterns 00100000000001 and 00000000010010 (two of the patterns not used by the EFM table). The frame is then considered eight 98-bit numbers. The 98 bits of P give the CD player information about the start and end of each song (each music track). The bits of Q contain more information, such as whether the CD is audio or ROM, and whether other digital recorders are legally permitted to copy the CD.

## C.6 Data Readout

To read the data off the CD track, a narrow, focused laser beam is shined on it, and the reflection is measured. The process is based on two physical principles: refraction and interference. Refraction is the case where a beam of light passes from one medium to another; both its wave length and its speed change, causing it to bend. If it passes from a rare medium (low index of refraction) to a dense one (high index of refraction), both its speed and its wave length decrease.

The refraction index of vacuum is defined as 1 (that of air is virtually the same), and that of the substrate must be 1.55. The wavelength of the laser beam in the air is 780 nanometers, so in the substrate it drops to about 500 nanometers. The height of a pit is about a quarter of that (actually a little less than a quarter), or 110 nanometers. The area of the beam hitting the metallic layer is about twice the area of a pit (Figure C.2a), so half of it hits the pit, and the other half, the land around the pit. The two halves are reflected with a phase difference of one-half the wavelength ($\lambda/2$), causing destructive interference (Figure C.2b). This is why the reflection intensity drops when the beam scans a pit. The reflection drops to about 25% of its maximum because the height of a pit is not exactly $\lambda/4$, and because the parts of the beam hitting the pit and the land are not exactly equal.

## C.7 Fabrication

The process of fabricating CDs consists of the following steps:

1. The digital information has to be recorded on a master tape. It may consist of sound samples for an audio CD, or data files for a CD-ROM. The tape has to checked carefully, since any errors would be transferred to all the final CDs made. Old music tapes can be translated to digital but the quality will normally be low.

2. The master tape is transferred to a CD master tape, with the subcode information.

3. A master CD is made of a glass disc. It is about 240mm in diameter and 5.9mm thick. It is polished smooth, and covered with a thin layer of photoresist material. The disc is then checked with a laser beam to make sure it is flat, and the thickness of the photoresist material is uniform.

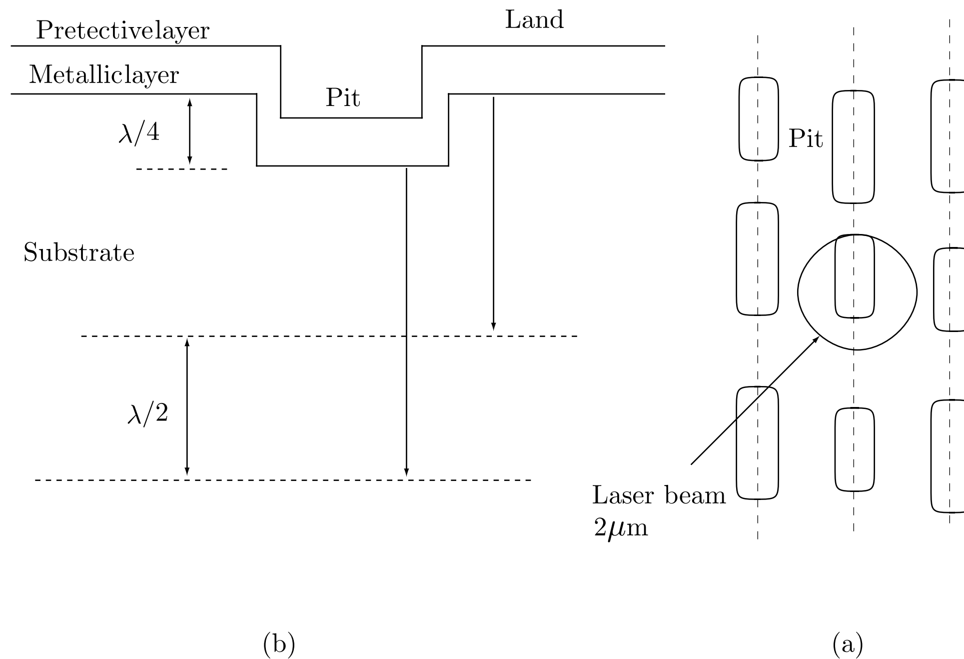



**Figure C.2**. Reading a CD

4. A special "cutting" machine is used to fabricate the track on the disc. A computer reads the CD master tape, encodes the information (i.e., performs the interleaving, calculates the parity bits and does the EFM modulation) and uses a laser beam to expose the disc where the pits should be. This process requires a quite environment, so the machine must be isolated from vibrations.

The glass disc is then flooded by a chemical that etches away the exposed areas of the photoresist (not the glass), thus creating the pits. A laser beam monitors this process by measuring the depth of the pits as they are being formed.

5. A thin silver layer is evaporated on the disc, which is then played (or read), to ensure reliability.

6. A container with electrolytic solution is used to make a "negative" of the silvered glass disc. The negative is made of nickel and is called the "father". Making the father destroys the photoresist layer, so the glass master can no longer be used. Several "mothers" may be made of the single father, and several "sons" (also called stampers) made of each "mother".

7. A stamper is used to made many copies of the CD. Each copy is made by injecting plastic over the stamper, thus transferring the pits to the plastic. This becomes the substrate. A metallic layer is evaporated over the track, followed by a top layer of acrylic, and by the printed label.

The entire process requires high precision machines, a very clean environment, and isolation from vibrations. In spite of this, the final price of a CD depends on market demands and popularity of the performer, more than on the technical difficulties of fabrication.

## C.8 The CD-ROM Format

Because of the large capacity of the audio CD, it can be used to store large quantities of digital data of any kind. This is the principle of the CD-ROM. It can be used as a read-only memory with a capacity of about 600Mbytes. The data is recorded as pits on a track, same as an audio CD, but the format is different. Data is recorded on an audio CD in 24-byte frames, with a subcode (of 1 byte per frame) for each group of 98 frames. In the CD-ROM, the basic data unit is a group of 98 frames, called a sector. The size of a sector is $98 \times 24 = 2352$ bytes, and it is divided into four parts: (1) a 12-byte sync pattern; (2) a 4-byte header; (3) 2048 bytes (=2K) of user data; (4) 288 bytes of system data (used for error correction, in addition to the CIRC code).

The maximum capacity of the CD-ROM is 333,000 sectors, which translates to 681,984,000 bytes (about

650.4Mbytes). If less data is required, the outer part of the CD remains blank, making it easier to handle the disc by hand. (Even 550Mbytes is quite a large capacity, equal to about 275,000 pages of text.) Because of the additional error correction information, CD-ROMs are extremely reliable. They typically create one uncorrectable error for every $10^{16}$ or $10^{17}$ bits read! For comparison, there are about $31.5\times10^{17}$ nanoseconds in a century.

The CD-ROM reader is not compatible with the audio CD player. It has the same laser, modulation, and error correction circuits, but instead of converting the information read to analog, the CD-ROM reader sends it to the computer as digital information in sectors of size 2K each. Today it is common to have a CD player and CD-ROM reader combined in one product. It uses the same laser to read both types of discs, and uses the Q subcode information to distinguish between them.

It is important to realize that the CD-ROM standard does not specify how to organize files on the disc, or even where to place the directory. This is why a CD-ROM for a PC cannot normally be read by a Macintosh. To overcome this, an additional standard has been developed by the High Sierra group—a group of CD-ROM makers—that specifies the logical organization of files on the disc. CD-ROMs made to this standard can be read by any operating system that has the High Sierra utility programs.

## C.9 Recordable CDs (CD-R)

Normal (nonrecordable) CDs are read-only; the content of such a CD is permanent and cannot be modified. It is obviously very desirable to be able to record data on a CD, which is why recordable CDs (CD-R) have been developed and are constantly being improved and becoming more reliable and cheaper. The principle of CD-R is to add a layer that can be modified by a laser beam. A nonrecordable CD has layers of (from bottom to top) plastic substrate, reflective aluminum-chromium, a protective acrylic lacquer overcoat, (some makers add another durable protective overcoat), and the label. In a CD-R, the layers are (1) plastic substrate, (2) reflective gold or aluminum, (3) an extra layer made of light-sensitive organic dye, (4) a protective acrylic lacquer overcoat, (some makers add another, fifth layer of durable protective overcoat), and (6) label.

Three types of dye are used in CD-R media. (1) Azo, Blue or Deep Blue. (2) Cyanine, Light Blue, Light Green, or Yellow. (3) Phthalocyanine. Silver or Gold.

In its original state, the dye is transparent enough so it does not block light reflection from the gold layer above. But when the dye is "zapped" by the writing laser (which has about ten times the power of the reading laser), a dark discoloration is created on the dye which blocks reflections and forms a permanent, readable mark. The dye molecules absorb enough energy during the writing process to break down physically and chemically, leaving an area that no longer reflects light. The marks made by the writing laser follow the same encoding scheme as the pits in a nonrecordable CD, so recordable CDs can be read by any CD drive.

When a recordable CD is read, the reading laser beam shines through those parts of the dye layer that haven't been burned. The beam is then reflected from the gold (or aluminum) layer, so the nonburned parts become lands. The zapping (or burning) of the dye is done by breaking its molecules, and is therefore permanent; the CD-R cannot be erased. The device that "burns" the CD-R is called a CD recorder or a CD-R drive (a popular name is a CD burner) and current drives can record CDs at speed of 24X to 40X (they can also read CDs, and do this at higher speeds).

▸ **Exercise C.2:** How long does it take to burn a CD-R at 24X and 40X speeds?

High quality CD-Rs have a gold reflective layer, so they look golden when viewed from the top (the label side). Low quality CD-Rs have an aluminum reflective layers. When viewed from the bottom, the green layer of dye may look either light-green (the high qulity CDs) or dark green (the low quality ones). Also, high-quality CD-Rs have unique serial numbers, written both in digits and in a bar code, thereby making it easy to identify and track them.

It's important to realize that the dye is fairly sensitive to light; it has to be in order for a laser to zap it fast. It is therefore important to avoid exposing CD-R discs to sunlight. The low-quality CD-Rs are especially sensitive, but it is good practice to keep CD-Rs in opaque containers both before and after recording data. Another consideartion is cleanliness. The laser beam that writes the CD-R has to penetrate through the thick plastic substrate, which is why scratches and especially fingerprints and dirt are much more dangerous to a recordable CD *before* writing than afterward. The fingerprint, dirt, or smudge can scatter the beam of the writing laser, possibly weakening it to the point where the burn it makes is too small

or too light to be read later. Thus, extra care must be taken in the handling of a recordable CD before any data is written on it.

## C.10 Summary

The discussion here has emphasized the main technical problems that had to be solved in order to achieve both the high-storage capacity, and the extreme reliability of the CD. The submicroscopic manufacturing details, and the complex circuits needed to read and decode the information at such high rates, make the CD player (and CD-ROM reader) a very complex consumer electronic device.

## C.11 DVD

DVD (digital versatile disc) is the next step in the CD saga. The DVD format was developed jointly by Philips and Sony between 1980 and 1984. We start with the name itself. Depending on who you listen to, DVD means "digital versatile disc," "digital video disc," or it is not an acronym and does not stand for anything.

DVD-ROM stands for DVD-Read Only Memory, a format (similar to the familiar CD-ROM) that is defined by Book A of the DVD specification. DVD-Video is defined by Book B of the specification. Basically, DVD-Video is DVD-ROM with an added application layer that limits its use to playing movies. Thus, a DVD-Video disc is DVD-ROM, but a DVD-ROM disc is not always DVD-Video. Divx stands for Digital Video Express. This is a DVD-Video with Triple DES encoding and a unique serial number added. This limits its application. A Divx player plays DVD-Video, but a DVD player will not play a Divx disc. DVD-Audio hasn't been not defined yet, but when it will, you will find it in Book C of the DVD specification. There are currently four competing proposals.

There are more DVD formats. DVD-R is DVD-Recordable, defined by Book D of the DVD specification. This type is currently made by Pioneer. DVD-RW stands for DVD-Rewritable. This type is rewritable up to 1,000 times. It has the same capacity as DVD-R, 3.95Gb, and this is expected to go up to 4.7Gb. A DVD-R/W drive will record DVD-R and rewrite DVD-R/W. Also, DVD-R/W discs can be read by first-generation DVD hardware. This backwards compatibility with old, existing players is, of course, a great marketing idea. Notice that DVD-R/W is not an official DVD format. It has been presented to the DVD Forum for ratification, but it faces competition from the already-accepted DVD-RAM, which isn't readable on anything but DVD-RAM.

The following paragraphs constitute a summary of DVD features and applications.

A TECHNICAL TRIUMPH

How is DVD different from CD? For greater data density, there are smaller pits, a more closely-spaced track and a shorter-wavelength red laser. The error correction is more robust. And thanks to Sony technology, the modulation scheme is more efficient. All this means that a standard DVD can hold 4.7 gigabytes of data—an amazing seven times the data capacity of a current Compact Disc! So you'll enjoy higher-resolution pictures, more channels of digital sound, richer graphics, and far more multimedia. Not enough? Dual-layer DVDs can hold more than twelve times the information of a CD on a single side. So there's no need to turn the disc over.

HOLLYWOOD SPECTACULAR

DVD is major news for anyone who owns a television and anyone who enjoys movies. Thanks to digital video decoding technology, DVD delivers far and away the best color, sharpness and clarity in home video. In fact, a DVD picture approaches the "D-1" TV studio production standard. And thanks to variable bit-rate MPEG2 compression, it all fits easily onto a single side of a 4-3/4-inch disc. In fact, a single-layer DVD can hold a two hour, 13 minute movie—with room to spare for Dolby AC-3(TM) discrete 5.1-channel digital sound in your choice of three languages! Dual-layer, single-sided discs can hold movies more than four hours long. And because DVD is an optical disc, you get instant access. You can play it repeatedly without wear and tear. You'll never need to rewind.

ULTIMATE MULTIMEDIA

Digital convergence is erasing the old distinctions between entertainment and information, recreation and education. So it's the perfect time for DVD-ROM. With up to 8.5 Gb capacity on a single side it's the optical disc that gives you more. More on-line capacity for software publishers. More room for large databases. And high-quality full-motion video for better interactive games. DVD-ROM drives are also

speed readers. Even a standard DVD-ROM blasts along at higher data transfer rates than even the fastest current CD-ROM. Since they're able to play your existing CD-ROMs, DVD-ROM drives also respect the past. With the recent development of DVD-Write Once and DVD-Rewritable, the DVD format is focused on the future. Best of all, DVD is supported by the world's leading hardware manufacturers and software companies (Table C.3).

| Format | Capacity per side | Hardware backers | Software backers |
|---|---|---|---|
| DVD-Video | 4.7 GB | DVD Forum (Hitachi, Matsushita, Mitsubishi, JVC, Pioneer, Sony, Toshiba, Philips, Thomson) | Time-Warner, MGM/UA, Columbia TriStar, Polygram, Disney |
| Divx | | Matsushita, Thomson, Zenith | Paramount, Universal, Dreamworks |
| DVD-R | 4.7 GB | DVD Forum | |
| DVD-RAM | 2.6 GB | Toshiba, Matsushita | |
| DVD+RW | 3.0 GB | Sony, Hewlett-Packard, Philips | |

**Table C.3**. DVD's many flavors

A compact disc player is perhaps the most sophisticated and electronically complex piece of home entertainment to ever reach the consumer.

—Ken C. Pohlmann, *The Compact Disc*

# D
# ISO Country Codes

| Country | A 2 | A 3 | Number |
|---|---|---|---|
| AFGHANISTAN | AF | AFG | 004 |
| ALBANIA | AL | ALB | 008 |
| ALGERIA | DZ | DZA | 012 |
| AMERICAN SAMOA | AS | ASM | 016 |
| ANDORRA | AD | AND | 020 |
| ANGOLA | AO | AGO | 024 |
| ANGUILLA | AI | AIA | 660 |
| ANTARCTICA | AQ | ATA | 010 |
| ANTIGUA AND BARBUDA | AG | ATG | 028 |
| ARGENTINA | AR | ARG | 032 |
| ARMENIA | AM | ARM | 051 |
| ARUBA | AW | ABW | 533 |
| AUSTRALIA | AU | AUS | 036 |
| AUSTRIA | AT | AUT | 040 |
| AZERBAIJAN | AZ | AZE | 031 |
| BAHAMAS | BS | BHS | 044 |
| BAHRAIN | BH | BHR | 048 |
| BANGLADESH | BD | BGD | 050 |
| BARBADOS | BB | BRB | 052 |
| BELARUS | BY | BLR | 112 |
| BELGIUM | BE | BEL | 056 |
| BELIZE | BZ | BLZ | 084 |
| BENIN | BJ | BEN | 204 |
| BERMUDA | BM | BMU | 060 |
| BHUTAN | BT | BTN | 064 |
| BOLIVIA | BO | BOL | 068 |
| BOSNIA AND HERZEGOWINA | BA | BIH | 070 |
| BOTSWANA | BW | BWA | 072 |
| BOUVET ISLAND | BV | BVT | 074 |
| BRAZIL | BR | BRA | 076 |
| BRITISH INDIAN OCEAN TERR, | IO | IOT | 086 |
| BRUNEI DARUSSALAM | BN | BRN | 096 |
| BULGARIA | BG | BGR | 100 |
| BURKINA FASO | BF | BFA | 854 |
| BURUNDI | BI | BDI | 108 |
| CAMBODIA | KH | KHM | 116 |
| CAMEROON | CM | CMR | 120 |
| CANADA | CA | CAN | 124 |
| CAPE VERDE | CV | CPV | 132 |
| CAYMAN ISLANDS | KY | CYM | 136 |
| CENTRAL AFRICAN REPUBLIC | CF | CAF | 140 |
| CHAD | TD | TCD | 148 |
| CHILE | CL | CHL | 152 |
| CHINA | CN | CHN | 156 |
| CHRISTMAS ISLAND | CX | CXR | 162 |
| COCOS (KEELING) ISLANDS | CC | CCK | 166 |
| COLOMBIA | CO | COL | 170 |
| COMOROS | KM | COM | 174 |
| CONGO | CG | COG | 178 |
| CONGO, (communist) | CD | COD | 180 |
| COOK ISLANDS | CK | COK | 184 |
| COSTA RICA | CR | CRI | 188 |
| COTE D'IVOIRE | CI | CIV | 384 |
| CROATIA (Hrvatska) | HR | HRV | 191 |
| CUBA | CU | CUB | 192 |
| CYPRUS | CY | CYP | 196 |
| CZECH REPUBLIC | CZ | CZE | 203 |
| DENMARK | DK | DNK | 208 |
| DJIBOUTI | DJ | DJI | 262 |
| DOMINICA | DM | DMA | 212 |
| DOMINICAN REPUBLIC | DO | DOM | 214 |
| EAST TIMOR | TP | TMP | 626 |
| ECUADOR | EC | ECU | 218 |
| EGYPT | EG | EGY | 818 |
| EL SALVADOR | SV | SLV | 222 |
| EQUATORIAL GUINEA | GQ | GNQ | 226 |
| ERITREA | ER | ERI | 232 |
| ESTONIA | EE | EST | 233 |
| ETHIOPIA | ET | ETH | 231 |
| FALKLAND ISLANDS | FK | FLK | 238 |
| FAROE ISLANDS | FO | FRO | 234 |
| FIJI | FJ | FJI | 242 |
| FINLAND | FI | FIN | 246 |
| FRANCE | FR | FRA | 250 |
| FRANCE, METROPOLITAN | FX | FXX | 249 |
| FRENCH GUIANA | GF | GUF | 254 |
| FRENCH POLYNESIA | PF | PYF | 258 |
| FRENCH SOUTH TERR. | TF | ATF | 260 |
| GABON | GA | GAB | 266 |
| GAMBIA | GM | GMB | 270 |

**Table D.1:** ISO country codes (1 of 3).

| Country | A 2 | A 3 | Number | Country | A 2 | A 3 | Number |
|---|---|---|---|---|---|---|---|
| GEORGIA | GE | GEO | 268 | LESOTHO | LS | LSO | 426 |
| GERMANY | DE | DEU | 276 | LIBERIA | LR | LBR | 430 |
| GHANA | GH | GHA | 288 | LIBYAN ARAB JAMAHIRIYA | LY | LBY | 434 |
| GIBRALTAR | GI | GIB | 292 | LIECHTENSTEIN | LI | LIE | 438 |
| GREECE | GR | GRC | 300 | LITHUANIA | LT | LTU | 440 |
| GREENLAND | GL | GRL | 304 | LUXEMBOURG | LU | LUX | 442 |
| GRENADA | GD | GRD | 308 | MACAU | MO | MAC | 446 |
| GUADELOUPE | GP | GLP | 312 | MACEDONIA, (YUGOSLAV REP.) | MK | MKD | 807 |
| GUAM | GU | GUM | 316 | MADAGASCAR | MG | MDG | 450 |
| GUATEMALA | GT | GTM | 320 | MALAWI | MW | MWI | 454 |
| GUINEA | GN | GIN | 324 | MALAYSIA | MY | MYS | 458 |
| GUINEA-BISSAU | GW | GNB | 624 | MALDIVES | MV | MDV | 462 |
| GUYANA | GY | GUY | 328 | MALI | ML | MLI | 466 |
| HAITI | HT | HTI | 332 | MALTA | MT | MLT | 470 |
| HEARD & McDONALD ISLS | HM | HMD | 334 | MARSHALL ISLANDS | MH | MHL | 584 |
| HOLY SEE (VATICAN) | VA | VAT | 336 | MARTINIQUE | MQ | MTQ | 474 |
| HONDURAS | HN | HND | 340 | MAURITANIA | MR | MRT | 478 |
| HONG KONG | HK | HKG | 344 | MAURITIUS | MU | MUS | 480 |
| HUNGARY | HU | HUN | 348 | MAYOTTE | YT | MYT | 175 |
| ICELAND | IS | ISL | 352 | MEXICO | MX | MEX | 484 |
| INDIA | IN | IND | 356 | MICRONESIA, FED. STATES | FM | FSM | 583 |
| INDONESIA | ID | IDN | 360 | MOLDOVA, REPUBLIC OF | MD | MDA | 498 |
| IRAN | IR | IRN | 364 | MONACO | MC | MCO | 492 |
| IRAQ | IQ | IRQ | 368 | MONGOLIA | MN | MNG | 496 |
| IRELAND | IE | IRL | 372 | MONTSERRAT | MS | MSR | 500 |
| ISRAEL | IL | ISR | 376 | MOROCCO | MA | MAR | 504 |
| ITALY | IT | ITA | 380 | MOZAMBIQUE | MZ | MOZ | 508 |
| JAMAICA | JM | JAM | 388 | MYANMAR | MM | MMR | 104 |
| JAPAN | JP | JPN | 392 | NAMIBIA | NA | NAM | 516 |
| JORDAN | JO | JOR | 400 | NAURU | NR | NRU | 520 |
| KAZAKHSTAN | KZ | KAZ | 398 | NEPAL | NP | NPL | 524 |
| KENYA | KE | KEN | 404 | NETHERLANDS | NL | NLD | 528 |
| KIRIBATI | KI | KIR | 296 | NETHERLANDS ANTILLES | AN | ANT | 530 |
| KOREA, (north) | KP | PRK | 408 | NEW CALEDONIA | NC | NCL | 540 |
| KOREA, REPUBLIC OF | KR | KOR | 410 | NEW ZEALAND | NZ | NZL | 554 |
| KUWAIT | KW | KWT | 414 | NICARAGUA | NI | NIC | 558 |
| KYRGYZSTAN | KG | KGZ | 417 | NIGER | NE | NER | 562 |
| LAO (communist) | LA | LAO | 418 | NIGERIA | NG | NGA | 566 |
| LATVIA | LV | LVA | 428 | NIUE | NU | NIU | 570 |
| LEBANON | LB | LBN | 422 | NORFOLK ISLAND | NF | NFK | 574 |

**Table D.1:** ISO country codes (2 of 3).

| Country | A 2 | A 3 | Number | Country | A 2 | A 3 | Number |
|---|---|---|---|---|---|---|---|
| NORTH MARIANA ISLS | MP | MNP | 580 | ST. PIERRE AND MIQUELON | PM | SPM | 666 |
| NORWAY | NO | NOR | 578 | SUDAN | SD | SDN | 736 |
| OMAN | OM | OMN | 512 | SURINAME | SR | SUR | 740 |
| PAKISTAN | PK | PAK | 586 | SVALBARD AND JAN MAYEN | SJ | SJM | 744 |
| PALAU | PW | PLW | 585 | SWAZILAND | SZ | SWZ | 748 |
| PALESTINIAN TERRITORY | PS | PSE | 275 | SWEDEN | SE | SWE | 752 |
| PANAMA | PA | PAN | 591 | SWITZERLAND | CH | CHE | 756 |
| PAPUA NEW GUINEA | PG | PNG | 598 | SYRIAN ARAB REPUBLIC | SY | SYR | 760 |
| PARAGUAY | PY | PRY | 600 | TAIWAN | TW | TWN | 158 |
| PERU | PE | PER | 604 | TAJIKISTAN | TJ | TJK | 762 |
| PHILIPPINES | PH | PHL | 608 | TANZANIA | TZ | TZA | 834 |
| PITCAIRN | PN | PCN | 612 | THAILAND | TH | THA | 764 |
| POLAND | PL | POL | 616 | TOGO | TG | TGO | 768 |
| PORTUGAL | PT | PRT | 620 | TOKELAU | TK | TKL | 772 |
| PUERTO RICO | PR | PRI | 630 | TONGA | TO | TON | 776 |
| QATAR | QA | QAT | 634 | TRINIDAD AND TOBAGO | TT | TTO | 780 |
| REUNION | RE | REU | 638 | TUNISIA | TN | TUN | 788 |
| ROMANIA | RO | ROM | 642 | TURKEY | TR | TUR | 792 |
| RUSSIAN FEDERATION | RU | RUS | 643 | TURKMENISTAN | TM | TKM | 795 |
| RWANDA | RW | RWA | 646 | TURKS AND CAICOS ISLANDS | TC | TCA | 796 |
| SAINT KITTS AND NEVIS | KN | KNA | 659 | TUVALU | TV | TUV | 798 |
| SAINT LUCIA | LC | LCA | 662 | UGANDA | UG | UGA | 800 |
| ST VINCENT & GRENADINES | VC | VCT | 670 | UKRAINE | UA | UKR | 804 |
| SAMOA | WS | WSM | 882 | UNITED ARAB EMIRATES | AE | ARE | 784 |
| SAN MARINO | SM | SMR | 674 | UNITED KINGDOM | GB | GBR | 826 |
| SAO TOME AND PRINCIPE | ST | STP | 678 | UNITED STATES | US | USA | 840 |
| SAUDI ARABIA | SA | SAU | 682 | US MINOR ISLANDS | UM | UMI | 581 |
| SENEGAL | SN | SEN | 686 | URUGUAY | UY | URY | 858 |
| SEYCHELLES | SC | SYC | 690 | UZBEKISTAN | UZ | UZB | 860 |
| SIERRA LEONE | SL | SLE | 694 | VANUATU | VU | VUT | 548 |
| SINGAPORE | SG | SGP | 702 | VENEZUELA | VE | VEN | 862 |
| SLOVAKIA (Slovak Republic) | SK | SVK | 703 | VIET NAM | VN | VNM | 704 |
| SLOVENIA | SI | SVN | 705 | VIRGIN ISLANDS (BRITISH) | VG | VGB | 092 |
| SOLOMON ISLANDS | SB | SLB | 090 | VIRGIN ISLANDS (U.S.) | VI | VIR | 850 |
| SOMALIA | SO | SOM | 706 | WALLIS AND FUTUNA ISLANDS | WF | WLF | 876 |
| SOUTH AFRICA | ZA | ZAF | 710 | WESTERN SAHARA | EH | ESH | 732 |
| S. GEORGIA & S. SANDWICH | GS | SGS | 239 | YEMEN | YE | YEM | 887 |
| SPAIN | ES | ESP | 724 | YUGOSLAVIA | YU | YUG | 891 |
| SRI LANKA | LK | LKA | 144 | ZAMBIA | ZM | ZMB | 894 |
| ST. HELENA | SH | SHN | 654 | ZIMBABWE | ZW | ZWE | 716 |

**Table D.1:** ISO country codes (3 of 3).

# References


Agerwala, T., and Arvind (1982) "Data Flow Systems," *Computer*, **15**(2)10–12, Feb.

Augarten, Stan (1984) *Bit By Bit: An Illustrated History of Computers*, New York, Ticknor and Fields.

boardwatch 2002 is part of `http://www.ispworld.com/`.

BPCS (2001) is `http://www.know.comp.kyutech.ac.jp/BPCSe/`.

Burks, A. W., H. H. Goldstine, and J. von Neumann (1976) "Preliminary Discussion of the Logical Design of an Electronic Computing Instrument," reprinted in E. Swartzlander (ed.) *Computer Design Development*, Rochelle Park, NJ, Hayden Publ.

Burks, Alice and Arthur (1988) *The First Electronic Computer: The Atanasoff Story*, Ann Arbor, MI, University of Michigan Press.

Comer, D. (1989) *Internetworking with TCP/IP: Principles, Protocols, and Architectures*, Volumes I and II, Englewood Cliffs, NJ, Prentice-Hall.

Data General Corp. (1969) *NOVA Computer Assembler Manual*, #093-000017, Southboro, Mass.

Data General Corp. (1970) *How to use the Nova Computers*, Southboro, Mass.

Digital Equipment Corp. (1968) *Introduction to Programming, PDP-8*, Maynard, Mass.

Digital Equipment Corp. (1981) *VAX Architecture Handbook*, Maynard, MA.

ENIAC 2000 is URL "http://www.library.upenn.edu/special/gallery/mauchly/jwmintro.html".

Flannery, Sarah (2000) *In Code: A Mathematical Journey*, London, Profile.

Gardner, Martin (1972) "Mathematical Games," *Scientific American*, **227**(2):106, August.

Goldstine, H. H. (1972) *The Computer from von Neumann to Pascal*, Princeton, NJ, Princeton University Press.

Hamming, Richard (1980) *Coding and Information theory*, Englewood Cliffs, NJ., Prentice-Hall.

Haynal 2000, R. Haynal, "Internet Backbones," `http://navigators.com/isp.html`

Heath, F. G. (1972) "Origins of the Binary Code," *Scientific American*, **227**(2):76, August.

Hennesy, J. (1982) "The MIPS Machine," *Proceedings COMPCON 1982*, pp. 2–7, San Francisco, CA.

Hillis, W. D. (1985) *The Connection Machine*, Cambridge, Mass., MIT Press.

Hewlett-Packard Corp (1972) *A Pocket Guide to the HP2100 Computer*, Cupertino, CA., Publ. 5951-4423, pp.2–12.

Huffman, D., (1952) “A Method for the Construction of Minimum Redundancy Codes,” *Proceedings of the IRE*, **40**,1098–1101, Sept.

Hyman, Anthony (1982) *Charles Babbage: Pioneer of the Computer*, Oxford, Oxford University Press.

IBM Corp (1963) *The IBM System/360 Principles of Operation*, form GA22-6821,

IBM Corp (1963a) *IBM 7040 and 7044 Data Processing Systems, Student Text*, IBM Form No. C22-6732, 1963

Intel Corp (1975) *Intel 8080 Microcomputer Systems User’s Manual*, ref. 98-153c, Santa Clara, CA.

Intel Corp (1985) *Introduction to the 80386*, Order No. 231252-001, Santa Clara, CA.

Intel Corp (1987) *Microprocessor and Peripheral Handbook,* Vol. I, Order #230843-004, Santa Clara, CA.

IowaState (2000) is URL `http://www.cs.iastate.edu/jva/jva-archive.shtml`.

ispworld 2002 is URL `http://www.ispworld.com/`.

Juffa (2000) is URL `ftp://ftp.math.utah.edu/pub/tex/bib/fparith.bib`.

Kane, G. (1981) *68000 Microprocessor Handbook*, Berkeley, CA, Osborne/McGraw-Hill,

Krol, E. (1994) *The Whole Internet*, Sebastopol, CA, O’Reilly and Assoc.

Leventhal, L. A. (1979) *6502 Assembly Language Programming*, Berkeley, CA, Osborne/McGraw-Hill.

Lin, Shu, (1970) *An Introduction to Error Correcting Codes*, Englewood Cliffs, NJ, Prentice-Hall.

Linde, Y., A. Buzo, and R. M. Gray (1980) “An Algorithm For Vector Quantization Design,” *IEEE Transactions on Communications*, COM-28:84–95, January.

List 1999 “The List: The Definitive ISP Buyer’s Guide,” `http://thelist.internet.com/`

Mano, Morris M. (1991) *Digital Design*, Englewood Cliffs, NJ, Prentice-Hall.

Mano, Morris M. (1997) *Logic and Computer Design Fundamentals*, Upper Saddle River, NJ, Prentice Hall.

Marks, Leo (1999) *Between Silk and Cyanide: A Codemaker’s War 1941–1945*, Free Press.

Matula, D., and P. Kornerup (1978) *A Feasibility Analysis of Binary Fixed-Slash and Floating-Slash Number Systems*, Tech. Rep. CS 7818, Southern Methodist University, Dallas, TX.

Metcalfe, R. M. and D. R. Boggs (1976) “Ethernet: Distributed Packet Switching for Local Computer Networks,” *Communications of the ACM*, **19**,395–404, July.

Mokhoff, N. (1986) “New RISC Machines Appear as Hybrids with both RISC and CISC Features.”

Mollenhoff, Clark (1988) *Atanasoff, Forgotten Father of the Computer*, Ames, Iowa State University Press.

Osborne, A. (1978) *An Introduction to Microcomputers*, Vol. III, Berkeley, CA, Osborne and Assoc.

Osborne, A. and G. Kane (1981) *4 and 8 Bit Microprocessor Handbook*, Berkeley, CA, Osborne/McGraw-Hill.

Patterson, D. A., and D. R. Ditzel (1980) “The Case for the RISC,” *Computer Architecture News*, **8**(6)25–33, Oct.

Patterson, D. (1985) “Reduced Instruction Set Computers,” *Communications of the ACM*, **28**(1)8–21, Jan.

Press, W. H., B. P. Flannery, et al. (1988) *Numerical Recipes in C: The Art of Scientific Computing*, Cambridge University Press. (Also available on-line from `http://www.nr.com/`.)

Radin, G. (1983) “The 801 Minicomputer,” *IBM Journal of Research and Development*, **27**(3)237–246, May.

Ralston, A., ed. (1985) *Encyclopedia of Computer Science and Engineering*, 2nd Ed., Van Nostrand.

Ramabadran, Tenkasi V., and Sunil S. Gaitonde (1988) “A Tutorial on CRC Computations,” *IEEE Micro* pp. 62–75, August.

Randell, B. ed. (1982) *The Origin of Digital Computers*, Berlin, Springer verlag.

RFC821 (1982) is `http://www.cis.ohio-state.edu/htbin/rfc/rfc821.html`.

Roja, Raul (2000) is URL `http://hjs.geol.uib.no/html/zuse/zusez1z3.htm`.

Rosin, R. F. (1969) "Contemporary Concepts of Microprogramming and Emulation," *Computer Surveys*, 1(4)197–212, Dec.

RSA (2001) is `http://www.rsasecurity.com/rsalabs/challenges/factoring/index.html`.

Salisbury, Alan B. (1976) *Microprogrammable Computer Architecture*, NY, American Elsevier.

Savard, John (2001) is `http://home.ecn.ab.ca/~jsavard/crypto/mi0604.htm`.

Scott McCartney (1999) *ENIAC: The Triumphs and Tragedies of the World's First Computer*, New York, Walker and Company.

Seitz, C. L. (1985) "The Cosmic Cube," *Communications of the ACM* **28**(1)22–33.

Siegel, H. J., and McMillan, R. J. (1981) "The Multistage Cube," *IEEE Computer*, **14**(12),65–76, Dec.

Siewiorek, D., et al (1982) *Computer Structures*, McGraw-Hill.

Singh, Simon (1999) *The Code Book*, New York, Doubleday.

Shallit 2000 is URL `http://www.math.uwaterloo.ca/~shallit/Courses/134/history.html`.

Smith, David Eugene (1923) *History of Mathematics*, reprinted by Dover Publications 1958, volume 2, p. 514.

Smith, D. A. (1929) *A Source Book in Mathematics*, New York, NY, McGraw-Hill.

Smotherman (1999) is URL `http://www.cs.clemson.edu/~mark/uprog.html`.

Stevenson, D. (1981) "A Proposed Standard for Binary Floating-Point Arithmetic," *IEEE Computer*, **14**(3)51-62.

Tabak, D. (1987) *RISC Architecture*, New York, John Wiley.

Watson, I., and J. Gurd (1982) "A Practical Data Flow Computer," *IEEE Computer*, **15**(2),51–57, Feb.

Zimmermann, Philip (1995) *PGP Source Code and Internals*, MIT Press.

Zimmermann, Philip (2001) is URL `http://www.philzimmermann.com/`.

Ziv J. and A. Lempel (1977) "A Universal Algorithm for Sequential Data Compression," *IEEE Trans on Information Theory*, IT-23(**3**), pp. 337–343.

No Roman ever died in contemplation over a geometrical diagram. (A reference to the death of Archimedes).
—Alfred North Whitehead

# Answers to Exercises

**1.1:** The main reason is that I/O devices have moving parts, while the processor is electronic and has no moving parts. Another reason is that I/O is often done serially, one bit at a time.

**1.2:** Normally, the PC is a special register, but in principle it can be one of the general-purpose registers. A good example is the VAX computer (Section 2.16) where register 15 is also the PC. The program can easily access and modify the PC on this computer, a feature that adds interesting addressing modes to the VAX, but may also lead to program crashes if the programmer accidentally uses the PC as a GPR.

**1.3:** An invalid opcode and a bad operand.

**1.4:** Procedure calls and interrupts. Interrupts (Section 1.7) are not instructions, but they affect the flow of control in the same way as jumps, by resetting the PC.

**1.5:** The PC would, in such a case, contain a random value, thus pointing to a random memory location. The control unit would start, in such a case, by fetching the content of the random memory location and executing it as an instruction. It would go on in this fashion until stopped by a bit pattern that is not the code of any instruction. Such a bit pattern causes the control unit to issue an interrupt.

**1.6:** The main reasons are:

1. The operating system is large. Most of the time, only parts of it are needed. Keeping the entire operating system in memory would occupy a large memory area while using just small parts of it.

2. The operating system is updated from time to time by the manufacturer. Keeping it in memory permanently would require replacing the memory itself (hardware) each time the operating system is updated. It is better to keep the operating system on a disk, where it can easily be replaced.

**1.7:** 1 PB=1024 TB = 1,048,576 GB = 1,073,741,824 MB $\approx 1.09951\cdot 10^{12}$ KB $\approx 1.1259\cdot 10^{15}$ bytes $\approx 9.0072\cdot 10^{15}$ bits.

**1.8:** 32M equals $2^5 \times 2^{20} = 2^{25}$. This is more than 32 million words, but there is no need to know or to memorize the precise decimal value of this number. What a professional should be able to figure out, however, is that addresses in a 32M memory are 25-bit numbers and *every* 25-bit number is a valid address.

**1.9:** The big endian order is 00000001|01011010|00110101|00001111 and the little endian order is the "natural" 00001111|00110101|01011010|00000001.

**1.10:** When the memory unit detects bad parity, it cannot tell whether the parity bit or one of the data bits are bad. However, by adding more parity bits, codes can be designed that can identify the location of an error in the word. Once the bad bit is identified, it is easy to correct it, by simply complementing it. Error-correcting codes are introduced in Section 3.9.

**1.11:** This rule has no exceptions.

**1.12:** No. Such instructions are necessary, since the programmer may want to disable certain interrupts during parts of the program.

**1.13:** No. The user can execute it, but when the user program is running, the mode is 'user' anyway.

**1.14:** Yes. The program sends the content of an entire page to the printer, then has to check status and wait until the printer has finished printing the page before it can send the next page. In practice, the user program invokes a *print manager* (part of the operating system), and it is the print manager that sends the output to the printer and checks status.

**1.15:** Register I/O is simple to use, and it requires a minimum amount of hardware. It make sense to use it in cases where only a few pieces of data need to be input or output; it also makes sense in cases where computer time is inexpensive and can be carelessly spent. This is why register I/O is commonly used on personal computers, but not on large, multiuser computers.

**2.1:** A `JMP` instruction in the direct mode is a good example. Such an instruction is long, because the entire address (25–35 bits) is included in the instruction. It is executed by resetting the PC, so it is fast.

**2.2:** It is obvious, from Figure 2.2, that there can be 16 and 64 instructions of formats 3 and 4, respectively.

**2.3:** "`LOD R5,#ARY`". This instruction uses the immediate mode.

**2.4:** In a computer using cascaded indirect, the assembler should have a special directive for this purpose.

**2.5:** Many stack operations use the element or two elements on top of the stack as their operands. Such an instruction removes the operands from the stack and pushes the result back into the stack. If the programmer wants to use the same operands in the future, they should stay in the stack, and an easy way to accomplish this is to generate their copies before executing a stack operation. Example:

```
   LOD #38    loads the element right below the top.
1. PUSH       it becomes the new top.
   LOD #51    loads the element right below the new top (the old top).
2. PUSH       it becomes the new top.
3. ADD        adds the two elements on the top and removes them.
```

The successive states of the stack are shown in Figure Ans.1.

| PUSH | PUSH | ADD | sum replaces operands |
|---|---|---|---|
| | 51 | 51 | |
| 38 | 38 | 38 | 89 |

**Figure Ans.1:** Using a stack

**2.6:** The identity operation, where each bit is transformed into itself.

**2.7:** The password is used to select cover files $C_1$, $C_2$, and $C_3$. They are XORed to obtain 1010, which is XORed with $F$ to yield 0011, which is XORed with one of the $C_j$'s, say $C_2$, to produce a new $C_2 = 1101$. Once this is done, file $F$ can be recovered (if $P$ is known) by

$$F = C_1 \oplus C_2 \oplus C_3 = 0011 \oplus 1101 \oplus 0111 = 1001.$$

**2.8:** That the most recent result was a zero.

**2.9:** There are $2^n$ $n$-bit numbers. Half of them $(= 2^{n-1})$ are negative, one is zero, which leaves $2^{n-1} - 1$ positive numbers.

**2.10:** No. The smallest mantissa is 0.5, so the smallest product of two normalized mantissas is $0.25 = 0.01_2$.

**2.11:** The exponents are equal, so the mantissas can be added directly. Adding the two fractions $0.10\ldots0_2 = 0.5$ and $0.11\ldots0_2 = 0.75$ produces 1.25, or in binary $1.010\ldots0$. The sum has to be normalized by shifting it to the right one position. To compensate, the exponent should be incremented by 1 but this causes overflow, since the exponent is already the largest one that can be expressed in three magnitude bits. The result is $0|1000|101\ldots0$ and the V flag should be set.

**2.12:** We start with an example of the real division $5/2 = 2.5$. We have to find a way to divide the integers 10 and 4 and end up with 5 (the integer representing 2.5). We already know that to obtain the right product we have to divide by the resolution, so it is natural to try to obtain the right quotient by multiplying by the resolution. We therefore try $(10 \cdot 2)/4 = 20/4 = 5$. The rule for dividing fixed-point numbers is: Multiply the dividend (10) by the resolution (2), then divide the result (which is greater than the dividend, since the resolution is normally greater than 1) by the divisor (4).

Next, we try the real division 2.5/2. Applying the rule above to this case results in $(5 \cdot 2)/4 = 10/4$ and yields the integer 2, which is the representation of the fixed-point number 1. Obviously, the real division 2.5/2 should produce 1.25, but this number cannot be represented in our fixed-point system whose resolution is 2. Therefore, the value 1.25 has to be truncated to 1.

**2.13:** The number of 4-bit codes where only 10 of the 16 combinations are used is

$$\frac{16!}{(16-10)!} = \frac{16!}{6!} \approx 29 \cdot 10^9.$$

There can be, of course, longer BCD codes, and the longer the code, the more codes are possible.

**2.14:** These codes are listed in Table Ans.2.

| | 5421 | 5311 | 7421 |
|---|---|---|---|
| 0 | 0000 | 0000 | 0000 |
| 1 | 0001 | 0001 | 0001 |
| 2 | 0010 | 0011 | 0010 |
| 3 | 0011 | 0100 | 0011 |
| 4 | 0100 | 0101 | 0100 |
| 5 | 1000 | 1000 | 0101 |
| 6 | 1001 | 1001 | 0110 |
| 7 | 1010 | 1011 | 1000 |
| 8 | 1011 | 1100 | 1001 |
| 9 | 1100 | 1110 | 1010 |

**Table Ans.2:** Three weighted BCD codes

**2.15:**

Table Ans.3 shows how individual bits change when moving through the binary codes of the first 32 integers. The 5-bit binary codes of these integers are listed in the odd-numbered columns of the table, with the bits of integer $i$ that differ from those of $i-1$ shown in boldface. It is easy to see that the least-significant bit (bit $b_0$) changes all the time, bit $b_1$ changes for every other number, and, in general, bit $b_k$ changes every $k$ integers. The even-numbered columns list one of the several possible reflected Gray codes for these integers. The table also lists a recursive Matlab function to compute RGC.

**2.16:** Such an addition is really a subtraction. The (absolute value of the) result is less than either of the two original operands. If each operand fits in a register, the result is certainly going to fit in a register, and there can be no overflow.

**3.1:** It is possible to have programs that do not input any data. Such a program always performs the same operations and always ends up with the same results. Examples are (1) a program that prints the ASCII code table, (2) a program that calculates $\pi$ to a certain, fixed precision, and (3) a program that generates and plays a song on the computer's speaker. Such programs are not very useful, though, and even they

| 43210 | Gray | 43210 | Gray | 43210 | Gray | 43210 | Gray |
|---|---|---|---|---|---|---|---|
| 00000 | 00000 | 0**1000** | 10010 | **10000** | 00011 | 1**1000** | 10001 |
| 0000**1** | 00100 | 0100**1** | 10110 | 1000**1** | 00111 | 1100**1** | 10101 |
| 000**10** | 01100 | 010**10** | 11110 | 100**10** | 01111 | 110**10** | 11101 |
| 0001**1** | 01000 | 0101**1** | 11010 | 1001**1** | 01011 | 1101**1** | 11001 |
| 00**100** | 11000 | 01**100** | 01010 | 10**100** | 11011 | 11**100** | 01001 |
| 0010**1** | 11100 | 0110**1** | 01110 | 1010**1** | 11111 | 1110**1** | 01101 |
| 001**10** | 10100 | 011**10** | 00110 | 101**10** | 10111 | 111**10** | 00101 |
| 0011**1** | 10000 | 0111**1** | 00010 | 1011**1** | 10011 | 1111**1** | 00001 |

**Table Ans.3:** First 32 Binary and Reflected Gray Codes.

```
function b=rgc(a,i)
[r,c]=size(a);
b=[zeros(r,1),a; ones(r,1),flipud(a)];
if i>1, b=rgc(b,i-1); end;
```

Matlab Code For Table Ans.3.

must generate output. A program that does not generate any output is useless, since there is no way to use its results. The output of a program, of course, does not have to be printed or displayed. It can be stored in memory or in registers, or it can be sent on a communications line to another device. An example is a computer that controls traffic lights at an intersection. The outputs control the lights, and no printer or monitor is used.

**3.2:** In principle, yes, but in practice an I/O device is normally assigned two or more select numbers, for its status and data.

**3.3:** It is unimportant to know the character codes in any base. When we want to use the character `A` in a computer, we can simply type `A`. In the few cases where we need to know the code of a character, perhaps a control character, it is preferable to know it in binary. It turns out that it is much easier to translate between binary and either octal or hex, than between binary and decimal.

**3.4:** Answer not provided.

**3.5:** Assuming that a file of $n$ bits is given and that $0.9n$ is an integer, the number of files of sizes up to $0.9n$ is

$$2^0 + 2^1 + \cdots + 2^{.9n} = 2^{1+.9n} - 1 \approx 2^{1+.9n}.$$

For $n = 100$ there are $2^{100}$ files and $2^{1+90} = 2^{91}$ can be compressed well. The ratio of these numbers is $2^{91}/2^{100} = 2^{-9} \approx 0.00195$. For $n = 1000$, the corresponding fraction is $2^{901}/2^{1000} = 2^{-99} \approx 1.578 \cdot 10^{-30}$. These are still extremely small fractions.

**3.6:** The receiver will interpret the transmission as "01 00 111 110 10 0", and will hiccup on the last zero.

**3.7:** A typical fax machine scans lines that are about 8.2 inches wide ($\approx$ 208mm). A blank scan line thus produces 1664 consecutive white pels.

**3.8:** There may be fax machines (now or in the future) built for wider paper, so the group 3 code was designed to accommodate them.

**3.9:** Each scan line starts with a white pel, so when the decoder inputs the next code it knows whether it is for a run of white or black pels. This is why the codes of Table 3.20 have to satisfy the prefix property in each column but not between the columns.

**3.10:** The code of a run length of one white pel is 000111, and that of one black pel is 010. Two consecutive pels of different colors are thus coded into 9 bits. Since the uncoded data requires just two bits (01 or 10), the compression ratio is $9/2 = 4.5$ (the compressed file is 4.5 times bigger than the uncompressed one).

**3.11:** The next step matches the space and encodes the string '`␣e`'.

| | | | |
|---|---|---|---|
| `sir␣sid` | `␣eastman␣easily␣` | $\Rightarrow$ | (4,1,'`e`') |
| `sir␣sid␣e` | `astman␣easily␣te` | $\Rightarrow$ | (0,0,'`a`') |

and the next one matches the '`a`'.

**3.12:** Figure Ans.4 shows two 32×32 matrices. The first one, $a$, with random (and therefore decorrelated) values and the second one, $b$, is its inverse (and therefore with correlated values). Their covariance matrices are also shown and it is obvious that matrix $\mathrm{cov}(a)$ is close to diagonal, whereas matrix $\mathrm{cov}(b)$ is far from diagonal. The Matlab code for this figure is also listed.

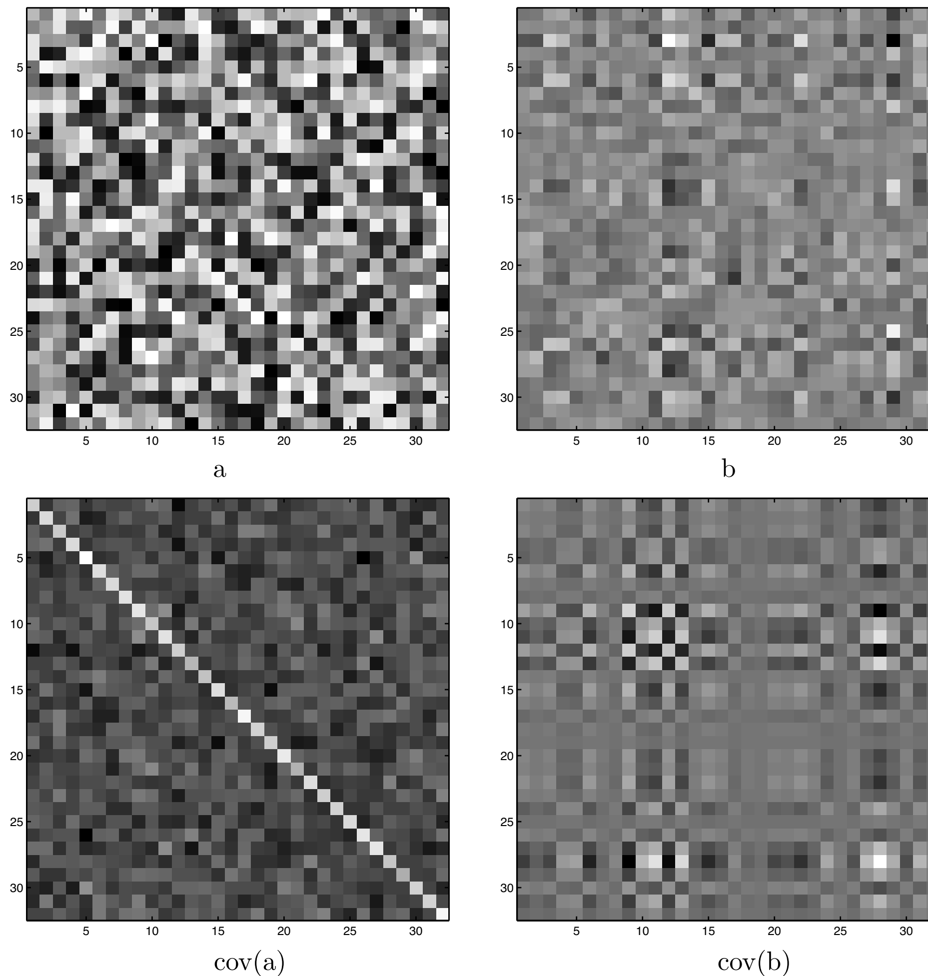



**Figure Ans.4:** Covariance Matrices of Correlated and Decorrelated Values

```
a=rand(32); b=inv(a);
figure(1), imagesc(a), colormap(gray); axis square
figure(2), imagesc(b), colormap(gray); axis square
figure(3), imagesc(cov(a)), colormap(gray); axis square
figure(4), imagesc(cov(b)), colormap(gray); axis square
```

Code for Figure Ans.4.

**3.13:** The Mathematica code of Figure Ans.5 yields the coordinates of the rotated points

$$(7.071, 0),\ (9.19, 0.7071),\ (17.9, 0.78),\ (33.9, 1.41),\ (43.13, -2.12),$$

(notice how all the $y$ coordinates are small numbers) and shows that the cross-correlation drops from 1729.72 before the rotation to $-23.0846$ after it. A significant reduction!

```
p={{5,5},{6, 7},{12.1,13.2},{23,25},{32,29}};
rot={{0.7071,-0.7071},{0.7071,0.7071}};
Sum[p[[i,1]]p[[i,2]], {i,5}]
q=p.rot
Sum[q[[i,1]]q[[i,2]], {i,5}]
```

**Figure Ans.5:** Code For Rotating Five Points.

**3.14:** In an $8\times 8$ template there should be $8\cdot 8/4=16$ holes. The template is written as four $4\times 4$ small templates, and each of the 16 holes can be selected in four ways. The total number of hole configurations is therefore $4^{16}=4{,}294{,}967{,}296$.

**3.15:** For 12 November 2001, the weighted sum is

$$50\cdot 1+51\cdot 2+52\cdot 1+53\cdot 1+54\cdot 0+55\cdot 1=312$$

and $312 \bmod 190=122$.

**3.16:** Using just one frequency allows the telecommunications hardware to select the highest frequency for which the line impedance is still low. This allows for the highest signaling rate. The use of multiple phases and amplitudes increases the bitrate.

**3.17:** Since each card is assigned a unique three-byte number, there can be $2^{24}\approx 16.78$ million such numbers.

**3.18:** Certain entities own many computers and use up many IPs. A university campus may have computer labs with thousands of computers, each having its own IP number. There are also devices other than computers that need IP numbers. A television cable interface box in a private home, for example, requires an IP, so the cable service provider would be able to let the home owner watch certain pay-per-view programs. There are also cameras located in many places that broadcast their pictures on the Internet.

**4.1:** Obviously, one per machine instruction. Modern computers normally have between 100 and 300 instructions in their instruction set, so this is the number of microprograms in the control store.

**4.2:** In addition to the 16-bit ALU output, the ALU produces another result; it updates the values of the two status flags Z and N. Sometimes all that's needed is the values of the flags, and not the ALU result itself.

**4.3:** New programs are being written all the time. The microprograms for a new computer, in contrast, need be written only once. This is one difference between a program and a microprogram. Another difference is that a program may be very large, whereas a typical microprogram is only a few microinstructions long. These two differences make it possible to completely debug microprograms.

**4.4:** The word size is a multiple of 8 since the ASCII code is 8 bits long (page 80). However, the control store contains microinstructions and not ASCII codes, so there is no reason why its word size should be a multiple of 8. Notice that our main memory has 16-bit words.

**4.5:** Both the 'ADDR' field and the MPC register (Figure 4.6) are 8 bits wide, which suggests that the control store contains $2^8=256$ words. Each word accommodates one microinstruction, so it should be 31 bits wide. The control store size is thus $256\times 31$. A look at the complete microcode (Table 4.14) shows that the control store is mostly empty, since our instruction set is so small.

**4.6:** The ALU looks at the 'function' input lines all the time. They get updated at the end of subcycle 1, after the new microinstruction has been fully loaded into the MIR. The ALU starts a new operation at that point, but it operates on old data, since the A and B latches will only be stable with their new data at the end of subcycle 2. This is another reason why the ALU output should be disregarded until the end of subcycle 3.

**4.7:** It should take place in subcycle 4, when output is normally sent to one of the registers. The reader should thus add the clock line for subcycle 4 as another input to the AND gates of Figure 4.10.

**4.8:** It makes sense to use the notation '`r1=mbr:=r2;`' (notice the use of the equal sign '`=`' and the assignment operator '`:=`').

**4.9:** There is no difference. The individual phrases of a microinstruction can be written in any order. As long as they are written on one line, the microassembler will assemble them into one, 31-bit microinstruction with the individual fields in the right order.

**4.10:** This is simply a memory write operation where the content of R1 is written in m[x].

```
mar:=ir; mbr:=r1; wr;
wr;
```

**4.11:** Because there is no direct path from the MBR to the MAR. The MBR can be moved only to the ALU and, from there, to the C bus. The MAR can be loaded only from the B latch. This is another aspect of the microinstructions being low level. Even though we use a high-level notation for our microinstructions, we cannot write anything that we want. We have to limit ourselves to existing registers and existing data paths. Our hypothetical microassembler has to be able to identify invalid microinstructions.

**4.12:** Perhaps the simplest format for a "Load Index" instruction is obtained when we assume that the index register is always R0. Such an instruction should have the usual 4-bit opcode, followed by a 12-bit `indx` field. The microinstructions should add the index and R0, using the sum as the effective address. Assuming that R0 has been preloaded with the number 5, the instruction '`LODX R1,7`' would load R1 with the content of memory location 12. The microprogram is.

```
axl F'X;
axr FF'X;          [ax=0FFF hex]
ax:=band(ax,ir);   [ax becomes the 12 lsb of the ir]
ax:=ax+r0;         [ax becomes the effective address]
mar:=ax; rd;
rd;
r1:=mbr;
```

Notice how similar it is to the microprogram for relative load.

**4.13:** None. Obviously, the PC and IR should not be disturbed by the microprograms. Also, the machine instructions may be using the SP and any of the four general-purpose registers at any time, so the microinstructions should not disturb these registers either. A real computer has more than one auxiliary register, that's invisible to the machine instructions, and that the control unit can use at any time. Also, the ALU in a real computer has internal registers that it uses for temporary, intermediate results.

**4.14:** The ADD R1 instruction is easy to implement. It adds memory location m[x] to R1 using the direct mode, but it is easy to modify the microprogram to use other modes.

```
mar:=ir, rd;
rd;
r1:=r1+mbr;
```

The "ADD R1,Ri" instruction requires more microinstructions, since Ri can be one of four registers. The microprogram is a little similar to the one for "PUSH Ri".

```
      axl 00001000'B;
      axr 0;          [ax becomes the mask 0000100...0]
      ax:=band(ax,ir); if Z goto r0r1; [isolate bit ir(11)]
r2r3: axl 00000100'B;  [ir(11)=1, check ir(10)]
      axr 0;
      ax:=band(ax,ir); if Z goto r2;
r3:   r1:=r1+r3; goto fech;
r2:   r1:=r1+r2; goto fech;
r0r1: ax 00000100'B;  [ir(11)=0, check ir(10)]
      axr 0;
      ax:=band(ax,ir); if Z goto r0;
r1:   r1:=r1+r1; goto fech;
r0:   r1:=r1+r0; goto fech;
```

**4.15:** When the computer starts, and also each time it is reset, a special hardware circuit resets many components in the computer, among them the PC and the MPC. The microcode stops either when the power to the computer is turned off, or when the HLT machine instruction is executed. The microprogram for this instruction is simply one microinstruction that jumps to itself, thus, '`l: jump l;`'. The computer is still running at full speed but it is effectively idle since no further machine instructions will be fetched. It can be restarted by hitting the 'reset' button.

**4.16:** In principle, it is possible to have procedures in the microcode, but in practice it is better to write the same sequence of microinstructions several times than to have the overhead associated with procedures. Procedures illustrate another difference between code and microcode. Procedures are used in programs because a program should be readable and easy to write. A microprogram, however, should satisfy one important criterion, namely it should be fast. Calling a procedure and returning from it are operations that require time, time which constitutes overhead associated with the procedure. This overhead can be tolerated in a program, but in a microprogram, where every microsecond counts, it may be degrading performance too much. This is why procedures are rarely used in microcode.

Procedures can be added to our microinstructions by adding two 1-bit fields 'CALL' and 'RET'. A microinstruction where 'CALL' is set to 1 saves the MPC in a special, auxiliary register PROC, then resets the MPC to the 'ADDR' field. A microinstruction where 'RET' is set to 1 resets the MPC from register PROC. These operations should take place in subcycle 4. They also require extra hardware (gates and buses) to move eight bits between the MPC and PROC.

**4.17:** Yes. It is possible to also move one of the eight registers into the MAR. The microinstruction '`r1=mbr:=r2; mar:=pc;`' does just that. It selects R2, moves it to the A bus, the A latch, through the ALU and, from there, to the C bus (where it ends up in R1) and to the MBR. It also selects the PC, moves it to the B bus, the B latch, and to the MAR.

**4.18:** The first approach that comes to mind is to isolate the 12-bit `x` field, save it in the AX register, and write a microprogram with a loop. In each iteration, this microprogram should shift R1 one position to the left, decrement AX by 1, and exit if the new AX is negative (by means of '`if N goto...`'). If AX is not negative, the microprogram should loop again. The only trouble is that decrementing is done by adding a $-1$, and the only way to get a $-1$ in our microprograms is to prepare it in the AX register. Since that register is being used, our approach cannot be implemented. This is an example of a relatively simple microprogram that requires *two* auxiliary registers (see also exercise 4.13).

**4.19:** Neither. They *translate* instructions from one language to another. The same is true for a compiler.

**5.1:** When generating the four-dimensional cube we have used a certain convention. All the nodes located in the left three-dimensional subcube were numbered $0xxx$, and those in the right subcube were numbered $1xxx$. To obtain different numbers, this convention can be reversed. Note that the particular number of a node does not matter as long as we realize that each bit position corresponds to one of the dimensions of the cube.

**5.2:** It has $2^{12} = 4\text{K} = 4096$ nodes, each connected to 12 other nodes. The node number has 12 bits, each corresponding to one of the 12 dimensions of the cube.

**5.3:** By changing the topology to a ring. The programs would also have to be generalized

**5.4:** To calculate $\int_a^b f(x)\,dx$, the interval $[a, b]$ is divided into many small subintervals. Each node calculates the area under the function in a number of intervals determined by the host, sums the areas, adds the total to those sent by its children, and sends the result to its parent.

**5.5:** The best way is to use a zero-dimensional subcube (just one node, typically #0). To take advantage of the parallel architecture, however, one must find a problem that can be divided into identical subproblems.

**5.6:** Start with a comparand of $00\ldots00$ and a mask of $10\ldots00$. A search will tell whether any words in memory have a zero on the left. If yes, then the smallest number has a zero in that position. Otherwise, it has a one. Generate that value in the leftmost position of the comparand, shift the mask to the right, and loop $N$ times.

**5.7:** By now it should be obvious that a good choice for the matching store is an *associative memory*. The matching store is an ideal application for an associative memory since a quick search is important.

**5.8:** A FIFO queue. In such a queue, the items on top are the oldest, so searching top to bottom always yields a match with the oldest candidate.

**5.9:** A recursive procedure. Each time a recursive procedure calls itself, the same instructions are executed. To match properly, however, the tokens generated by each call should be different.

**6.1:** A recursive procedure calling itself repeatedly because of a bug.

**6.2:** Because it is transparent to the user.

**7.1:** An OR gate outputs a 1 when any or all of its inputs are 1's. In fact, the only case where it outputs a 0 is when all its inputs are 0's. An XOR gate is similar but it excludes the case where more than one input is 1. Such a gate outputs a 1 when one of its inputs, and only one, is a 1.

**7.2:** Figure Ans.6 shows how NOR gates can be combined to produce the outputs of NOT, OR, and AND.

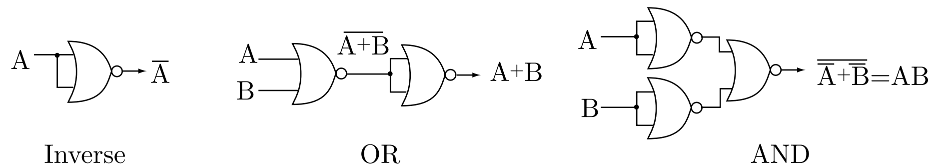



**Figure Ans.6:** The universality of NOR

**7.3:** Figure Ans.7 shows how an XOR gate is constructed from a NAND, OR, and AND gates. Simply replace the OR and AND gates with NAND gates, or replace all three with NOR gates.

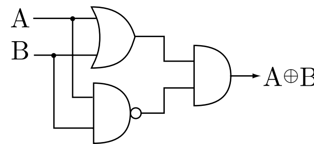



**Figure Ans.7:** The XOR logic gate

**7.4:** The last output `Q`$_3$, goes high after four input pulses.

**7.5:** Figure Ans.8 shows this design. The idea is to keep both inputs `J` and `K` permanently high.

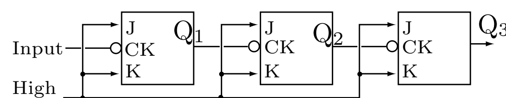



**Figure Ans.8:** A Ripple Counter with `JK` flipflops

**7.6:** Generalizing the approach shown in Figure 7.14, we (1) select the smallest $n$ such that $2^{n-1} < N \le 2^n$, (2) construct a modulo-$n$ binary counter, (3) select all the $Q_i$ outputs that go high at the $N$th step and connect them to an AND gate, and (4) use the output of that gate to reset the counter. Another approach is to connect a decoder to the outputs of the counter, and use the $N$th decoder output line to reset the counter.

**7.7:** This is done by triggering the T flipflop on a rising clock edge (Figure 7.15b).

**7.8:** Yes. All that is needed is a set of AND gates connecting the outputs of $B$ to the inputs of $A$. These gates can be fabricated regardless of whether $A$ can be transferred to $B$ or not.

**7.9:** The number $c$ of control lines should satisfy $k = 2^c$, so the number of control lines needed is $c = \log_2 k$.

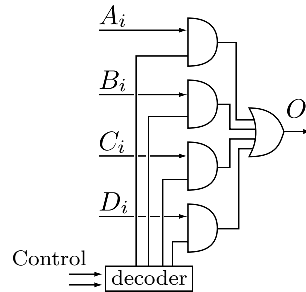



**Figure Ans.9:** A 4-to-1 multiplexor

**7.10:** This is shown in Figure Ans.9. Notice how the two control lines are decoded, so only one of the four AND gates is selected.

**7.11:** Table Ans.10 is the truth table of this device. It is clear, from this table, that the three outputs are defined by

$$Y_2 = I_4 + I_5 + I_6 + I_7,$$
$$Y_1 = I_2 + I_3 + I_6 + I_7,$$
$$Y_0 = I_1 + I_3 + I_5 + I_7.$$

Each of the three outputs is the output of an OR gate. Each of the three gates has four inputs taken from among the eight input lines $I_i$.

| | $I_0$ | $I_1$ | $I_2$ | $I_3$ | $I_4$ | $I_5$ | $I_6$ | $I_7$ |
|---|---|---|---|---|---|---|---|---|
| $Y_2$ | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| $Y_1$ | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| $Y_0$ | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |

**Table Ans.10:** A truth table for an encoder

**7.12:** Yes, it can have fewer than $2^n$ inputs. An encoder with three outputs generally has eight inputs. If it happens to have, say, five inputs, then only five of the eight possible outputs can be obtained.

**8.1:** Destructive memory must have hardware that reconstructs a word after it's been read.

**8.2:** The address decoder should be enabled each time memory is accessed. This can be achieved by connecting both the $R$ and $W$ lines to the decoder's "enable" input through an OR gate. The $R$ line should also be connected to all the BCs (in the same way the R/W line is connected in Figure 8.2a), so they receive a 1 for a memory read and 0 otherwise.

**8.3:** The 20 address lines needed by a 1M memory can be divided into four sets of five lines each. Each set is decoded by a 1-of-5 decoder, so the total number of decoder outputs is $4\times32 = 128$. Each BC receives four select lines, so it should have AND gates with four inputs.

**8.4:** A good example of such an application is the control store of a new computer. When a new, microprogrammable computer is developed, the microprograms should be written, debugged, and stored permanently in the control store. The best candidate for the control store is ROM. However, while the microprograms are being debugged, that ROM has to be modified several times, suggesting the use of EPROM.

**8.5:** Yes. This would require a $4\text{K}\times8$ ROM where only 96 of the 4096 locations are actually used.

**8.6:** Address 1010 has 11. Addresses 0001 and 0101 contain 01, and addresses 0000, 0001, 0010, 0011 contain 10.

**9.1:** The design is shown in Figure Ans.11.

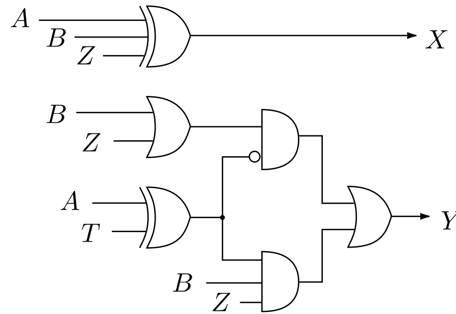



**Figure Ans.11:** An adder/subtractor

**9.2:** This observation is based on the fact that $1+x+\bar{x}$ equals either $1+1+0$ or $1+0+1$. In either case the sum bit is 0 and there is a carry bit. Once this is understood, it is easy to see that the sum

$$
\begin{array}{r} xx\ldots x10\ldots 0 \\ + \\ \bar{x}\bar{x}\ldots\bar{x}10\ldots 0 \end{array}
$$

equals all zeros (plus a carry bit that should be ignored).

**9.3:** If such a bit is 1, it attempts to set the latch, but it is already set. If the bit is 0, it will not reset the latch, since this can be done only through the R input.

**9.4:** It is easy to see, from the expressions for $C_1$, $C_2$, and $C_3$, that $C_4$ requires gates with a fan-in of 5.

**9.5:** The method proposed in the exercise reduces the fan-in of the AND gate but not that of the final OR gate. At the same time, the method delays the calculation of $C_3$ until that of $C_2$ is complete.

**9.6:** The problem is that the clock pulse $\phi$ is sent to the counter directly. In order to stop it, the "done" signal and the clock pulse should be connected to an OR gate and the ouptut of that gate should clock the counter. This way, once "done" goes high, the gate output stays high regardless of the clock.

**9.7:** The results are summarized in Table Ans.12. In both cases the product is $11110111000_2 = -72$.

| (a) |
|---:|
| 11010 |
| 01100 |
| 00000 |
| 00000 |
| 1111010 |
| 111010 |
| 00000 |
| 1110111000 |

| (b) | |
|---:|---|
| 01100 | |
| 11010 | |
| 00000 | |
| 01100 | |
| 00000 | |
| 01100 | |
| 01100 | |
| 0100111000 | |
| 0110000000 | subtract ($2^5\times 01100_2$) |
| 1110111000 | |

**Table Ans.12:** Multiplying signed integers

**9.8:** An integer with alternating bits, such as $0101\ldots01$. Table 9.12 implies that the compact representation of this multiplier is $1\bar{1}1\bar{1}\ldots1\bar{1}$. This multiplier does not reduce the number of partial products, and there is also the overhead associated with two's complementing the multiplicand.

**9.9:** Yes, but the overhead and the complex algorithm more than cancel out any gains from the shorter loop.

**10.1:** Suppressing the listing file makes sense if a listing exists from a previous assembly. Also, if a printer is not immediately available, the user may want to save the listing file and to print it later.

**10.2:** In many higher-level languages, a semicolon is used to indicate the end of a statement, so it seems a good choice to indicate the end of an instruction. Also, we will see that many other characters already have special meaning to the assembler.

**10.3:** The `F` indicates a floating-point operation, so this is a floating-point add.

**10.4:** The number of registers should be of the form $2^k$. This implies that a register number is $k$ bits long, and fits in a $k$-bit field in the instruction. A choice of 20 registers would mean a 5-bit register number—and thus a 5-bit field in many instructions—but that field would not be fully utilized since with five bits it is possible to select one of 32 registers.

**10.5:** To indicate an index register and a base register (these were special registers used for addressing; they are not discussed in this book).

**10.6:** An addressing mode should be used to assemble the instruction. Addressing modes are discussed in Section 2.3.

**10.7:** The symbol is simply undefined, an error situation that's discussed, among other errors, at the end of this chapter.

**10.8:** Certain characters, such as '–' and '_' allow for a natural division of the name into easy-to-read components. Other characters, such as '$' and '=' make labels more descriptive. Examples: `NO_OF_FONTS` is more readable than `NoOfFonts`. `REG=DATA` is more descriptive than `RegEqData`.

**10.9:**

```
a. type
   address=0..4095; an address in a 4K memory
   node=record
        info: address;
        next: ^node
        end;
   list=^node; the type 'list' is a pointer to the beginning of such a list
b. const
   lim=500; max =1000; some suitable constants
   type
   list=o..lim; type 'list' is the index of the first list element in array house
var
   house: array[1..lim] of 0..max; lists of pointers are housed here
house2: array[1..lim] of 0..lim; pointers pointing to
 the next element inside each list, are housed here.
```

**10.10:** Using logical operations and, perhaps, shifts to mask the rest of the instruction.

**10.11:** Yes, it is valid, its value is $-11$ and its type is absolute. It simply represents a negative distance.

**10.12:** External symbols, and the way they are handled by the loader, are discussed in texts on systems programming. In the above example, the assembler calculates as much of the expression as it can (`A-B`) and generates two `modify` loader directives, one to add the value of `K` and the other, to subtract `L`, both executed at load time.

**10.13:** Address 24, because of the earlier phrasing ...with the name '1' and a value $\geq 17\ldots$

**10.14:** '`JMP *`' means jump to the current instruction. Such an instruction jumps to itself and thus causes an infinite loop. In the early microprocessors, this was a common way to end the program, since many of those microprocessors did not have a `HLT` instruction.

'`JMP *-*`' means jump to location 0. However, since the LC symbol '*' is relative, the expression `*-*` is $rel - rel$ and is therefore absolute. The instruction would jump to location 0 absolute, not to the first location in the program.

**10.15:** Because the `USE` is supposed to have the name, not the value, of a LC, in its operand field.

**10.16:** Yes. The only problem is that the loader needs to be told where to start the execution of the entire program. This, however, is specified in the `END` directive and is a standard feature, used even where no multiple LCs are supported.

**10.17:** It is identical to:

```
     JMP TMP
       .
       .
       .
TMP: DC *
```

The computer will branch to the location labeled `TMP` but that location contains an address, not an instruction. The likely result will be an interrupt (invalid opcode).

**10.18:** It depends on the characters allowed. The ASCII codes of the characters ‘<’, ‘=’, ‘>’, ‘?’, ‘@’ immediately precede the code of ‘A’. Similarly, the codes of ‘[’, ‘\’, ‘]’ immediately follow ‘Z’ in the ASCII sequence. If those codes are used, then it is still easy to use buckets. Given the first character of a symbol name, we only need to subtract from it the ASCII code of the first of the allowed characters, say ‘<’, to get the bucket number. If other characters are allowed, then buckets may not be a good data structure for the symbol table.

**11.1:** Either make the label a parameter, use local labels (see Chapter 10), or use automatic label generation.

**11.2:** Generally not. In the above example, it makes no sense for the assembler to substitute `X` for `B`, since it does not know if this is what the programmer wants. However, there are two cases where double substitution (or even multiple substitution) makes sense. The first is where an argument is the name of a macro. The second is the case of an assembler where parameters are identified syntactically, with an ‘&’ or some other character. The first case is called nested macro expansion and is discussed in Section 11.6.1. The second case occurs in the IBM360 where parameters must start with an ‘&’ and are therefore easy to identify. The 360 assembler performs multiple substitution until no ‘&’ are left in the source line.

**11.3:** It depends on the MDT organization and on the size of the data structures used for binding. Typically, the maximum number of parameters ranges between a few tens and a few hundreds.

**11.4:** If the last argument is null, it should be preceded by a comma. A missing last comma indicates a missing argument.

**11.5:** Add another pass (pass $-1$?) to collect all macro definitions and store them in the MDT. Pass 0 would now be concerned only with macro expansions.

**11.6:** Nested macro definition. In such a case, each expansion of certain macros causes another macro definition to be stored in the MDT, and space in the MDT may be exhausted very rapidly.

**11.7:** All three parameters are bound to the same argument and are therefore substituted by the same string.

**11.8:** To retain the last binding and use default values. Thus, `P2` is first bound to `MAN` and then to `DAN`. Since `P3` is not bound to any argument, it should get a default value.

**11.9:** The macro processor would continue scanning the source, reading and assigning more and more text to the second argument, until one of the following happens:

1. It finds a period-space combination somewhere in the source. This would terminate the scan and would bound the second parameter to a (long) argument.
2. It gets to the end of the source and realizes that something went wrong.
3. It finds a character (rather, a token) in a context that's invalid inside a macro argument. In the case of TeX, such a token could be the start of a new paragraph.

In cases 2, 3 the macro processor would issue a ‘run away argument’ error message, and either terminate the macro expansion or give the user a chance to correct the source file interactively.

**11.10:** The difference is in scope. Attributes of macro arguments exist only while the macro is expanded. Attributes of a symbol exist while the symbol is stored in the symbol table.

**11.11:** The programmer probably meant a default value of null for parameter `M2`. However, depending on the syntax rules of the assembler, this may also be considered an error.

**11.12:** It should be, since it alerts the reader to the fact that some of the listing is suppressed.

**11.13:** Because it allows the implementation of recursive macros.

**11.14:** Yes, many assemblers support simple expressions in the `AIF` directive.

**11.15:** A constant, a `SET` symbol, or a macro argument.

**11.16:** No, one '`ENDM X`' is enough. It signals to the assembler the end of macro `X` and all its inner definitions.

**11.17:** The string `'A'[AY](AYX)`.

**C.1:** The capacities are shown in the table

| | Mode 1<br>345000×2048 | Mode 2<br>345000×2336 |
|---|---|---|
| Bytes | 706,560,000 | 805,920,000 |
| Mbytes | 673.828 | 768.585 |

Notice that mega is defined as $2^{20} = 1,048,576$, whereas a million equals $10^6$. The playing time (at 75 sectors per second) is 76 min, 40 s in either mode.

**C.2:** A full CD contains 700 Mbytes which is the equivalent of 80 minutes. At 24X speed it therefore takes $80/24 = 3.33$ minutes and at 40X it takes $80/40 = 2$ minutes to record. In addition, the drive needs a few second to finish up the CD and (optionally) 2–3 minutes to verify the recording by comparing the new CD with the original data.

Kneading is difficult to describe in words, but
many cookbooks have pictures. The idea is to
exercise the dough and work some more flour into it.

—Harry S. Delugach

# Index


4004 microprocessor, 272
6800 microprocessor, 273
8008 microprocessor, 273
8080 microprocessor, 48, 273
  instruction set, 48–49
8086 microprocessor, 273
9/19/89 syndrome, 70
68000 microprocessor, 37, 273
68020 microprocessor, 273
6502 microprocessor, 39, 40

AC coefficients (of an image transform), 106
accumulator (design of), 252
accumulator addressing mode, 41
ACE computer, 269
adaptive Huffman coding, 96–97
addition
  of floating-point numbers, 60–61
  of integers, 247–250
address
  in memory, 5
  size of, 7
  space, 73
addressing modes, 3, 14, 34, 36–41
  accumulator, 41
  autodecrement, 41
  autoincrement, 41
  byte displacement, 51
  byte relative deferred, 51
  cascaded indirect, 40
  current page direct, 40
  direct, 37, 180
  immediate, 37–38
  implicit, 40
  implied, 40
  index, 37–38
  indirect, 16, 37, 39–40
  longword relative, 51
  post-indexed indirect, 39
  pre-indexed indirect, 39
  register, 41
  register direct, 51
  register indirect, 41
  relative, 37–38, 180
  stack, 41
  stack relative, 41
  VAX, 51
  word displacement, 51
  word relative, 51
  zero page, 40
Adleman, Leonard M. (1945–), 125
Aiken, Howard Hathaway (1900–1973), 268–269
Alberti, Leon Battista (1404–1472), 110
Altair computer, 273
ALU, 1, 30–31, 247–265
Amis, Kingsley (1922–1995), 116
amplitude modulation, 133
amplitude shift keying, 133
AMSCO cipher, 120
analytical engine, 268
AOI gate (digital device), 228
applications of associative computers, 208
arithmetic errors, 65
arithmetic operations, 44, 247–261
arithmetic shift, 45
array processors, 190, 193–199
  applications, 193
  as special purpose computers, 195
  branching, 195
  instruction types, 194
ASCII codes, 79, 80, 245, 304
  control characters, 79
associative computers, 10, 190, 206–209
  applications, 207
  comparand, 207
  design of, 208
  mask, 207
  response store, 208
associative memory, 206
  search, 206
astable (digital device), 228, 232
asynchronous serial I/O, 137, 139
Atanasoff, John Vincent (1904–1995), 269–270
Atanasoff-Berry computer, 269–270

ATM (Asynchronous Transfer Mode), 148
Auden, Wystan Hugh (1907–1973), 188
Augarten, Stan, 228
authenticating an encrypted message, 126
autodecrement mode, 41
autoincrement mode, 41
average size of opcode, 35

Babbage, Charles (1791–1871), 112, 268
baud (definition of), 135
baud (origin of term), 83, 132
Baudot code, 67, 82, 84
Baudot, Jean Maurice Emile, 67, 82
BCD (binary coded decimal), 64–67, 81
BCD numbers, 7, 44, 64–65
Berners-Lee, Tim (html developer), 163
Berry, Clifford, 269
bi-level image, 102
bias (of a floating point number), 60
big endian, 7, 299
binary numbers (reasons for use), 228
bistable (digital device), 228–231
bit (origin of name), 131
bitrate (in wave modulation), 135
Boggs, David (ethernet inventor), 150
book cipher, 108
Boole, George (1815–1864), 54
boolean operations, 54
Booth multiplication algorithm, 256–259
bootstrap loader, 4, 8
bottleneck, von Neumann, 10, 190, 206
branch
  conditional, 46–47
  in array processors, 195
  unconditional, 46–47
break (artificial interrupt), 20, 24, 25
burst DRAM, 242
bus (in the computer), 9
bus (memory), 10
bus organization, 72–73
buses (CPU), 17
byte (definition of), 6, 7
byte displacement mode, 51
byte relative deferred mode, 51

cache RAM, 242
Caesar cipher, 108
Caesar, Julius, 108
Carroll, Lewis, ii, 85
carry, 68–70
  definition of, 68
  propagation, 267
  status flag, 46, 252
carry look-ahead (CLA), 250
cartoon-like image, 102
CD (compact disc), 283–290
  error correction in, 91–92, 285–286
CDC computers, 7
  Cyber, 62, 204
  STAR, 219
central processing unit, *see* processor
cgi (common gateway interface), 165
channel I/O, 78–79
characters (data type), 53
check bits, 86–87
cipher (definition of), 107
ciphers
  AMSCO, 120
  Myszkowsky, 120
  transposition, 114–120
circular queue, 101
circular shift, 45
CISC, *see* complex instruction set computers, 48, 219
CLA, *see* carry look-ahead
Cleland, John, ii
clock (in the computer), 169
coarse-grained architecture, 190, 221
Cocke, John, 222
codes
  adaptive Huffman, 96–97
  ASCII, 79, 80, 245, 304
  Baudot, 67, 82
  BCD, 65–67, 81
  book cipher, 108
  check bits, 86–87
  cipher, 107
  EBCDIC, 79–81
  error-correcting, 84–93
  error-detecting, 84–93
  excess-3, 65
  Hamming, 89–90
  Hamming distance, 85, 87–88
  homophonic substitution, 113–114
  Huffman, 95–97
  monoalphabetic substitution, 109
  nomenclator, 108
  one-time pad, 113, 121
  parity bits, 87
  periodic, 91
  polyalphabetic substitution, 110, 126
  public key, 125–126
  reflected BCD, 67
  reliable, 84–93
  SEC-DED, 90–91
  secure, 107–131
  self-complementing BCD, 67
  stream cipher, 45, 121–122
  unit distance BCD, 67
  variable size, 94–95
  Vernam, 121
  Vigenère, 110–113
  voting, 85–86
  weighted BCD, 65–67
collating sequence, 53
Colossus computer, 269

columnar transposition ciphers, 119–120
  double encryption, 119
combinational digital device, 228
compact disc (CD), 283–290
comparand register (associative computers), 207
comparisons, 46, 265
compiler (as part of the operating system), 4
compiler (optimizing in RISC), 220
complex instruction set computers, 219
computer (definition of), i
computer architecture
  classification, 189–190
  comparisons of, 219
  definition of, i
  Flynn classification, 189–190
  granularity, 189, 190
  scalar, 189
  superscalar, 189
  vector processor, 189
computer arithmetic, 65
computer bus, 9, 71
computer history, 267–273
  first generation, 270–272
  second generation, 272
  third generation, 272
computer networks, 148–165
  domain names, 156–160
  internet, 152–153
    addresses, 156–160
    IP, 155–156
    TCP, 156
  LAN, 132, 148–151
    ethernet, 149–150
    token ring, 150–151
  wide area, 151–152
computer organization
  definition of, i
  multilevel, 184
computing (definition of), i
concatenation (of characters), 53
conditional branch, 46
connection machine, 197–199
  clusters, 198
  flags, 198
  messages, 197, 198
  routers, 198
constellation diagram, 135
content addressable memory, 206
continuous-tone image, 102
control signals, 3, 13
control store, 17, 18, 167–170, 176, 177, 181, 184, 187, 304
  horizontal, 185
  vertical, 185
control unit, 1–5, 11–17
  and effective address, 14
  control signals, 13
  definition of, 1
  fetch-execute cycle, 2–5, 19, 20
  hardwired, 168
  in RISC, 220
  in SIMD, 193
  microprogrammed, 17, 167
  register transfer, 13
  sequencer, 13
  stopped, 5
core memory, 7, 272
correlations
  between quantities, 103
correlations between pixels, 104
cosmic cube, 202
counters (digital device), 232–236
  asynchronous, 233–234
  ring, 235
  ripple, 233–234
  synchronous, 234
country codes, 291–295
cover image (in steganography), 128
CPU, *see* processor
CPU buses, 17
CRAY 1, 62
CRAY computers, 204
Cray, Seymour, 222
CRC (cyclic redundancy code), 93, 122, 139, 140
cross correlation of points, 106
cryptography, 107–126
  authentication, 126
  Diffie-Hellman-Merkle key exchange, 123–124
  PGP, 126
  public key, 125–126
current page direct mode, 40
cyber (origin of word), 165
cyber-squatting, 157
cyclic redundancy character, *see* CRC

D latch, 230–231
Darwin, Charles, 246
data compression, 93–107
  and redundancy, 93–94
  dictionary methods, 100–102
  fax, 97–100
  group 3, 97
  Huffman method, 95–97
  images, 102–107
  lossy, 106, 107, 128
  LZ77, 100–102
  principle of image compression, 102
  run lengths, 98
  transforms, 103–107
  variable size codes, 94–95
data encryption standard, *see* DES
data flow computers, 10, 190, 209–217
  branching, 215
  definition, 210
  destination address, 211
  instruction formats, 214

Manchester, 210
matching store, 210
principle of, 210
tokens, 210, 211
data movement (general rule), 8
data movement instructions, 42
data types, 53–70
definition, 53
DC coefficient (of an image transform), 106
DEC computers
PDP-11, 41, 62, 196
PDP-8, 40, 272
VAX, 48, 62, 196, 219, 299
decimal (BCD) numbers, 64–65
decoders (digital device), 228, 237
decorrelated pixels, 103
delay line (in computer design), 231
delayed jumps on RISC I, 224
demodulator, 133
demultiplexors (digital device), 237
DES, 122–123
device select number, 47
DG computers (Nova), 41
diameter (of a topology), 201
difference engine, 268
differencing (image compression), 102–103
Diffie, Whitfield (1944–), 123, 125
Diffie-Hellman-Merkle key exchange, 123–125
digital devices
AOI gate, 228
astable, 228, 232
bistable, 228–231
combinational, 228
counters, 232–236
decoders, 228, 237
demultiplexors, 237
encoders, 238–239
latches, 228–231
logic gates, 227–228
monostable, 228, 231
multiplexors, 237
multivibrators, 228–232
priority encoders, 23, 238–239
registers, 236
sequential, 228
digram (letter pair), 110
frequencies of, 111
Dijkstra, Edsger, 31
direct mode, 37, 180
discrete cosine transform (DCT), 106
displacement (in addressing modes), 38, 180
distribution of letters, 109
Ditzel, David R., 222
division of integers, 260–261
DMA, 76–78
as a memory requestor, 9, 77
as memory requestor, 241
command, 76
device, 8, 76
Dodo (quotation from), ii
domain names, 156–160
advertising of, 158
double shift, 46
double-bus organization, 72
double-precision arithmetic, 69
and carry, 69
DRAM (dynamic RAM), 241
DSL (digital subscribers line), 144–145
`DUP` (in data flow computers), 214
DVD (digital versatile disc), 289–290
dynamic RAM (DRAM), 241

EA, *see* effective address
EBCDIC code, 79–81
Eckert, John Presper (1919–1995), 270, 272
`EDIT` instruction, 47, 219
EDSAC computer, 168
EDVAC computer, 271
effective address
and instruction execution, 14
definition of, 2, 36
Einstein, Albert (1879–1955), 110, 155
Ellis, James (British cryptographer), 125
emulation, 168, 184
encoders (digital device), 238–239
endian (byte order in a word), 7, 299
energy
concentration of, 106
of a distribution, 106
English text
frequencies of letters, 111
ENIAC computer, 270, 271
Enigma code machine, 113, 268–269
ensemble (SIMD computer), 193
EPROM, 243
error-correcting codes, 84–93, 299
in a CD, 91–92, 285–286
error-detecting codes, 84–93
ETAOINSHRDLU (letter probabilities), 111
ethernet, 149–150
history of, 150
excess-3 code, 65
exclusive OR (XOR), 44–45, 300
exrad in a floating point number, 60
extended data out DRAM, 242

fast page mode DRAM, 242
Fast, Arnold (first computer programmer), 268
fax compression, 97–100
group 3, 97
Feistel, Horst (Lucifer designer 1915–1990), 122
fetch-execute cycle, 2, 3, 19, 20
start and end, 4–5
field programmable logic array (FPLA), 246
fine-grained architecture, 190, 221

fingerprinting (digital data), 131
firmware, 184
first generation computers, 270–272
fixed point numbers, 63–64
  resolution of, 63
fixed-size opcodes, 35
fixed-slash rational, 68
flipflop, *see* bistable
floating point numbers, 57–63
  division by zero, 63
  normalization of, 59
  on RISC, 221
  reason for name, 63
  resolution of, 62
floating-point zero, 59
floating-slash rational, 68
flow of control, 3
Flowers, Thomas Harold (1905–1998) and Colossus, 269
Flynn classification, 193
forced procedure call, 21
format of a machine instruction, 33
Forrester, Jay Wright (1918–), 272
FPLA, *see* field programmable logic array
frequencies of symbols in LZ77, 101
frequency modulation, 133
frequency shift keying, 133
full adder, 30, 247
  on MPP, 197
full subtractor, 247
full-duplex, 141

Galois fields, 92
Gardner, Martin, 202
general purpose register, 236
general-purpose register, 2, 14, 38, 48, 50, 299
  accumulator, 252
generating polynomials, 92–93
Gibson, William, 165
Giga (= $2^{30}$), 7
golden ratio, 55
government communications headquarters (GCHQ), 108
GPR, *see* general-purpose register
granularity (in computer architecture), 189, 190
Gray codes, 67
Gray, Frank, 67
grayscale image, 102

half adder, 30
half-adder-subtractor, 247
half-duplex, 141
Hamming codes, 89–90
Hamming distance, 85, 87–88
Hamming, Richard, 87
hardware (definition of), i
hardwired control unit, 168
Hayes modem commands, 143
Hayes, Dennis C. (modem developer), 143
Hein, Piet, 202
Hellman, Martin E. (1945–), 123
Hennesy, John, 222
Heuring, Vincent, 226
Hilbert space-filling curve (and steganography), 128
Hillis, Daniel W., 197
history of computers, 267–273
`HLT` instruction, 25, 47, 306
Hoff, Marcian Edward (Ted, 1937–), 272
homophonic substitution codes, 113–114
horizontal microprogramming, 185, 188
host computer, 194, 201, 202
HP computers
  1000, 40
  2100, 40
html (definition of), 164
Huffman codes, 95–97
hunt mode (in synchronous serial I/O), 138
Husson, Samir, 168
hypercube
  architecture, 191–192, 200
  messages, 192
  node numbering, 191
  properties, 191

I/O
  channel, 78
    multiplexor, 79
    selector, 79
  channels, 78–79
  DMA, 76–78
  instructions, 47
  interrupt, 20, 26–28, 75
  memory-mapped, 47, 74–75
  polled, 74
  processor, 1, 71–79
  register, 74
  serial, 132–141
i860 microprocessor, 189
IANA, 163
iAPX 432 microprocessor, 273
IAS computer, 271
IBM computers
  360, i, 42, 219, 272
  370, 62, 219
  701, 272
  7000 series, 272
  801 RISC, 222
  RT, 222
ICANN, 163
IDEA (encryption algorithm), 126
IEEE (Institute of Electrical and Electronics Engineers), 62
IEEE floating-point standard, 62
image
  cartoon-like, 102
  continuous-tone, 102
  definition of, 102

grayscale, 102
resolution of, 102
transforms, 103–107
types of, 102
image compression, 102–107
by transforms, 103–107
differencing, 102–103
principle of, 102
immediate mode, 37–38
immediate operand, 34
impedance, 133
implicit mode, 40
implied mode, 40
index mode, 37–38
index register, 38
indirect mode, 16, 37, 39–40
infinity (in the IEEE f.p. standard), 63
information theory
and computer communications, 131
and increased redundancy, 85
instruction register (IR), 2, 13, 14, 19
instruction types on array processors, 194
instructions
execution, 3
formats, 33, 35, 37, 49, 52
formats (data flow), 214
size, 3
types, 41–48
integer addition, 247–250
integer division, 260–261
integer multiplication, 253–260
Booth algorithm, 256–259
three-bit shift, 259–260
integer subtraction, 247–250
integers (signed), 54–57
integrated circuit (invention of), 272
Intel computers
4004, 272
8008, 273
8080, 73, 273
8086, 273
8087, 62
i860, 189
iAPX 432, 273
iPSC/1, 202–204
iPSC/2, 204
iPSC/860, 204
Pentium, 273
international standard book number (ISBN), 120, 127
International Telecommunications Union (ITU), 97, 142
Internet, 148
backbones, 153–155
domain names, 156–160
history of, 150–152
ICANN and IANA, 163
ISPs, 153–155
layout, 153–155
organization, 152–153
protocol (IP), 155–156
protocol (TCP), 156
interrupt I/O, 75
interrupts, 1, 3, 20–30, 47
acknowledge (IACK), 22
break, 20, 24
daisy chain, 23–24
device priorities, 23
handling procedure, 21
I/O, 20, 26–28
in a microprocessor, 28–30
mask register, 24
nested, 1, 22
priority encoder, 24
request lines (IRQs), 20–21
simultaneous, 1, 22
software, 20, 24
supervisor call, 20
vectored, 1, 21–23
inverse of a rational number, 68
IP (internet protocol), 155–160
IR, *see* instruction register
ISBN, *see* international standard book number
ISDN, 145, 147
ISO country codes, 291–295
ITU, *see* International Telecommunications Union
ITU-T, 97, 142

Jacquard loom, 268
Jefferson, Thomas (and cryptography), 113
JK latch, 230–231
Johnson, Samuel, (1709–1784), 148
JPEG image compression standard, 107

K (=1024), 6
Kasiski, Friedrich Wilhelm (1805–1881), 112
Kawaguchi, Eiji (BPCS steganography), 129
Kerckhoffs' principle, 109, 123
key (in cryptography)
asymmetric, 125
distribution problem, 123, 124, 126
symmetric, 125
keyword in transposition ciphers, 119–120
Kilby, Jack St. Clair (1923–), 272

Larcom, Lucy, 265
latches, 228–231
D, 230–231
JK, 230–231
RS, 229
SR, 122, 229–231
T, 231
layers (of a network), 152
LC, 37
Leahy, Joe, 25
Leibniz, Gottfried Wilhelm von (1646–1716), 54, 267
letter distribution in a language, 109

letter frequencies
  transposition ciphers, 114
level (in computer organization), 184
little endian, 7, 299
LOAD instruction, 42
local area network (LAN), 132, 148–151
lockstep (in array processors), 190
lockstep operation, 193, 199
logic gates (digital device), 227–228
logic operations, 44
logical shift, 45
long real (IEEE floating point format), 63
longword, 51
longword relative mode, 51
lossy data compression, 106, 107, 128
Lucifer (predecessor of DES), 122
LZ77 dictionary compression, 100–102
LZH dictionary compression, 100

machine instructions, 33–53
  formats, 33
machine language, 33
Manchester data flow computer, 210
Manhattan project, 271
mantissa (of a floating-point number), 58, 59
MAR (memory address register), 9
Mark I computer, 269–270, 272
mask register (associative computers), 207
massively parallel processor (MPP), 195–197
matching store, 210, 212
Mauchly, John William (1907–1980), 270, 272
MBR (memory buffer register), 9
McLuhan, Herbert Marshall (1911–1981), 131
Mega (= $2^{20}$), 6
memory, 1, 5–11, 241–246
  access time, 9
  address register (MAR), 9
  addresses, 5
  bandwidth in vector processors, 205
  banks in vector processors, 204
  buffer register (MBR), 9
  burst DRAM, 242
  cache RAM, 242
  conflicts, 200
  core, 272
  cycle time, 9
  delay lines, 241
  destructive, 241
  distributed, 190
  DRAM, 241
  extended data out DRAM, 242
  fast page mode DRAM, 242
  FPLA, 246
  magnetic cores, 7, 241
  no blank words, 5, 8
  nonvolatile, 8, 241
  operations, 8–11
  PAL, 246
  parity bit, 8
  PLA, 245–246
  PROM, 243
  RAM, 241–242
  read, 8
  ROM, 7–8, 243–245
  semiconductor, 7
  shared, 190
  size of, 5, 6
  SRAM, 241–243
  synchronous DRAM, 242
  synchronous graphics DRAM, 242
  video DRAM, 242
  volatile, 7, 241
  word size, 5
  words, 5
  write, 8
memory banks, 200
memory bus, 10
memory reference instructions, 47
memory requestor, 9
  DMA, 9, 77
memory-mapped I/O, 47, 73–75
Merkle, Ralph C., 123
message passing in hypercube, 192
Metcalfe, Robert (ethernet inventor), 149–150
microprocessors (history of), 272–273
microprogram (definition of), 18, 167
microprogrammed control unit, 17, 167
microprogramming, 17–20, 167–188, 273
  advantages of, 19
  horizontal, 185, 188
  vertical, 185
MIMD computers, 190, 199–200
MIR (micro instruction register), 18
miscellaneous instructions, 47
modems, 133, 141–144
  cable-tv, 144
  Express 96, 143
  Hayes commands, 143
  Smartmodem 1200, 143
  Smartmodem 300, 143
  Smartmodem 9600, 143
  standards of, 142–143
modes of processor, 24–25
modulation, 133–137
  amplitude, 133
  bitrate, 135
  constellation diagram, 135
  frequency, 133
  PAM, 135
  phase, 133
  QAM, 134–135
  signaling rate, 135
modulator, 133
monoalphabetic substitution codes, 109
monostable (digital device), 228, 231

Motorola computers
 6800, 273
 68000, 273
 68020, 273
 PowerPC, 273
`MOVE` instruction, 42
MPC (micro program counter), 18
MPP (SIMD computer), 195–197
multicomputers, 190, 199
multilevel indirect mode, 40
multiplexors (digital device), 237
multiplication
 of floating-point numbers, 58–60
 of integers, 253–260
multiprocessors, 190, 199
 memory banks, 200
 memory conflicts, 200
multiprogramming, 28
multivibrators (digital devices), 228–232
Myszkowsky cipher, 120

$N$-operand computer, 48
$N$-operand instructions, 48
nanoinstructions, 186
nanoprogramming, 186
nanosecond (ns), 9, 169
nanostore, 186
national security agency (NSA), 108
negative zero, 56
 absent in 2s complement, 57
nested interrupts, 1, 22
networks, 148–165
 internet, 152–153
  addresses, 156–160
  IP, 155–156
  TCP, 156
 local area, 148–151
 wide area, 151–152
Newman, Max (and Colossus), 269
Nieuwenhoff, Auguste Kerckhoffs von, 109
nomenclator (secure code), 108
non von Neumann architectures, 190
non von Neumann computer, 206
nonnumeric data types, 53
`NOP` instruction, 47
normalization of floating-point numbers, 58–59
not a number (NaN in the IEEE f.p. standard), 63
Noyce, Robert Norton (1927–1990), 272
numbers
 BCD, 64–65
 fixed point, 63–64
 floating point, 57–63
 rational, 68
 signed, 54–57
numeric data types, 54
Nyquist theorem, 283

octaword, 51
one's complement, 54–56
one-time pad cipher, 113, 121
opcode, 19, 33
 average size, 35
 fixed size, 35
 size, 34–36
 variable size, 34–36
opcode decoder, 11
operand specifier byte, 51
operating system, 28
 and initializing the PC, 4
 bootstrap loader, 4, 8
 definition of, 4
 resident, 4
operation code, 33
operations (type of machine instruction), 43
Opler, A., 168
orthogonal transform, 103
orthonormal matrix, 104, 106
overflow, 68–70
 definition, 68
 status flag, 69

packet switching, 152
PAL, *see* programmable array logic
PAM modulation, 135
parallel adder, 30
parallel algorithms, 200–201
parallel computers, 189–217
 distributed memory, 190
 shared memory, 190
parallel processing, 189–217
 main problems of, 189
parallel programming
 integration, 201
 matrix multiplication, 201
 primes, 200
 rank sorting, 200
 sorting on a tree, 201
 summing numbers, 201
parity bits, 8, 87
 in ASCII, 80
Pascal, Blaise (1623–1662), 267
Patterson, David A., 222
PC, *see* program counter
PE (processing element), 193
Peano space-filling curve (and steganography), 128
pel (picture element), 98
Pentium microprocessor, 273
perfect shuffle
 as a transposition cipher, 121
 in DES, 122
periodic codes, 91
permutations
 transposition ciphers, 114–120
PGP, *see* pretty good privacy
phase modulation, 133

phase shift keying, 133
PIC, *see* priority interrupt controller
Picasso, Pablo (1881–1973), 126
pipelining, 184
pixels
  correlated, 104
  decorrelated, 103
  definition of, 102
  highly correlated, 102
PLA, *see* programmable logic array
Poe, Edgar Allan, 110
Poe, Edgar Allan (1809–1849), 70
Poel, L. W. van der, 168
Pohlmann, Ken C., 290
points (cross correlation of), 106
polled I/O, 74
polyalphabetic substitution codes, 110
  compared to RSA, 126
polynomials
  generating, 92–93
  primitive, 122
position independent code, 38
post-indexed indirect mode, 39
PowerPC microprocessor, 189, 273
pre-indexed indirect mode, 39
prefix codes, 95
  used with opcodes, 35
prefix property, 35, 95
pretty good privacy (PGP), 126
primitive polynomial (in cryptography), 122
priorities of interrupts, 24
priority encoders (digital device), 23, 238–239
  and interrupts, 24
priority interrupt controller, 28–30
privileged instructions, 24–25
procedure call
  conditional, 47
  forced, 21
processing elements (PEs), 190
processor, 1, 2
  modes, 24–25
program counter, 2, 48
  and computer start, 4
  as a GPR, 2
  in relative mode, 37
  incrementing, 2, 47
  register 15, 51, 299
  resetting, 3, 13, 46, 299, 300
  return address, 16
  saving, 16
program status (PS), 25
programmable array logic, 246
programmable logic array (PLA), 19, 176, 185, 245–246
PROM, 243
protocol in serial I/O, 139–141
PSW updating, 47
PU (processing unit), 193
public key cryptography, 125–126
Pyramid computer, 222

QAM modulation, 134–135
quadword, 51
quantization
  image transform, 103

RALU (registers and ALU), 2
RAM, 241–242
random access (definition of), 8, 241
random access memory, *see* RAM
raster order scan, 104
rational numbers, 68
read only memory, *see* ROM
reduced instruction set computers (RISC), 219–226
redundancy
  and data compression, 93–94
  and reliable codes, 84–93
  in data compression, 94–95
  in images, 103–107
Reed, Irving S., 92, 285
Reed-Solomon Code (CIRC), 92, 285
reflected BCD codes, 67
reflected Gray codes (RGC), 67
register direct mode, 51
register I/O, 74
register indirect mode, 41
register mode, 41
register transfer notation, 1
register transfers, 236
  and instruction execution, 13
registers, 236
  accumulator, 252
  auxiliary, 170, 181
  general-purpose, 170
  shift, 263
relative mode, 37–38, 180
reliable codes, 84–93
resolution of images (defined), 102
RGC, *see* reflected Gray codes
Ridge computer, 222
ring counters, 235
RISC, 185
  advantages of, 221
  definition of, 220
  disadvantages, 221
  floating point numbers on, 221
  history, 222
  optimizing compiler for, 221
RISC I computer, 48, 52, 222–226
  delayed jumps on, 224
  instruction set, 52–53
RISC II computer, 222
Rivest, Ronald L., 125
Robinson computer, 269
ROM, 7–8, 243–245
Rosin, Robert F., 168, 186

RS latch, 229
RSA, *see* public key cryptography
run lengths (data compression), 98

S status flag, 46
scatter (vector instruction), 200
Scherbius, Arthur (Enigma inventor), 113
Scheutz, George & Edvard, 268
Schickard, Wilhelm (1592–1635), 267
search (associative memory), 206
SEC-DED codes, 90–91
second generation computers, 252, 272
secure codes (cryptography), 107–126
secure codes (steganography), 127–129
secure codes (watermarking), 131
self-complementing BCD codes, 67
semiconductor memories, 7
sequencer (in the control unit), 13, 17, 18
sequential digital device, 228
Sequin, Carlo, 222
serial I/O, 132–141
  asynchronous, 137, 139
    start bit, 139
    stop bits, 139
  protocol, 139–141
  synchronous, 137–138
    hunt mode, 138
serial interface, 132
serial port, 132
Shakespeare, William (and letter frequencies), 94, 96
Shamir, Adi, 125
Shannon, Claude Elwood (1916–2001), 85, 131
shift registers, 263
shifts, 45–46, 263
  arithmetic, 45
  circular, 45
  double, 46
  logical, 45
  through carry, 46, 252
short real (IEEE floating point format), 62
sieve of Eratosthenes, 200
sign bit, 54
sign convention (of a floating point number), 60
sign-magnitude, 54
signaling rate (in wave modulation), 135
signed integers, 54–57
  and carry/overflow, 69
significant digits of a floating point number, 58
SIMD computers, 193
simple mail transfer protocol, 140–141
simplex, 141
simulation, 184
simultaneous interrupts, 1, 22
single-bus computer organization, 47, 72
`SKIP` instruction, 47
sliding window compression, 100–102
smallest positive floating-point, 62
SMTP, *see* simple mail transfer protocol
software (definition of), i
software interrupt, 20, 24–25
Solomon, Gustave, 92, 285
SONET (Synchronous Optical NETwork), 147
sorting (associative computer), 208
Sprague, Ruth M., 239
SR latch, 122, 229–231
SRAM (static RAM), 241–243
stack mode, 41
stack relative mode, 41
start bit, 139
static RAM (SRAM), 241–243
status bit of a PU, 195
status flags, 16, 46
steganography, 107, 127–129
  cover image, 128
  hiding a file, 45
stop bits, 139
`STORE` instruction, 42
stream ciphers, 45, 121–122
subband transform, 103
substring operation, 53
subtraction of integers, 247–250
superadditive sequence, 55
supervisor call, 20
synchronous DRAM, 242
synchronous graphics DRAM, 242
synchronous serial I/O, 137–138

T latch, 231
T1 (speed of serial transmission), 132, 145
T3 (speed of serial transmission), 132, 145
Tanenbaum, Andrew, 167, 170, 185
taps (wavelet filter coefficients), 123
TCP (transmission control protocol), 152, 156
telecommunications (and serial I/O), 132–141
temporary real (IEEE floating point format), 63
Tera ($= 2^{40}$), 7
text
  English, 111
third generation computers, 252, 272
three-address instructions, 41
three-bit shift (integer multiplication), 259–260
time slices, 25–26
token matching, 211
token ring LAN, 150–151
topology, 201
training (in data compression), 96, 98
`TRAnsfer` instruction, 42
transforms
  AC coefficients, 106
  DC coefficient, 106
  images, 103–107
  orthogonal, 103
  subband, 103
transmission control protocol (TCP), 152, 156
transposition ciphers, 114–120

columnar, 119–120
letter frequencies, 114
turning template, 117–118
transputer, 222
Trask, Maurice, 165
trigram (of letters), 110
Trithemius, Johannes, 127
Trollope, Anthony, 70
truth table, 30
Turing, Alan Mathison (1912–1954), 268
and Enigma, 268–269
turning template transposition ciphers, 117–118
twisted ring counters, 235
two's complement, 56–57, 249
observations about, 249, 255

unconditional branch, 46
unicode, 81, 121, 275–282
unit distance BCD codes, 67
UNIVAC, 272

variable size codes, 94–95
variable size opcodes, 34–36
variance (as energy), 106
VAX
addressing modes, 51
instruction set, 50–52
vector instructions, 205
vector processing, 44, 61
vector processors, 204–205
memory bandwidth, 205
memory-memory, 204, 205
register-register, 204, 205
vectored interrupts, 1, 21–23
Vernam cipher (one-time pad), 121
vertical microprogramming, 185
video DRAM, 242
Vigenère cipher, 110–113
Vigenère, Blaise de (1523-1596), 110
virtual memory, 50
VLSI, 189, 272
von Neumann architecture, 190
von Neumann bottleneck, 10, 190, 206, 210
von Neumann machine, 10, 271
von Neumann, John (1903–1957), 271
voting codes, 85–86
Vázsonyi, Andrew, 184

watermarking (digital data), 131
web (world wide), 163–165
web client, 164
web server, 164
weighted BCD codes, 65–67
Whitehead, Alfred North (1861–1947), 297
wide area networks, 151
Wiener, Norbert, 165
Wilkes, Maurice Vincent, 168
word displacement mode, 51
word relative mode, 51
word size, 5, 7
words in memory, 5
Wostrowitz, Eduard Fleissner von (turning template cipher), 117
www (world wide web), 163–165

XOR (exclusive OR), 44–45, 300

Z status flag, 46
Z-80 microprocessor, 273
Z-8000 microprocessor, 273
zero page mode, 40
Zimmermann, Philip R. (PGP, 1954–), 126
Zuse, Konrad (1910–1995), 268

The first day Holmes had spent in cross-indexing his huge book of references. The second and third had been patiently occupied upon a subject
which he had recently made his hobby—the music of the Middle Ages.

—*Arthur Conan Doyle, His Last Bow*