Lecture Notes on Computer Systems and Operating Systems

Jonathan G. Campbell
Department of Computing,
Letterkenny Institute of Technology,
Co. Donegal, Ireland.

email: jg.campbell@ntlworld.com, jonathan.campbell@lyit.ie
URL:http://homepage.ntlworld.com/jg.campbell/lyitcsys/
Report No: jc/04/0001/r
Revision 5.2

4th January 2004

Contents

Chapter 1

Introduction

1.1 Purpose

This document is an introduction to the architecture and organisation of computer systems, including operating systems. It provides course notes for the courses: *Computer Systems* and *Operating Systems I*, part of the course *National Certificate in Computing (I.T. Support)*.

Let me explain why the notes for the two courses are together. Originally, I kept Computer Systems notes separate from those of Operating Systems. But this meant that there was much duplication — because a good knowledge of Computer Systems (hardware and basics of programs/software) is a prerequisite for a study of Operating Systems, and when teaching Operating Systems I could never be sure that students had a sufficient knowledge of Computer Systems. So now there is just one set of notes.

1.2 The Courses

These courses (Computer Systems and Operating Systems I), as contained in the syllabuses and as taught by me, are meant to be introductions to the hardware and software of computer systems. They are meant to provide a foundation for learning about practical systems. Neither course is meant to make you into a fully trained computer technician in twelve weeks. What you learn in these courses provide a basis for lifelong learning in a rapidly evolving subject area.

1.3 Lectures, Learning . . .

Education is an *active*, not *passive* process. Thus, I strongly encourage you to:

- Take notes during lectures, rather than just following along with these notes. Make sure to bring extra notepaper — these notes will quickly become cluttered if you try to superimpose significant additional notes on them;
- Do the exercises. At least attempt them — they, or problems similar to them, crop up in exams;

- Participate in the tutorials – by *participate* I mean attempt the exercises and enter into discussion if the presented solution is different from yours; ask questions about things that remain unclear to you. Don't forget, *there are no stupid questions, just people too stupid to ask them*!
- Form study groups; if you, individually, have a problem with the course, it's your problem, but if it becomes clear that a good number of students have the same problem, then it can become the lecturer's problem!
- Spend a decent amount of time on the assignments and practicals — they are designed to help you learn.

1.4 Reading — Computer Systems

These notes, and whatever is handed out at lectures, constitute the *essential* reading. However, in addition to handing out some extra notes and exercises during lectures, I may also inform you that certain sections of these notes are *not* part of the course, and as such are *off limits* for examinations and other assessment.

You will learn a lot from reading from other textbooks and articles — I strongly encourage you to read as widely as possible.

These notes were originally based on (Tanenbaum 1990). This is now in its 4th edition (Tanenbaum 1999), but the earlier edition suits our needs best. The reason I have developed these notes is that I figure the books are slightly advanced for a course like this — alright to read once you've attended the lectures, but not *teach-yourself*.

One book that would be a very nice supplement to this course is (Petzold 2000).

Hillis (Hillis 1999) gives a very readable introduction for lay-persons, but could be very interesting to you for its numerous insights on the basics of computers. (BBC 1992) is the book of the TV program that the BBC ran at the end of 1991. We may get a chance to watch parts of the videos.

(Ferry 2003) is a very readable account of the early history of computing.

If this course gives you an appetite for electronics, and you want to teach yourself more, there is no better than (Horowitz & Hill 1989).

If you want an easy but comprehensive introduction to computer science (Brookshear 1999) is clear and easy to read; it covers many of the topics on this course. The two books by Patterson and Hennessy are achieving the status of *bibles* of computer architecture theory, (Hennessy & Patterson 2002), (Patterson & Hennessy 1994), but go well beyond what is needed for this course. (Stallings 2000a) is very popular and maybe worth looking at in the library.

For practicals, we will occasionally refer to (Dick 2002) and (Mueller 2001).

If you want help on taking PCs apart, swapping disks, upgrading CPUs and motherboards, (Mueller 2001) is encyclopedic; see also (Thompson & Thompson 2000).

1.5 Reading — Operating Systems

Most of the notes on operating systems were based on (Tanenbaum 2001) and (Silberschatz, Galvin & Gagne 2002).

See also (Stallings 2000b), (Harris 2002), (Flynn & McHoes 2001), (Sybex 2000), and (but don't spend money buying it!) (Ritchie 1997).

For operating systems practicals, where we work mostly with Windows 2000, I use (Wallace 2000). (Dick 2002) also contains much useful information as does (Tulloch 2001).

For easier introductions to Windows 2000 and Windows in general, see (Meyers 2001), (Stewart & Scales 2000).

Chapter 2

Overview of Computer Systems and Operating Systems

2.1 Introduction

The objective of this chapter is to give a quick overview of the composition of a computer system — a sort of *Dummies Guide to Computer Systems in 24 Minutes*!

2.2 Simple Model of a Computer – Part 1

A very simple model of a computer is shown in Figure 2.1. At its simplest, the *Central Processing Unit (CPU)* is a glorified calculator, the *memory* stores the results of calculations – it is a set of labelled cells – the label is called the *address*, each cell contains a number – *data*. The *bus* is simply a pipeline along which data and addresses can flow.

Both data and addresses are stored as numbers, though data numbers may be *interpreted* otherwise, e.g. as text characters. As well as containing data, the memory also contains the *program* – in the form of numeric instructions. Ultimately, everything is reduced to numbers.

What is a computer program? A program is a sequence of instructions for the computer to obey. Take, for example, the C/C++/Java instruction:

```
z = x + y; /* add variables x and y, result in z*/
```

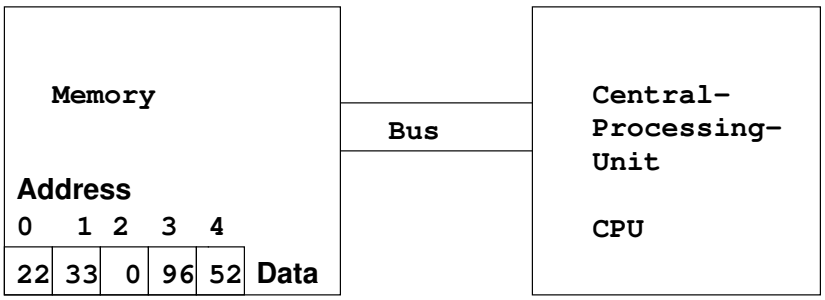


Figure 2.1: Overview of a Computer: CPU and Memory

This says: (i) take the number in the cell labelled `x`, (ii) take the number in the cell labelled `y`, (iii) add them, (iv) put the result in the cell labelled `z`.

In a programming language, cells are called *variables*, variable names are easy to remember names for memory cells: e.g. `x` at memory cell 0, `y` at 1, `z` at 2; therefore in Figure 2.1 the value in x is 22, in y, 33, and in z, 0. Note the difference between address (for y, it is 1) and data or value (for y it is 33). Note: the terms *memory cell*, *location*, *address* are often used interchangeably.

P1. Read the contents of memory cell corresponding to x.

P2. Read the contents of memory cell corresponding to y.

P3. Do the addition,

P4. Write the result in the memory cell corresponding to z.

Four or *five* machine instructions per high-level language instruction is typical.

Although we usually talk of just *bus*, there are three parts to a normal bus: the *data-bus*, the *address-bus*, and the *control* part. The bus operates as follows, take P2 above: first the address (1) is put on the address-bus, then control asserts *read-memory*, some time later the contents of address 1 (33) is put on the data-bus.

Some detail is missing:

- Control of the CPU, e.g. stepping it through P1 ... P4.
- Where do the machine instructions P1 ... P4 come from?

The answer to these is that the machine instructions are just another form of data, and the controller in the CPU must fetch them from memory, and execute them; thus, there are two sorts of data flowing along the bus - actual numeric values, and machine instructions; the controller is in a continuous cycle:

- Fetch P1.
- Execute P1.
- Fetch P2 ... Execute P2.
- ... and keep on until it reaches a *halt* instruction.

This is the famous *fetch-execute cycle*.

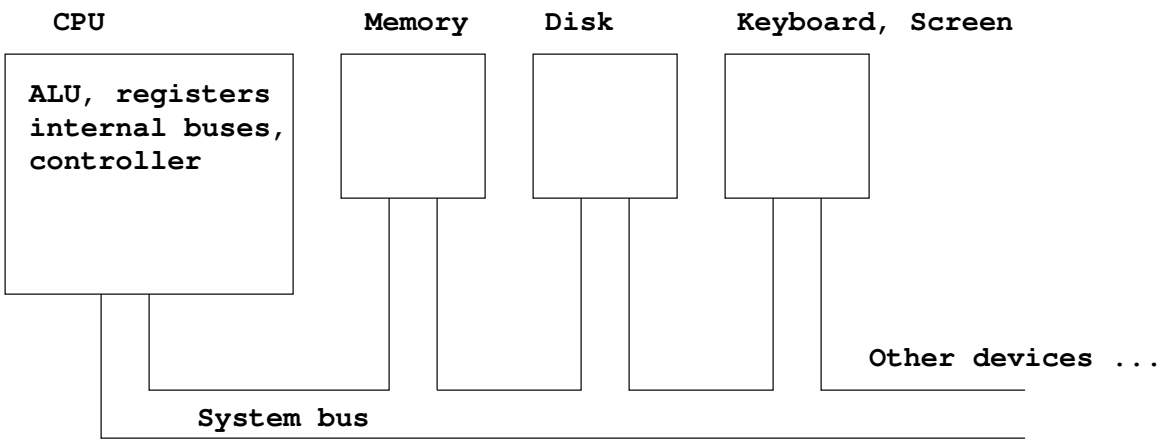


Figure 2.2: Organisation of a Simple Computer System

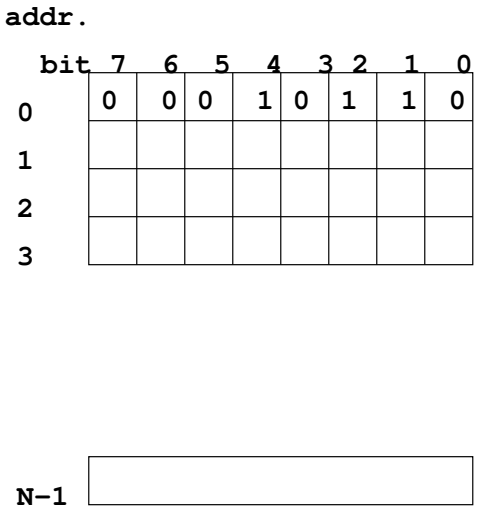


Figure 2.3: Memory

2.3 A Computer System

A CPU and memory aren't much use without a keyboard and screen, and a disk drive, i.e. a useful computer is actually a *computer system*.

Figure 2.2 shows the organisation of a simple computer system with one CPU, memory and three input-output (I/O) devices – *peripherals*: a disk and a keyboard & screen. As always, the components are connected by a system bus – this is the same as the bus mentioned above; communication with peripherals can be done in a manner similar to memory reading/writing.

Figure 2.3 shows main memory; it is essentially a collection of numbered (addressed) cells, each of which, typically, can store eight binary digits – *bits* (**BI**nary digi**T**s). Each bit is like a *sub-cell* that can be '0' or '1'. Bits have much in common with light-bulbs: they are either *on* — value 1, or *off* — value 0.

A collection of eight bits is called a *byte*.

Each cell has an address. Addresses run from 0 to N-1, where N is the size of the memory; on a typical PC these days, N is 64 Mega(bytes). Mega = 'million' — actually, $1048576 = 2^{20}$.

Numeric values Each bit is a single binary value: you can count only from 0 to 1 with it; however, a collection of eight bits, properly interpreted, allows you to count from 0 up to 255; and 16 bits, 0 ...65535 (64K - 1). The byte shown in address 0 of Figure 2.3 contains *binary* `0001 0110` which is *decimal* (base-10) 22, *hexadecimal* (base-16) 0x16, or *octal* (base-8) 026. (We will cover number systems in chapter 2).

Essentially, Figure 2.2 conforms to the so-called *von Neumann* computer architecture, which has five basic parts:

1. The arithmetic-logic unit (ALU).
2. The control unit. The ALU and the control unit comprise the CPU.
3. The memory.
4. Input equipment.
5. Output equipment.

As we have already said, memory may contain, in addition to data, program instructions, coded as numbers. One major breakthrough of the von Neumann architecture was that data (what is processed), and instructions — that describe the processing to be done — are both numbers, and both reside in the same memory. But, *inside the machine*, there is nothing to distinguish data from instructions, the program must keep track of what parts of memory are data and what part instructions.

A memory cell (location, variable), therefore, can store:

- True numerical values;
- Numerical values which are eventually translated to other meanings, e.g. ASCII text character codes;
- Program instructions; and, wait for it,
- Addresses of other locations/cells).

You should note carefully the difference between *data* and *address*. Conceptually, the difference is the same as that between *you* and (for example) your house address or your student number. The confusion arises because, in computers, data and addresses are the same form, i.e. numbers. But, conceptually, they are as different as chalk and cheese.

2.4 Central Processing Unit

2.4.1 Introduction

As indicated, the CPU consists of an Arithmetic-Logic-Unit (ALU) – the calculator, and a controller – to do the fetching and executing of the program instructions; in addition, we need a few local memory registers – for storage of temporary data, and buses to transfer data and addresses throughout the CPU.

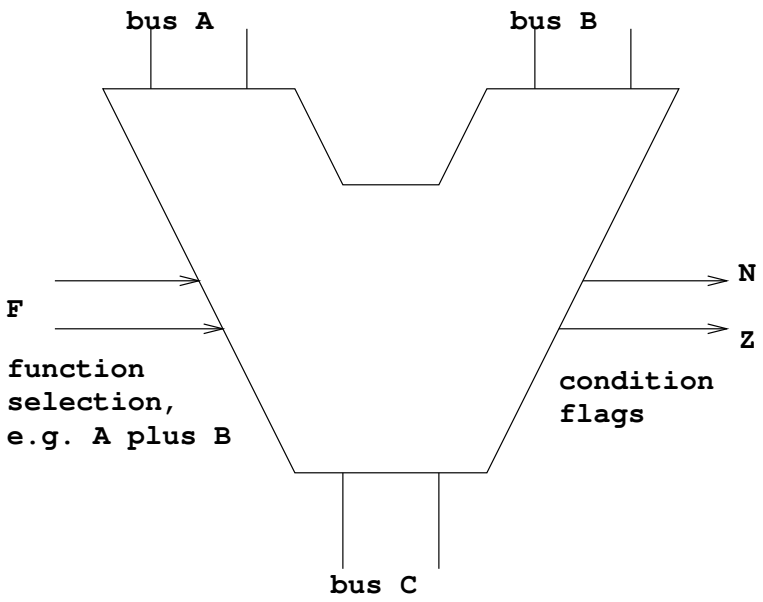


Figure 2.4: Arithmetic and Logic Unit (ALU)

2.4.2 Arithmetic Logic Unit – ALU

The ALU is the calculator part. The ALU takes the contents of ALU input buses A and B and combines them in some way (add, subtract, logical OR, etc.) and passes the result onto a C data bus. Let us assume that this is a 16-bit machine, i.e. the buses and ALU take 16-bit numbers – they are 16-bits wide.

A simple ALU is shown in Figure 2.4. It has a number of functions which are selected by function selection control lines F; the control lines are equivalent to selecting the operation on a calculator, e.g. add, subtract. Two condition flags are output: N, set to 1 if last operation caused a negative result; Z, last operation, if a zero result; result flags are important for constructing conditional instructions (*if*, *else*).

2.4.3 Simplified CPU

Figure 2.5 gives a view of a simplified CPU – but not so simplified that we cannot describe how an actual program might operate in it; with few additions figure 2.5 would actually work. The major thing we have left out is *control*.

In addition to the buses – both internal and system – the system bus is one on the far left), and ALU, all mentioned before, we have just three new items: the *memory address register (MAR)*, the *memory buffer register (MBR)*, and the *accumulator register(AC)*.

Conceptually, registers are no different from main memory cells.

The MAR and MBR are used for communicating with memory: when the CPU needs to read *load* from memory, it puts the address of the required item into MAR, asserts *read* on the control part of the system bus, some time later, the system bus will transfer the relevant data item into the MBR; for write *store*, it puts the address in MAR, the data in MBR, and asserts *write*, the data are then copied into the relevant memory location.

The AC is used as a general holding register for operand (input) data and results.

All will become clear if we return to the example program statement:

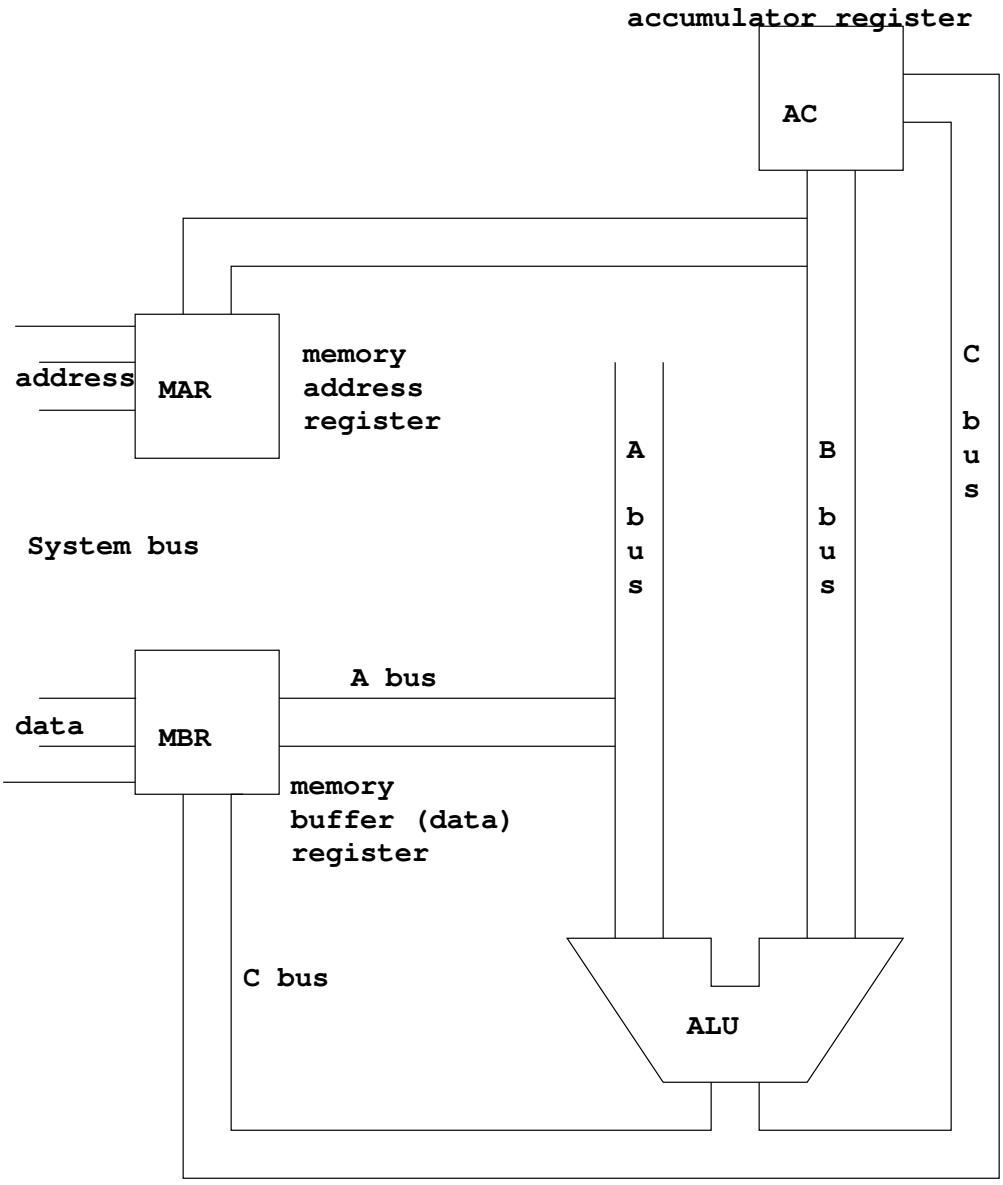


Figure 2.5: Central Processing Unit

```
z = x + y;
```

As in the introduction, assume that z corresponds to location 2, x to 0, and y to 1. The program, as it would appear in machine instructions, is:

P1. Load contents of location 0 (x, 22) into AC.

P2. Add contents of 1 (y, 33) to AC, with result going to AC.

P3. Store contents of AC in location 2 (z).

2.5 Software

Without software, the system in Figure 2.2 just sits there quietly, doing nothing!. It must be programmed with appropriate *software*. Roughly speaking, software can be divided into two categories:

System software: operating system and utility software;

Application software: e.g. word processing, spreadsheets, . . . ; also user-developed software.

The divide, between system and application, can be somewhat blurred. In general, the more general purpose and *ordinary* the function performed – like copying or deleting files, or compiling or assembling programs – the closer to system software; the more special purpose – like a database management system – the closer to application category.

Originally, the system software was likely to be developed by the computer *hardware* manufacturer; however, since the 1980s, the PC revolution has turned this on its head.

The most important item of system software is the *operating system*.

2.6 Computer Programs

As we have said earlier, a computer program is a sequence of instructions for the computer. (There are other ways of viewing programs, especially when programming in a high-level language like VisualBasic, but, considering the *native* language of the machine (numerically coded machine language), this is definitely true.)

In the lectures, I will frequently refer to the analogy between computer programs and a cookery recipes, see Figure 2.6 below.

211. Soufflé Omelet. Cooking time 15 minutes

Quantities for 1 large or 4 small omelets:

4 fresh eggs
¼ c. water
½ tsp. salt
Pinch of pepper
½ oz. melted butter (1 Tbs.)
Chopped parsley

1. Separate the yolks and whites of the eggs and beat the yolks with the water until thick and lemon-coloured. Add the seasonings.

2. Beat the egg whites stiffly.

3. Melt the fat in the pan.

4. Fold the egg yolks into the whites very gently and pour into the pan. Cook very slowly for about 5 minutes, when the omelet should be golden underneath and beginning to rise up in the pan. If the heat is too great it will rise up quickly and then collapse and be tough.

5. Continue cooking in a moderate oven or under a slow grill for 8–10 minutes, until the top looks dry. Do not cook too long, or it will shrivel and be tough.

6. Fold in half and turn on to a hot dish. Sprinkle with chopped parsley and serve at once.

N.B. For variety this omelet may be served with a sauce poured over – e.g. Onion Sauce, No. 85; Tomato Sauce, No. 89; Cheese Sauce, No. 75; Mussel Sauce, No. 83; or Mushroom Sauce, No. 82.

89. Tomato Sauce (for fish, meat, vegetables, eggs)

1 oz. fat (25 g.)
1 onion, chopped
2 or 3 bacon rinds
1 oz. flour (3 Tbs.)
½ bay leaf
½ pt. tomato juice or 2 Tbs. tomato paste made up to ½ pt. with water (1 c.)
½ pt. stock (1 c.)
½ tsp. sugar
Salt and pepper to taste

Measures level. Fry the onion and bacon rinds in the fat, and then add the flour and finish as for the Roux Method, No. 69, adding the bay leaf with the stock and tomato. Boil for ½ hour. Strain before using and add the sugar and seasonings

69. The Roux Method of Making a Sauce. Cooking time 10 minutes or longer

Quantities for 4–8 helpings: (When making only half a recipe use a little less than half the thickening or thin as necessary before serving; a small quantity of sauce is always thicker because there is greater evaporation during cooking.)

Ingredients	*Thin or Pouring Sauce*	*Thick or Coating Sauce*
Butter, margarine, fat or oil	1 oz. (25 g.)	2 oz. (50 g.)
Flour	1 oz. (3 Tbs.)	2 oz. (6 Tbs.)
Liquid (milk, stock, or half and half)	1 pt. (2 c.)	1 pt. (2 c.)

Measures level.

1. Melt the fat and stir in the flour. Mix well and cook very gently for 1–2 minutes or until it looks mealy for a white sauce, or until brown for a brown sauce. This mixture of fat and flour is called a 'roux'.

2. Add the cold or warm (not hot) liquid and stir or whisk vigorously until the sauce is smooth and boiling. Boil it gently for 5 minutes or cook for 10–20 minutes over boiling water, stirring occasionally. If the sauce shows any tendency to be lumpy, either beat it hard with a small wire whisk or put it in the blender for a few seconds. Whisking or blending very much improves texture and appearance.

3. If the sauce is to be kept hot for any length of time it must either be put in a double boiler or the saucepan stood in another pan of simmering water; stir occasionally to prevent a skin from forming or run a thin layer of melted butter or margarine over the top. Stir this in before serving the sauce.

Figure 2.6: Computer programs have much in common with recipes (recipes from B. Nilsson, The Penguin Cookery Book)

A computer program is a set of instructions for a computer. The instructions must be correct – to the finest detail; computers know nothing other than what is in their programs. (That is where the cookery recipe analogy can fail a little – in a cookery recipe, we assume that the cook needs instructions only to a certain level; for example, we don't see instructions on how to boil water, or how to crack an egg.)

At present, we will assume that the computer knows how to add, subtract, read/load from memory, store/write to memory, but little more.

The following is not unlike the sort of administrative task that I have to do at the end of terms.

I have to compute coursework marks for each student. Assume that there are three courseworks, one worth 20%, the other two worth 40% each. I have the courseworks marked, all marked out of 10. Let's say I have a file for each student, marked a, b, c, d, ... For now, I want to assume that there are just four students (a, b, c, d).

In the files, I have coursework forms, marked `a1, a2, a3` for student a's courseworks 1, 2, 3. I also have a form marked `at` for the total mark, and a form marked `ag` for the grade (100–70, grade A; 69–60, grade B, 59–40, grade C; 39–0, grade F).

Similarly, `b1, b2, b3, bt, bg` for the next student ....

My program for computing coursework is as follows. I'm going to complicate things a little by assuming that I cannot multiple, but have to do multiply by adding.

My data: `a1, a2, a3, at, ag;`

`b1, b2, b3, bt, bg;`

`c1, c2, c3, ct, cg;`

`d1, d2, d3, dt, dg;`

And I need some 'working' data: `total`; assume that I have a sheet on my desk marked `total`.

1. Write 0 on the sheet marked `total`;
2. What we want to do here is add 2× `a1` to `total`.

2.1 Take sheet `a1`, and sheet `total`; add the numbers you see (one number on `a1`, the other on `total`); take the result, erase the number on `total`, and write the result there;

2.2 Take sheet `a1`, and sheet `total`; add the numbers you see; take the result, erase the number on `total`, and write the result there;

3. What we want to do here is add 4× `a2` to `total`.

3.1 Take sheet `a2`, and sheet `total`; add the numbers you see (one number on `a2`, the other on `total`); take the result, erase the number on `total`, and write the result there;

3.2 Take sheet `a2`, and sheet `total`; add the numbers you see; take the result, erase the number on `total`, and write the result there;

3.3 Take sheet `a2`, and sheet `total`; add the numbers you see; take the result, erase the number on `total`, and write the result there;

3.4 Take sheet `a2`, and sheet `total`; add the numbers you see; take the result, erase the number on `total`, and write the result there;

4. What we want to do here is add 4× `a3` to `total`.

   I won't repeat the steps; they are the same as 3.1–3.4; however, note that you *would* have to repeat them in a computer program.

   At the end of 1–4, we have the courswork total percentage in `total`.

5. Read the number on `total` write that number onto `at` (students total);

6. Decide on grade. Read `total`;

6.1 If greater-than-or-equal-to 70, write A to sheet `ag`; jump to step 7.

6.2 If greater-than-or-equal-to 60, write B to sheet `ag`; jump to step 7.

6.3 If greater-than-or-equal-to 40, write C to sheet `ag`; jump to step 7.

6.4 Write F to sheet `ag`. Note: we arrive here only if all of the tests in 6.1–6.3 evaluate to false.

7. Write 0 on sheet marked `total`;

8. What we want to do here is add $2 \times B1$ to `total`.

   Etc, repeat the whole lot for `b`; then `c`, `d`.

   Of course, you've spotted that we could shorten the program a lot by writing at 7.

7. Repeat 1.–6. substituting `b`s for `a`s

8. Repeat 1.–6. substituting `c`s for `a`s

9. Repeat 1.–6. substituting `d`s for `a`s

   But we cannot, because we are still too dumb to understand the concept of `repetition` — i.e. `for` loops, arrays, ...

In the example, we can identify two major computer program structures:

Sequence The simplest structure, do step 1, do step 2, ...;

Selection Do a test; if it is true, do something; otherwise fall through onto the next instruction.

There are three further structures:

Repetition Repeat a set of steps: (a) a number of times; or, (b) while some condition remains true;

Procedure (or subprogram). You could imagine writing a program which deals just with one student, call it `SProg1`.

Then at 2. Do `SProg1` and so on for students b, c, d.

(I've left out some detail – we would have to copy the marks to some common area, e.g. a blackboard, so that the person carrying out the instructions in `SProg1` would see them; and we would have to modify the instructions in `SProg1` to refer to places on the blackboard, rather than to `a1, a2, a3`. At the end of each run through `SProg1`, we'd have to copy the `total` and `grade` results to the appropriate student sheet.)

Procedure with parameters . Again imagine writing a program which deals just with one student, call it `SProg2(mark1, mark2, mark3)`. Make `SProg2` have not just instructions on it, but also three blank areas onto which you can write the three *input marks*; also a blank space for the `total`, and `grade`.

Then at 2. Do `SProg2(a1, a2, a3)`, i.e. copy `a1, a2, a3` onto the appropriate blank areas; when done, copy `total, grade` onto the student sheet.

Likewise 3. Do `SProg2(b1, b2, b3)`.

And so on, for students c and d.

That's it. All possible programs can be described in terms of these five structures: sequence, selection, repetition, procedure, procedure with parameters. This fact should ensure that you can never claim that computer programming (or understanding a computer program) is other than *very simple*!

As mentioned earlier, we will frequently refer to the similarities between computer programs and recipes. The set of recipes shown in Figure 2.6 will be used to demonstrate this point.

2.7 Simple Model of a Computer – Part 2

At the beginning of the chapter, we introduced a simple model of a computer as a calculator together with a set of pigeon holes in which data (numbers) are stored. Later, in section 2.4, and Figure 2.5, we showed something which is more or less a true depiction of the processing part of an actual CPU (you could make that CPU and it would work).

Now, since at the beginning of a course, funny diagrams may be hard to understand, and their operation hard to imagine, I want to develop a mechanical model of a CPU that we can use in class. I will be as brief as possible here – you will need to come to the class to get the full details.

The components are as follows:

- CPU, consisting of:
  - ALU;
  - AC register;
  - MAR and MBR registers;
  - some buses connecting these parts.
- Memory;
- a bus connecting Memory to MBR and MAR.

The model is built as follows. The CPU is a person sitting at at desk; on that desk is a simple calculator, the ALU; the calculator can handle only integers. Also on the desk is a tray (like an in-tray) that can hold a sheet of paper, marked AC (i.e. this models the AC register). Likewise, at the edge of the desk, we have two other trays, one marked MAR, the other MBR. As we have remarked before, the MAR and MBR form a gateway between the CPU and the bus and Memory. As well a MAR and MBR, there are two indicators, one marked Read, one marked Write.

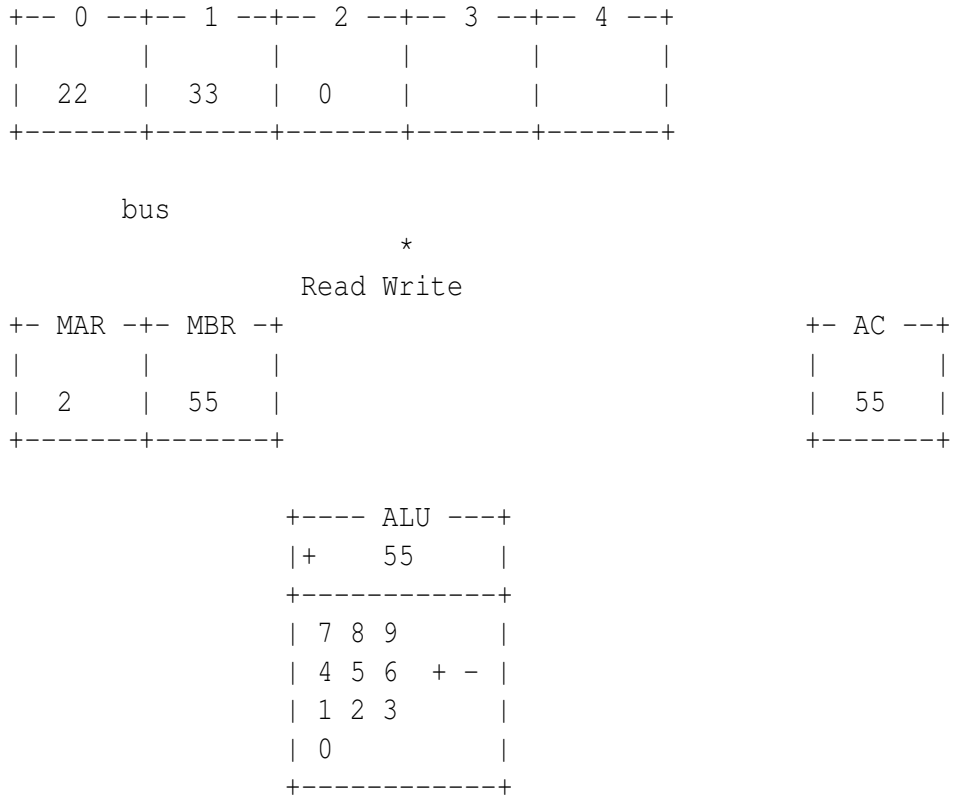


Figure 2.7: Mechanical Computer

Then, some distance away, we have a set of pigeon holes labelled 0, 1, 2, ... This is the memory, each pigeon hole (memory cell) can hold one card capable of holding one number.

There are two people involved: (a) the operator of the CPU; (b) the person who operates as the bus between memory and MAR, MBR.

Figure 2.7 shows the layout.

The person operating the bus has an uninteresting job: when the person operating the CPU shouts 'bus', the bus person does the following (from now on, I use *cell* instead of pigeon hole):

B1. Check whether Read or Write is indicated, it must be one or the other;

B2. If it is Write, (a) take the number in MAR and use it as a label; (b) take the number in MBR, write down a copy of it; (c) walk over to memory and place that copy in the cell labelled with the number indicated by MAR.

Thus, in the case shown, a piece of paper with 55 on it would be put in cell 2.

B3. If it is Read, (a) take the number in MAR (e.g. 1) and use it as a label; (b) walk over to memory, read the number (e.g. 33) in the cell labelled with the number indicated by MAR; (c) write down a copy of that number (33) ; (d) walk back to the CPU desk and place the copy of the number (33) in MBR.

The person operating the CPU has a list of instructions – a program. The following is a set of instructions to do the following: add the contents of memory cell 0 to the contents of memory cell 1, store the result in cell 2; if the result is greater-than-or-equal-to 40, put 1 in cell 3, otherwise put 0 in cell 3. (We are

adding marks, and cell 3 contains an indicator of Pass (1) or Fail (0). I hope you can see the similarity with the program in section 2.6, but I've simplified things for the purposes of this part.

Remember, computers are very stupid, and have no memory other than registers or memory cells. Here is the program:

P1. Load contents of memory 0 into AC.

This entails: (a) write 0 on a piece of paper and place it in MAR; (b) put a tick against Read; (c) shout "Bus"; (d) some time later, the contents of cell 0 (22) will arrive in MBR; (d) write down a copy of what is in MBR and place it in AC.

P2. Add contents of memory 1 to contents of AC.

This entails: (a) write 1 on a piece of paper and place it in MAR; (b) put a tick against Read; (c) shout "Bus"; (d) some time later, the contents of cell 1 (33) will arrive in MBR; (e) look at what is in AC and in MBR, use the calculator to add them (22 + 33); (f) write down a copy of the result and put it in AC. Thus, in the case shown, a piece of paper with 55 on it would be put in to AC.

P3. Store the contents of AC in memory 2.

This entails: (a) write 2 on a piece of paper and place it in MAR; (b) make a copy of what is in AC and place it in MBR; (c) put a tick against Write; (d) shout "Bus"; (e) some time later, the contents of MBR (55) will arrive in memory 2. Thus, in the case shown, a piece of paper with 55 on it would be put in cell 2.

P4. Load the constant 40 into the AC.

This entails: write 40 on a piece of paper and place it in AC.

P5. Store the contents of AC in memory 4.

This entails: see P3 above. The result is that memory 4 will have 40 in it.

P6. Load the contents of memory 2 into AC.

This entails: see P1. above. The result is that AC has 55 in it.

P7. Subtract contents of memory 4 from contents of AC.

This entails: (a) write 4 on a piece of paper and place it in MAR; (b) put a tick against Read; (c) shout "Bus"; (d) some time later, the contents of cell 4 (40) will arrive in MBR; (e) look at what is in AC (55) and in MBR (40), use the calculator to subtract them (55 - 40); (f) write down a copy of the result and put it in AC. Thus, in the case shown, a piece of paper with 15 on it would be put in to AC.

P8. Look at the contents of AC; if it is positive (we'll aggree that 0 is positive) jump to instruction P12. (Otherwise go on to the next step.)

P9. Load the constant 0 into the AC.

This entails: see P4. above.

P10. Store the contents of AC in memory 3.

This entails: see P3 above. The result is that memory 3 will have 0 in it.

P11. Jump to P14.

P12. Load the constant 1 into the AC.

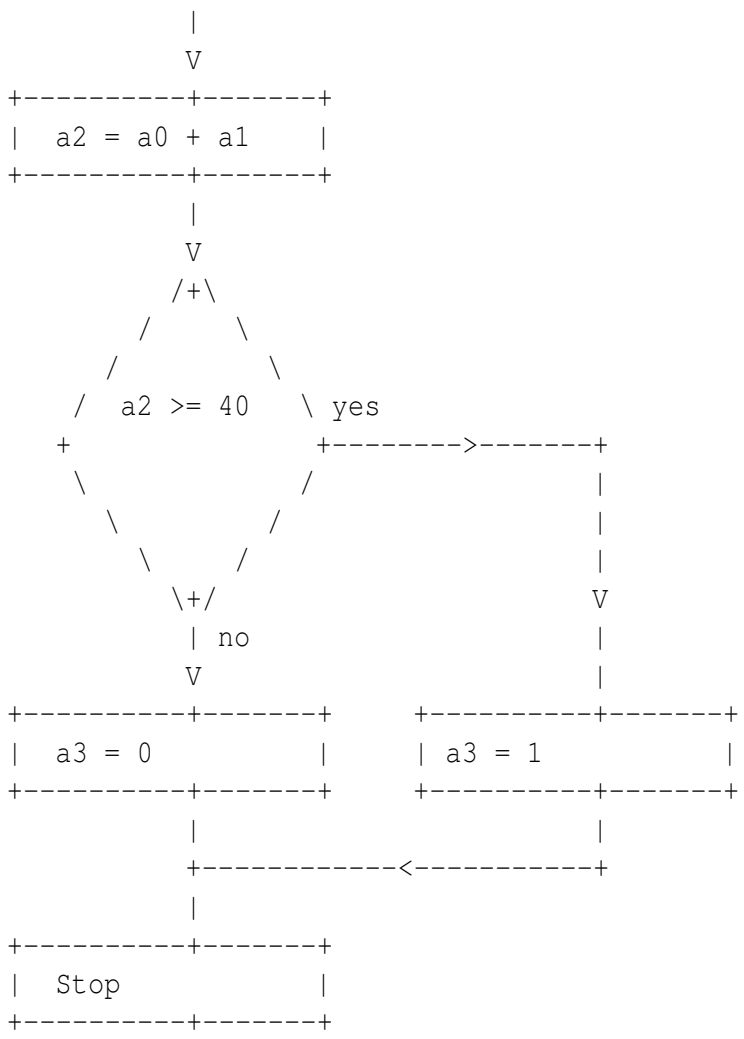


Figure 2.8: Program Flowchart

P13. Store the contents of AC in memory 3.

P14. Stop.

And that's it! A lot of work you may think; but that's how a computer must be programmed – everything down to the finest detail must be explicitly stated.

We could express our little program if a language like C (or Java) as:

```
a2 = a0 + a1;
if(a2 >= 40) a3 = 1;
else a3 = 0;
```

or as in the flowchart given in Figure 2.8.

I have an admission – that's only half the story! The program is also stored in memory, coded as numbers; and every time we step onto a new instruction, we have first to get it (fetch it) from memory. But we'll come back to that in Chapter 6.

2.8 Virtual Machines, Abstraction, Levels of Concern

Here, and in other modules, you will frequently encounter the term *abstraction*, most often in the context of an antidote to *complexity*. It crops up under a variety of guises: separation of concerns, structured levels, modular this and that, information hiding, modelling.

A dictionary definition of abstraction (the verb) is: the process of considering (extracting) certain essential properties of objects while omitting other properties that are not relevant (usually inessential details); the act of abstraction or abstracting produces an abstraction (noun) which collectively refers to the set of chosen properties.

From our point of view abstraction does have two important facets which are related but different, namely, extraction (of the essential from the inessential) and idealisation (i.e. the irrelevant details are deviations from the ideal). Often, somewhat loosely, an abstraction may be called a model.

The key concept here is probably that of a *black-box* abstraction; a black-box model simplifies a system down to:

- what it expects as inputs;
- what it produces as outputs; and,
- the relationship between the inputs and the outputs.

2.8.1 Multilevel View of Computing Machines

It is impossible and undesirable for engineers or programmers to carry in their minds the hundreds of thousands of transistors, resistors, connections . . . , and the millions of software instructions that make up a computer system. Thus it is necessary to decompose (break-up) the system into chunks or sub-systems. But we have to be methodical about this: if you break a 2,000,000 component computer into 200,000 sub-systems, you are not much further ahead — where next?

A computer can be viewed as a hierarchy of levels as shown below. At any one time you make up your mind what level concerns you.

Level 5 Application Level Here the computer is viewed as a machine which performs a specific function, e.g. a spreadsheet, a word-processor; the user at this level shouldn't have to be concerned with any details of computer science or engineering, just as the average car driver need not be concerned with the internal details of car engines.

Level 4 High-level Language Level Here the computer is viewed as a (virtual) machine which obeys the instructions or understands the statements of a high-level language such as C, C++, Java, COBOL, etc. A compiler or interpreter converts these instructions into lower level machine or assembly code.

Level 3 Operating System Machine Level At this level, we have operating system software in charge of executing the machine language (see level 2). In some contexts, this level is absent.

Level 2 Machine Language Level Here are the same sort of instructions that we described for the previous level: major difference, they are now coded as numbers, rather than alphanumeric symbols (e.g. 2501 instead of (say) ADDD x). Two points: (a) they are ready for execution in the processor – the Level 1 machine 'understands' them; and, (b) human programmers have great difficulty in understanding them!

Level 1 Microprogram Level This level is concerned with the (underlying) engine that implements individual Level 2 instructions by appropriately directing the the components and their interconnections at that level. In some (mostly earlier, but also some modern RISC machines) machines this level does not exist and machine instructions are implemented directly by digital logic.

Level 0 Digital Logic Level Here we are at the level of gates and memory cells. The language is made up of 1s and 0s or, at best, binary numbers.

There are certainly levels below level 0 – transistors, p-n junctions, molecules, atoms, . . . !), but the six above should be enough for us except for brief mention of how to make digital logic components out of transistors.

At each level, the user, programmer, engineer, or physicist must pay attention to different sorts of object or quantity. At level 4 (high-level language,), the programmer deals with the data types provided by the language. At level 2 you are forced to deal with physical storage locations, registers and the like. And so on, until at level 0 you are concerned with bits (1s and 0s).

The major point is that, the higher the level of abstraction, the more powerful the machine becomes. For example, you want to type a letter on a word-processor. At level 5 you just do it. At level 4 you first have to write a word-processor program – allocate a few years or so for that!

And so on. Down to level 0, where you first have to make a computer from thousands of chips, then write the microcode, . . . , then write the compiler, the write the word-processor, . . . , then type the letter.

The difference between working at level 4 (say) and level 2 is like discovering that you can take 100 metre strides, instead of the 1.5 meters of ordinary mortals.

2.9 Operating Systems

These days, if you are buying a computer system, operating system software is just as important as the hardware.

An Operating System is a Program An *operating system* is a program. When you switch on a computer (a computer system), the hardware ensures that the operating system is read into the memory (a process called *booting* – *bootstrapping*) and that control is transferred to the OS (i.e. the OS starts running).

The Operating System Executes all the Time — well, sort of I am tempted to say that, from then on, the operating system (program) runs all the time until the the computer is switched off. However, we all know that only one program can be running on a single processor at any time; and most computers operate with just one processor. What about the applications we want to run — our word processor, or spreadsheet, our games . . .

The Operating System is in charge of the computer We can revise that statement to say that the *operating system* is *in charge of the computer all of the time*. To show how a simple program (the operating system) can be *in charge*, or manages, the computer, yet relinquish the processor to other programs, is one of the major objectives of this course. Note: I use the term *in charge* because I want to reserve the term *control*; when we say *transfer of control* (to a program), we mean that the instructions of this program beging to execute on the processor – and the operating system (and other programs), for the moment, slide into the background.

The Operating System as a platform for executing software Another useful way of thinking about the operating system is that it provides a *platform* or environment (a *virtual machine*) upon which to run your application software and provides an interface with which *you* the user interact with the machine. In this view, the operating system provides a *layer* that separates the application software and the user from the hardware.

The Operating System provides services When an operating system is viewed as a platform for executing software (and for interfacing with humans), we can see that it *provides services*. For example: reading from, and saving to disks; sending data to a printer, etc.

The Operating System as police or government When modern operating systems (Windows 2000, XP, UNIX, Linux) go to the trouble of *providing services* they also *force* application programs to avail of these services. For example, unlike on MS-DOS and early versions of Windows (3.1 and to an extent 95 and 98), application programs are barred from directly accessing hardware like disks, network cards, etc., they must use the appropriate operating system service. Since accessing hardware directly is difficult and some application programs are written by less skilled programmers, directly accessing hardware was a common source of errors and frequently resulted in the systems crashing.

Operating systems that adopt this more modern approach are therefore more reliable — there are fewer occurrences of *the blue screen of death*!

In this sense, the operating system can be seen as a police force or government. And just like governments require you to pay taxes (for the services they claim to provide) the operating system demands a certain amount of CPU time for itself.

2.10 Self assessment questions

These are also *recall* type questions that appear as parts of examination questions.

1. Figure 2.2 depicts a computer system. Briefly describe:

   (a) the purpose of an ALU; (b) the purpose of a bus (generally); (c) memory (give a simplified description); (d) a register.

2. Figure 2.5) depicts a Central Processing Unit. Briefly describe:

   (a) the purpose of an ALU; (b) the role of the MAR and MBR in CPU-memory interaction; (c) in addition to address, data, what additional information needs to be transferred via the *system bus* between CPU and Memory; (d) the AC register.

3. The computer in Figure 2.5 is to compute the equivalent of the high-level language statement:
   `z = x + y;`

   (a) What does this statement mean? (b) Considering just data, describe what data must flow to and from memory. (c) Considering just program instructions, describe what *data* must flow to and from memory.

4. The following list (see section 2.8) identifies six levels of abstraction for computer systems.

   Identify, with discussion, the appropriate level of abstraction (view) for:

   (a) an accountant; (b) a programmer who writes operating systems; (c) an engineer who designs CPU chips; (d) and engineer who designs memory chips; (e) a programmer who writes payroll software; (f) a programmer who writes i-o device software; (g) a programmer who writes compiler software; (h) an engineer/physicist who is developing next generation *light-based* computing machinery.

5. Why might you choose to program in assembly language as opposed to a high-level language such as VisualBasic or Java?

6. Briefly, state *what an operating system does*.

   General point: a three hour exam. lasts 180 minutes; let's cut that back to 150 minutes to account for reading questions, checking at the end, revising answers ... Therefore, 1.5 minutes per mark.

   A question like this could be answered by giving five significant points very briefly; or, two/three points with more substantial explanation.

   For five marks, in terms of writing, we probably require no more than five or so sentences – where each sentence expresses a single relevavt idea/point.

Chapter 3

Computer Number Systems and Arithmetic

3.1 Introduction

This chapter describes how numbers and other data are represented, stored and processed in digital computers.

We will see that positive integer numbers are conceptually easy to handle in a computer; if we extend that to include negative numbers, things get more difficult; finally, when you extend to the real number system a further complexity is introduced and you need to use floating point representation. However, we do not need to cover floating point in this course.

We will also see how other entities are stored in computers – like text characters; essentially, computers can store only numbers, so anything else must be coded as a number; and, for these codes to be any use at all, the coding scheme must be agreed by those who wish to exchange information – within the same computer and across different computers.

3.2 Real Numbers versus Integer Numbers

First of all we have the *natural* numbers $\{0, 1, 2, \ldots\}$; these are the numbers we use when we count.

Integers include the natural numbers together with their negatives, $\{\ldots, -2, -1, 0, 1, 2, \ldots\}$.

Real numbers are an entirely different matter. When we measure things, e.g. the weight of a piece of cheese, the length of a piece of string, we have a *real* number. You might say, the weight of this piece of cheese is 25 grams. Not much difference from a natural number, I hear you say. But, almost certainly, in coming up with the 25, you merely took the nearest natural number. If you were in a laboratory, using a very accurate instrument, you might have originally had 25.4124359267 grams. At home you might have got 25.4 grams. Which is correct, 25, 25.4, 25.4124359267 grams? Actually, none of them. If you wanted to be fully correct, you'd have to use thousands and millions of digits; to be exact, infinity of them. Real numbers form a continuum. Between any two real numbers, no matter how close, there are an infinite number of other real numbers. The more precision you use in your measurement, the more digits you get.

In contrast, between 3 and 6, there are just two other integer numbers, 4 and 5. You cannot be any more precise than counting the people in a classroom and stating, for example, there are 23 students present.

It is my intention to spend considerable time on this in class – don't allow me to pass on until you understand it.

3.3 Representation of Values in Finite Data Word Sizes

[Here we use the term *word* for the contents of a location or a register; a word is usually a collection one or more bytes, interpreted as a unit.]

Because of finite data word size, when we represent numbers on a digital computer, we normally need to approximate. It should be clear that approximation of the natural and integer numbers is not such a big problem; you just cut off at some sufficiently large positive and negative numbers, say $-N_m$ and $+N_p$ to replace $-\infty$ and $+\infty$ as the bounds.

Rational numbers could be handled similarly – as a pair of integers (numerator, denominator) – but this representation is rarely ever used for general purpose arithmetic.

Real numbers are the big problem – to represent them completely requires an infinity of digits. Note: between any two real numbers, no matter how close, there is an infinite set real numbers – the reals form a continuum. However, in a digital computer they must be represented by selected examples – approximations; thus, on a very limited system, we might have 2.5, 2.55, 2.3, ... and in such a system, 2.512345... would be approximated by 2.5.

In summary:

1. Integers are easy to simulate. Normal integer arithmetic and computer integer arithmetic are effectively the same. Handling a sign introduces some extra complexity.
2. Reals are more difficult to simulate. You have to be careful with the computer approximations of real arithmetic; even the best approximation – floating point numbers and floating point arithmetic – are inherently inaccurate.

For an encyclopedic account of this subject see (Knuth 1997).

3.4 Addition, Multiplication, Exponentiation (Powers)

3.4.1 Multiplication

Multiplication is shorthand for lots of additions of the same thing. Let's say we have $2+2+2$, we know that is 6, or 3×2 (three *times* 2, i.e. two three times).

If we go to algebra, we can have $a+a+a$, we have $3 \times a$ (three *times* a, i.e. a three times). Let me amend this a little to $0+a+a+a = 3 \times a$.

In general, starting out with 0, we repeatedly add a, b times, $0+a+a+\ldots+a$, b of them, we get $0+a+a+\ldots+a = b \times a$.

Handy shorthand — that's what much of mathematics is about.

3.4.2 Exponentiation (Powers)

Exponentiation (powers) is shorthand for lots of multiplications of the same thing. Let's say we have $2 \times 2 = 4 = 2^2$, two squared, or two to the power of two.

$2 \times 2 \times 2 = 8 = 2^3$, two cubed, or two to the power of three.

Like I included the zero in the addition-multiplication, I want to include 1 in the story: $1 \times 2 \times 2 \times 2 = 8 = 2^3$

Now go to algebra. $1 \times x \times x \times x = x^3$ – just shorthand, and very useful if you are multiplying by x loads of times such that you would lose count writing down all the $\times x$ terms. And generalising further, $1 \times x \times x \times x$ (p of them) $= x^p$.

Examples. $1 \times 2 \times 2 \times 2 = 2^3 = 8$. $1 \times 2 \times 2 \times 2 \times 2 = 2^4 = 16$. $1 \times 10 \times 10 \times 10 = 10^3 = 1000$.

Let us return for a moment to the addition and multiplication. What happens if we subtract an a? $0+a+a+a-a = (3-1) \times a = 2 \times a$. And if we subtract three of them, we get $0+a+a+a-a-a-a = (3-3) \times a = 0$; we get back to the zero.

What is the equivalent of addition's subtraction for multiplication, i.e. what cancels out one multiplication? The answer is *division*.

$\frac{1 \times 10 \times 10 \times 10}{1 \times 10}$ (three on the top, and one on the bottom) $= 10^{3-1} = 100$.

Generalising, $\frac{1 \times x \times x \times x}{1 \times x} = x^{3-1} = x^2$.

What if we have $\frac{1 \times x \times x \times x}{1 \times x \times x \times x}$ and end up with $x^{3-3} = x^0$? Look at it another way for a moment, the three $\times x$ on the top cancel the three on the bottom (do it for $x = 10$ or $x = 2$ if you don't believe me), and we have $\frac{1}{1} = 1 = x^0$.

Anything, other than zero (0) itself, raised to a zero power equals 1; i.e. $x^0 = 1$, for $x \neq 0$, $0^0 = 0$.

Now you can see why I included the $1\times$; from now on we can leave it out (I hope you are comfortable with the fact that $1 \times$ anything = anything).

Let's generalise further, $\frac{1 \times x \times x \ldots}{1 \times x \times x \ldots} = x^{p-q}$, if we have p multiplies on top (numerator), and q on the bottom (denominator).

Therefore, $\frac{1}{1 \times x^y} = x^{-y}$. Make sure you understand this – i.e. work at it yourself, and get me to go through it in class until you do.

And, $\frac{1}{1 \times x^{-z}} = x^z$.

As I mentioned before, we often leave out the leading $1\times$, e.g. $\frac{1}{x^{-z}} = x^z$.

Example. If we have frequency, f, = 1 Megahertz = 1 million cycles per second, and we want to find the time (period), T, of once cycle, we have:

$T = \frac{1}{f} = \frac{1}{1 \times 10^6} = 10^{-6}$ seconds, which is 1 microsecond.

3.5 Symbolic Representations of Numbers – Bases

3.5.1 General

Digital computers use binary (base 2) as their 'native' representation; however, in addition to binary, and decimal, it is also useful to discuss hexadecimal (base 16) and octal (base 8) representations. Our everyday representation of numbers is *decimal*; it uses the 10 symbols $\{0,1,2,3,4,5,6,7,8,9\}$ – *digits*. 10 is its *base* or *radix*.

The decimal number system is a **positional** system: the meaning of any digit is determined by its position in the string of digits that represent the number. Thus:

$$\begin{aligned} 123 &= 1\times 10^2 + 2\times 10^1 + 3\times 10^0 \\ &= 1\times 100 + 2\times 10 + 3\times 1 \\ &= 100 + 20 + 3 \end{aligned}$$

In general, given a base b, and number of digits p, the number is written: $d_{p-1}d_{p-2}\ldots d_2 d_1 d_0$, and,its value is:

$$value = d_{p-1}\times b^{p-1} + d_{p-2}\times b^{p-2} + \ldots d_2\times b^2 + d_1\times b^1 + d_0\times b^0$$

The leftmost, d_{p-1}, is the **most significant digit**, d_0 , the rightmost is the **least significant digit**.

3.5.2 Binary Representation

Base 2 requires just two symbols, $\{0,1\}$, but the pattern remains the same as for decimal. The digits are termed **bi**nary digi**t**s – **bit**s.

Example. [Subscript: x_{10} signifies that x is decimal, y_2 signifies binary].

$$\begin{aligned} 101_2 &= 1\times 2^2 + 0\times 2^1 + 1\times 2^0 \\ &= 4 + 0 + 1 \\ &= 5_{10} \end{aligned}$$

3.5.3 Conversion – Binary-Decimal, Decimal-Binary

The example of $101_2 \rightarrow 5_{10}$, above, shows how it is done: simply add the powers of two corresponding to digits

Example. Convert 22_{10} to binary.

Method 1: successively divide by 2 until you get to 0; the remainders (r), read back-wards, are the binary digits.

$$\begin{aligned}
\frac{22}{2} &= 11\ r\ 0 \leftarrow \text{least significant digit} \\
\frac{11}{2} &= 5\ r\ 1 \\
\frac{5}{2} &= 2\ r\ 1 \\
\frac{2}{2} &= 1\ r\ 0 \\
\frac{1}{2} &= 0\ r\ 1 \leftarrow \text{most significant digit}
\end{aligned}$$

$$22_{10} = 10110_2.$$

Clearly, this will work for any base, and, actually, is the method that you would use to extract the digits (base 10, or any base) from a number to enable printing.

To check, convert back to decimal:

$$\begin{aligned}
10110_2 &= 1\times 16 + 0\times 8 + 1\times 4 + 1\times 2 + 0\times 1 \\
&= 16 + 0 + 4 + 2 + 0 \\
&= 22
\end{aligned}$$

Method 2: Same principle as method 1, but easier to do in your head. First you must remember powers of 2: 1, 2, 4, 8, 16, 32, 64, 128, ... Then:

1. Find power of two just below the number (22), i.e. 16 $16 = 2^4$, so, bit 4 is set;
2. Subtract 16, to get 6;
3. Loop back with 6 as number; repeat until done.

3.5.4 Hexadecimal

Hexadecimal is base 16: the sixteen symbols used are $\{0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F\}$. A table of the values is shown in Table 3.1. For completeness, we also give binary and octal (base 8).

To distinguish them from decimal, hexadecimal numbers are often written with a `0x` prefix. Thus, decimal `15` is `0xF` or `0xf`; case is not significant.

When dealing with computers at a low-level it is almost essential to be able to work with hexadecimal ("hex") or octal; hex is more common now, but it may be difficult to memorise the symbol, number, bit-pattern table above. If you need to use hex in an examination or otherwise, make up the table on a sheet of paper – it takes only a few seconds.

Hexadecimal representation effectively translates groups of four bits into $0 \ldots F$, whilst an octal digit translates three bits into $0 \ldots 7$.

3.5.5 Conversion – Decimal-Hexadecimal

Example. Convert 24_{10} to hex.

Dec	Bin	Oct	Hex
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

Table 3.1: Decimal Binary, Octal, Hexadecimal

Method 1 Successively divide by 16 until you get to 0; the remainders (r), read backwards, are the hex digits.

$$\frac{24}{16} = 1\ r\ 8 \leftarrow \text{least significant digit}$$
$$\frac{1}{16} = 0\ r\ 1 \leftarrow \text{most significant digit}$$

$$24_{10} = 18_{16}, \textit{i.e.}\ \texttt{0x18}.$$

To check, convert back to decimal:

$$\begin{aligned} 18_{16} &= 1 \times 16^1 + 8 \times 16^0 \\ &= 1 \times 16 + 8 \times 1 \\ &= 16 + 8 \\ &= 24 \end{aligned}$$

Method 2 Convert 22_{10} to hex. First, convert this to binary, i.e. 11000_2. Then, starting from the right (least significant digit) divide into groups of four; then use Table 3.1.

$$\begin{array}{c} 1\ 1000 \\ 1\ 8 \end{array}$$

$11000_2 = 18_{16}$

3.5.6 Octal

Example. Convert 25_{10} to octal. Ans. binary: 11001_2; group into threes starting from right: 11 001. This is 31_8.

Q. Why do real programmers (who, at least in the old days, always worked in octal) confuse Halloween and Christmas? A. Because Dec 25 = Oct 31!

3.6 Binary Arithmetic

3.6.1 Addition

Assuming you do decimal addition as follows: $(19+22)$ $2+9=11$ i.e. 1 and carry 1; $2+1=3+$ carry $1=4$; Answer: 41.

$$\begin{array}{r} 19 \\ +22 \\ =41 \end{array}$$

Binary addition (and other bases too) follows the same pattern – the addition of single digits, and the carry rule simply need to be amended. Thus:

$$\begin{array}{r} 101 \\ +001 \\ =110 \end{array}$$

$1+1=10$ i.e. 0 and carry 1; $0+0=0+$ carry $1=1$, $0+1=1$ (no carry this time).

3.6.2 Hexadecimal addition

Example. $1e_{16}+25_{16}$ $(30_{10}+37_{10}=67_{10})$

$$\begin{array}{r} 1e \\ +25 \\ =43 \end{array}$$

$5+e=3$ and carry 1; $2+1=3+$ carried $1=4$; Check: $43_{16}=64+3=67$.

3.7 Representation of Sign in Binary Numbers

The three primary methods used to represent signed binary numbers are: sign-magnitude, twos-complement, and biased (also called excess). Of these, twos-complement is by far the most common.

3.7.1 Sign-magnitude

In sign-magnitude, one bit is used for sign, typically $1 = -$, $0 = +$; usually the most significant bit is sign, and and the low-order $p-1$ bits are the magnitude of the number – where we have a total of p bits.

3.7.2 Biased

In a biased system, a fixed bias is added – such that the sum of the bias and the number being represented will always be non-negative; thus, what would normally be interpreted as 0 represents the most negative number.

3.7.3 Twos-complement

Twos-complement makes adding and subtracting particularly easy to implement; in a twos-complement system, negating a number is done by taking its twos-complement. Computing the twos complement of a binary number involves two steps:

1. Complement each bit, i.e. ones complement; for example, in a four bit number: $0011 \rightarrow 1100$;
2. Add 1, $1100 \rightarrow 1101$.

Thus, $0011_2 = 3_{10} \rightarrow 1101$.

Example. In a four bit number system, represent -3 in each of the formats.

- The number itself: +3 in unsigned, straight binary, is 0011
- Sign-magnitude: $-0011 \rightarrow 1011$.
- Twos-complement: $0011 \rightarrow 1100 \rightarrow 1101$.
- Biased: (bias of 8): $3+8 \rightarrow 11_{10} = 1011_2$.

Table 3.2 shows a comparison of the three methods of representing signed numbers.

Note:

1. (see * in Table 3.2) In twos-complement and biased, there is only one zero!
2. In both biased and twos-complement there is one more negative number than positive.
3. In a p-bit number, the set of possible twos-complement numbers is $\{-2^{p-1}, \ldots 2^{p-1}-1\}$.

Value	Sign-mag.	Twos-comp.	Biased (8)
-8	not poss.	1000	0000
-7	1111	1001	0001
-6	1110	1010	0010
-5	1101	1011	0011
-4	1100	1100	0100
-3	1011	1101	0101
-2	1010	1110	0110
-1	1001	1111	0111
-0	1000	*	*
0	0000	0000	1000
+1	0001	0001	1001
+2	0010	0010	1010
+3	0011	0011	1011
+4	0100	0100	1100
+5	0101	0101	1101
+6	0110	0110	1110
+7	0111	0111	1111

Table 3.2: Comparison of signed representations (four bits)

3.7.4 Twos-complement arithmetic

Addition

Addition is straightforward: simply add and *discard* any carry out from the most significant bit.

Example. $2 + (-3) = -1$

$$\begin{array}{rr} & 0010 \\ + & 1101 \\ 0 & 1111 \end{array}$$

Example. $3 + (-3) = 0$

$$\begin{array}{rr} & 0011 \\ + & 1101 \\ 1 & 0000 \end{array}$$

and the 1 carried out of the most significant bit is discarded – note, this does **not** signify *overflow*, see the next subsection.

Subtraction

Subtraction is done as follows: first negate the number to be subtracted, then perform addition, i.e. `z = x - y`: negate `y` to get `y'`, then, `z = x + y'`.

3.7.5 Overflow

Overflow comes about when the result of an operation does not fit the representation being used. It is a common trap for the unwary, especially in programming languages, for example C, C++, various assembly languages, and machine language, which have no built-in checks for it.

Example. Positive overflow. In our four-bit number system, $3 + 6$

$$\begin{array}{r} 0011 \\ 0110 \\ 1001 \end{array}$$

Hmmm! Consulting Table 3.2, we find that $3 + 6 \rightarrow -1$.

Example. Negative overflow. In our four-bit number system, $-3 - 6$

$$\begin{array}{r} 1101 \\ 1010 \\ 1\ 0111 \end{array}$$

As is normal, the 1 carried out of the most significant bit is discarded. Hmmm again! Consulting Table 3.2, we find that $-3 - 6 \rightarrow +7$.

Detection of Overflow For unsigned numbers, detecting overflow is easy, it occurs exactly when there is *carry out of the most significant bit*.

For twos-complement things are trickier; overflow occurs exactly when the *carry-in* to the most significant bit is different from the *carry-out* of that bit.

3.7.6 The twos-complement circle, wrap-around

If you examine Table 3.2, you will see (obviously enough when you think about it) that you can iterate in a positive direction along the twos-complement numbers by simply adding 1. Finally, when you get to $7_{10} = 0111$

$$\begin{array}{r} 0111 \\ 0001 \\ 1000 \end{array}$$

In other words, *most positive number* $+1 =$ *most negative number*; so, we wrap-around.

Exercise. Draw the circle for four bit numbers.

3.8 Binary Numbers — Summary

Let's say we must compute the (decimal) value of the four bit unsigned binary number 1101; here is the quick way:

```
  Power of two (p)   = 3     2    1     0
  Value of bit = 2^p = 2^3  2^2  2^1   2^0
  i.e. value =         8     4    2     1

  Number               1     1    0     1
                       ^     ^    ^     ^ 1 x 1 = 1
                       |     |    |
                       |     |    +-----  0 x 2 = 0
                       |     |
                       |     +----------- 1 x 4 = 4
                       |
                       +----------------- 1 x 8 = 8
                                                --- add
                                                13
```

What about eight bits unsigned? Take 1100 1101.

```
Power of two (p)   = 7     6     5     4    3     2    1    0
Value of bit = 2^p = 2^7  2^6   2^5   2^4  2^3  2^2  2^1   2^0
i.e. value =         128   64    32   16    8     4    2     1

Number                1    1     0     0    1     1    0     1
So:                 128 + 64 +  0  +  0  + 8  +  4  + 0  +  1

which is:                          205
```

The situation is slightly different for twos-complement. Again, we'll look at four bits.

```
  Power of two (p)   = 3*    2    1     0      * -- but a different meaning.
  Value of bit = 2^p = -2^3 2^2  2^1   2^0
  i.e. value =        -8     4    2     1

  Number               1     1    0     1
                       ^     ^    ^     ^ 1 x 1 = 1
                       |     |    |
                       |     |    +-----  0 x 2 = 0
                       |     |
                       |     +----------- 1 x 4 = 4
                       |
                       +----------------- 1 x 8 =-8
                                                --- add
                                                -8 + 5 = -3
```

```
Verify: (a) ones-complement 1101 -> 0010

         (b) add one        0010
                               1
                           ----
                           0011 = 3
```

Eight-bit twos-complement. Again, take 1100 1101.

```
Power of two (p)   =  7*   6     5     4    3     2    1    0
Value of bit = 2^p = -2^7 2^6   2^5   2^4  2^3  2^2  2^1   2^0
i.e. value =        -128   64    32    16    8     4    2     1

Number                1    1     0     0    1     1    0     1
So:                 -128 + 64 +  0  +  0  + 8  +  4  + 0  +  1
which is:           -128 + 77       = - 51
```

which is easily verified by ones-complementing and adding 1.

3.9 Fractions and Floating Point

3.9.1 Introduction

Often, the quantities we want to represent and compute are not integers. Furthermore, the intermediate results in a computation may not be integer. To represent non-integers, the following are required:

1. Representation of the digits of the number – on both sides of the *point*;
2. Representation of the position of the *point*.

In a *fixed point* representation, the point is assumed to be in some fixed position – and, since it is fixed, the position need not be stored. In a *floating point* representation, the position of the point can be varied according to the size of the number. Floating point is by far the most common and useful on general purpose computers.

3.9.2 Fixed Point

To represent fractional numbers, we simply extend the positional system, introduced in section 3.5.1, to include digits corresponding to negative powers. We do this by introducing a *point* '.'; the integer number is as before, and is shown to the left of the point; the fraction part is shown to the right of the point. Therefore, generalising what was given in section 3.5.1, a number which includes a q-bit fractional part, for a total of $p+q$ bits, is written:

$d_{p-1}d_{p-2}\ldots d_2d_1d_0.d_{-1}d_{-2}\ldots d_{-q+1}d_{-q}$

and its value is:

$value = d_{p-1} \times b^{p-1} + d_{p-2} \times b^{p-2} + \ldots d_2 \times b^2 + d_1 \times b^1 + d_0 \times b^0 + d_{-1} \times b^{-1} + d_{-2} \times b^{-2} + \ldots + d_{-q+1} \times b^{-q+1} + d_{-q} \times b^{-q}$

Thus, 123.45_{10}:

$$\begin{aligned} 123.45_{10} &= 1 \times 10^2 + 2 \times 10^1 + 3 \times 10^0 + 4 \times 10^{-1} + 5 \times 10^{-2} \\ &= 1 \times 100 + 2 \times 10 + 3 \times 1 + 4 \times 0.1 + 5 \times 0.01 \\ &= 100 + 20 + 3 + 0.4 + 0.05 \\ &= 123.45 \end{aligned}$$

In binary,

$$\begin{aligned} 0.1 &= 1 \times 2^{-1} = 1 \times 0.5 = 0.5 \\ 0.01 &= 1 \times 2^{-2} = 1 \times 0.25 = 0.25 \\ 0.001 &= 1 \times 2^{-3} = 1 \times 0.125 = 0.125 \\ &\ldots \end{aligned}$$

Example. Convert 11.101_2 to decimal.

$$\begin{aligned} 11.101 &= 1 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} \\ &= 2 + 1 + 0.5 + 0 + 0.125 \\ &= 3.625 \end{aligned}$$

The the magnitude of the number depends on the *fixed* position of the point. The problem arises that sometimes we want most of the digits after the 'point', i.e. small numbers, sometimes we want most of them before the 'point', i.e. large numbers. A single fixed-point representation cannot do both.

Other methods that have been proposed for storage of fractions:

- Store the number as a logarithm – actually, logarithms are peripherally involved in floating point representation;
- Store the number as a pair (a, b) representing a rational number $\frac{a}{b}$.

However, floating point is the universally adopted solution for general purpose computers.

3.9.3 Floating Point

A floating point number is composed of two parts:

1. The digits of the number, the *fraction*; the fraction is often called the *mantissa*, but, strictly, this is an abuse of the term.
2. The position of the point – the *exponent*.

The floating point representation is similar to the 'scientific', or 'exponential' representation of very large and very small numbers on some calculators and in books; e.g. $123.56 = 0.12345\ E\ +03$, i.e. 0.12345×10^3. In this scientific representation, you can think of the exponent as representing the 'point' position.

In fact, the exponent is the integer part of the *logarithm* to which the *base* must be raised in order to place the point in the right position. Examples of decimal floating-point with five digit fraction and three digit exponent are shown in Table 3.3.

fraction	exponent	floating-point	fixed-point
12345	003	0.12345×10^3	123.45
12345	002	0.12345×10^2	12.345
12345	001	0.12345×10^1	1.2345
12345	000	0.12345×10^0	0.12345
12345	−001	0.12345×10^{-1}	0.012345

Table 3.3: Examples of floating-point

3.9.4 Binary Floating Point

[The remainder of this section on floating-point is included just for interest — it is not part of either Computer Systems or Operating Systems I courses.]

We will explain binary floating point using a total of 8 bits (1 for sign, 3 for exponent, 4 for fraction); most practical systems use 32 bits, but the following explains the principles without the confusion of many bits. IEEE 754, the floating point standard used on most computers these days, uses 32 bits for *single precision* and 64 bits for *double precision*.

In our 8-bit system, *flob* (*flo*ating-point *b*yte), the bits are arranged as follows,

$$\underbrace{sign}_{1\ bit}\ \underbrace{exp}_{3bits}\ \underbrace{frac}_{4bits}$$

The fraction (mantissa) is just a straight fraction, i.e. the (virtual) point is at the left hand side of the four bits. The exponent is stored as a signed number, using excess 4 (biased by 4) representation, i.e. biased by 4 – recall Table 3.2. Thus, the bit patterns in the exponent represent values as shown in Table 3.4.

bits	111	110	101	100	011	010	001	000
value	3	2	1	0	-1	-2	-3	*

Table 3.4: 3-bit, excess 4

Of course, you don't have to look up Table 3.4, simply subtract 4 from the stored number.

The value, v, of the stored floating point number is given by eqn. 3.1.

$$v = f \times r^e \tag{3.1}$$

where f is the fraction, r is the radix (base) of the exponent and e is the exponent.

Or, to make the excess in the exponent more explicit,

$$v = f \times r^{e-q} \tag{3.2}$$

where q is the excess.

Normalisation Unless otherwise stated, f is bounded as follows:

$$.1000 \leq f \leq .1111 f \times r^{e-q} \tag{3.3}$$

i.e. the top bit of the fraction is always set.

The most common practical floating point number system (IEEE Standard 754, single precision) uses 8 bits for exponent, 23 for fraction – including 1 for sign; a total of 32 bits. There is a double precision format at 64 bits and extended precision at 80 bits.

Hidden Bit In a normalised number, see eqn 3.3, the top bit is always set; thus, we don't really have to store it – this is a trick that the IEEE standard uses to 'steal' an additional bit for the fraction; they work it so that the top bit is assumed to be just to the left of the 'point'. We will adopt this trick for our floating-point-byte system. Finally we end up with eqn. 3.4, where we show the hidden bit, and make the sign explicit,

$$v = (-1)^s \times 1.f \times r^{e-q} \tag{3.4}$$

Denormalised Numbers In special circumstances, IEEE 754 allows numbers to be stored with a denormalised fraction; for such cases they use exponent 000 which flags the special case. The major advantage is that we can store a 'pure' zero.

Henceforth, we will call this small floating point number system **flob** – **flo**ating-point **b**yte.

Example: Stored representation is 01101011, what is the value?

Sign. $0 \rightarrow +$. Stored exponent $= 110 = 6_{10}$, actual exponent $= +2$. Fraction $= 1011$, i.e. with hidden bit, 1.1011. Therefore,

$$\begin{aligned} v &= 1.1011 \times 2^{+2} \\ &= 110.11 \times 2^0 \text{ (why?)} \\ &= 110.11 \\ &= 4+2+0+0.5+0.25 \\ &= 6.75 \end{aligned}$$

Example: Stored representation is 10111100, what is the value?

Sign. $1 \rightarrow -$. Stored exponent $= 011 = 3_{10}$, actual exponent $= -1$. Fraction $= 1100$, i.e. with hidden bit, 1.1100. Therefore,

$$\begin{aligned} v &= -1.1100 \times 2^{-1} \\ &= -.11100 \times 2^0 \\ &= -.11100 \\ &= -(0.5+0.25+0.125) \\ &= -0.875 \end{aligned}$$

None: In the following we will assume that the **only** allowed denormalised number is 0; i.e. $0000000 \rightarrow 0$; there is no other allowable case where stored exponent can be 000.

3.9.5 Exercises

1. (a) What is the largest positive number that can be represented using our 'flob' number system? Ans. 15.5. Explain.
   (b) Largest negative number?
   (c) Smallest positive number? Ans. 0.125. Explain.
   (d) Smallest negative number?

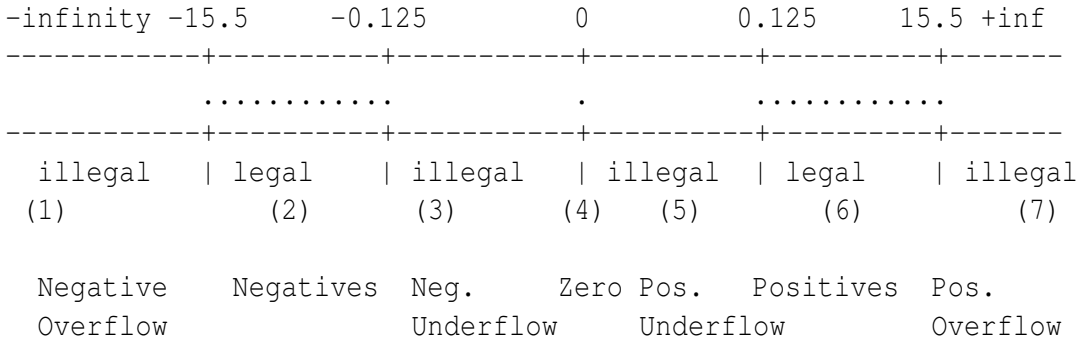


Figure 3.1: The Seven Regions of The Real Number Line.

(e) Smallest number in magnitude?

2. (a) Explain how many individual numbers can be represented by flob? Hint 1: how many positive, how many negative, and don't forget zero. Hint 2: for the positives (say), how many different fraction patterns, how many exponent patterns; multiply to get the total. Ans. 225. Explain.

   (b) Normally a byte can represent 256 different values; explain how we have lost the remaining (256-225) = 31 values?

3.9.6 Rounding and Truncation

As mentioned in the introduction, section 3.3, the real numbers form a continuum. However, instead of infinity of them, exactly 225 can be represented in flob. In flob there are three legal regions on the real line, and four illegal, see Figure 3.1.

The dots `(...)` in Figure 3.1 are meant to show that, even within the legal regions, we do not have a continuum, but a discrete set of numbers. So, in reality, there are many other *illegal* regions, besides those mentioned above.

For actual numbers, we must compromise and represent them with a close legal number; this entails *rounding* or *truncation*.

Rounding Rounding is probably the best – it takes the closest; e.g. in a one digit representation, $0.34 \rightarrow 0.3$, $0.36 \rightarrow 0.4$; therefore, the maximum error $= 0.05$, i.e. half the step between legal numbers (0.1).

Truncation Truncation chops off the insignificant digits, e.g. $0.39 \rightarrow 0.3$; in this case, the maximum error is 0.09.

3.9.7 Normalisation

Normalisation is introduced to make maximum use of the fraction, i.e. ensure that the leftmost digit is non-zero. Thus, as mentioned above, $1.0 \leq f \leq 1.1111$.

The normalisation process is simple: shift the fraction one digit left at a time, each time decreasing the exponent by 1; when the left-most digit is non-zero, normalisation is complete.

Example: 0.0010×2^0.

$fraction = 0\ 0010$, $exponent = 000$; we show the hidden bit.

The fraction must be shifted left 3 places so the exponent has 3 subtracted from it. The normalised value is 1.0000×2^{-3}. Therefore, we have,

$fraction = 1\ 0000$, $exponent = 001\ (-3)$ and this would be stored as 0 001 0000 (N.B. the top bit is hidden).

3.9.8 Conversion to Floating Point

Example. Convert 9 to 'flob', and express in hexadecimal format.

1. $Sign = 0$.
2. Convert fraction to binary: 1001.
3. Normalise, i.e. shift the 'point' left until you have $1.xxx$,

   1.0010×2^3 – point shifted 3 left = divide by 2^3, therefore we need to add 3 to exponent – which starts off at 0.
4. Discard the (hidden) '1' – before the point.

   $.0010 \times 2^3$.
5. Convert the exponent to excess representation (excess = bias = 4); i.e. $stored\ exponent = actual\ exponent + 4$.

   $3 + 4 = 7 = 111_2$
6. Assemble full representation,

$$\overbrace{0}^{s}\ \overbrace{111}^{exp}\ \overbrace{0010}^{frac}$$

7. Group into fours for conversion to Hexadecimal, 0111 0010
8. Convert to Hex.: 72.

Example. Convert $-0.625 = -\frac{5}{8}$ to flob, and express in hexadecimal format.

1. $Sign = 1$.
2. Convert fraction to binary: .1010.
3. Normalise, i.e. shift the 'point' left until you have $1.xxx$,

   1.0100×2^{-1} – point shifted 1 right = divide by 2^1 = multiply by 2^{-1}, therefore we need to subtract 1 from exponent – which starts off at 0.

4. Discard the (hidden) '1' – before the point.

   $.0100 \times 2^3$.

5. Convert the exponent to excess representation (excess = bias = 4); i.e. $stored\ exponent = actual\ exponent + 4$.

   $-1+4 = 3 = 011_2$

6. Assemble full representation,

$$\overbrace{1}^{s}\ \overbrace{011}^{exp}\ \overbrace{0100}^{frac}$$

7. Group into fours for conversion to Hexadecimal, 10110100

8. Convert to Hex.: $B4$.

Example. $0.625 = \frac{5}{8}$

See above, same except: $sign \rightarrow 0$

$0011\ 0100 = 34_{16}$; normally write as 23H.

Example. 0 – exactly zero.

$0000\ 0000 = 00H$. Zero is a special case, see above, which is not normalised.

Example. Convert $00H$ from flob representations to decimal value.

You just have to remember that this is zero see above.

Example. Convert $35H$ from flob representations to decimal value.

1. Express in binary: 0011 0101.

2. Pick off sign, bit 7: $0 \rightarrow +$.

3. Extract stored exponent (bits 6,5,4): $011 = 3$.

4. Compute $actual\ exponent = stored\ exponent - 4 = 3 - 4 = -1$.

5. Extract fraction: 0101 $(.0101)$.

6. Now add in 1 before the point, i.e. the one that was 'hidden': 1.0101.

7. Express as 'fixed point', i.e. make the exponent 0 – reverse the normalisation process: $1.0101 \times 2^{-1} = 0.10101 \times 2^0$.

8. Convert to decimal: $0 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} + 0 \times 2^{-4} + 1 \times 2^{-5} = 0.5 + .125 + .03125 = 0.65625$

3.9.9 Addition and Subtraction in Floating Point

Here, we merely show the principle; and, to avoid some detail we use a decimal format.

Example. $2 \times 10^6 + 13 \times 10^3 = 2.013 \times 10^6 = 2013 \times 10^3 = 2013000$.

The general method for addition and subtraction, $(A +/- B)$ is:

1. Line up points by making exponents equal:
   (a) Identify the operand with the smaller exponent, say A, exponent e_A;
   (b) **Do**. Successively shift A's fraction right while increasing its exponent;
   (c) **until** $e_A = e_B$.
2. Add (or subtract) the fractions.
3. Normalise the result.

Example. $0.28 \times 10^1 + 0.12 \times 10^0$.

$$\begin{aligned} & 0.280 \times 10^1 \\ & +0.012 \times 10^1 \\ & = 0.292 \times 10^1. \end{aligned}$$

Example. $0.28 \times 10^2 - 0.24 \times 10^2$.

$$\begin{aligned} & 0.280 \times 10^2 \\ & -0.240 \times 10^2 \\ & = 0.040 \times 10^2 \\ & = 0.400 \times 10^1 \quad \text{after normalisation.} \end{aligned}$$

3.9.10 Multiplication and Division

These are easier than add and subtract.

Multiplication

1. Multiply the fractions;
2. Add the exponents;
3. Normalise the result.

Division

1. Divide the fractions;
2. Subtract the exponents;
3. Normalise the result.

$$\begin{aligned} x &= f_x \times r^{e_x} \\ y &= f_y \times r^{e_y} \end{aligned}$$

$$x \times y = (f_x \times f_y) \times r^{(e_x + e_y)}$$

$$\frac{x}{y} = \frac{f_x}{f_y} \times r^{(e1-e2)}$$

Example. $0.120 \times 10^1 \times 0.253 \times 10^2 = 1.2 \times 25.3$.

$$\begin{aligned} 0.120 \times 10^1 \times 0.253 \times 10^2 &= 0.030360 \times 10^3, \\ &= 0.030 \times 10^3 \quad \text{retaining only 3 significant digits,} \\ &= 0.300 \times 10^2 \quad \text{normalised.} \end{aligned}$$

3.9.11 Fixed versus Floating Point

Speed: where there is no dedicated floating-point hardware, fixed – point arithmetic can be performed faster and easier, see below;

Ease of Implementation: fixed point may be easier to implement easier; in the past, many computers – like the one we encounter in later chapters – have only integer arithmetic units, and so must use software (using algorithms like those shown above) to do floating point. Up to the Intel 80386 series – great-great-grand-daddy of the Pentium III – you had to have a special purpose floating-point co-processor (the 80387) for Intel XX86 microprocessors; modern digital-signal-processors (DSPs) may not have any floating-point;

Ease of programming: generally, floating point saves the programmer any worry about over/under-flow;

Range: fixed point limits the range of the numbers being represented. Floating point gives greater **range** – at the cost of resolution; see below for definitions of range and resolution.

3.9.12 Accuracy

Strictly speaking, **accuracy** is not a feature of the number system. Accuracy is determined by the error inherent in the production of the number, for example, the error in reading a metre. Thus, 12.34 ± 0.01; this indicates that the actual number could be anywhere between 12.33 and 12.35. It might be possible to include better *resolution* e.g. express it as 12.34567, but, due to the *accuracy* limitation, the 567 would be meaningless.

3.9.13 Resolution

Resolution is measured by δv – signifying *value difference*. δv is the difference in values represented by neighbouring states. In floating-point systems, δv varies according to the magnitude of the number (v); in fixed-point systems δv remains constant across the range of v.

Example. Consider 01000111 in flob. $v_1 = 1.1000_2 \times 2^3 = (1.5 \times 8)_{10} = 12.0$. Next value up is $v_2 = 1.1001 \times 2^3 = (1.5625 \times 8)_{10} = 12.5$. Hence, $\delta v = v_2 - v_1 = 0.5$.

At $v_1 = 1.1000 \times 2^{-3} = 1.5 \times \frac{1}{8} = 0.1875$. Next value up is $v_2 = 1.1001 \times 2^{-3} = 1.5625 \times \frac{1}{8} = 0.1953125$. Hence, $\delta v = v_2 - v_1 = 0.0078125 (= \frac{1}{128})$.

If you used the whole eight bits for fixed-point then δv would be constant at $2^{-7} = \frac{1}{128} = 0.0078125$ over the whole range.

3.9.14 Range

Range or **dynamic range** is the difference between the largest positive number and the largest negative number. In 'flob' this is $15.5 - -15.5 = 31.0$, see above.

3.9.15 Precision

Precision is defined as the ratio of **resolution** to **value**, i.e. $\frac{\delta v}{v}$. Floating point gives an almost constant precision across the dynamic range.

Example. See section 3.9.13. At $v = 1.1000 \times 2^3 = 12.0$, $\frac{\delta v}{v} = \frac{0.5}{12.0} = 0.042$.

At $v = 1.1000x2^{-3} = 0.1875$, $\frac{\delta v}{v} = \frac{\frac{1}{128}}{0.1875} = 0.04$.

Exercise . What about eight bit fixed point? What is the precision at: (a) .00000001? (b) .11111111? Note the large difference.

3.10 Common Number Representations

3.10.1 Various Sizes of Integers and their Ranges

Table 3.5 shows some commonly used integer formats and their ranges.

Size	Name in Java	Minimum	Maximum
8-bit signed	byte	-128	127
16-bit signed	short	-32768	32767
32-bit signed	int	$-2,147,483,648$	$2,147,483,647$

Table 3.5: Commonly Used Integer Representations

3.10.2 Floating Point

IEEE single precision (32 bits): $\pm 1.2 \times 10^{-38}$ to 3.4×10^{38}, 7 significant digits.

$value = (-1)^s \times 1.f \times 2^{exp-127}$.

bit 31 = sign, bits 30 to bit 23 = 8 bit exponent, bits 22to bit0 = fraction.

IEEE floating-point uses the hidden bit trick mentioned above.

3.11 Alphanumeric codes

Text characters, as well as numbers must be binary coded to get them into a computer memory. A code that copes with alphabetic characters, $\{\text{a} \ldots \text{z}, \text{A} \ldots \text{Z}\}$ as well as digits $\{0 \ldots 9\}$ is called *alphanumeric*.

Up to recently, the most frequently used code was the ASCII (American Standard Code for Information Interchange) code; ASCII uses 7-bits, (0 to 127); this includes *printable* and *control* characters, see Table 3.6.

Name	Hex	if control
bel	07	ctrl-G
line-feed	0a	ctrl-J
carriage-return	0d	ctrl-M
blank-space	20	
digits		
0	30	
...		
9	39	
alphabetic upper-case		
A	41	
...		
Z	5a	
alphabetic lower-case		
a	61	
...		
z	7a	

Table 3.6: ASCII table

3.11.1 UNICODE

ASCII has proved adequate for English text. But what about languages with accents? And those with funny shaped characters? UNICODE is a new international standard, governed by the UNICODE consortium, aimed at allowing a richer character set. In fact, Java uses UNICODE a 16-bit code – so that Java `char` type is 16-bit, compared to 8 bit in C and C++. According to (Tanenbaum 1999) the total of known characters in the world's languages number some 200,000 – thus, there is already pressure on UNICODE.

3.11.2 Types

$0x41$ could be 'A' or the number $0x41$. "AB" together in a 16-bit word could be part of a string, or the number $0x4142H$, or part of an instruction, or .... The computer program must know beforehand what each memory cell represents – thus, *types*.

3.12 Error Detection

When information is transferred between various parts of a computer system it is possible that errors may be introduced. These errors take the form of transmitted 0s inadvertently being received as 1s or vice-versa.

emphParity provides single error detecting capability. The scheme involves appending an additional bit to the string of 1s and 0s – the parity bit. Depending on the chosen convention, the appended bit is chosen so that the total number of 1s in the string is always even *even parity* or always odd *odd parity*.

Bit 7 is often used – pure ASCII uses 7 bits, bits $0 \ldots 6$..

Example. ASCII character 'A' using odd parity.

$A = 0x41 = P100\ 0001 = 1100\ 0001$ with odd parity bit placed in bit 7.

3.13 Exercises

1. Does the Roman number system (where 2002 is MMII) fit into the *positional number system* model?

2. I chose a number between 0 and 15. You are allowed to ask me questions about it which require a binary (yes/no) answer. Devise a strategy for getting the answer in a *minimum* of questions. (a) What is that minimum? (b) What if the number is between 0 and 255? (c) In the optimum strategy, what in fact is the nature of each question? (Hint: bits).

3. Convert the following 4-bit binary numbers to decimal (assume unsigned): 0000, 1111, 0001, 0101, 0111, 1000, 1001.

4. Convert the following decimal numbers to binary – use as many bits as you need (assume unsigned): 2, 4, 7, 8, 23, 127, 129, 192.

5. Convert the following 4-bit binary numbers to hexadecimal (assume unsigned): 0000, 1111, 0001, 0101, 0111, 1000, 1001.

6. Convert the following 8-bit binary numbers to hexadecimal (assume unsigned): 0000 0001, 0001 0000, 1111 0001, 1001 1000.

7. Convert the following decimal numbers to hexadecimal – use as many digits as you require (assume unsigned): 2, 4, 7, 8, 23, 127, 129, 192; Hint: it may be easier to complete a previous question first.

8. (a) Suggest symbols for a 'base-12' number system. (b) What is 33 base-12 in decimal? (c) What is 53 decimal in base-12? (d) What is 59 decimal in base-12?

9. (a) Suggest symbols for a 'base-36' number system. (b) What is 22 base-36 in decimal? (c) What is 73 decimal in base-36? (d) What is 71 decimal in base-36?

10. How many states are possible with: (a) 12 bits, (b) 16 bits, (c) 4 bits, (d) 1 bit.

11. How many bits are required to uniquely represent: (a) 2048 things, (b) 1000 things, (c) 100 things, (d) the decimal digits, (e) the letters 'a' to 'z', (f) 1 thing.

12. Perform the following arithmetic on 4-bit binary values: (a) $0111 + 0110$; (b) $0111 + 0110$; (c) $0101 + 0101$ — this is 101×2, notice what multiplying by 2 does; what do you thing division by 2 does? (d) (problems here! — explain) $0111 + 1101$.

13. Perform arithmetic on the following two digit hexadecimal values (assume that they must fit into 8-bits unsigned – point out any invalid results/overflows): (a) $0x2E + 0x15$; (b) $0x7E + 0x25$; (c) $0x1F + 0x31$; (d) $0xFF + 0x01$.

14. (a) Give the amended rules for base-36 arithmetic. (b) Add 11 base-36 to 22 base-36. (c) Check your work in decimal.

15. In an exam, I'd expect you to be able to describe the three methods of representing signed numbers – with examples and brief compare/contrast.

16. For this question, assume 4-bit twos-complement representation; in each case check for overflow and in your answer, note the validity of the result. Convert the following decimal numbers to binary: 2, 4, 7, -2, -4, -7, 8, -8.

    Do the same assuming biassed representation – bias is 4.

17. For an 8-bit 2-s complement format construct, and (where appropriate) give decimal value of: (a) the most negative number; (b) the most positive; (c) -1; (d) most negative plus 1; (e) zero.

18. For this question, assume 4-bit twos-complement representation. In each case check for overflow and in your answer, note the validity of the result. Perform the following arithmetic: (a) $0001 + 0101$; (b) $0001 + 1101$; (c) $1101 + 0101$; (d) $1000 + 0111$; (e) $0111 + 0110$; (f) $1001 + 1001$; (g) $1001 - 1001$; (h) $1001 + 0001$; (i)1101 + 1101.

19. Draw the circle formed by 4-bit twos-complement representation.

20. Convert the following binary numbers with fractional parts to decimal: (a) 11.11, (b) 101.111;

21. Convert the following decimal numbers with fractional parts to binary: (a) 5.5; (b) 2.25; (c) 3.625; (d) 5.875.

22. Translate the following ASCII (8-bit) code string (in hexadecimal) to text characters: `68656C6C6f20776f726C640D0A4279650D0A`; see Table 3.6, extrapolate as necessary, if 'a' is 0x61, 'b' is 0x62, 'c' is 0x63, etc.

23. (a) Translate the text string "Good day" to ASCII (hex.) (b) Add three 'beeps' to it.

24. If you did a hex print out of memory and found the following four contiguous bytes: `30313233`. Make some suggestions as to what is represented. In the context of *types* comment on this state of affairs.

25. Insert *odd parity* bits (top bit) to the following character string `68656C6C6f20`.

26. What is the major difference between UNICODE and ASCII?

27. You have a file containing unsigned bytes. You want to transmit it to another computer using a line normally used for ASCII characters: (i) identify the major problems that you will encounter; (ii) design a failsafe (and fairly efficient) solution (involving a simple encoding and decoding); (iii) if your file starts off at 1000 bytes, how many characters will you transmit; (iv) how many bits.

28. Suggest appropriate number systems for each of the following: (use integer where possible, use a few bits as possible, consider the use of code if appropriate; when I say code I mean that the 'real-world' quantity may be represented by other than a 'straight' representation, e.g. if I had student marks in the range 33 to 63 (out of 100), I could code this into 5-bits, using a biased representation — see notes above): (a) speeds of vehicles (e.g. in a radar speed trap machine); (b) identifiers for all the students in this class; (c) identifiers for all the people in Northern Ireland; (d) identifiers for all the people in the world; (e) identifiers for all the computers in the world (well, those connected to the Internet).

29. Explain how many bits are required for a *completely* faithful representation of a *Real* number; let's say it we are restricted to the range the range 0.0 to 1000.0 — but that doesn't affect the answer at all! Hint: it's an awful lot!

30. On most computer systems, a character (letter or digit) is stored as 8-bits (or one byte). A floppy disk can hold 1.4 Mbytes. (a) What does M mean? (b) In the same context, what does K mean?

3.14 Self assessment questions

These are also *recall* type questions that appear as parts of examination questions.

1. A computer system has eight (8) bit addressing. (a) What, in decimal, is its range of addresses? (b) in binary? (c) in hexadecimal? (d) What is the total number of memory cells it can address?

2. Repeat the previous question for 12, 16, 20, 32 bits.

3. How many bits do Pentiums use for addressing?

4. Convert to binary (give a justification/check if possible): (a) 24 decimal(base 10); (b) ff hex.; (c) 77 hex; (d) ff77 hex; (e) ffff hex; (f) fffff hex; (g) 10000 hex; (h) 256 decimal; (i) 255 decimal; (j) 128 decimal; (k) 127 decimal; (l) 32767 decimal; (m) 65538 decimal; (n) 4095 decimal;

5. Perform the following four-bit binary additions: 0111 + 0001; 0111 + 0011; 0000 + 0111; 1111 + 0000; 1111 + 0001;

6. Using Table XX, (a) convert the following ASCII string 68656C6C6f20 to uppercase.

7. Explain the use of *parity* for error detection in memory and data transmission systems. Convert the ASCII string 68656C6C6f20 to odd parity.

8. Explain the major difference between UNICODE and ASCII.

9. The ASCII code for 'B' = 0100 0010 binary, 'D' = 0100 0100, 'a' (lower-case) = 0110 0001. Hence, give the ASCII codes for: 'A', and 'd'.

10. Explain *two* major limitations of an 8-bit computer. [Hint: address word size, data word size].

Chapter 4

Digital Circuits and Logic

4.1 Introduction

We now tackle the lowest of the six levels mentioned in the introduction — the **digital logic** level. (In one part of the chapter we go to a lower level and show how logic circuits can be made from switches and transistors.)

Digital: means that we are dealing with 'digits', i.e. discrete quantities as opposed to continuous or analogue quantities;

Logic: indicates that the electronics is implementing a form of mathematics based on (propositional) logic.

First we will introduce propositional logic. Then we will show how simple logic functions are implemented in computer hardware.

4.2 Logic

4.2.1 Introduction

Since at least 400 B.C. human beings have been studying what it means to think and to reason. Two closely related aims can be identified:

- To understand the process of mental reasoning.
- To develop methods to mechanise thinking and reasoning, e.g. to develop procedural methods for doing mathematical proofs, or simply to give procedures by which a thinker could come up with a rational answer to a difficult question, or, more lately, as in the study of Artificial Intelligence, to develop methods by which computers may be programmed to perform tasks that require *intelligence*.

Broadly speaking, logic is the study of the principles and patterns of reasoning, argument and proof. Again generalising, we can say that logic uses a language that is a formalisation of certain types of natural language statements.

4.2.2 Propositional Logic

Propositional logic deals with statements (propositions) which make claims which can be either *true* or *false*. Thus, valid propositions are:

- Dublin is the capital of Ireland.
- Belfast has more than 100,000 inhabitants.
- Manchester United are Premiership Champions.
- Joe Bloggs is a first year LYIT student.

The test for validity is as follows: you must be able to meaningfully append to the statement "is true" (or "is false"). Thus "Dublin is the capital of Ireland *is true*". Notice, however, how "Manchester United (or Arsenal, God forbid!) are Premiership Champions" may need time/date qualification.

Symbols We can make symbols *stand-for* any proposition, e.g. $p =$ "Dublin is the capital of Ireland". Hence, in propositional logic, the variable p has the truth value *true*. $q =$ "Letterkenny has more than 10,000 inhabitants".

Propositional logic variables, e.g. p and q, above, are sometimes called *Boolean*s, or *Boolean* variables — because a mathematician call George Boole (mid 1800s) did a lot of work on the mathematics of logic.

4.2.3 Compound Propositions

We can now combine *elementary* propositions using logic *connectives* to make *compound* or *complex* propositions.

Conjunction – and Dublin is the capital of Ireland *and* Letterkenny has more than 10,000 inhabitants is *true*. I.e. `p AND q = true`. The combined proposition is true if-and-only-if both elements are true. However, the compound statement: Arsenal are European Champions *and* Letterkenny has more than 10,000 inhabitants is *false*.

The interpretation of logical *and* can be neatly summarised with a *truth-table* using abstract symbols:

p	q	p and q
F	F	F
F	T	F
T	F	F
T	T	T

Table 4.1: Truth-table for conjunction

Other symbols for conjunction are: $\wedge$ (looks like an A), and, in C/C++/Java, `&&`.

Disjunction – or Note: *or* has two natural language meanings: exclusive-or, only one of the statements can be true at any time; inclusive-or, only either one or the other needs be to be true, or *both*; without any qualification, *or* generally means inclusive-or. The interpretation of *or* is given by the following truth-table:

p	q	p or q
F	F	F
F	T	T
T	F	T
T	T	T

Table 4.2: Truth-table for disjunction

Other symbols for disjunction: $\vee$, and, in C/C++/Java, $||$.

Negation – not If a proposition p is *true*, then **not**(p) is *false*. The meaning of *not* is given by the truth-table:

p	not q
F	T
T	F

Table 4.3: Truth-table for negation

Other symbols for negation: $\neg$, $\sim$, and, in C/C++/Java, $!$.

4.3 Digital Circuits, Logic Circuits

Digital computers are constructed from switching circuits which implement propositional logic. 'Switch-on' might be associated with *true* and 'off' with *false*. However, equally, so long as everyone is agreed as to the convention, 'on' could be associated with *false* and 'off' with *true*.

Likewise, the physical entities representing the values could be {voltage-low, voltage-high} or {light-off, light-on}, or whatever. Very often, we use {0, 1} – the binary digits – to represent {false, true}, but please note that, in this case, they are **not** numbers.

The essence of logic circuits is that we have only two allowable values, which we can label whatever we like, usually: {false, true}, or{F, T}, or {0, 1}, and the two *binary* operations mentioned above: *conjunction* (*and*), and *disjunction* (*or*), together with the *unary* operation *negation* (*not*). There are other binary operations, but they can be expressed in terms of these three.

Note: in this context, we use the term **binary** to indicate that the operation/function operates on two values/inputs. It is a coincidence that the values can be interpreted as binary digits (bits)!

Let us now examine these operations or functions – to emphasise that they are functions, we write them in *prefix* notation, remarking, however, that the *infix* notation conveys exactly the same meaning.

Conjunction – and

$$\begin{aligned}\textbf{and}(F,F) &= F\\ \textbf{and}(F,T) &= F\\ \textbf{and}(T,F) &= F\\ \textbf{and}(T,T) &= T\end{aligned}$$

Of course, this is the same thing as Table 4.1. Because there is a slight analogy with numerical multiplication, *and* is sometimes denoted by the infix '.', or just concatenation of the input variables; also, *and* itself is often written in infix form. Thus, for Boolean variables p, q, and r, the following are equivalent:

$$\begin{aligned}r &= \textbf{and}(p,q)\\ r &= p\ \textbf{and}\ q\\ r &= p \wedge q\\ r &= p.q\\ r &= pq\\ r &= p\&q\end{aligned}$$

Disjunction – or

$$\begin{aligned}\textbf{or}(F,F) &= F\\ \textbf{or}(F,T) &= F\\ \textbf{or}(T,F) &= F\\ \textbf{or}(T,T) &= T\end{aligned}$$

Of course, this is the same thing as Table 4.2. Because there is a slight analogy with numerical addition, *or* is sometimes denoted by '+'. Thus, for Boolean variables p, q, and r, the following are equivalent:

$$\begin{aligned}r &= \textbf{or}(p,q)\\ r &= p\ \textbf{or}\ q\\ r &= p \vee q\\ r &= p+q\end{aligned}$$

Negation – not

$$\begin{aligned}\textbf{not}(F) &= T\\ \textbf{not}(T) &= F\end{aligned}$$

This is the same thing as Table 4.3.

Because there is a slight analogy with numerical negation, *not* is sometimes denoted by variations of $-$ and $\sim$. Thus, for Boolean variables p, q, the following are equivalent:

$$\begin{aligned} q &= \mathbf{not}(p) \\ q &= \neg p \\ q &= \bar{p} \\ q &= \sim p \\ q &= p' \end{aligned}$$

4.3.1 And, or are binary, not is unary and the associative law

Although it is not uncommon to see $p+q+r = p$ **or** q **or** r, you must remember that both *or* and *and* are *binary* – meaning two parameter functions. Thus, strictly,

$$p+q+r = p+(q+r) = \mathbf{or}(p, \mathbf{or}(q,r)),$$

just as in Java or C, `if(x==2||y==3||z==4)` strictly could be written `if(x==2||(y==3||z==4))`.

Of course, in 'ordinary' algebra, $+$ and $\times$ $-$, etc. are also binary, but the *associative law*, see section 4.3.2 for these operations allows us to string a series of them together. Note: in 'ordinary' algebra, there are two minuses, a unary one (-1) and a binary one $(3-1=2)$; always remember that if you have to write an expression evaluator program.

4.3.2 Equivalences of Propositional Logic

Just as it helps to know that in ordinary algebra, $a \times x + b \times x + a \times y + b \times y = (a+b) \times (x+y)$, or that $x + (-x) = 0$, it is useful to be aware of the more useful equivalences (or identities or laws) of propositional logic. Some of them are shown in Table 4.4. Some of these are intuitive, some not. For each equivalence there are duals – where **and** is swapped for **or** and *vice-versa*. For completeness, we show the table in two forms, Table 4.4, using the mathematicians $\wedge, \vee, \neg$ notation, and Table 4.5, using the engineers $.,+,\bar{}$ notation.

Name	**and** dual	**or** dual
Identity	$T \wedge p = p$	$F \vee p = p$
Null	$F \wedge p = F$	$T \vee p = T$
Idempotency	$p \wedge p = p$	$p \vee p = p$
Inverse	$p \wedge \neg p = F$	$p \vee \neg p = T$
Commutation	$p \wedge q = q \wedge p$	$p \vee q = q \vee p$
Distribution	$p \vee (q \wedge r) = (p \vee q) \wedge (p \vee r)$	$p \wedge (q \vee r) = (p \wedge q) \vee (p \wedge r)$
Association	$p \wedge (q \wedge r) = (p \wedge q) \wedge r$	$p \vee (q \vee r) = (p \vee q) \vee r$
Absorption	$p \wedge (p \vee q) = p$	$p \vee (p \wedge q) = p$
de Morgan	$\neg(p \wedge q) = (\neg p) \vee (\neg q)$	$\neg(p \vee q) = (\neg p) \wedge (\neg q)$

Table 4.4: Basic equivalences of logic — mathematics notation

Name	**and** dual	**or** dual
Identity	$T \,.\, p = p$	$F + p = p$
Null	$F \,.\, p = F$	$T + p = T$
Idempotency	$p \,.\, p = p$	$p + p = p$
Inverse	$p \,.\, \bar{p} = F$	$p + \bar{p} = T$
Commutation	$p \,.\, q = q \,.\, p$	$p + q = q + p$
Distribution	$p + (q \,.\, r) = (p + q) \,.\, (p + r)$	$p \,.\, (q + r) = (p \,.\, q) + (p \,.\, r)$
Association	$p \,.\, (q \,.\, r) = (p \,.\, q) \,.\, r$	$p + (q + r) = (p + q) + r$
Absorption	$p \,.\, (p + q) = p$	$p + (p \,.\, q) = p$
de Morgan	$\overline{(p \,.\, q)} = (\bar{p}) + (\bar{q})$	$\overline{(p + q)} = (\bar{p}) \,.\, (\bar{q})$

Table 4.5: Basic equivalences of logic — engineering notation

4.3.3 Truth-Tables used in Proofs

We have already shown truth-tables for **and**, **or** and **not** – Tables 4.1, 4.2, and 4.3. They can also be used for proving results like those above.

Example. Verify: $p + p.q = p$. Answer, see Table 4.6.

p	q	$p.q$	p	$p + p.q$
F	F	F	F	F
F	T	F	F	F
T	F	F	T	T
T	T	T	T	T

Table 4.6: Truth-table proof of $p + p.q = p$

Comparison of columns 4 (p) and 5 ($p + p.q$) proves the result.

Exercise. Use a truth-table to prove $p + \bar{p}.q = p + q$.

How many rows in a truth-table? In the example above, we have two input variables, p, q, hence we have 2^2 rows – these enumerate all the possibilities of $(p,\ q)$, where p, q can take on values from only $\{F, T\}$.

Ex. How many rows for a truth-table involving a, b, c? a, b, c, d? a, b, $\bar{c}$? a, b, c, $\bar{c}$?

Ex. In integer arithmetic, you want to prove that $x.y + (-x).y = 0$. Is it possible to do this using a table? Hint: how many rows? Answer, infinity. However, in logic, for p symbols, we need only 2^p rows.

4.4 Digital Logic Gates

Logic gates are the hardware implementation of logic functions. Again, all logic (Boolean) functions can be implemented using a combination of *and*, *or*, and *not* gates. The term *gate* comes from the operation of the *and* gate, see below, which, if one of its inputs is *false*, blocks or gates the other input, even if it is true.

Microprocessors, and other digital logic integrated circuits (chips) are simply massive collections of interconnected gates.

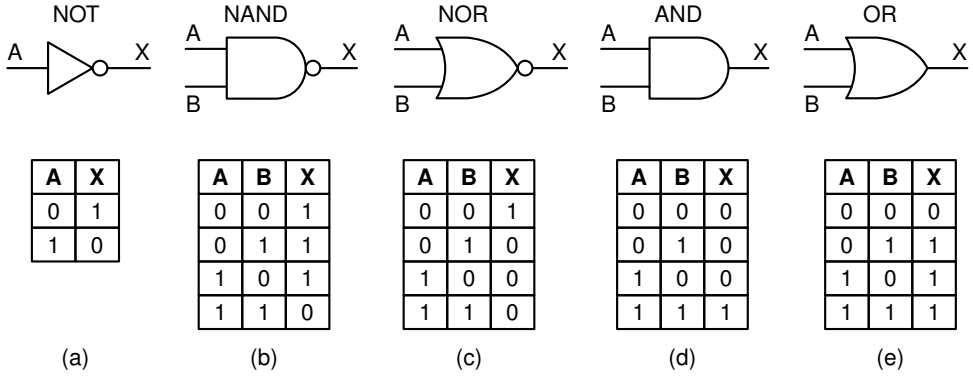


Figure 4.1: Symbols and truth-tables for (a) not (b) nand (c) nor (d) and (e) or
; note that 0 is used instead of F and 1 for T

Because of a particular twist of fate in the physics of transistors – of which logic gates are constructed – it turns out that real circuits tend to employ *nand* and *nor* gates a good deal in preference to *and* and *or*:

nand: p **nand** $q =$ **not**$(p$ **and** $q)$

nor: p **nor** $q =$ **not**$(p$ **or** $q)$

[A good many of the figures in the remainder of this chapter are from (Tanenbaum 1999), chapter 3].

Symbols for Logic Gates Figure 4.1 shows the symbols, and truth-tables, for the five basic logic gates.

The circles denote negation – *not*, inversion. It is possible to have negation on inputs too.

4.5 Logic Circuit Analysis

There are two fairly distinct activities in working with digital logic, analysis, and synthesis.

Analysis: given a circuit, what does it do? i.e. what logic (Boolean) function does it implement.

Synthesis: given a Boolean expression, produce a circuit that implements it – preferably using a minimal number of gates. This is synthesis or design.

This section covers analysis.

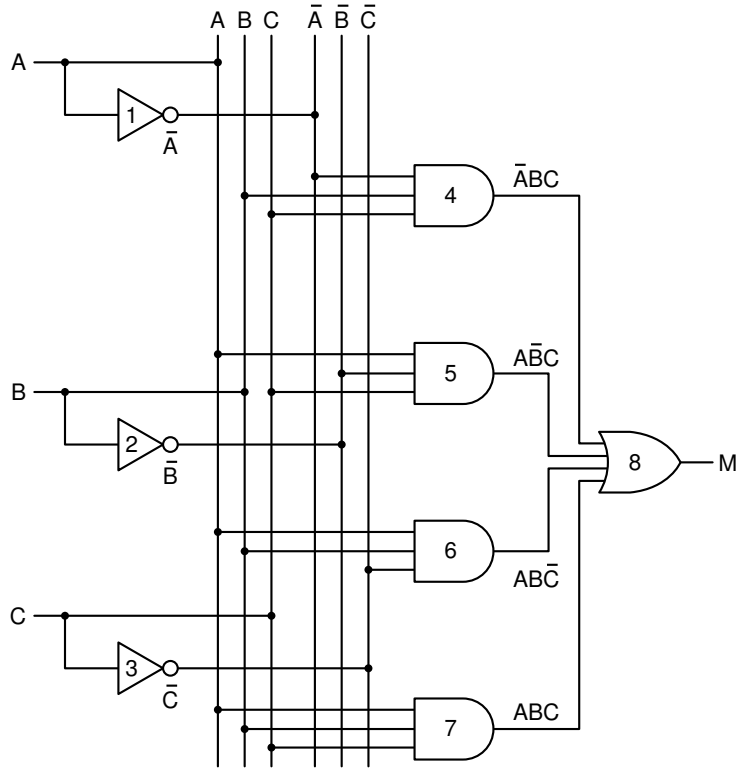
Example. What does the circuit in Figure 4.2 do?

It takes a majority vote over the three inputs.

Exercise. Verify the truth-table in Figure 4.2(a) by creating an expanded truth table containing columns for $A, B, C, \bar{A}, \bar{B}, \bar{C}, \bar{A}BC, A\bar{B}C, AB\bar{C}, ABC$. Note: starting from scratch, and not knowing the answer, this is the way you would normally do it – you cannot hope to work out column M in your head.

A	B	C	M
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

(a)



(b)

Figure 4.2: Truth-table (a) and circuit (b) for majority of three circuit

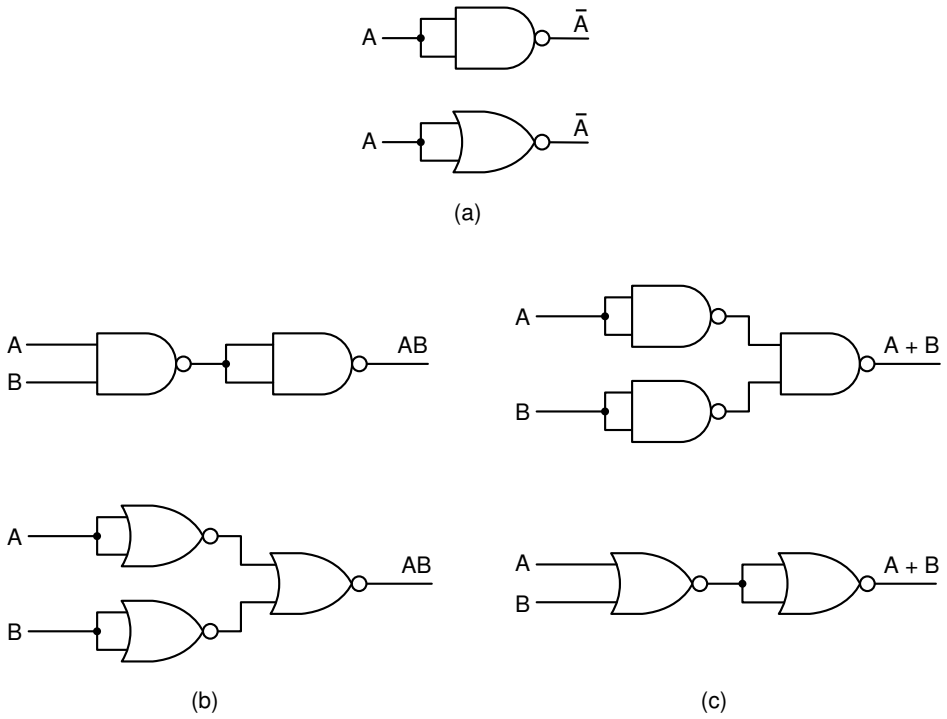


Figure 4.3: (a) not, (b) and, and (c) or functions implemented using nands and nors

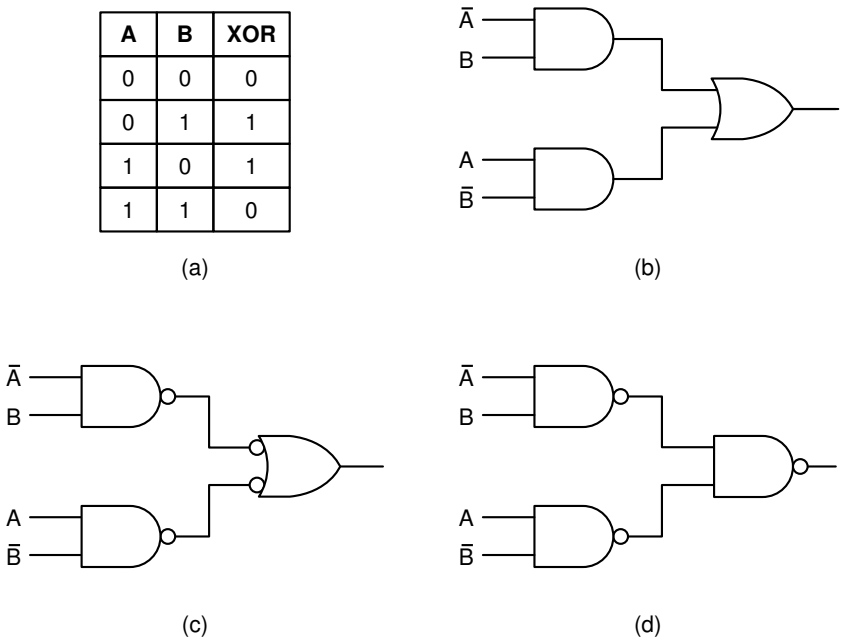


Figure 4.4: Exclusive or – three implementations

4.5.1 Equivalent Circuits

Quite often, because of a preference for *nand*, *nor*, or because there are spare gates left over, it may be desired to implement *and*, *or*, and *not* using *nand*, *nor*. Figure 4.3 shows some examples.

Let us examine the *nand* gate (top) in Figure 4.3(a).

$\overline{A.A} = \overline{A},$ since, from the idempotency law $A.A = A$.

Now the *nand* version (top) of Figure 4.3(b): the input to the second gate is clearly $\overline{AB}$, and we already know that, connected like this, the second *nand* gate performs negation, and two negations cancel, hence AB.

Exercise. Use (a) a truth-table; (b) *Boolean algebra* to verify that the two circuits in Figure 4.3(c), do, in fact, perform *or*.

4.5.2 Exclusive-or

Exclusive-or, commonly abbreviated to *xor*, as opposed to inclusive-or, see the discussion above, captures the meaning *either one or the other, but not both* – i.e. it outputs a T if the inputs are different. We will have cause to refer to it later, so it is worthwhile to show a circuit – Figure 4.4 shows three possible circuits and a truth-table. Figure 4.4(b) directly implements the Boolean expression most usually associated with *xor*: $\bar{A}B + A\bar{B}$.

4.6 Bitwise Logical Operations

In most assembly languages, and in `C` and `C++` there are binary operations which logical operations on each corresponding pair of bits. Here, we interpret 0 as F (false), 1 as T (true).

4.6.1 AND

$$\begin{aligned} a &= 0011\ 0000 \\ b &= 0101\ 0111 \\ a\ \&\ b &= 0001\ 0000 \end{aligned}$$

4.6.2 Masking

Let us say that we have two four bit numbers packed into a byte. How do we disentangle them. Take the byte 1001 0111; the two four bit numbers are 7 and 9. First, the lower four bits; here we *mask* with 0000 1111.

$$\begin{aligned} a &= 1001\ 0111 \\ lowMask &= 0000\ 0111 \\ a\ \&lowMask &= 0000\ 0111 \end{aligned}$$

For the higher four bits, we shift right by four bits (see subsection4.6.4, and use the same mask.

4.6.3 OR

$$\begin{aligned} a &= 0011\ 0000 \\ b &= 0101\ 0111 \\ a\ |\ b &= 0111\ 0111 \end{aligned}$$

4.6.4 Shift

$$\begin{aligned} a &= 0011\ 0000\ (=16+32=48_{10}) \\ a \text{ shiftleft } 1 &= 0110\ 0000\ (=96_{10}) \text{ i.e. } 48\times 2 \\ a \text{ shiftright } 1 &= 0001\ 1000\ (=24_{10}) \text{ i.e. } \frac{48}{2} \end{aligned}$$

Thus, you sometimes find assembly language programmers using shift for multiplying (shift left) and dividing (shift right) by powers of two.

4.6.5 Rotate

$$\begin{aligned} a &= 1011\ 0000 \\ a \text{ rotateleft } 1 &= 0110\ 0001 \\ a \text{ rotateright } 1 &= 0101\ 1000 \end{aligned}$$

Rotate is like shift, only bits can wrap-around.

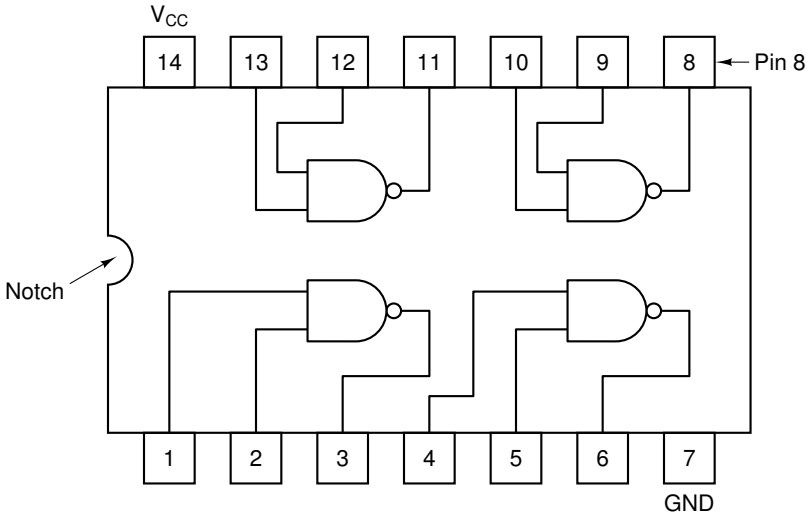


Figure 4.5: 7400 Integrated circuit – four nand gates

4.7 Generations of Integrated Circuits

Integrated circuits started to appear in the late 1960s. A typical single chip of that age, still available to buy today, a Texas instruments transistor-transistor logic (TTL) 7400 chip, is shown in Figure 4.5; it contains four *nand* gates and has 14 pins.

Nowadays, an integrated circuit such as Figure 4.5 is said to be *small scale integrated* (SSI). Small scale integrated circuits contain up to 10 gates (up to about 100 transistors). In the late 1960s, complete computers were made from SSI; still, this was a lot better than the individual devices (transistors, resistors, etc.) of the 1950s and 60s. And, a vast improvement over the vacuum tubes used in the 1940s.

You will still find SSI circuits, e.g. TTL 7400 quad 2-input *nand* used on the periphery of computer systems, and as the 'glue' which connects the more complex circuits/chips together.

In SSI chips, as many gates of a given type as possible are put in the package (chip). The number of gates is limited by the number of input-output pins, and to a lesser extent by the ability to dissipate the heat generated by the circuit. On the 7400 chip above (which is typical of the whole 7400 series), we have pins as follows:

- Power. V_{cc} and ground: 2 pins;
- Each gate. Two inputs and one output = 3; four gates: 12;
- Total: 14 pins.

An additional consideration in chip design is the total area taken up. In an SSI chip like the 7400, this is roughly 2.5 cm $\times$ 1 cm. Of course, the actual circuitry (silicon) would be less than this, maybe 2 mm $\times$ 2 mm.

Each 7400 series chip would consume about 10 milliWatts of power, i.e. it would draw 2 milliAmps of current (5 volt power supply).

Later medium scale integrated (MSI) circuits began to appear – containing 10–100 gates. Later on large scale integrated (LSI) containing 100–100,000 gates. The first microprocessors (1971 – the 4 bit Intel

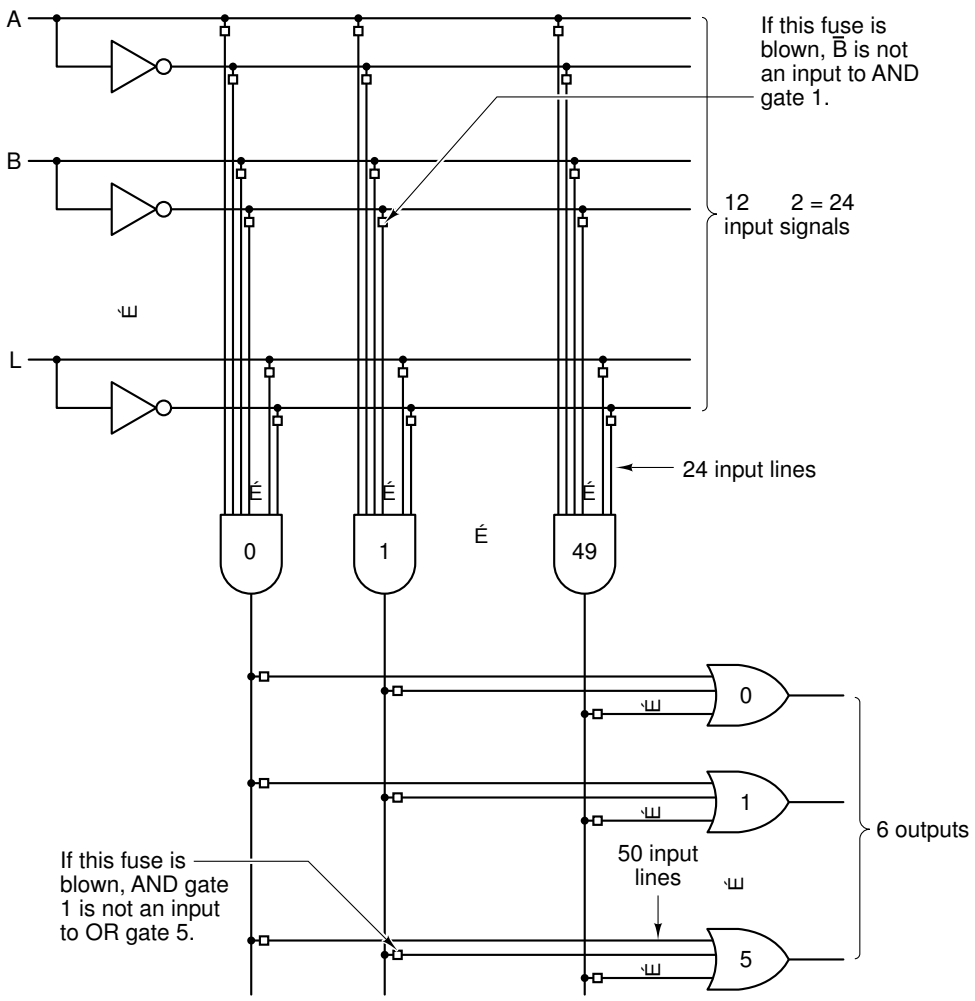


Figure 4.6: A programmable logic array. The squares represent fuses which can be blown to disable connections, and so determine which function is implemented, i.e. programmed

4004) would have been classified LSI. Now we have very large scale integrated (VLSI), and beyond. The latest Pentium IV contains in the region of 50,000,000 transistors.

A great many very complex complete components are now available on single chips – microprocessors, disk controllers, modems, ... On the other hand, there is still a need to design circuits from simple gates. But, building large circuits from SSI is simply impractical – it would take too much space and consume too much power. Hence, programmable logic arrays (PLAs) and field programmable gate arrays (FPGAs).

4.8 Programmable Logic Arrays

A programmable logic array contains a large array of *not*, *and* gates and *or* gates on a single chip. The user can connect certain gates by breaking certain (fuse) connections. Figure 4.6 contains an example.

4.8.1 Transistor implementations

For completeness, we show transistor implementations of *not* (also called an *inverter*), *nand* and *nor*. Again for completeness, we give the following model of a transistor when it is operated in so-called

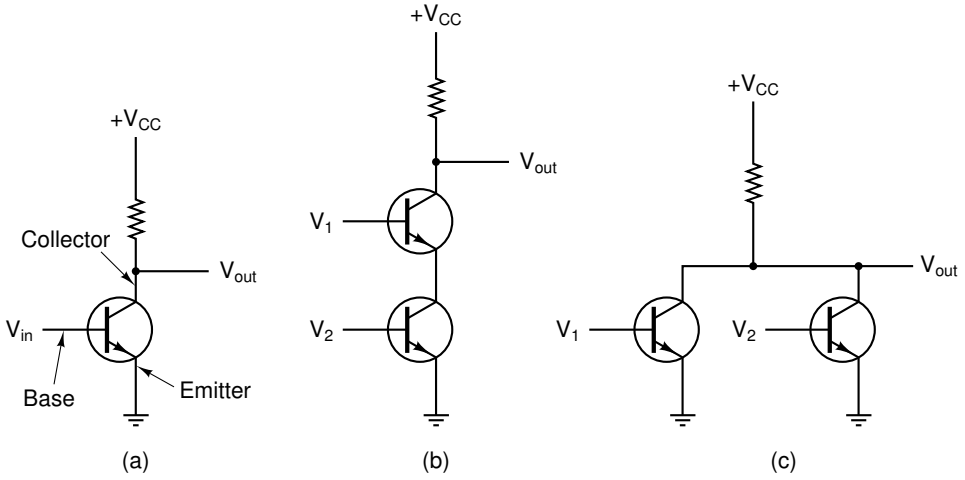


Figure 4.7: (a) Not (b) nand (c) nor

switching mode.

Transistor as a Switch There are three connections – collector, base, and emitter. The positive power supply (e.g. +5 volts) is applied to the collector via a resistor; the emitter (the one with the arrow) is connected to ground/earth. The input signal is applied to the base. When the input signal is *high*, say > 3 volts when the power is 5 volts, the transistor switches *on*, i.e. current is allowed to flow down to the emitter and out to ground. When the input signal is *low*, the transistor switches *off* i.e. little or no current is allowed to flow.

The net effect of the transistor in Figure 4.7(a) switching *on* is that the output is connected to ground and V_{out} takes the voltage of ground – ~ 0 volts. Thus *high* input (V_{in}) – logic 1 – yields *low* output – logic 0; thus, *not*, inversion.

The net effect of the transistor switching *off* is that the output takes on the voltage $\sim V_{cc}$ volts. Thus *low* input (V_{in}) – logic 0 – yields *high* output – logic 1; thus, *not*, inversion.

With this model in mind, it is easy to verify that the circuits in Figures 4.7(b) and (c) do in fact operate as *nand* and *nor*.

If you would like to see mechanical implementations of gates (levers, also water valves), see (Hillis 1999). If you want to see switch and relay implementations, see (Petzold 2000).

4.9 Exercises

1. Besides bits in computers, light-bulbs, and the truth-values of propositions, describe *five* other activities that use, or are suitably described by, Boolean values (on/off, true/false).
2. (Section 3.3) Use a truth-table to verify the *or* version of de Morgan's Law: $\overline{(p+q)} = (\bar{p})\ .\ (\bar{q})$
3. Use a truth-table to verify the inverse law: $p + \bar{p} = T$.
4. Prove that $pq + p\bar{q} = p$ Hint: use the *or* form of the inverse law: $p + \bar{p} = T$.
5. Compute the truth of the compound statement *Letterkenny is in County Donegal OR the population of Letterkenny is greater than 500,000*.

6. Compute the truth of the compound statement *Letterkenny is in County Donegal AND the population of Letterkenny is greater than 500,000*.

7. Simplify the following using Boolean algebra: (a) $\bar{A}\bar{B}+AB+\bar{A}B$; (b) $AB\bar{C}+ABC+A\bar{B}\bar{C}+A\bar{B}C$.

8. Simplify the following, explaining your steps: (a) $F.T$; (b) $F+T$; (c) $(T.T).(T+F)$; (d) $(T+T).A$; (e) $\bar{A}.(A+B)$; (f) $A.(A+B')$; (g) A**xor**A; (h) A**xor**$\bar{A}$.

9. See Exclusive-Or (xor) (a); discuss how *xor* relates to *equivalence/equality*.

10. How would you use two *xor*s, two *inverter/not*s, and an *and* to create a circuit to compare two two-bit numbers.

11. Look back at Chapter 3 and the rules for binary addition; assume that you are adding bits a,b. Create a truth table for a,b,sum; compare with *xor*. Create a truth table for $a,b,carry$; compare with the truth-table for *and*.

12. Consider the task of making a Pentium IV microprocessor out of 7400 chips; see Figure 4.5. I reckon you can get about 100×7400 chips onto a card the size of a PC motherboard, and about 25 of those cards into a cabinet the size of a filing cabinet, and I could get 100 filing cabinets into my room; how many rooms for your Pentium?

    What will be the power consumption? Assume that each 7400 chip consumer 10 milliWatts of power.

13. The following equation describes a *two*-input multiplexer: when A is $false$ the output is (the same as) D_0, when A is $true$ the output is (the same as) D_1: $D_0.\bar{A}+D_1.A$. (a) Verify this statement using a truth table.

    (b) Draw a diagram for a two-input multiplexer.

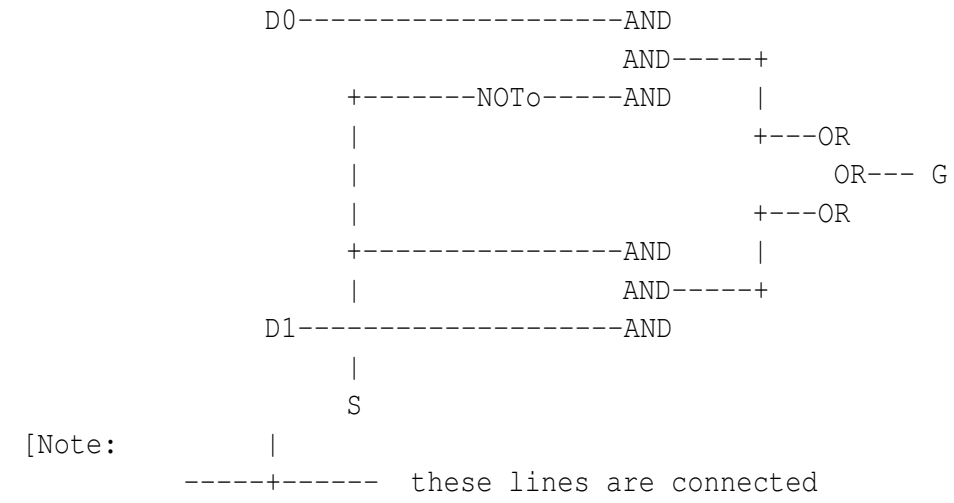
    (c) A *four* input multiplexer will have four inputs — and one output; how many control lines (A, B, C, ...).

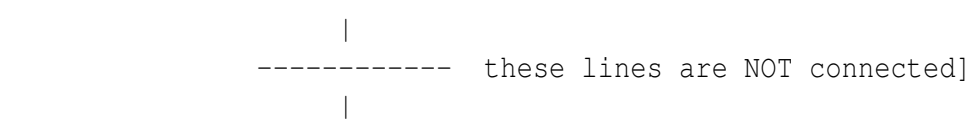
    (d) Sketch a diagram for a four-input multiplexer.

14. Draw a diagram of a two-input multiplexer that uses on/off switches.

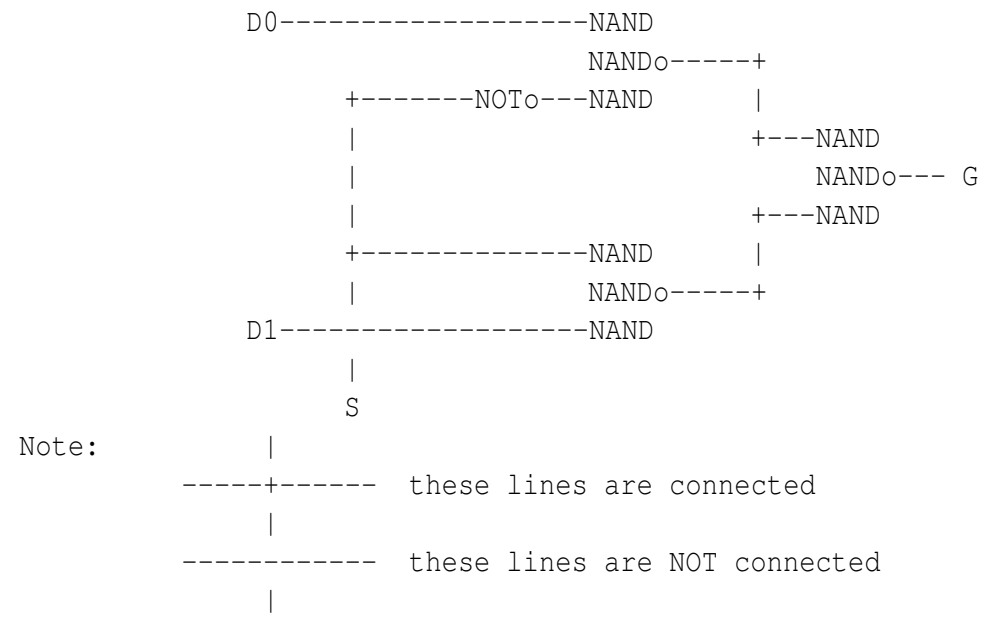
15. Show how you would construct a 2-to-1 multiplexer using a 7400 quad NAND gate, see Figure 4.5; concentrate on the fact that it has 4 NAND gates.

16. Create a truth-table to verify that the following circuit performs the action of a two-to-one multiplexer. Compare G to D0 and D1, and see what happened for S(witch) = 0 (F), S = 1 (T).



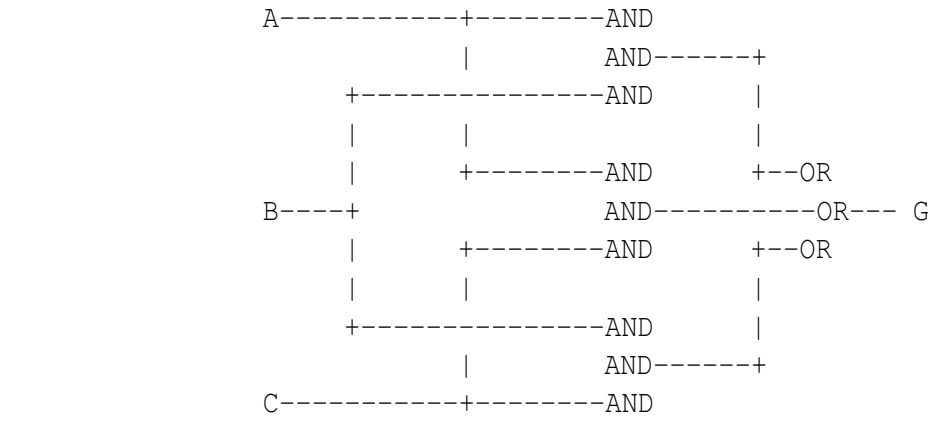


17. Create a truth-table to verify that the following circuit performs the action of a two-to-one multiplexer. Compare G to Y and Z, and see what happened for S(witch) = 0 (F), S = 1 (T).



Note:

18. Verify that the following circuit computes a *majority* decision over A,B,C, i.e. for $(A,B,C) = (T,F,F)$, the output is F, whilst for (F,T,T) or T,T,T etc. the output is T. A is connected to AND gates 1 and 2, B is connected to AND gates 1 and 3, and C is connected to AND gates 2 and 3.



19. Perform the following bitwise logical operations: (a) 0101 1001 **and** 00001111; (b) 0101 1001 **and** 00000110; (c) 0101 1001 **or** 00001111; b) 0101 1001 **or** 00000110.

20. Perform the following *shift* operations: (a) 0000 0110 shiftright 1; (b) 0000 0110 shiftleft 1; (c) convert the originals and the shifted to decimal; hence deduce the arithmetic equivalent of shift-right? shift-left?

Chapter 5

The Components of a Computer

This chapter describes the intermediate sized building blocks of computers, i.e. those which are intermediate between simple gates at one extreme, and full microprocessors on a chip at the other extreme. With an understanding of these components, in the next chapter, we will be able to describe the construction of a working computer.

Most of the circuits below are available as CMOS or TTL MSI chips. In a microprocessor such as a Pentium IV, obviously these packages are not used; however, if you examined the circuit diagram for a microprocessor, you would find similar *building blocks* used throughout the chip.

5.1 Multiplexers and routing circuits

5.1.1 Multiplexer

A multiplexer (mux) connects via switches a number (typically, some power of two, 2^n) of input lines to **one** output line.

Figure 5.1 shows a two input multiplexer.

You can analyse it in a number of ways. First, think of the AND gates as *gates* in the open-closed sense. A '1' *true* on one input will allow the other input to pass through: $1.x = x$. Thus, when S is 0, what come out of the NOT gate (inverter) is 1, the gate will open and D0 will appear at the output of the top AND gate; in the case of the bottom AND gate, S will be inputting a 0, and that gate will be closed. We combine the outputs with an OR. When S is 1, the bottom gate will be open, and the top gate closed.

Second. Do the following as an exercise: derive an expression for F – in terms of D1, D2, and S.

Third. Figure 5.2 gives a truth-table for the two input multiplexer. Notice that when S is 0, F is the same as D0; when S is 1, F is the same as D1.

Figure 5.3 shows an eight input multiplexer. A, B, C are the selection (or switching or control or address) lines; when $(A,\ B,\ C) = (0,\ 0,\ 0)$, input D_0 – and no other input – is fed through to F.

Exercise. Construct a truth-table for the eight-input multiplexer shown in Figure 5.3 and thereby verify that it does as promised, i.e. $F = D_0$ when $(A,B,C) = (0,0,0)$, etc.

Exercise. By examining Figure 5.3, derive an expression for F in terms of $A,B,C,D_0,D_1,\ldots D_7$. Check it for $(A,B,C,D_0) = (0,0,0,1)$ and $(0,0,0,0)$.

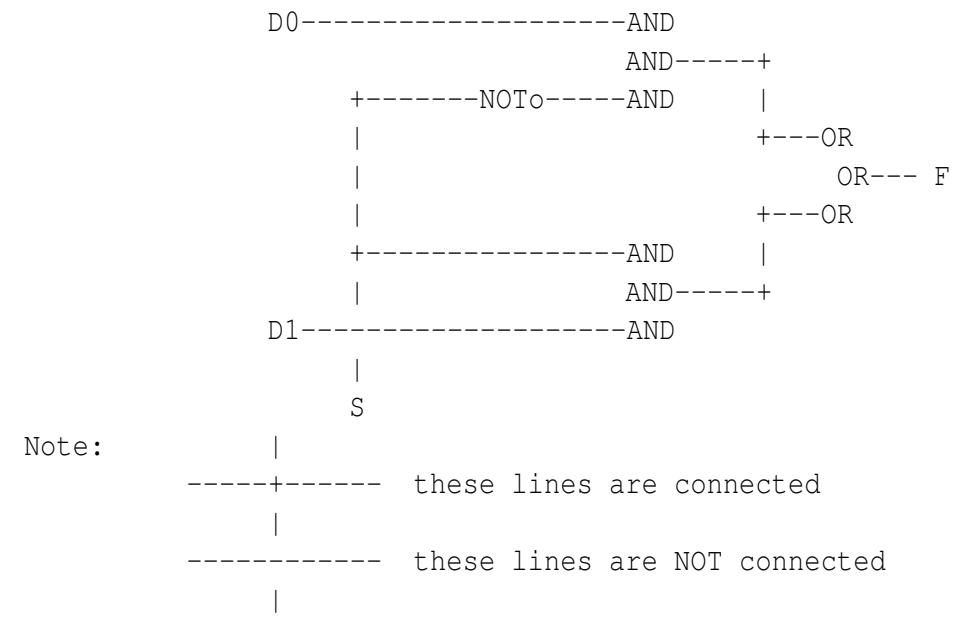


Figure 5.1: Two input multiplexer

D1	D0	S	S′	D0&S′	D1&S	F= (D0&S′)OR(D1&S)
0	0	0	1	0	0	0
0	1	0	1	1	0	1
1	0	0	1	0	0	0
1	1	0	1	1	0	1
0	0	1	0	0	0	0
0	1	1	0	0	0	0
1	0	1	0	0	1	1
1	1	1	0	0	1	1

Figure 5.2: Two input multiplexer, truth table

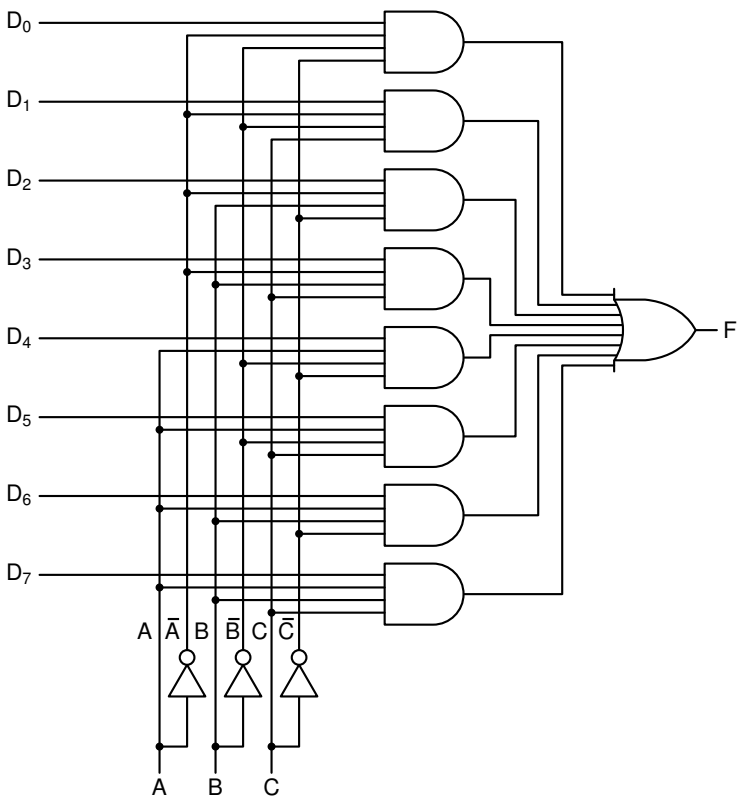


Figure 5.3: Eight input multiplexer

5.1.2 Demultiplexer

A demultiplexer does the opposite to a multiplexer: it has one input, and many (2^n) outputs. A demultiplexer uses select/address lines just as the multiplexer, i.e. the appropriate output is addressed by the select lines.

5.1.3 Decoder

A decoder takes n inputs, has 2^n outputs, and, according to the select/address lines, one (and one only) of the outputs goes to a logic 1.

In principle, a decoder operates like a demultiplexer, but with a logic 1 tied to the (single) input all the time, i.e. in a decoder, the selected output goes to 1. Figure 5.4 shows a 3-to-8 decoder.

Exercise. By examining Figure 5.4, derive an expression for F in terms of $A, B, C, D_0, D_1, \ldots D_7$. Check it for $(A, B, C, D_0) = (0, 0, 0, 1)$ and $(0, 0, 0, 0)$.

Consider the following application of a decoder: a hotel with 8 (2^3) rooms needs to send room identity, e.g. for fire-alarm, to a central location. Instead of using eight lines, being knowledgeable of computer science, they decide to use just three lines, and code the room identity as a three bit number. But, they want any alarm to light one of eight lights; hence, they need a 3-to-8 decoder.

Similarly, in a computer, it may be wasteful to use eight lines for eight actions: so code them and transmit them as three bits, then decode the three bits to eight useful bits when you need them.

Exercise. Examine Figure 5.4. Devise a circuit for a one-to-two decoder. One input, A, two outputs D_0 and D_1; when A is 0, D_0 is 1 and D_1 is 0; when A is 1, D_0 is 0 and D_1 is 1.

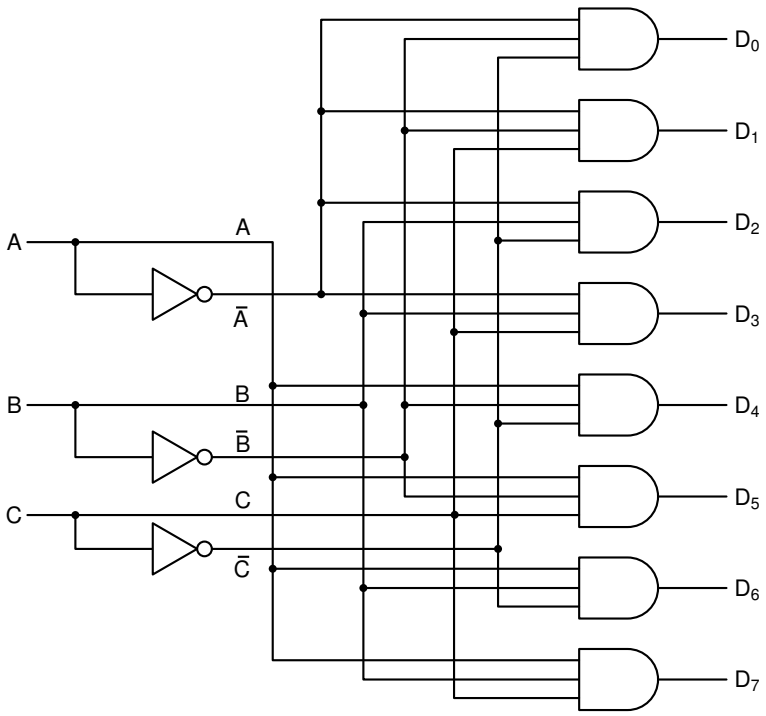


Figure 5.4: A 3-to-8 decoder

A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Figure 5.5: Two input multiplexer, truth table

Write down the truth-table for the one-to-two decoder.

5.2 Arithmetic Circuits

5.2.1 Adders

Half Adder

Think back to decimal addition using carries, e.g. $6804 + 1236$ ($6 + 4$ is 10, put down 0 and carry one .... Binary addition goes as follows: 0 plus 0 is 0 and no carry; 0 plus 1 is 1 and no carry; 1 plus 0 is 1 and no carry; 1 plus 1 is 0 and carry 1. Or we can rephrase as: 0 plus 0 is 0 and carry 0; 0 plus 1 is 1 and carry 0; 1 plus 0 is 1 and carry 0; 1 plus 1 is 0 and carry 1. This can be expressed in the truth-table in Figure 5.5.

We see that the Carry part is easy, it is just the A AND B. Sum is 1 only when A is different from B, and zero otherwise; this can be implemented using an exclusive-or gate – recall section 4.5.2 and Figure 4.4.

A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

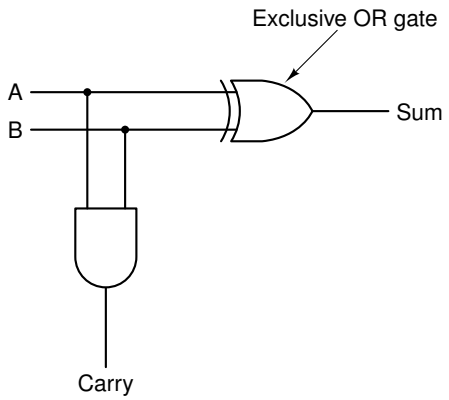
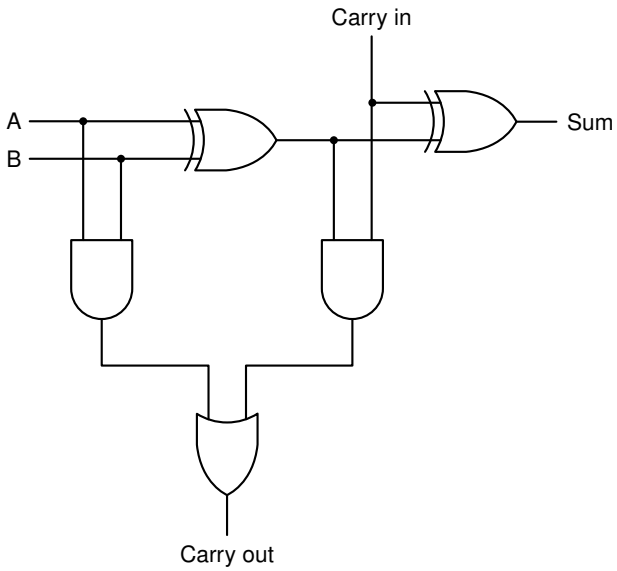


Figure 5.6: A half-adder. (a) truth-table; (b) circuit.

A	B	Carry in	Sum	Carry out
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

(a)



(b)

Figure 5.7: A full adder. (a) truth-table; (b) circuit.

A half adder is shown in Figure 5.6 together with its truth-table adds together two single bit inputs bits to give a sum bit S, and a carry bit C. We see that $S = \bar{A}.B + A.\bar{B} = A$ **xor** B; and $C = A.B$.

Full Adder

The full-adder is a bit more useful; the inputs are two bits to be added, plus carry – from another previous addition. A full-adder may be built from half-adders and an OR gate, see Figure 5.7.

5.2.2 Arithmetic and Logic Unit (ALU)

Getting more ambitious, Figure 5.8 shows a one bit ALU. It uses two function bits F_0, F_1 – bottom left-hand corner – to choose between four possible operations:

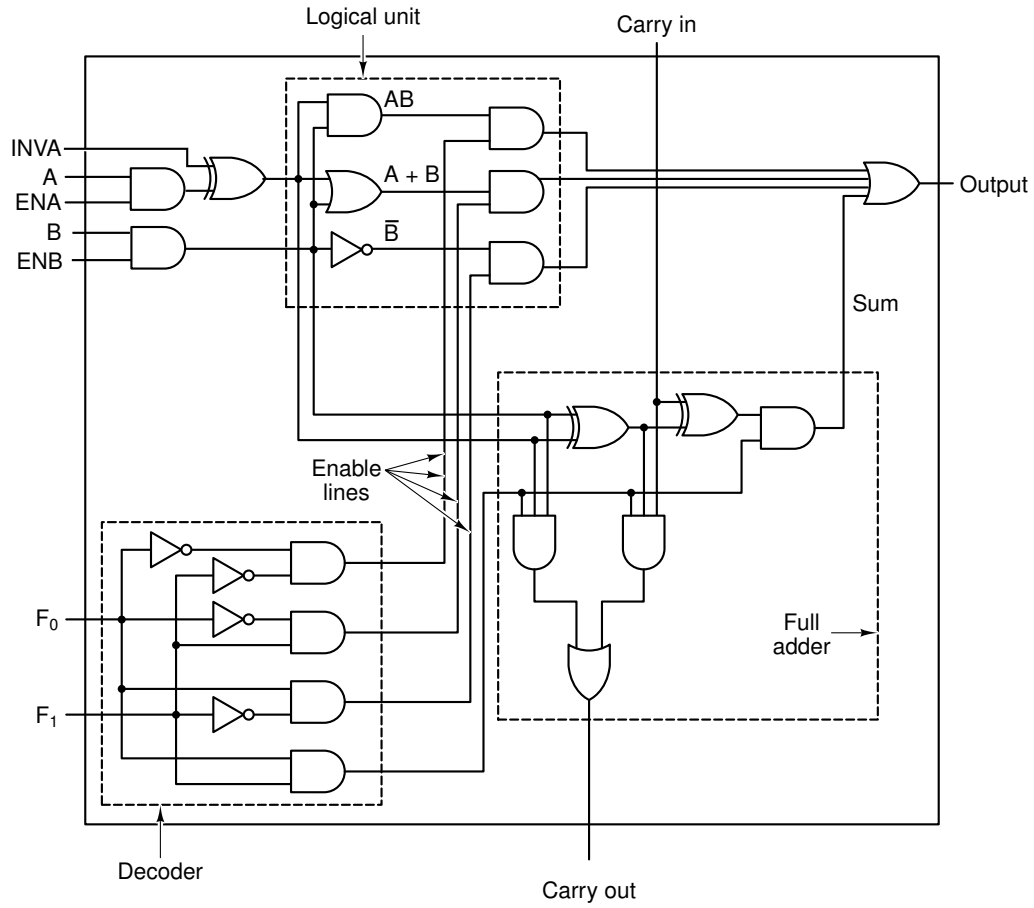


Figure 5.8: A one bit ALU

A **and** B, A **or** B, **not** B, and arithmetic sum, $A + B$. In addition, via *INVA*, $\bar{A}$ may be substituted for A in any of the four. Moreover, either A or B, or both, may be enabled via *ENA*, *ENB*; if *ENx* is 1 then the value for x is *enabled* (allowed to pass into) the circuit, otherwise 0 is passed in.

Notice the full adder on the bottom right, and the decoder, see section 5.1.3, on the bottom left.

Connecting one bit ALUs together

We can make an n-bit ALU by connecting n one bit ALUs together; such an 8 bit ALU is shown in Figure 5.9. In some contexts, such one bit circuits (or sometimes more than one) are called 'bit-slices'.

In the next chapter we will use an ALU, viewed functionally as a subsystem, like that in Figure 5.10. That one can compute *four* functions which are selected by control lines F_0, F_1. The functions are:

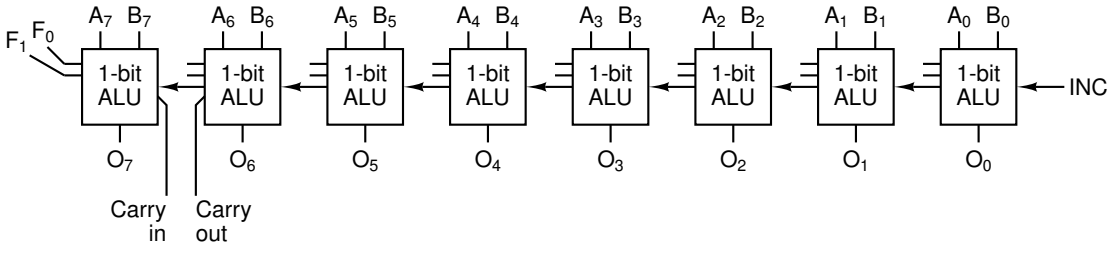


Figure 5.9: An 8 bit ALU constructed from one bit ALUs

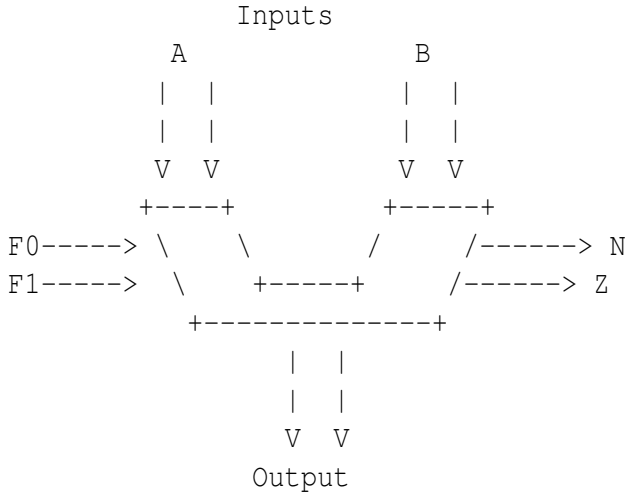


Figure 5.10: ALU – Functional view

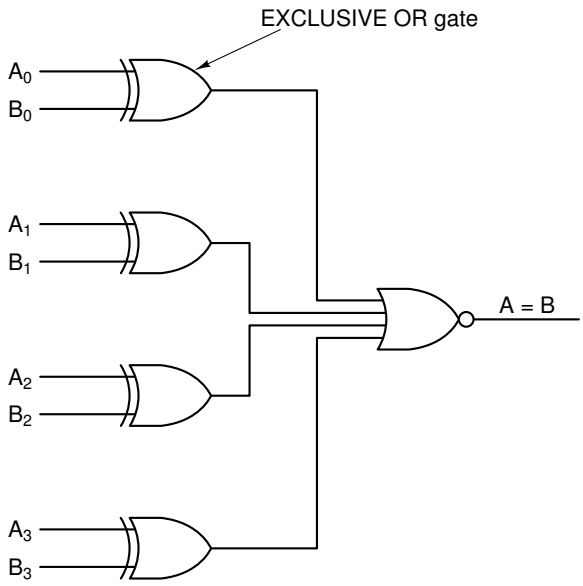


Figure 5.11: A four bit comparator

A plus B, *A* **and** *B*, *A* 'straight through', and $\bar{A}$. Two condition flags are output, which give the 'condition' of the result of the last operation. These are: *N*, set to 1 if the last operation caused a negative result, *Z*, set to 1 if the last operation resulted in zero. **N.B. please do not get confused by the difference between the one bit ALU in Figure 5.8 and the one in the next chapter**.

5.2.3 Magnitude Comparator

Figure 5.11 shows a four bit magnitude comparator.

Exercise. Verify that Figure 5.11 does in fact output 1 when the inputs are equal, 0 when unequal. Hint 1: see section 4.5.2, **xor** is **true** if and only if the inputs are different. Hint 2: use de Morgan's law to show $\overline{X_0 + X_1 + X_2 + X_3}$ in terms of **and**, where X_i is the output of the ith **xor** gate.

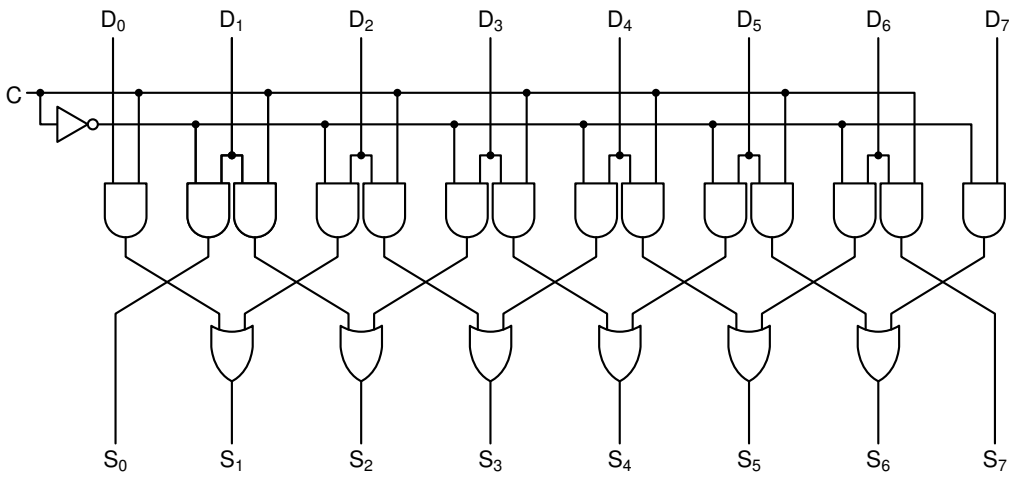


Figure 5.12: An eight bit shifter

5.2.4 Shifter

Figure 5.12 shows an eight bit shifter. $C = 1$ shifts right, $C = 0$ shifts left.

5.3 Flip-Flops and Latches – Memory

5.3.1 Introduction

Up to now, we have dealt with **combinatorial** logic, i.e. the outputs depend only on the current inputs – the outputs disappear when the inputs disappear. In many cases, definitely in a computer, we require retention of **state** – **memory**.

Digital circuits with memory are called **sequential** circuits.

After showing a theoretical model of sequential circuits, we will describe some basic memory circuits – flip-flops, etc. Then section 5.4 will give an overview of larger memory subsystems such as are used as the main memory of computers.

5.3.2 Sequential Circuit as Combinatorial plus Memory

A sequential circuit can be modelled as a separate combinatorial circuit connected to memory or storage, as shown in Figure 5.13. However, this diagram is useful only from a theoretical point of view, and, in practice, the memory and gates are all mixed up.

5.3.3 Set-Reset (SR) Latch

[Incidentally, the terms ‘latch’ and ‘flip-flop’ tend to get mixed up; (Tanenbaum 1999) makes the distinction that latches are level triggered, whilst flip-flops are edge triggered, see section 5.3 below.

Figure 5.14 shows a Set-Reset(SR) latch implemented with **nor** gates.

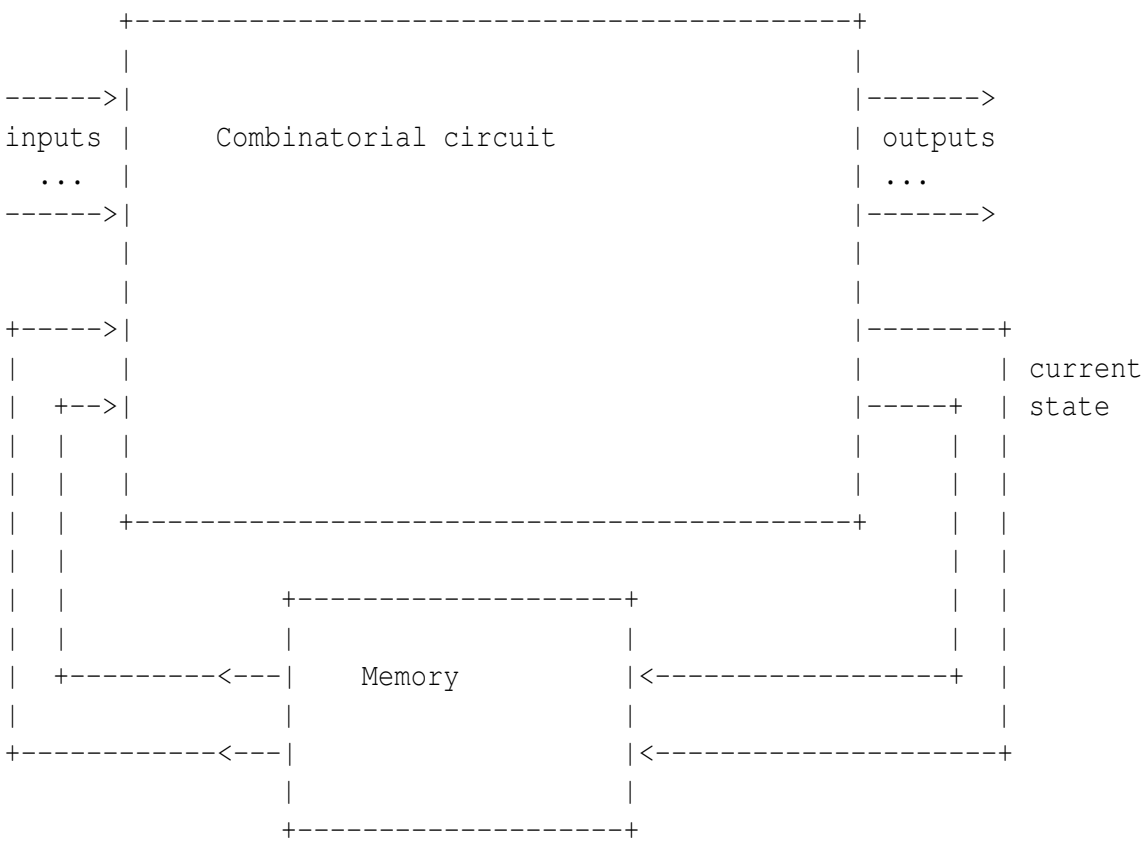
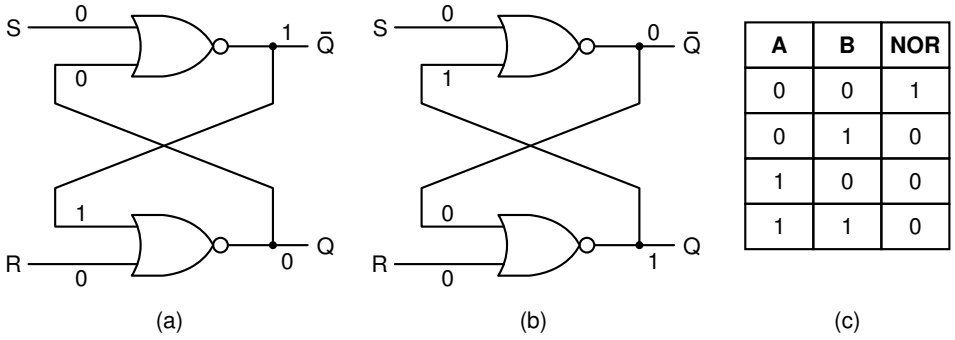


Figure 5.13: Sequential Circuit as Combinatorial + Memory



A	B	NOR
0	0	1
0	1	0
1	0	0
1	1	0

Figure 5.14: SR Latch (a) State 0 (b) State 1 (c) Truth-table for nor

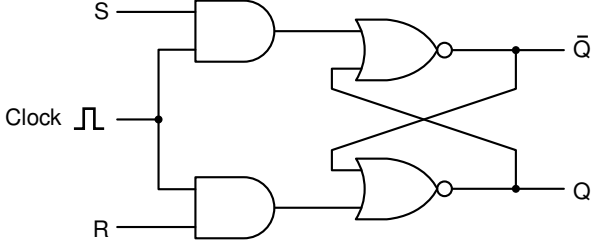


Figure 5.15: Clocked SR Latch

Analysis Two inputs: S, R; one Sets – output goes to 1, the other Resets – output goes to 0. Two outputs: Q, $\bar{Q}$, which are inverses of one another.

1. Assume S and R both 0; and, $Q = 0$. Gate 1 (G1) has $(0,0)$ as input, i.e. $\bar{Q} \to 1$. This 1 goes to Gate 2 (G2) which now has $(1,0)$ as input, so $Q \to 0$.
2. Assume $S = R = 0$; but that now, $Q = 1$. So, $(0,0)$ into G1 gives $\bar{Q} \to 0$. And, $(0,0)$ into G2 gives $Q \to 1$.
3. The states $Q = \bar{Q} = 0$, and $Q = \bar{Q} = 1$ are inconsistent, so for $S = R = 0$, Q will remain 0 or 1; it is **stable** in either state.
4. If we start at $Q = 0$, and S changes to 1. $(1,0)$ into G1 gives $\bar{Q} \to 0$. This 0 goes to G2 i.e. $(0,0)$ into G2 which gives $Q \to 1$. Thus, setting S to 1 switches the latch from 0 to 1.
5. At $Q = 0$, setting R to 1 has no effect since $(1,0)$ into G2 has the same effect as $(1,1)$.
6. At $Q = 1$, $R \to 1$. $(0,1)$ into G2 gives $Q \to 0$.
7. At $Q = 1$, $S \to 1$ has no effect. Verify as exercise.
8. What happens if $S = R = 1$, i.e. someone is not using the device correctly? While S, R are held at 1, the only 'consistent' output state is $Q = \bar{Q} = 0$. As soon as one drops, the output goes into the appropriate stable state. If they drop together, then the stable state is chosen randomly.

5.3.4 Clocked SR Latch

It is often important to restrict changes to specified times which are specified by a **clock** pulse – also called **enable** or **strobe**. See Figure 5.15. In this case, S, R get through the **and** gates only when the clock pulse is present (1).

5.3.5 Clocked D-type Latch

Figure 5.16 shows a D-type latch; this gets rid of the nonsense of $S = R = 1$ together, i.e there is only one input, D (Data).

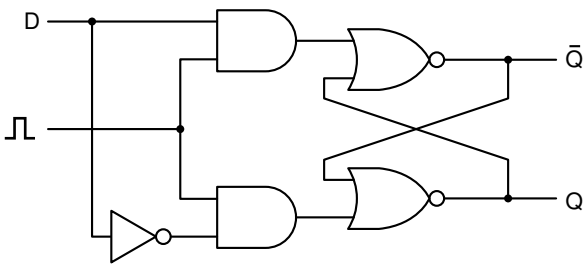


Figure 5.16: Clocked D-type Latch

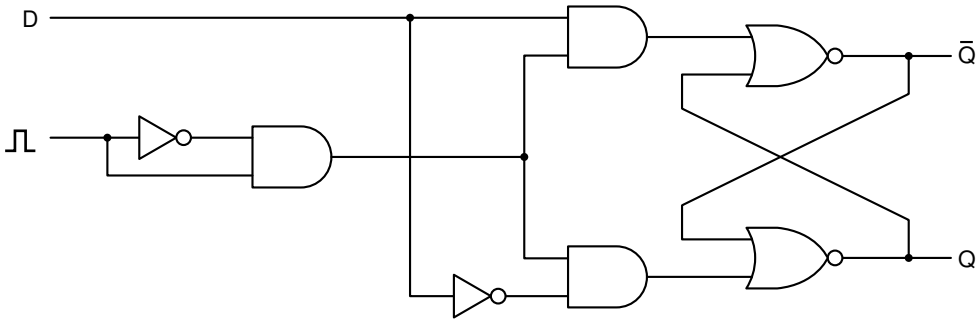


Figure 5.17: D-type edge triggered flip-flop

5.3.6 D-Type Edge Triggered Flip-Flop

Enabling/clocking with a level is uncomfortable for engineers. Edges or transitions are better – they are more clearly defined. Thus, D-type edge triggered, see Figure 5.17 for a circuit.

Figure 5.18 shows the symbols used for some D-type latches and flip-flops. *CK* stands for clock, a circle on it signifies clocked by 0 level, or falling edge – as opposed to level 1, or rising edge.

5.4 Memory

It would be possible to make a (main) memory, or individual registers, from D-type flip-flops, but pretty impractical if you need a number of Megabytes. And, it's not only that multi-million chips is a problem, but the multi-million lines in and out, not to mention the clock lines.

Figure 5.19 shows the logic diagram for a four word three-bits memory system.

It is instructive to understand how it works in principle, because all bigger chips operate similarly.

Inputs: I_0, I_1, I_2 data in; two address bits: A_0, A_1 – for 4 words. Note: a single address refers to all 3-bits of a word; you cannot subdivide or access individual bits.

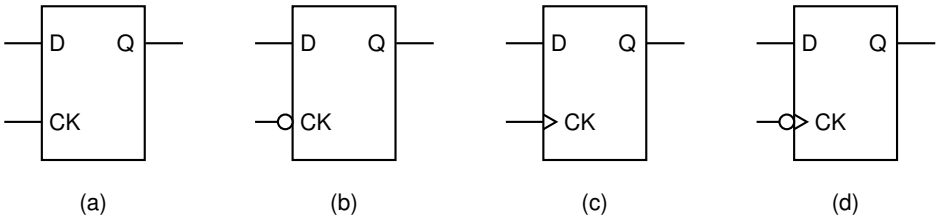


Figure 5.18: D-type latches and flip-flops

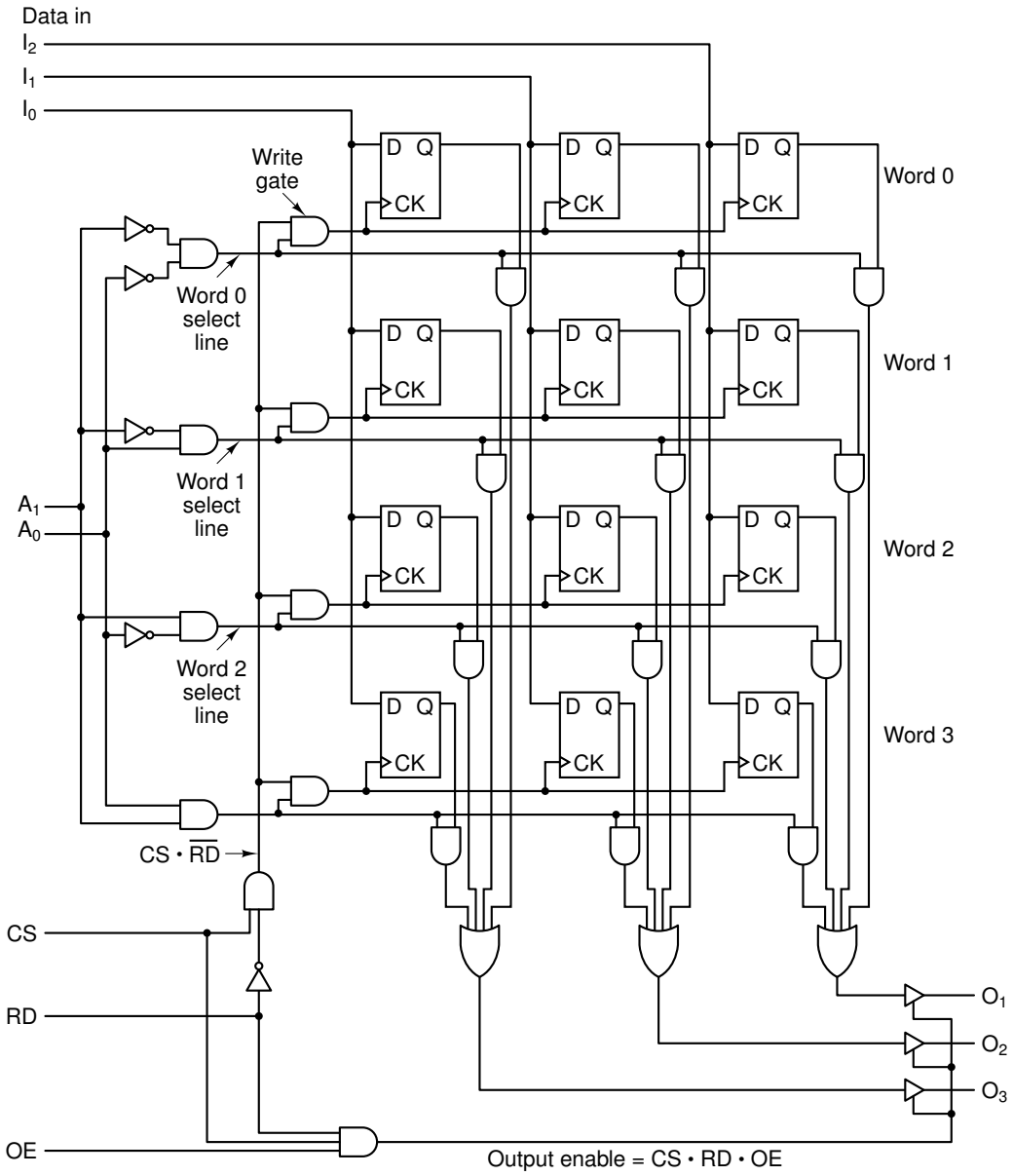


Figure 5.19: Four word three-bit per word memory

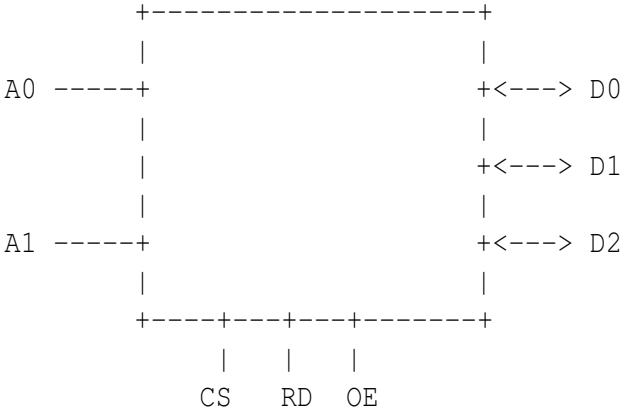


Figure 5.20: Four word three-bit per word memory showing bi-directional data bus

Outputs: $O_0,\ O_1,\ O_2$ data out.

Control: CS – chip select; RD – read, if 1 read if 0 write; OE – output enable.

Operation:

1. Write. $A0$, $A1$ are decoded (see section 5.1.3 and **and**ed with CS **and** $\overline{RD}$ to produce a clock CK on the appropriate row of 3-bits. I_0 is on the D-input of of all four bits (rows) of the first column; ditto $I_1,\ I_2$ on columns 1 and 2 respectively. CK enables the data into the flip-flop. Thus writing is finished.
2. Read. $A0$, $A1$ are decoded to select the appropriate row (word). The Q (output) data are enabled onto the O_i lines.
3. Slight Enhancement. Data in $(I_0,\ I_1,\ I_2)$ and data out $(O_0,\ O_1,\ O_2)$ are never used at the same time (verify as an exercise), so pins on a package can be saved by replacing these six with three, simply called Data $(D_0,\ D_1,\ D_2)$, used as in/out. What we need is an electronic switch that completely disconnects the output **or** gates when writing, or whenever OE is not set. **Tri-state** circuits – the triangular 'things' on the bottom right-hand corner – provide this facility; see section 5.5. In fact, the outputs of the **or** gates are connected to the outside world only when $CS.OE.RD$ is true.

   This is reflected in the schematic diagram (summary) shown in Figure 5.20.

5.4.1 Memory Mapped Input-output

With little more added, the decoding and enabling schemes shown in Figure 5.19 can be used to handle input-output **ports** as well as memory. Thus, some memory addresses are retained, e.g. FFF0 Hex to FFFF Hex, and when, for example, a memory write is performed to FFF0 Hex, this does not go to memory, but to an output port, which in turn is connected to some output device. Likewise, memory-mapped input.

5.4.2 Graphics memory

On PC compatibles, the screen graphics is represented by data in memory. This memory resides in the same *memory space* as ordinary memory. Hence, writing to appropriate (low) memory addresses can change the text or graphics on the screen.

```
Bit no.   15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
          +--++--++--++--++--++--++--++--++--++--++--++--++--+
          |1||0||0||0||1||1||0||0||0||1||1||0||0||1||1||0|
          +--++--++--++--++--++--++--++--++--++--++--++--++--+
```

Figure 5.21: A 16-bit register

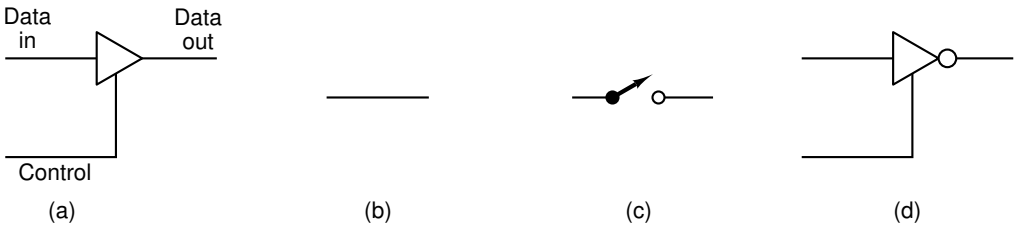


Figure 5.22: Tri-state buffer (a) Non-inverting (b) Effect when Control = 1 (c) = 0 (d) Inverting Buffer

5.4.3 Registers

Conceptually, a register is no different from a main memory word.

A 16-bit Register can store 16 bits, see Figure 5.21. Please note the numbering scheme for bits (now fairly universally agreed).

When we talk of CPU registers, we usually mean registers located in the CPU; these may be *special purpose* – and so not directly addressable by programmers –, or *general purpose*, in which case they are addressed by some alphanumeric code (e.g. AC, or X, or R1). Because of the speed of the memory making up the register, and/or its proximity, data transfer to and from a CPU register is normally an order of magnitude faster than main memory access.

5.5 Tri-State

Don't worry, you don't have to learn a new calculus of a logic with three logic states! Simply, as well as 1 and 0, a **tri-state** buffer can have a third state: **disconnected** or **open-circuit**, i.e. it operates purely as an electronic switch. A control line operates the switch. See Figure 5.22.

5.6 Buses

Crudely, a bus is a pipeline along which data can flow; but, usually, it's not as simple as connecting two devices with wires, usually the bus is multi-purpose and can be used by a number of devices; the devices must take their turn; thus, in addition to the physical connection, we need to define rules – these rules are called the bus **protocol**.

In general, you can have many devices connected by the same bus. Like a group of people talking – or even closer analogy, a group of people on a telephone conference – some order has to be preserved. Not everyone can talk at once. A bus has no problem with many listeners – but there can only be one talker at any one time – a bit like a lecture!)

Essentially, bus = physical connection + protocol.

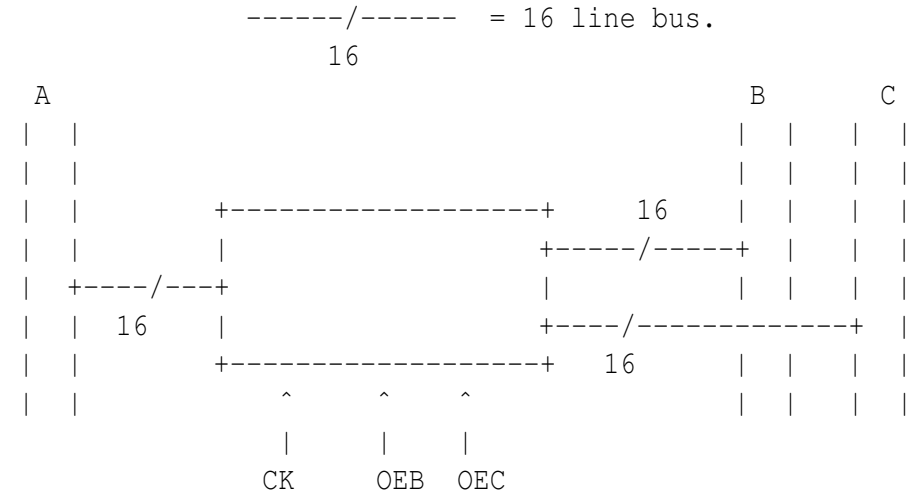


Figure 5.23: 16-bit register connected to buses A, B and C

The complexity of the protocol depends on how general purpose is the design of the bus. Internal computer buses can be simple; external buses, which may be used for printers, analogue interfacing etc., need a well specified protocol.

For the meanwhile, think of a bus as a pipe along which data flow, and access to the bus is controlled by a system of 'taps'; the taps are often in the form of multiplexers on the input to the bus, and tri-state buffers on the output.

Figure 5.23 shows a 16-bit register connected to buses C, A and B.

The signals OEB, OEC open tri-state buffers to allow, respectively, the contents of the register onto bus B, or bus C, or both; the controller which handles OEB, OEC ensures that this register will never be enabled onto any bus (B, or C) at the same time any other register is enable onto that bus – let us reiterate: a bus can tolerate no more than one talker.

Whilst one must use tri-state to output-enable each register that is connected to a bus, normally, a multiplexer is sufficient to select between multiple buses capable of writing to a register. In addition, a clock (CK) operates to select when the data actually gets clocked into the register.

Note the shorthand for a multi-line bus.

5.7 ROM and RAM

ROM = Read-only-memory. RAM = Random-access-memory.

But, unfortunately, the names are misleading. Both are in fact Random Access, or Direct Access – as opposed to Sequential Access. ROM is so-called because, usually, it is only ever written to once, and thereafter is only read (i.e. Read Only).

Random Access versus Sequential Access *Random access* means that you can access **any** memory sell on demand; you can read the addresses in any order;

Sequential access means that to get to memory address N, you have to read address 1, 2, 3, ... up to $N-1$, finally, N;

Sequential access is more common in disk and tape files; in that case, a record contains, not only its own data, but a pointer to the next record. In a random access disk file, there is a table, at the beginning of the file, giving the pointers to all records.

The circuit discussed in section 5.4 is a four word $\times$3-bit RAM; you can read and write, and both read and write are random access.

A ROM is logically quite similar, except you cannot write it in the normal way; it is fixed during manufacture; or it written with special equipment – PROM, Programmable ROM. ROM stays the same even when the power is off: it is **non-volatile**. Thus: RAM: Read/Write, volatile; ROM: Read only, non-volatile

5.8 Timing and the Clock

5.8.1 Introduction

Nearly everything in a computer happens on the rising (or falling – depends on convention) edge of a pulsed signal (the computer's equivalent of a 'beat' or orchestra conductor's hand signals. Thus, we need a **clock**, which produces a periodic sequence of pulses. The period – or cycle time – is typically around 0.01 microsecond, $0.01x10^{-6}$ secs which is 0.01 of a millionth of a second.

A *period* of 0.01 microsecs implies a clock rate (or *frequency* of 100 MHz. 100 MegaHertz is 100×10^6 cycles per second.

5.8.2 Frequency and Period of Periodic Events

In what follows times (t and T) are measured in seconds, and frequencies, f, in Hertz (Hz); Hertz is a synonym for cycles per second; Hertz is credited with the discovery of radio-waves.

The *period* – usually denoted T – of a periodic event (a repetitive event) is the time between the beginning of one event and the beginning of the next one; e.g. if the event 0 starts at $t = 0.0$, event 1 at $t = 0.1$, event 2 at $t = 0.2$, etc. the period $T = 0.1$ seconds.

The *frequency* (or rate) refers to the number of events that happen in *one* second; thus the events above have a frequency of 10 cycles per second, or 10 Hz.

It is a simple matter to convert from period (T) to frequency (f):

$$f = \frac{1}{T}\ Hz.$$

where T is measured in seconds. And,

$$T = \frac{1}{f}$$

We need shorthand for large frequencies, and small times; it is common practice to deal in multiples of 10^3:

- 10^{-3} secs.$= \frac{1}{1000} = 1$ millisec (msec); 1000 Hz = 1 KiloHz (KHz);

- 10^{-6} secs.$= \frac{1}{10^6} = 1$ microsec (μsec); 10^6 Hz $= 1$ MegaHz (MHz);
- 10^{-9} *secs.* $= \frac{1}{10^9} = 1$ nanosec (nsec); 10^9 Hz $= 1$ GigaHz (GHz);

Note the difference between **m** – **milli-**, 10^{-3}, and **M** – Mega, 10^6 !

Exercise. If the period is 0.002 μsec, or 2 *nsec*, what is the clock rate? Ans: 500 MHz.

Exercise. If you wanted a clock rate of 1000 MHz what is the period? Ans: 1$nsec$.

5.8.3 Clock Periods and Instruction Times

Q. If you had a 2 GHz PC, i.e. one on which the clock *beats* at a rate of 2000 million beats per second, could it do 2000 million adds per sec.? A. Definitely *no*. There are two main reasons:

- First a minor one, if you look at the simple addition 'program' above, you will see that, if the program is to be useful, you need to load the operands from memory, and store the result; therefore any useful add will take a minimum of *three* instructions.
- Second, what about individual instructions? Look at the fetch-execute cycle (if this is not clear, ask for a more detailed explanation of *fetch-execute*), there are five steps in fetch, five in execute; typically each of these would take one clock cycle – a total of *ten*; therefore we are down to 35 million instructions per second; for most current processors, this is a sensible number. Then the three machine instructions above, which leaves us at about 12 million adds per second.
- Moreover, if the adds were within a Java program (interpreted), see chapter 14, you would have also to account for the cycles used in the interpretation of the *bytecode* by the Java Virtual Machine (an interpreter program).

5.9 Exercises

1. The following equation describes a **two**-input multiplexer: when A is $false$ the output is (the same as) D_0, when A is $true$ the output is (the same as) D_1: $D_0.\bar{A} + D_1.A$.

   (a) Verify this statement using a truth table.

   (b) Draw a diagram for a two-input multiplexer – see Figure 5.3.

   (c) A **four** input multiplexer will have four inputs – and one output; how many control lines (A, B, C, ...); again, see Figure 5.3.

   (d) Sketch a diagram for a four-input multiplexer.

2. Draw a diagram of a two-input multiplexer that uses on/off switches.

3. How would you use two **xor**s, an **or** and an bf inverter/not to create a circuit to compare two two-bit numbers; Hint: see Figure 5.11.

4. A single input decoder (see Figure 5.4 for a 3 input decoder) takes one input (say A) and has two outputs D_0, D_1; for $A = true$, $D_1 = 1$, and $D_0 = 0$; $A = false$, vice-versa; (a) Give the equations for D_0, D_1; (b) Based on Figure 5.4, draw a diagram of a 1-to-2 decoder.

5. (a) Draw a block diagram for a one-bit full adder: a rectangle with the (three) inputs coming in at the top, and the (two) outputs emerging from the bottom.

   (b) Hence, or otherwise, draw a block diagram for a two-bit full adder.

   (c) Hence, or otherwise, draw a block diagram for an n-bit full adder – only the lower two components, and the top one.

6. See Figure 5.8. (a) Convince yourself that the decoder subsystem (bottom left-hand corner) does its work properly; name the output lines, respectively from the top, $F_{and}, F_{or}, F_{compB}, F_{sum}$, and determine what values of (F_0, F_1) generate these signals.

   Note the use of **and** gates to *enable* the appropriate outputs into the output **or** gate.

   (b) ENA, ENB are used to *enable* (or the opposite – block) A and B into the logic and adder units; likewise $INVA$ is used to cause A to be replaced by $\bar{A}$, explain qualitatively, how, using both F_0, F_1 and $ENA, ENB, INVA$, you would generate: $\bar{A}$ **or** B, A 'straight-through', B 'straight-through', $\bar{B}$ 'straight-through'.

7. Verify that Figure 5.11 does in fact output 1 when the inputs are equal, 0 when unequal. Hint 1: **xor** is **true** if and only if the inputs are different. Hint 2: use de Morgan's law to show $\overline{X_0 + X_1 + X_2 + X_3}$ in terms of **and**, where X_i is the output of the ith **xor** gate.

8. Design a circuit (as simple as possible, hint, hint!) that will test if a 4-bit number is zero.

9. Examine the 4 word $\times$ 3-bit memory cell in Figure 5.19. It uses four rows $\times$ 3 columns of 1-bit cells; (a) how many rows and columns for 256×8; 1024×4.

10. The 4 word $\times$ 3-bit memory cell in Figure 5.19 uses 22 **and** gates and 3 **or**s. If the circuit were expanded to 256×8, how many of each?

11. (a) Draw a block diagram which summarises Figure 5.19; as inputs to the (i) left of the rectangle, show two address lines A_0, A_1; (ii) as input to the bottom of the rectangle show the three control lines CS, RD, OE, and, (iii) to the right, as bi-directional input-output, show three data lines D_0, D_1, D_3

    (b) Draw a similar diagram for 256×8 memory.

12. (a) How many address lines are required to address 64 Megabytes of memory?

    (b) How many memory locations can be addressed using 12 address lines.

13. As memory volumes on chips get larger, you need more address lines (n address lines for 2^n cells). Devise a method for using less than n address lines.

14. Implement a $8-bit \times 8-bit$ multiplier using a ROM. Describe: (a) Number of locations required; (b) Hence, number of address lines; (c) Number of bits in each word.

15. Find a type of memory that is **not** *random access*.

16. Buses. Explore the following. Family (of 4) A in Belfast, family (of 8) B in Dublin. Think of the steps that person A_1 must go through to speak to person B_5; and, within a pairwise dialogue, the manners (protocol) that must be observed in order for them to communicate effectively. Think of the telephone system providing a *bus*. Draw a diagram.

17. What sort of device/circuit is used to permit only one (of many) devices (e.g. registers) to put their data on a bus.

18. Refer back to Chapter 2. Assume that, in the bus connecting MAR and MBR with memory, we have have a 12-bit address part, and a 16-bit data part. However, more lines will be required – make some suggestions.

19. A normal arithmetic-and-logic-unit (ALU) has two inputs and an output, i.e. $input1 + input2 \rightarrow output$; what other input(s) and output(s) should an ALU have?

20. (a) On the original IBM PC (using an Intel 8086), the memory space was limited to 1 Megabyte. How, why? Actually, only 640K was available for normal program and data memory – what was the remainder used for?

21. (a) What is *cache* memory? (b) On modern processors, where is cache normally located? (c) How can cache improve the performance of a system? (d) In order to maximally benefit from cache, how should programs be organised? (as far as possible) – consider both program and data. We will return to this matter in later chapters.

22. What is the clock period of a 500 MHz Pentium III.

23. What does 1 GHz this mean in terms of clock period?

24. A computer is to read two numbers from memory, add them and place the result in memory. The memory is located one foot away from the processor. It is required to do the task in $\frac{1}{10}$ nanosecond. Why is this impossible; explain what is possible. Hint: What is the speed of light! Translate this to feet per nanosecond.

25. One we get towards clock periods of 1 GigaHz and beyond, some physical limitations will start to kick in. (a) What is the speed of light? (b) What is the fastest a (data) signal can move? What is the furthest data can move in 1 nanosec. (1×10^{-9}) seconds?

26. Describe the purpose of a CPU clock.

27. The period of a CPU clock is 0.5 nanoseconds.

    (a) in terms of fraction of a second, what is a nanosecond? (b) what is the frequency? (c) if we halved the frequency: (i) what would be the frequency? (ii) what would be the period?

Chapter 6

The Central Processing Unit (CPU)

6.1 Introduction

This chapter gives some more detail on the Central Processing Unit (CPU) and leads up to where we can write significant programs in assembly/machine code. First we will give an overview of how a processor and memory function together to execute a single machine instruction – the famous *fetch-decode-execute cycle*.

A CPU consists of three major parts:

1. The internal registers, the ALU and the connecting buses – sometimes called the *data path*;
2. The input-output interface, which is the gateway through which data are sent and received from main memory and input-output devices;
3. The control part, which directs the activities of the data path and input-output interface, e.g. opening and closing access to buses, selecting ALU function, etc. We will avoid going into much detail about the control.

A *fourth* part, main memory, is never far from the CPU but from a logical point of view is best kept separate.

We will pay most attention to the data path part of the processor, and what must happen in it to cause useful things to happen – to cause program instructions to be executed.

In the system we describe, the control part is implemented by *microprogram*, i.e. how the fetching, decoding and execution of a machine instruction can be implemented by execution of a set of sequencing steps called a microprogram. Note on terminology: the term *microprogram* was devised in the early 1950s (see (Ferry 2003)), long before microprocessors were ever dreamt of.

6.2 The Architecture of Mic-1

Figure 6.1 shows the data path part of our hypothetical CPU from (Tanenbaum 1990), page 170 onwards. Here, we briefly describe the components of Figure 6.1. Then we give a qualitative discussion of how it executes program instructions. Finally we describe the execution of instructions in some detail.

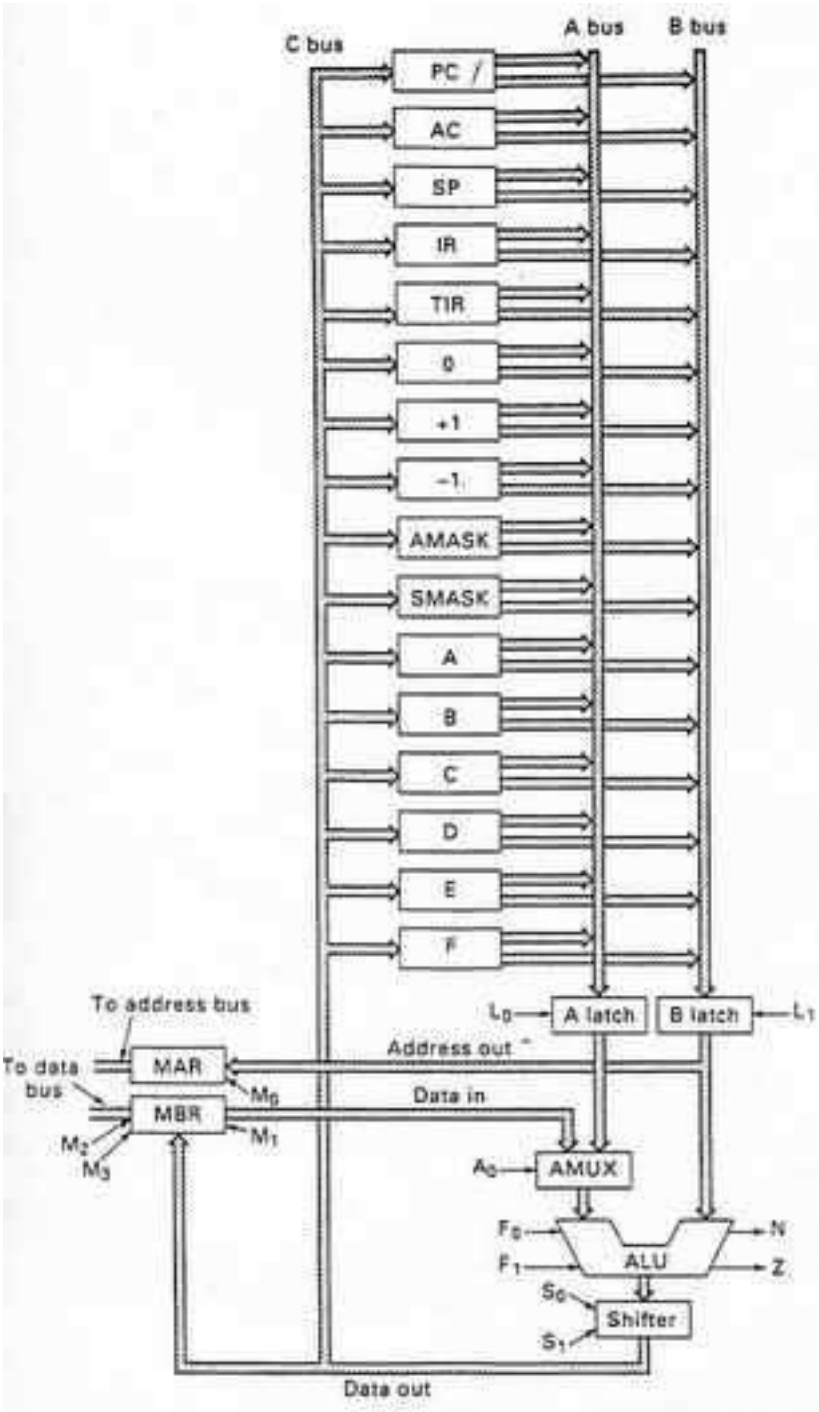


Figure 6.1: Mic-1 CPU (from Tanenbaum, Structured Computer Organisation, 3rd ed.)

6.2.1 Registers

There are 16 identical 16-bit registers. But, they are not general purpose, each has a special use:

PC, program counter The PC points to memory location that holds the next instruction to be executed;

AC, accumulator The accumulator is like the display register in a calculator; most operations use it implicitly as an unmentioned input, and the result of any operation is placed in it.

For now, we can ignore all the others, though we give brief descriptions below.

SP, stack-pointer Used for maintaining a data area called the *stack*; the stack is used for remembering where we came from when we call subprograms; likewise for remembering data when an interrupt is being processed; it is also used as a communication medium for for passing data to subprograms; finally, it is used as storage area for local variables in subprograms;

IR, Instruction Register Holds the instruction (the actual instruction data) currently being executed.

TIR, Temporary Instruction Register Holds temporary versions of the instruction while it is being decoded;

0, +1, -1 Constants; it is handy to have copies of them close by – avoids wasting time accessing main memory.

AMASK Another constant; used for masking (**and**ing) the address part of the instruction; i.e. AMASK **and** IR $\rightarrow$ address.

SMASK ditto for *stack* (relative) addresses.

A, B, . . . F General purpose registers; but general purpose only for the microprogrammer, i.e. the assembly language cannot address them.

6.2.2 Internal Buses

There are three internal buses, A and B (source) buses and C (destination) bus.

6.2.3 External Buses

The address bus and the data bus. Minor point to note: many buses, in particular those in many of the Intel 80X86 family, use the same physical bus (connections) for both address and data; it's simple to do – the control part of the bus just has to make sure all users of the bus know when it's data, when it's address.

6.2.4 Latches

A and B latches hold stable versions of A and B buses. There would be problems if, for example, AC was connected straight into the A input of the ALU and, meanwhile, the output of the ALU was connected to AC, i.e.. what version of AC to use; the answer would be continuously changing.

6.2.5 A-Multiplexer (AMUX)

The ALU input A can be fed with either: (i) the contents of the A latch; or (ii) the contents of MBR, i.e. what was originally the contents of a memory location.

6.2.6 ALU

In Mac-1a the ALU may perform just one of four functions:

0 $A+B$; note 'plus', rather than **or**; $(F_1,F_0)=(0,0)$;

1 A **and** B; $(F_1,F_0)=(0,1)$;

2 A straight through, B ignored; $(F_1,F_0)=(1,0)$;

3 **not** A; $(F_1,F_0)=(1,1)$.

Any other functions have to be programmed.

6.2.7 Shifter

The shifter is *not* a register – it passes the ALU output straight through: shifted left, shifted right or not shifted.

6.2.8 Memory Address Register (MAR) and Memory Buffer Register (MBR) and Memory

The MAR is a register which is used as a gateway – a 'buffer' – onto the address bus. Likewise the MBR (it might be better to call this memory *data* register) for the data bus.

The memory is considered to be a collection of *cells* or locations, each of which can be addressed individually, and thus written to or read from. Effectively, memory is like an array in C, Basic or any other high-level language. For brevity, we shall refer to this memory 'array' as M and the address of a general cell as x and so, the contents of the cell at address x as $M[x]$, or $m[x]$.

To *read* from a memory cell, the controller must cause the following to happen:

1. Put an address, x, in MAR;
2. Requests *read* – by asserting a *read* control line;
3. At some time later, the contents of x, $M[x]$ appear in MBR, from where, the controller can cause it to be ...
4. ... transferred to the AC or somewhere else.

To *write* to a memory cell, the controller must cause something similar to happen:

1. Put an address, x, in MAR;
2. Put the data in MBR;
3. Requests *write* – by asserting a *write* control line;
4. At some time later, the data arrive in memory cell x.

It is a feature of all general purpose computers that executable *instructions* and *data* occupy the same memory space. Often, programs are organised so that there are blocks of instructions and blocks of data. But, there is no fundamental reason, except tidiness and efficiency, why instructions and data cannot be mixed up together.

6.2.9 Register Transfer Language

To describe the details of operation of the CPU, we use a simple language called Register Transfer Language (RTL). The notation is as follows.

$M[x]$ denotes *contents* of location x; sometimes $m[x]$, or even just $[x]$. Think of an envelope with £100 in it, and your address on it.

Reg denotes a register; Reg = PC, IR, AC, R1 or R2.

$[M[x]]$ denotes contents of the *address* contained in $M[x]$. Think of an envelope containing another envelope.

We use $\leftarrow$ to denote transfer: $A \leftarrow B$. Pronounce this as 'A gets B'. In the case of $A \leftarrow M[x]$, we say 'A gets contents of x'.

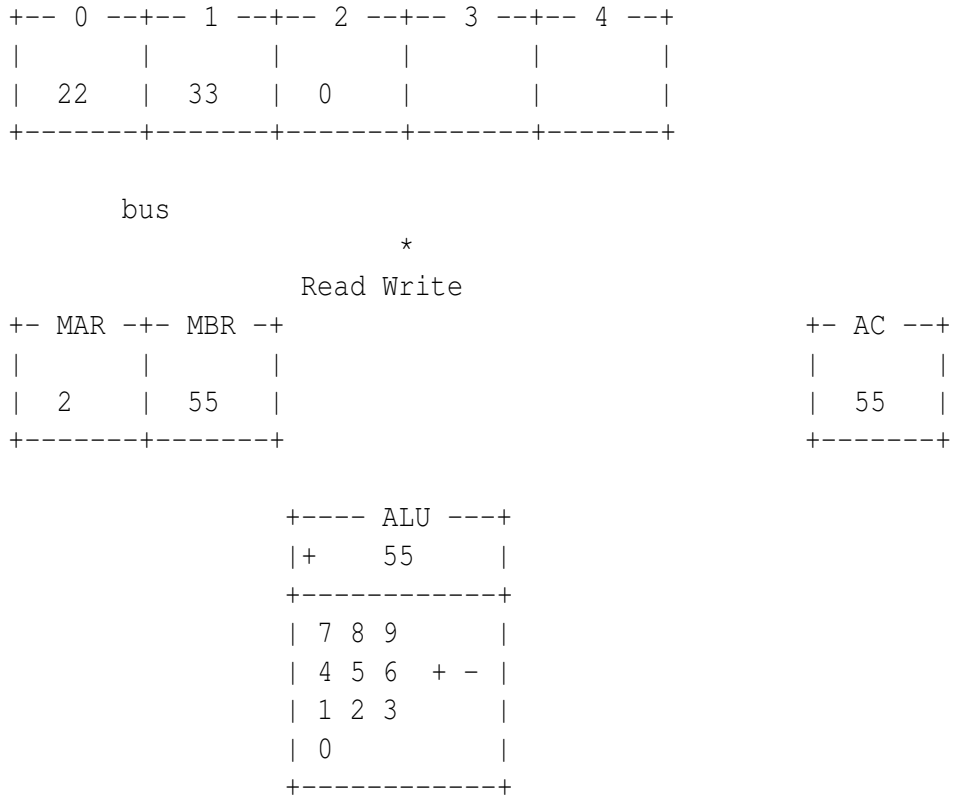


Figure 6.2: Mechanical Computer

6.3 Simple Model of a Computer – Part 3

Back in section 2.7, we produced a simple model of a computer. Here we show it again, Figure 6.2.

At the end of section 2.7 we admitted that we had been telling only half the truth! And we admitted that we had to fit the program into memory as well. Fine, here goes. Were going to use the same program.

In this more realistic model, the person operating the CPU has **no** list of instructions available on the desk, but must read one instruction at a time from memory.

Recall what was needed: add the contents of memory cell 0 to the contents of memory cell 1, store the result in cell 2; if the result is greater-than-or-equal-to 40, put 1 in cell 3, otherwise put 0 in cell 3. (We are adding marks, and cell 3 contains an indicator of Pass (1) or Fail (0).

And the program, with appropriate numerical code (so that instructions be stored in memory). The numerically coded instruction is given in four Hexadecimal digits; the first digit gives the operation required (load, add, store, . . .) – the *opcode*; the last three digits give the address or data – the *operand*.

The opcodes are as follows:

I have to renumber the program steps from P1–P14 to P101 . . . , for reasons which will soon become evident. Also, we will use hexadecimal numbering.

P101. Load contents of memory 0 into AC. Code: 0 000

P102. Add contents of memory 1 to contents of AC. Code: 2 001

P103. Store the contents of AC in memory 2. Code: 1 002

Operation	Code
Load memory	0
Store in memory	1
Add memory to AC and put result in AC	2
Subtract memory from AC and put result in AC	3
If AC is Positive, Jump	4
If AC is Zero, Jump	5
Jump (unconditionally)	6
Load constant	7

Figure 6.3: Opcodes

P104. Load the constant 40 into the AC. Code: 7 028 (40dec is 28Hex)

P105. Store the contents of AC in memory 4. Code: 1 004

P106. Load the contents of memory 2 into AC. Code: 0 002

P107. Subtract contents of memory 4 from contents of AC. Code: 3 004

P108. If AC is positive, jump to instruction P10c. Code: 4 10c

P109. Load the constant 0 into the AC. Code 7 000

P10a. Store the contents of AC in memory 3. Code 1 003

P10b. Jump to P10e. Code 6 10e

P10c. Load the constant 1 into the AC. Code 7 001

P10d. Store the contents of AC in memory 3. Code 1 003

P10e. Stop.

We now have to revise Figure 6.2 to show the program, Figure 6.4. The revisions are as follows:

- Show the additional memory (containing the program).
- Show a Program Counter (PC) register that keeps track of the address of the next instruction.
- Show the Instruction Register (IR); this tells the CPU operator what to do for the current step.

In this revised model, the CPU operator has no list of instructions on his/her desk (the CPU); he/she must go through the following cycle of steps for each instruction step:

Fetch (a) Take the number in the PC; (b) place it in MAR; (c) shout ”Bus”; (d) add one to the number in the PC – to make it point to the next step; (e) wait until a new number arrives in the MBR; (f) take the number in the MBR and put it in the Instruction Register;

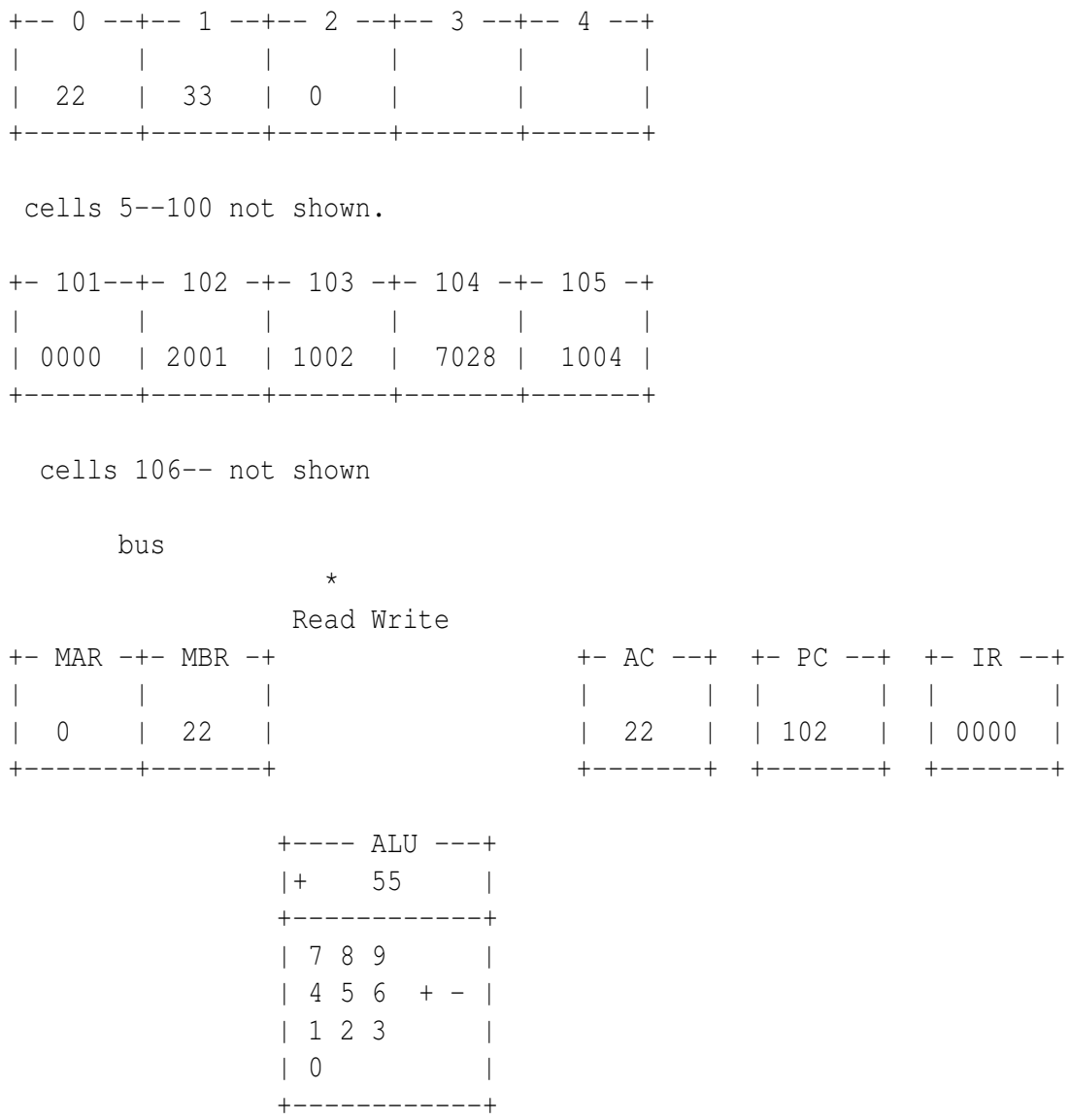


Figure 6.4: Mechanical Computer with Program

Decode (a) Take the number in IR; (b) Take the top digit (opcode), look it up in Figure 6.3, and see what has to be done; (c) take the number in the bottom three digits – this signifies the operand.

Execute Perform the action required. E.g. Add contents of memory 1 to contents of AC (2 001). Opcode is 2, operand is 001. We've already done this: (a) write 1 on a piece of paper and place it in MAR; (b) put a tick against Read; (c) shout "Bus"; (d) some time later, the contents of cell 1 (33) will arrive in MBR; (e) look at what is in AC and in MBR, use the calculator to add them (22 + 33); (f) write down a copy of the result and put it in AC. Thus, in the case shown, a piece of paper with 55 on it would be put in to AC.

If the operation is a jump, then all the operator does is take the operand (the jump-to address) and place it in the PC – thus stopping the PC pointing to the next instruction in sequence.

There we have it. The famous *fetch-decode-execute* cycle. The CPU is a pretty busy place!

6.4 The Fetch-Decode-Execute Cycle

How does the CPU and its controller execute a sequence of instructions? Let us start by considering the execution the instruction at location `0x100`; what follows is an endless loop of the so-called *fetch-decode-execute* cycle.

Fetch: read the next instruction and put it in the Instruction Register. Point to the next instruction, ready for the next Fetch.

Decode: figure out what the instruction means;

Execute: do what is required that instruction; if it is a JUMP type instruction, then revise the pointing to the jumped-to instruction. Go back to **Fetch**.

6.5 Instruction Set

We now examine the *instruction set*, by which assembly programmers can program the machine. We will call the machine Mac-1a; Mac-1a is a restricted version of Tanenbaum's Mac-1. The main characteristics of Mac-1a are: data word length 16-bit; address size 12-bits.

Exercise. What is the maximum number of words we can have in the main memory of Mac-1a? (neglect memory mapped input-output). How many bytes?

There are two addressing modes : *immediate* and *direct*; we will neglect Tanenbaum's local and indirect for the meanwhile.

It is accumulator based: that is, everything is done through AC; thus, 'Add' is done as follows: put operand 1 in AC, add to memory location, result is put in AC; if necessary, i.e. we want to retain the result, the contents of the AC is now copied to memory.

The Mac-1a programmer has no access to the PC or other CPU registers. Also, for present purposes, assume that SP does not exist.

A limited version of the Mac-1 instruction set is shown in Figure 6.5. The columns are as follows:

```
 ------------------------------------------------------------
 Binary         Mnemonic  Name                    Action(s)
 ------         --------  ----                    ---------
-------------------------------------------------------------
0000XXXXXXXXXXX  LODD   Load Direct              AC<-M[X]
0001XXXXXXXXXXX  STOD   Store Direct             M[X]<-AC
0010XXXXXXXXXXX  ADDD   Add Direct               AC<-AC+M[X]
0011XXXXXXXXXXX  SUBD   Subtract Direct          AC<-AC-M[X]
0100XXXXXXXXXXX  JPOS   Jump on pos.         IF AC>= 0 : PC<-X
0101XXXXXXXXXXX  JZER   Jump on zero         IF AC=  0 : PC<-X
0110XXXXXXXXXXX  JUMP   Jump uncond.             PC <- X
0111CCCCCCCCCCC  LOCO   Load constant            AC <- C
-------------------------------------------------------------
```

Figure 6.5: Mac-1a Instruction Set (limited version of Mac-1)

Binary code for instruction. I.e. what the instruction looks like in computer memory, *Machine code*.

Mnemonic. The name given to the instruction. Used when coding in *assembly* code.

Long name. Descriptive name for instruction.

Action. What the instruction actually does, described formally in *register transfer language (RTL)*.

6.6 Microprogram Control versus Hardware Control

Control of the CPU – fetch, decode, execute – is done by a microcontroller which obeys a program of microinstructions. We might think of the microcontroller as a *black-box* such as that shown in Figure 6.6. The microcontroller has a set of inputs and a set of outputs – just like any other circuit, ALU, multiplexer, etc. Therefore, instead of microprogramming, it can be made from logic hardware.

To design the circuit, all you have to do is prepare a truth-table (6 input columns - op-code (4 bits) and N, Z, 22 output columns), and generate the logic.

There is no reason why this hardware circuit could not decode an instruction in ONE clock period, i.e. a lot faster than the microcode solution.

The microprogrammed solution allows arbitrarily complex instructions to be built-up. It may also be more flexible, for example, there were many machines that users could microprogram themselves; and, there were computers which differed only by their microcode, perhaps one optimised for execution of C programs, another for COBOL programs.

On the other hand, if implemented on a chip, control store takes up a lot of chip space. And, as you can see by examining (Tanenbaum 1990), microcode interpretation may be relatively slow — and gets slower, the more instructions there are.

Figure 6.7 shows the full Mac-1 CPU with its microcontrol unit.

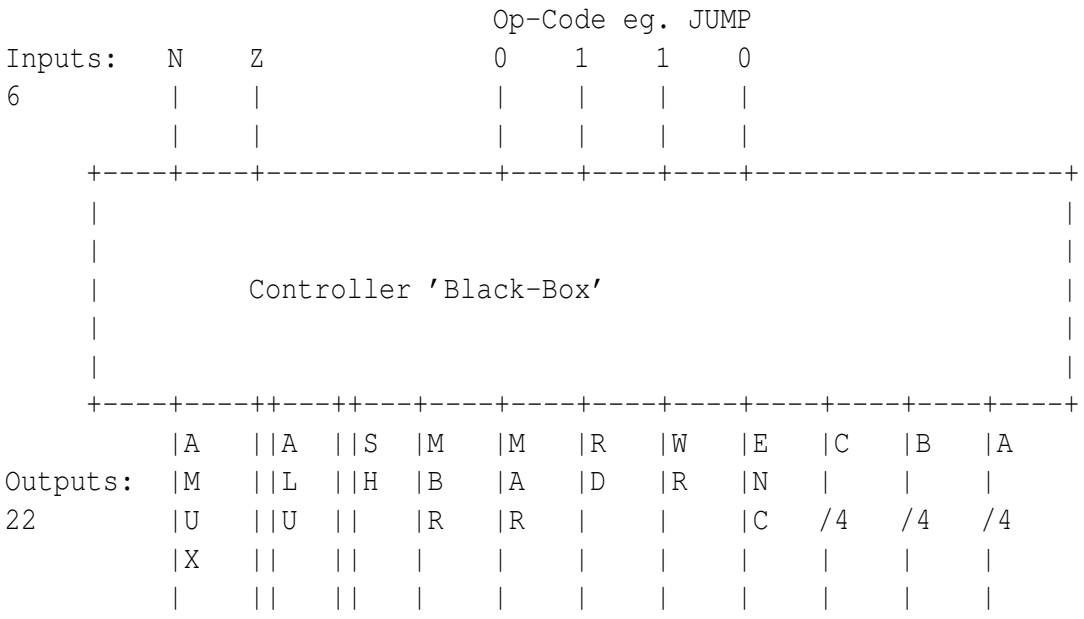


Figure 6.6: Controller Black-box, either Microcontroller or Logic

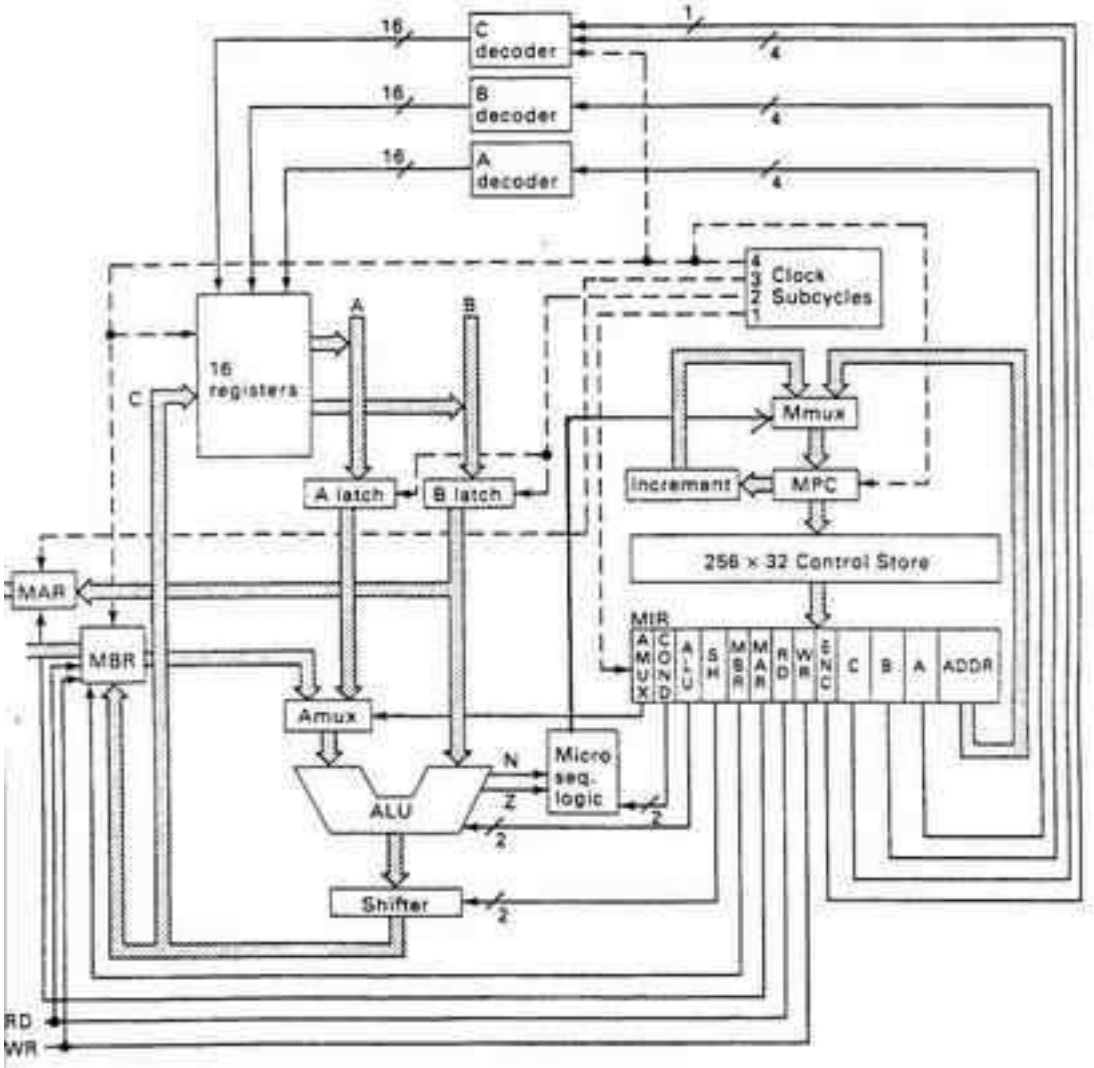


Figure 6.7: Mac-1 CPU including control (from Tanenbaum, Structured Computer Organisation, 3rd ed.)

6.7 CISC versus RISC

Machines with large sets of complex (and perhaps slow) instructions (implemented with microcode), are called CISC – **c**omplex **i**nstruction **s**et **c**omputer.

Those with small sets of relatively simple instructions, probably implemented in logic are called RISC – **r**educed **i**nstruction **s**et **c**omputer.

Most early machines – before about 1965 – were RISC. Then the fashion switched to CISC. Now the fashion is switching back to RISC, albeit with some special *go-faster* features that were not present on early RISC.

CISC machines are easier to program in machine and assembly code (see next chapter), because they have a richer set of instructions. But, nowadays, less and less programmers use assembly code, and compilers are becoming better. It comes down to a trade off, complexity of 'silicon' (microcode and CISC) or complexity of software (highly efficient optimising compilers and RISC).

6.8 Exercises

1. Consider the Mac-1a assembly code to do the equivalent of: `a0 = a1 + a2`:

   ```
   lodd a1
   addd a2
   stod a0
   ```

   Taking into account the fetch-execute cycle, and that there is a controller which also uses MAR and MBR, and assuming that the program starts at 100Hex (`lodd a1` is there), and that `a0, a1, a2` are at 100Hex, 101Hex, and 102Hex, respectively, describe precisely, and in order, all the data travel along the bus, to and from memory. Distinguish addresses and data.

2. Which Mac-1a instructions make use of the N, Z condition flags in their execution (I say execution to distinguish it from decode).

3. Some machines allow assembly programmers access to registers such as A, B, C ... in the Mac-1 CPU. Why might programs be speeded up by using these registers? For example, let us assume that there are instructions such as LODDA address, LODDB address, ... which cause registers a, B, etc. to be loaded instead of AC; also, corresponding ADDDA, ADDDB, which cause registers A, B, etc. to be added to AC.

4. If access to main memory is a bottleneck (the 'von Neumann bottleneck'), think of ways a alleviate the problem; hint: find a definition of (a) *cache memory*, (b) *pipelining*.

5. Using assembly language, show how to clear the accumulator on Mac-1A.

6. Many machines have a HALT instruction – which causes the machine to stop dead at the current PC address. Using assembly language, show how to halt Mac-1A, e.g. make it sit at PC = 100 effectively doing nothing.

7. Many machines have a NOOP – no-operation, i.e. the instruction wastes a bit of time and no other effect. Using assembly language, show how to program the same effect as a NOOP.

8. List and describe two major shortcomings of Mac-1A, i.e. the limited set of instructions in Figure 6.5.

9. Make a prioritized list of instructions you think would improve Mac-1A. Explain and give a rationale in each case.

10. Assuming that Mac-1a uses 16-bit twos complement to store signed integers, why is it impossible to LOCO 'load constant' minus 1, or indeed, any negative number. Suggest a method to circumvent this problem – which actually impacts positive numbers as well.

6.9 Self assessment questions

These are also *recall* type questions that appear as parts of examination questions.

1. Figure 6.1 depicts a Central Processing Unit. Briefly describe:

   (a) the purpose of the ALU; (b) the role of the MAR and MBR in CPU-memory interaction; (c) in addition to address, data, what additional information needs to be transferred via the 'system bus' between CPU and Memory; (d) the AC register; (e) the PC register; (f) the AMUX (A-multiplexer); (g) the A and B latches; (h) the F0F1 bits input to the ALU; (i) the N,Z bits output from the ALU.

2. Figure 6.1 depicts a Central Processing Unit.

   (a) Is it possible to write, in one step, to transfer the output of the ALU/Shifter to both the A and B registers?

   (b) Is it possible to write, in one step, to transfer the contents of both the AC and PC registers to the A Latch?

3. In the context of Figure 6.1 describe the fetch-decode-execute cycle.

4. In the context of Figure 6.1 and the instruction lodd 500 describe the fetch-decode-execute cycle.

5. In the context of Figure 6.1 and the instruction stod 500 describe the fetch-decode-execute cycle.

6. In the context of Figure 6.1 and the instruction addd 501 describe the fetch-decode-execute cycle.

7. Why must a normal arithmetic-and-logic-unit (ALU) have, in addition to one data output (the result), extra output(s)?

8. Why, in general, will a machine with *cache memory* run faster?

9. At any instant, a (data) bus can have many receivers/listeners, but only *one* active transmitter/talker; why? explain.

Chapter 7

Assembly Language Programming

7.1 Introduction

Chapter 6 showed how digital logic components can be connected together, and, with a bit of *glue* formed by the controller, can be made to perform useful elementary operations – like adding the contents of a memory location to the accumulator (ADDD).

We now come to constructing sequences of these elementary operations (ADDD, LODD, JUMP etc. ...) to do useful tasks, i.e. to *write programs* This chapter will concentrate on the basics: sequence, selection, repetition.

In addition we will describe the operation of a *two-pass assembler* and show how this can be done manually.

7.2 Programs in Assembly Code

We now want to write some programs for the Mac-1A. For completeness, we repeat Figure 6.5 as Figure 7.1. We can write in binary machine code, using Figure 7.1; but, that is tedious and error prone – because of the distraction of having to translate into a numeric code. It is better to use the mnemonics. The assembly code can then be *assembled* to machine code; this is done using a program called an *assembler* program.

For the purposes of this chapter, *assembly language* is a language in which there is one-to-one relationship between symbolic instructions and machine instructions; we will come across pseudo-instructions that are really instructions to the *assembler program*, but these will be obvious.

7.2.1 Conventions

In the following examples, if we need to mention actual addresses, we will start all code at $0x100$ and all data at $0x500$. Note again that the largest address is $0xFFFH$ (12 bits). Also, be careful with $0xFFFC$ onwards (4092_{10} to 4095_{10}, which are used for memory-mapped input-output, see Chapter 7.

Often, for brevity, we will assume variables that variables a0, a1, a2 ...(integers) have been declared and are already in locations 0x500, 0x501, 0x502, ... Alternatively, we will use symbols for data. Use these assumptions in the exercises where appropriate.

```
 ------------------------------------------------------------
 Binary         Mnemonic  Name                    Action(s)
 ------         --------  ----                    ---------
------------------------------------------------------------
0000XXXXXXXXXXX  LODD   Load Direct              AC<-M[X]
0001XXXXXXXXXXX  STOD   Store Direct             M[X]<-AC
0010XXXXXXXXXXX  ADDD   Add Direct               AC<-AC+M[X]
0011XXXXXXXXXXX  SUBD   Subtract Direct          AC<-AC-M[X]
0100XXXXXXXXXXX  JPOS   Jump on pos.         IF AC>= 0 : PC<-X
0101XXXXXXXXXXX  JZER   Jump on zero         IF AC=  0 : PC<-X
0110XXXXXXXXXXX  JUMP   Jump uncond.             PC <- X
0111CCCCCCCCCCC  LOCO   Load constant            AC <- C
------------------------------------------------------------
```

Figure 7.1: Mac-1a Instruction Set (limited version of Mac-1)

7.2.2 Simple Program – add two integers

Program. Add the integer in `a1` to the integer in `a0` and put the result in `a2`, i.e.

```
a2 = a0 + a1;
```

In assembly code:

```
Start:    LODD a0   /get a0 into accum.
          ADDD a1   /accum. gets a0+a1
          STOD a2   /store result in a2
```

There are *four* fields in an assembly instruction:

Label. Optional, only necessary if you need to jump to that instruction; can also be useful as comment, e.g. above. Always followed by ':', otherwise assembler will try to interpret it as an instruction.

Instruction mnemonic.

Operand I.e. address of thing to suffer the operation. Either:

- Symbolic name, e.g. above.
- Symbolic label, or numeric address, to jump to.
- Constant – for immediate addressing.

Comment Comments are even more important for assembly language programs than the are for high-level language programs. *Make your code maintainable by someone else!*

7.2.3 Declaring variables

For a real assembler, we would have to declare all variables. And, tell it where to start main program at. E.g. in the example above,

```
DW a0 0x10  /a0 is a WORD type, initialised to 10H
DW a1 0x22  /these are not executable instructions
DW a2       /but pseudo-instructions
            /ie. directives to the assembler to allocate
            /storage and associate names with that storage
            / just like you declare variables in Pascal

ORG       0x100      /start prog. at 0x100
```

7.2.4 If-then

High-level language

```
      if(a0==a1) a2 = a2 + 1;
      else  next instructions...
```

Assembly language

```
If:       LODD a0
          SUBD a1
          JZER Then
          JUMP Next
Then:     LOCO 1   /load constant 1
          ADDD a2
          STOD a2
EndThen:  JUMP Next  /Unnecessary
Next:     ...
```

7.2.5 If-then-else

High-level language

```
     if(a0==a1) a2 = a2 + 1;
     else a2 = a2 + 2;
     //next instructions...
```

Assembly language

```
If:       LODD a0
          SUBD a1
          JZER Then
Else:     LOCO 2
          ADDD a2
          STOD a2
          JUMP Next
```

```
Then:     LOCO 1
          ADDD a2
          STOD a2
EndThen:  JUMP Next  /Unnecessary
Next:     ...
```

7.2.6 Repetition

High-level language

```
//sum the first n-1 integers
int sum=0;
int i=0;
while(i < n){
  sum= sum + i;
  i++;
}
```

Assembly Language

```
init:    LOCO 0
         STOD sum
         STOD i
loop:    LODD i
         SUBD n
         JPOS loopend /          if i-n>=0,
                      / same as: if i>=n
                      / same as: if NOT i<n
         /otherwise continue with body of loop
loopbody:
         LODD sum  /load sum into accumulator
         ADDD i    /add i to it
         STOD sum  / and don't forget to store the result!
incr:    LOCO 1
         ADDD i
         STOD i
         JUMP loop
loopend: ...
```

Exercise. Draw a flowchart representing the program above.

It is straightforward to define deterministic loops, for example,

```
for(i = 0; i<n; i++){
  sum= sum + i;
}
```

in terms of `do-while`. Likewise `do-until` – the continuation test is merely carried out at the *end* of the loop, instead of at the beginning for `do-while`.

7.3 Machine Code, Memory Maps

Let us see how the *If-then-else* from the previous section would *assemble* into *machine code*. We use Figure 7.1 to translate.

For ease of understanding, we show the assembly language in the table. Assume that, initially, `a0` contains 8, `a1`, 4 and `a2`, 2.

```
<-- Assembly language --->  <------- Machine code ------------>
                             Address   Instruction/Data
If:       LODD a0            0100      0500
          SUBD a1            0101      3501
          JZER Then          0102      5107
Else:     LOCO 2             0103      7002
          ADDD a2            0104      2502
          STOD a2            0105      1502
          JUMP Next          0106      610B
Then:     LOCO 1             0107      7001
          ADDD a2            0108      2502
          STOD a2            0109      1502
EndThen:  JUMP Next          010A      610B
Next:     ...                010B      ...
                             ...
a0                           0500      0008
a1                           0501      0004
a2                           0502      0002
```

Thus, we can identify two separate sections of memory: (a) program, (b) data. Of course, there is nothing really separating these, and there is nothing to stop assembly programmers modifying instructions as data, nor, for that matter, runaway programs attempting to execute data!

Actually, as shown in Figure 7.2, in a C or C++ program, we can identify *four* sections of memory. Exact implementation detail may differ depending on compiler, machine architecture and operating system. In some contexts, Figure 7.2 is called a *memory map*.

7.4 The Assembly Process

Assembly is the process of translating the *symbolic* assembly code into *numeric* machine code.

Example Assemble (translate) the following assembly program into binary machine code. Assume a0, a1, . . . at 0x500, 0x501, ... Start code at 0x100.

```
DW a0  /shown here for completeness
DW a1
   ....
DW a10
DW one 1
```

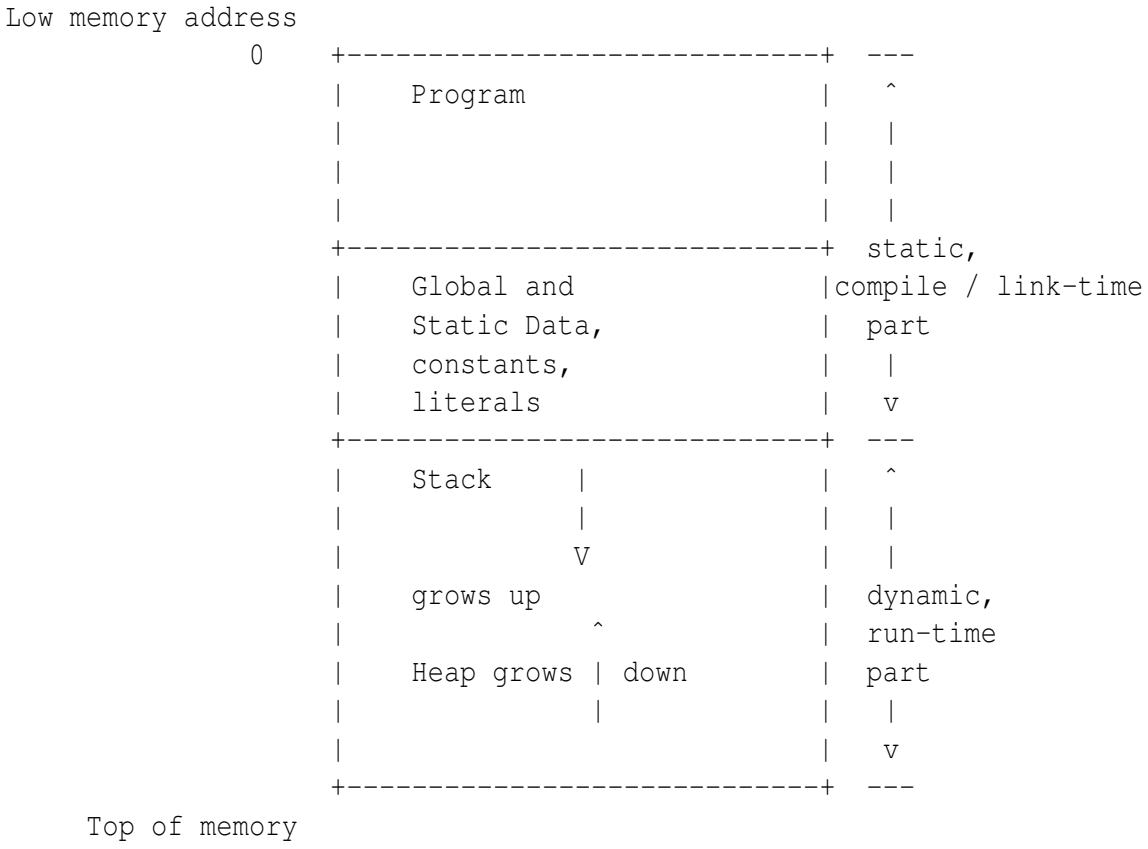


Figure 7.2: Memory Layout of a Program – Memory Map

```
If:       LODD a0
          SUBD a1
          JZER Then
          JUMP Next
Then:     LODD a10
          ADDD one
EndThen:  JUMP Next  /Unnecessary
Next:     .....
```

7.4.1 Two-Pass Assembler

This section describes how a two-pass assembly program works. We said that one assembly language instruction maps to one machine instruction; thus, it would appear natural to run through the list of assembly instructions and translate them using a table such as Figure 7.1.

This works fine until we get to `JZER Then`; what address is Then?

You don't know until you have assembled up to `Then`.

Therefore, we must have a *symbol table* - - which is a list of *(symbol, address)* pairs. If a symbol has been entered in the table, then you can look up its value. Initially, the table is empty.

Also we must have an operation-code (*op-code*) table – similar to the symbol table, only it is fixed, and contains a table relating instruction code (LODD, STOD, . . .) to the 4-bit *op-code* part of the instruction

```
     Symbol    Value     Other Information
     ------    -----     -----------------
     a0        500
     a1        501
     ... etc.
     a10       50a
     one       50b
```

Figure 7.3: Symbol Table – variables

```
     Assembly                                        Machine
Label Oper Oper      Comment        Instr.    ILC    OpC Oper
     ation and                      Length          ode  and
---  ----- ---       -------        ------    ----  --- ----
If:  LODD  a0                       1         100* 0
     SUBD  a1                       1         101  3
     JZER  Then                     1         102  5
     JUMP  Next                     1         103  6
Then:LODD  a10                      1         104* 0
     ADDD  one                      1         105  2
EndT:JUMP  Next     /Unnecessary   1        106* 6
Next:     .....                              107*
```

Figure 7.4: Partial Translation, pass one

(0001, 0010, ...). In general, this should also contain the length of each instruction, or some means of calculating it. Unlike most machines, Mac-1a has a single fixed instruction length of one; most machines have instruction lengths which vary according to the instruction.

7.4.2 Pass One

The main job of pass one is to build the symbol table.

1. Insert the data variables in the symbol-table, Figure 7.3;
2. Next go through the program to be translated, and build the ILC (instruction location counter); start at 100 (we assume that all programs start at 100) and cumulatively add each instruction length. See Figure 7.4. Note: I have abbreviated some label names.
3. Also, in this pass we might as well translate from symbolic op-code to numeric; see right-hand column of Figure 7.4;
4. Finally, add the label symbols – marked '*' in Figure 7.4 – to the symbol table; see Figure 7.5.

7.4.3 Pass Two

Now the symbol table is complete and we can complete the translation by filling in the operand fields; see Figure 7.6. Also, we may require the data area must be initialised – to zero or some appropriate default values.

Symbol	Value	Other Information
a0	500	
a1	501	
etc.		
a10	50a	
one	50b	
If	100	
Then	104	
EndT	106	
Next	107	

Figure 7.5: Symbol Table – at end of pass one – completed

Label	Assembly Operation	Assembly Operand	Comment	Instr. Length	ILC	Machine OpCode	Machine Operand
If:	LODD	a0		1	100	0	500
	SUBD	a1		1	101	3	501
	JZER	Then		1	102	5	104
	JUMP	Next		1	103	6	107
Then:	LODD	a10		1	104	0	50a
	ADDD	one		1	105	2	50b
EndT:	JUMP	Next	/ Unnecessary	1	106	6	107
Next:	.....				107		

Figure 7.6: Completed Assembly, pass two

7.5 Manual Assembly

Here in a slightly less formal way, is a recipe for assembling manually. We do it in a table, first write down the assembly code, the make two columns, one for *address*, the other for *instruction* or *data*. Then translate using Figure 7.1.

Do it in two passes. Leave Jump addresses blank first time round – you don't know the Jump destination addresses until after you have coded the Jump destination.

Use Hexadecimal for the machine code. If necessary, separate the binary fields in Figure 7.1 into groups of four – one for each Hex. digit, and add another column to Figure 7.1 to contain the Hex. representation of the operation code (*op-code*).

```
LABEL   ASS. INSTR.      ADDR.     BINARY INSTR.
-----------------------------------------------
If:       LODD a0        100       0500
          SUBD a1        101       3501
          JZER Then      102       5(Then)
          JUMP Next      103       6(Next)

Then:     LODD a10       104       050A
          ADDD one       105       250B

EndThen:  JUMP Next      106       6(Next)

Next:     .....          107
```

Now, we know that `Then = 104, Next = 107`. So we can complete the assembly:

```
LABEL   ASS. INSTR.      ADDR.     BINARY INSTR.
-----------------------------------------------
If:       LODD a0        100       0500
          SUBD a1        101       3501
          JZER Then      102       5104
          JUMP Next      103       6107

Then:     LODD a10       104       050A
          ADDD one       105       250B

EndThen:  JUMP Next      106       6107

Next:     .....          107
```

7.6 Linking and Loading

If the program above was viable to work on their own, it would be possible to load it into the addresses given: 0x100 to 0x107 for program and 0x500 to 0x50b for data, set the PC to 0x100, and the program would run.

Unfortunately, programs are usually made up from more than one module, and the results of each module translation – *object* files – must be linked together to produce an *executable* program.

Essentially, at the end of pass two, there will be symbols for which you do not have an address; these will refer to subprograms (see Chapter 7) that are contained in other modules. Also, the ILC will need to contain *relative* addresses, not *absolute* addresses as above.

In linking and loading the main jobs to be done are:

- Resolve references to external objects (symbols for which you have no value after pass two), these will be given in the symbol tables of another module.
- Relocate relative addresses to yield absolute addresses.

Refer also to Figure 7.2.

We further discuss linking (along with issues surrounding compilation and interpretation) in the course on Operating Systems, see the chapter at the end or miscellaneous topics.

7.7 Exercises

In the following, unless specified, assume the executable code is to start at is at 0x100, and `a0, a1, a2, ...` at 0x500, 0x501, 0x502, ...

1. Translate the following (separate) Java/C/C++ like instructions into Mac-1A.

   (a) `a10 = a0;`

   (b)
   ```
   if( a1==a0 ) a10 = 1;
   else         a10 = 0;

   if( a1>a0 ) a10 = 1;
   else        a10 = 0;

   if( a1<a0 ) a10 = 1;
   else        a10 = 0;
   ```

   (c) `a3 = a2 - a1;`

   (d) `a3 = a2*2;`

   (e) `a3 = a2*3;`

   (f) `a3 = a2*4;`

   (g) (more difficult) `a3 = a2 * a1;` (assume a2, a1 both less than 128).

   (h) `a0 = - a1;`

   (i) `a0 = - 1;`

2. Translate the following Java like code into Mac-1A.

```
int a,b,c,d,eq;

a = 16;
b = 32;
d = 64;
c = a + b;
if(c==d) eq = 1;
else     eq = 0;
```

3. Assuming `start` is at 100Hex, and `a0, a1, a2` at 500, 501, 502 (Hex), manually assemble the following:

```
Start:    LODD a0
          ADDD a1
          STOD a2
```

4. Manually assemble the following code.

```
DW a0  (at 500Hex)
 ...
DW a10 (at 50AHex)
DW x       50B
DW y       50C
DW z       50d

Start:    LODD a0
          ADDD a1
          STOD a2
          JZER Zero
          LOCO 0
          STOD x
          STOD y
          LOCO 10
          STOD z
          JUMP Lab1
Zero:     LOCO FF
          STOD x
          LOCO 1
          STOD y
          LOCO 2
          STOD z
Lab1:     ...
```

5. . What does the code in the previous question above do? Translate it into Java.

6. Manually assemble:

```
DW one 1 (one is at 50BHex)

If:       LODD a0
```

```
            SUBD a1
            JZER Then

Else:       LODD a10
            ADDD one
            ADDD one
            JUMP Next

Then:       LODD a10
            ADDD one

EndThen:    JUMP Next
Next:       .....
```

7. Assuming that Mac-1a uses 16-bit twos complement to store signed integers, why is it impossible to 'load constant' (LOCO) minus 1, or indeed, any negative number. Explain how you would, using LOCO and one other instruction, load a negative constant (say -1000 decimal) into the AC register.

8. Note the restriction on LOCO for positive numbers as well. Explain how, using LOCO and one other instruction, how you would cause the number 1000H to be loaded into the AC register.

9. Write any program that will modify continuously its behaviour by modifying its code – i.e. it treats some of its code as 'data'.

10. Here is a machine language program; usual layout, the program starts at 100; data variables a0, a1, ..., a10 are at 500, 501, 50a, respectively.

Addr.	Instruction
100	0500
101	2500
102	2500
103	1501
104	3502
105	4109
106	0505
107	1503
108	610c
109	0504
10a	1503
10b	610c

    (a) Disassemble it, i.e. translate back into assembly code;

    (b) What does it do? Ans:

```
a1 = 3*a0;
if(a1>a2) a3 = a4;
else      a3 = a5;
```

    Explain.

7.8 Self assessment questions

These are also *recall* type questions that appear as parts of examination questions.

1. Referring to Figure 7.1 convert the following into assembly language:

```
(a) a2 = a0 + a1;
(b) a2 = 32;
(c) a2 = a0 - a1;
(d) if (a0 == a1) then a2= 1;
(e) if (a0 == a1) then a2 = 1;
    else a2 = 2;
(f) if (a0 >= a1) then a2= 1;
(g) if (a0 < a1) then a2= 34;
(h) if (a0 >= a1) then a2= 1;
    else a2= 34;
```

2. Consider the following program.

```
loco 29
stod a1
loco 31
stod a2
loco 33
stod a3
lodd a1
addd a2
stod a3
stod a4
```

   When the program is completed: (a) What is in a3? (b) What is in a4? (c) What is in a1? (d) What is in a2?

3. Consider the following program.

```
loco 29
stod a1
loco 31
stod a2
loco 33
stod a3
stod a4
lodd a2
subd a1
stod a3
stod a4
```

   When the program is completed:

   (a) What is in a3? (b) What is in a4? (c) What is in a1? (d) What is in a2?

4. Consider the following program.

```
   loco 29
   stod a1
   loco 31
   stod a2
   loco 0
   stod a3
   stod a4
   lodd a2
   subd a1
   jzer then
   loco 100
   stod a3
   jump end
then loco 200
   stod a4
```

When the program is completed:

(a) What is in a3? (b) What is in a4? (c) What is in a1? (d) What is in a2?

5. (a) The following example shows an implementation of an IF ... THEN ... ELSE construct in Mac-1 assembly language. Add comments to each line of the assembly code, to explain how the assembly language performs the task, and replace the labels L1, L2, L3 with more explanatory labels (eg. THEN, ELSE, NEXT).

```
     IF a0>=a1 THEN a2:=0;
               ELSE a2:=1;
       next instructions...

If:       LODD a0
          SUBD a1
          JPOS L2
          JZER L2
L1:       LOCO 1
          STOD a2
          JUMP L3
L2:       LOCO 0
          STOD a2
L3:     .....
```

Chapter 8

Further Assembly Programming

8.1 Introduction

The instruction set (Mac-1a) introduced in chapter 7 was severely limited – particularly in its inability to call subprograms. We now extend our coverage to the remainder of the Mac-1 instruction set, and attempt some more ambitious programs.

Firstly, we will introduce the additional instructions - especially those based on the *stack*; we will explain the purpose of a stack and its use for handling subprograms. Next, we will show how some more real programs can be constructed, from simple three or four liners, to subroutines, input-output and interrupts.

In addition, we will describe the memory addressing modes found on most general purpose computers.

Although this chapter is entirely about Mac-1, the presentation is such that the principles of general-purpose computers are emphasised. Thus, someone who follows this chapter will have little difficulty in understanding Motorola 68000, Intel 80x86 series (Pentium), or virtually any existing computer.

8.2 Mac-1 Instruction Set Extensions

Figure 8.1 shows the Mac-1 instruction set extended to the full repertoire given in (Tanenbaum 1990); we do not bother with the binary version of the instruction – as in Figure 6.5 and Figure 7.1, since we will not be assembling programs using the additional instructions, nor writing them in machine code.

8.2.1 The Additional Jumps

```
jneg x    Jump on negative    if ac<0 : pc <- x
jnze x    Jump on nonzero     if ac!=0 : pc <- x
```

These save some of the trouble encountered using just `jpos` and `jzer`; however, as we have seen, they are not essential.

```
Mnemonic  Name                Action(s)
--------  ----                ---------
lodd x    Load Direct         ac <- m[x]
stod x    Store Direct        m[x] <- ac
addd x    Add Direct          ac<- ac + m[x]
subd x    Sub. Direct         ac <- ac - m[x]
jpos x    Jump on pos.        IF ac>=0 : pc <- x
jzer x    Jump on zero        IF ac=0 : pc <- x
jump x    Jump uncond.        pc <- x (0 <= x <=4095)
loco c    Load constant       ac <- c (0<=c<=4095)
lodl x    Load local          ac <- m[sp+x]
stol x    Store local         m[sp+x] <- ac
addl x    Add local           ac <- ac + m[sp+x]
subl x    Subtract local      ac <- ac - m[sp+x]
jneg x    Jump on negative    if ac<0 : pc <- x
jnze x    Jump on nonzero     if ac!=0 : pc <- x
call x    Call procedure      sp<-sp-1; m[sp] <- pc; pc <- x
pshi      Push indirect       sp <- sp-1; m[sp]<-m[ac]
popi      Pop indirect        m[ac] <- m[sp]; sp <- sp+1
push      Push onto stack     sp <- sp-1; m[sp] <- ac
pop       Pop off stack       ac <- m[sp]; sp <- sp+1
retn      Return from proc.   pc <- m[sp]; sp <- sp + 1
swap      Swap ac, pc         temp <- ac; ac <- sp; sp<-temp
insp y    Increment sp        sp <- sp + y; (0 <= y <= 255)
desp y    Decrement sp        sp <- sp - y; (0 <= y <= 255)

Notes:  1. c is a constant in range 0 - 4095
        2. x is an address in range 0 - 4095
        3. y is an address offset in range 0 - 255
```

Figure 8.1: Complete Mac-1 Instruction Set

8.3 The Stack

The *stack* is a special memory area. Although we give a lot of detail below, you will be able to get by knowing the major uses of the stack. In general, the stack is somewhere for the CPU to place a data item that it needs to memorise temporarily:

- When the CPU goes off to a subprogram, the CPU needs to be able to come back, so it stores the place where it departed from on the stack; this is just like you placing a marker at the current page in a book before you temporarily refer to some other part;
- See above, the calling program (the part of the program which calls/uses the subprogram) may need to pass some data to the subprogram; both the calling program and the subprogram can *see* the stack, so it is an ideal intermediate communication mechanism;
- Likewise, the subprogram may need to pass data back to the calling program;
- The subprogram (or any program) may need to create a temporary working data.

In Mac-1, the register SP is the *stack-pointer* and is dedicated to maintaining the stack; the stack itself – the data *pointed-to* – is actually part of main-memory.

The stack is a memory into which values can be stored and from which they can be retrieved on a last-in-first-out (LIFO) basis. Ideally, you store with a PUSH and retrieve with a POP. It may help to think of an analogy such as a spring loaded canteen tray dispenser, or a bus conductor's coin dispenser; the main point is that you can only put on the top (PUSH), get from the top (TOP) or remove the value at the top (POP). In spite of its simplicity this device has a remarkably large impact on the computational capability of a computer. A stack gives us a *sort-of* indirect addressing and also a *sort-of* indexed addressing via a the stack pointer; but a stack does much more than that, it is the basis of the implementation of functions and procedures, and blocks in block-structured high-level languages.

SP points to the top of the stack – i.e. to the memory location where the last value was *push*ed.

8.3.1 Direct Accumulator-Stack Instructions

```
push      Push onto stack     sp <- sp-1; m[sp] <- ac
pop       Pop off stack       ac <- m[sp]; sp <- sp+1
```

In the case of Mac-1, the stack grows from high memory towards low memory. `push` increases the size of the stack by one and places a value in the new memory cell (at the top). `pop` exactly reverses the process, i.e.. retrieves the last value written (the top) and decreases the size of the stack. `push` followed by `pop` has exactly *no* effect. And, as usual with Mac-1 most things are done through the accumulator (AC); `push` pushes the number in the AC, and `pop` removes the top of the stack and places it in the AC.

`push` operates as follows:

```
push      /sp <- sp - 1  ;SP decremented, NB. this INCREASES
                         size of stack.
          /m[sp] <- ac   ;put contents of AC into the memory
                         cell that SP POINTS TO
```

`pop` operates as follows:

```
pop       ;ac <- m[sp]   ;get contents of cell pointed to by
                         SP, into AC.
          ;sp <- sp + 1  ;decrease size of stack.
```

Note carefully again that the stack actually grows *down*wards, one word at a time – actually this is the case on a great many machines. Normally, in Mac-1 programs, we will assume that SP starts off pointing at memory cell 4020.

Example. The tables below show how the state of the stack and memory cells change, in response to the following code fragment, (assume SP initially set to 4020, and that a0 is at 500, and contains 30, that a1 is at 501 and contains 91):

```
            /(a)
lodd a0     /ac <- [a0] (=30)
push        /(b)
lodd a1     /ac <- [a1] (=91)
push        /(c)
pop         /ac <- m[sp]; sp <- sp -1
stod a0     /(d)
pop         /
stod a1     /(e)
```

In examples like this, to show the address of a memory cell and what it contains, we use the notation:

```
address: contents
500:      30
```

At the beginning (a):

```
      a0   500: 30
a1   501: 91
     AC : ?                    4018:      ?
                               4019:      ?
     SP: 4020 --points to---> 4020:      ?
```

At (b):

```
       a0   500: 30
a1   501: 91
     AC : 30                   4018:      ?
                             +->4019:      30
     SP: 4019 --points to--+  4020:      ?
```

At (c):

```
       a0   500: 30
a1   501: 91
     AC : 91               +-->4018:       91
                           |   4019:       30
     SP: 4018 --points to--+   4020:       ?
```

At (c):

```
       a0   500: 91
a1   501: 91
     AC : 91                   4018:      ?
                             +->4019:      30
     SP: 4019 --points to--+  4020:      ?
```

At (e):

```
       a0   500: 91
a1   501: 30
     AC : 30                   4018:      ?
                               4019:      ?
     SP: 4020 --points to---->4020:      ?
```

Comments:

1. The contents of A0 and a1 have been swapped; if we had wanted the same values POPped as PUSHed we would have had to POPped in the reverse order of PUSHing;
2. Once the SP moves back (after POP) we show ? in the stack area; the value would probably remain, but it would be exceptionally foolish to rely on this happening - as we shall see later when we mention interrupts.

8.3.2 Indirect Accumulator-Stack Instructions

```
pshi      Push indirect       sp <- sp-1; m[sp]<-m[ac]
popi      Pop indirect        m[ac] <- m[sp]; sp <- sp+1
```

Thus, the AC is used 'indirectly' – see *indirect* addressing mode, section 8.13; i.e. the value that is contents of the memory cell that is *pointed-to* by [ac] is pushed and popped.

8.3.3 Call and Return – CALL and RETN

`call` and `retn` are used for `call`ing subprograms (methods or functions in Java) and RETurNing from them.

8.3.4 CALL

`call` performs three steps.

```
call x    Call procedure   sp<-sp-1;     make room on stack for PC

                           m[sp] <- pc;  save PC

                           pc <- x;      put jump-to address in PC
```

`call` causes all of the following to happen:

1. Decrement the stack pointer – so that we will not overwrite last thing put on stack,
2. The contents of PC – which is pointing to NEXT instruction, the one just after the CALL – is pushed onto the stack, and,
3. Jump to 'x', which is the address of the start of the subprogram is put in the PC register, this is all a *jump* does. Thus, we go off to the subprogram – just as in `JUMP label`, but the important difference is that we remember where we were in the calling program, i.e. we must remember where we came from, so that we can get back there again.

8.3.5 RETN

`retn` performs two steps.

```
retn      Return from procedure

               pc <- m[sp]; take top of stack and put in PC

               sp <- sp + 1; decrease size of stack, i.e. delete what was
                                                          on the top.
```

`retn` causes all of the following to happen:

1. Pops the stack, to yield an address; if program is correct, the top of the stack will contain the address of the next instruction after the call from which we are returning; it is this instruction with which we want to resume in the *calling* program;
2. Jump to the popped address, i.e. put the address into the PC register.

8.4 Subprograms

8.4.1 Introduction

Note: In computer science, the terms *subprogram*, *subroutine*, *procedure*, *method* (in object-oriented programming languages such as Java), *function* (in C and C++) are largely equivalent. As mentioned above, `call` and `retn` are used for CALLing subroutines and RETurNing from them. One major objective of subroutines is to avoid having to repeat large chunks of code.

In section 2.6 we showed some cookery recipes. In those, notice how the writer of recipes can improve the readability of the cookery book by avoiding repetition of common *sub-recipes*, e.g. making a sauce, that crop up frequently, let's say 20 times. Not only does this decrease the size of the cookery book (19 half pages saved), but also increases readability of the book; in addition, it means that if the sub-recipe is to be altered, it need be altered only in one place, rather than 20.

I'll start off with a simple example; this example may fail to impress you; if so, imagine that it is something large an complex, e.g. reading a string from the keyboard, see section 8.10 — such that (i) you wouldn't want to type more than once; (ii) would use up a large amount of memory for each repetition; (iii) would, if such in here there and everywhere, would hinder the readability of the program; and, finally, (iv) if it ever had to be changed (e.g. from `y = x * 4 + 3;` to `y = x * 5 + 9;`), you would prefer to have to change it in one place only.

```
y = x * 4 + 3;
```

which can be written in assembly language as:

```
loco 3
addd x
addd x
addd x
addd x
stod y
```

However, for reasons that will soon become clear, I want to make this code a little more general (not specific to `x`); let's write a program that multiplies the AC by four and adds three, leaving the result in the AC.

Because we cannot add constants such as 3, we'll have to change the program a little and use a temporary variable `tmp`. In addition, we'll give it a label – so that we can jump to it when needed.

```
/ this program multiplies AC by four and adds 3, leaving the result in AC
m4p3      stod tmp         /200
          loco 3           /201
          addd tmp         /202
          addd tmp         /203
          addd tmp         /204
          addd tmp         /205
```

From now on, let us assume that `m4p3` is assembled and loaded at address 200.

Now, let us say we have:

```
          a2 = a1 * 4 + 3;
          a4 = a3 * 4 + 3;
          a6 = a5 * 4 + 3;
```

and we want to use just one copy of `m4p3`.

8.4.2 The Wrong Way – using JUMP!

A simplistic solution would be:

```
startprog: (assume startprog is at 100)
          lodd a1              /100
          jump m4p3            /101
          stod a2   /**        /102
          lodd a3
          jump m4p3
          stod a4
          lodd a5
          jump m4p3
          stod a6
```

But, without subprograms, the is a major problem: the program never gets to '**', because when it's finished `m4p3` it continues on to the next instruction *after* 205 and *not* 105 as desired.

Note, however, there is nothing to stop you (at 206) JUMPing to 102, but this defeats the whole purpose of the subprogram — you won't be able to use it for `a3` or anywhere else.

This gets us to one of the crucial differences between JUMP and CALL (subprogram). With JUMP, it's a one way ticket, you don't ever come back! With CALL you can remember where you came from and JUMP back there (using RETN) when you're finished in the subprogram.

8.4.3 The Correct Way – CALL and RETN

Figure 8.2 shows how we can make `m4p3` into a proper subprogram — all we need to do is add `retn` at the end; the label `m4p3` is all we need to name it.

```
/ this subprogram multiplies AC by four and adds 3, leaving the result in AC
m4p3      stod tmp         /200
          loco 3           /201
          addd tmp         /202
          addd tmp         /203
          addd tmp         /204
          addd tmp         /205
          retn
```

And here is how to `call` it.

```
          lodd a1
          call m4p3
          stod a2
          lodd a3
          call m4p3
          stod a4
          lodd a5
          call m4p3
          stod a6
```

Figure 8.2 shows the sequence of actions. Note: there is no *explicit* use of the stack, all PUSHes and POPs are done implicitly by CALL, RETN.

8.4.4 Subprograms with Parameters

Subprogram `m4p3` above uses AC to pass *parameters*. To make it completely general, we need to use the stack for passing parameters to the subprogram and returning results from it. In addition, the use of `tmp` is messy, and bad practice.

First, we must introduce load, store and arithmetic instructions that operate on stack memory.

```
lodl x    Load local          ac <- m[sp+x]
stol x    Store local         m[sp+x] <- ac
addl x    Add local           ac <- ac + m[sp+x]
subl x    Subtract local      ac <- ac - m[sp+x]
```

The term *local* come from *local variables – local* to the subprogram. From now on, if we want to pass something to a subprogram, we `push` it, and if we need store a value in a temporary variable, we also `push` it.

Each of these instructions allows you to access the memory cell `x` below the top of the stack; for example,

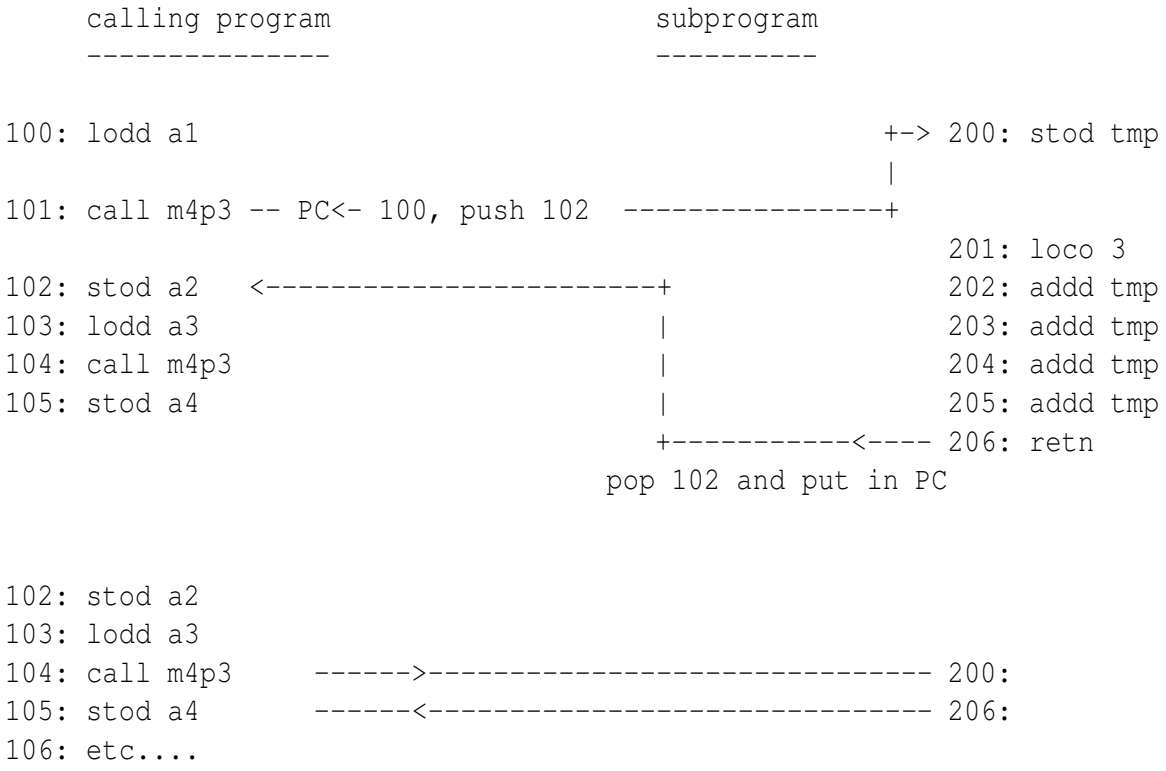


Figure 8.2: Subprogram Call and Return

```
          lodl 0   /loads into AC the most recently pushed value
                   /NOT the same as pop, as nothing is removed from
                   /the stack

          lodl 1   /loads into AC the last pushed but one
          stol 2   /stores value in AC into cell 2 from top
                   /NOT the same as PUSH, as no new space is
                   /created on stack, i.e. you may be overwriting
                   /something valuable.
```

Beware, when using the stack, it is easy to unintentionally overwrite important values, e.g. the return address of a subprogram, or to write to parts of memory that are not really part of the stack.

Now, we can revise `m4p3`.

```
/ this subprogram multiplies AC by four and adds 3, leaving the result in AC
m4p3      loco 3
          addl 1
          addl 1
          addl 1
          addl 1
          stol 2
          retn
```

Notice how the stack removes any need for named temporary variables. Question: why `addl 1` and not `addl 0`. Answer: because the return address is at 0 – it was the last thing `push`ed.

And here is how to `call` it.

```
loco 0      /actually, you could push anything
            /what is important is making space for the result
push        /make space for return value
lodd a1
push        /push input value
call m4p3
pop         /pop input value (remove from stack)
pop         /pop output value
stod a2
loco 0
push
lodd a3
push
call m4p3
pop
pop
stod a4
        etc...
```

8.5 Stack Frame

In connection with subprograms, there are *four* uses for the stack:

1. Passing parameters:
   (a) Passing parameters **to** the subprogram;
   (b) Returning values **from** the subprogram.
2. Storing the return address;
3. Saving the environment (registers);
4. Finally, all local variables are created on the stack.

In general, the stack looks like Figure 8.3. This is called the subprogram's *stack frame*.

```
    4020
High Memory ^
            |  Parameters
               Return Address
               Save area for registers (not used in our example)
               Local Variables.
```

Figure 8.3: Stack Frame

8.6 Recursive Subprograms

If you called the procedure `m4p3` *recursively* three times – or, indeed, there were three nested calls of different procedures – the situation would look like Figure 8.4. In this way a procedure can call itself again and again, without one call interfering with the other; the only limit is the size of the stack.

```
     4020
High Memory ^
            |  Parameters (in and out)
               Return Address
call 1         Save area for registers (not used in our example)
               Local Variables (not used in our earlier example).

               Parameters
               Return Address
call 2         Save area for registers (not used in our example)
               Local Variables.

               Parameters
               Return Address
call 3         Save area for registers (not used in our example)
               Local Variables.
```

Figure 8.4: Stack Frames for Recursive or Repeated Procedure Calls

8.7 Parameters Passed By Value

In the scheme mentioned in the previous two subsections, parameters are passed by *by value/copy*. Thus, subprogram `m4p3` can do whatever it likes to the memory location that contains the input values — the *copy* of `a1` (the parameter) is on the stack and `a1` itself are separate and so `a1` in the caller will never change; in fact, subprogram `m4p3` can treat it as a local variable.

8.8 Reentrant Subprograms

Subprograms which use the stack for passing parameters, and for their working (local) storage can be in use by more than one process at a time (e.g. in a multitasking operating system); such subprograms are called *reentrant*.

If a subprogram used global data, or used some local storage in its own program space – rather than using the stack, then the different (multiple and simultaneous) users of it would get their data mixed up.

Multitasking operating systems make much use of reentrant subprograms – there needs to be just one copy of the subprogram, even if it is being used by a great many processes.

8.9 Macros

As indicated above, a non-subprogram solution could have been used to repeat the `mp4p3` code as many times as was required. And as we mentioned, this repetition of code would have made the overall program larger, as well as other more serious problems.

Now, if we are content to accept the increase in program size, use of a *macro* avoids the other problems (i.e. more than one copy to maintain, difficulty of reading code with large chunks repeated).

Essentially, you declare a macro containing the working bits of the subroutine (no need for the house-keeping bits at the top and bottom) and then insert the macro code wherever the CALL appears. Macros are used whenever you want to trade memory for speed — you waste no time PUSHing and POPping the stack.

8.10 Input-Output Instructions

There are no *direct* instructions for input- output; instead Mac-1a uses memory-mapped input-output, whereby some memory cells are *mapped* to input-output ports; for simplicity we assume that there are only two ports, one connected to a standard-input device, the other connected to a standard-output device:

- Input, mapped to 4092 (0xFFC); status 4093 (0xFFD).
- Output, mapped to 4094 (0xFFE); status 4095 (0xFFF).

Note: recall that 0x signifies Hexadecimal.

We assume that each device works with bytes (i.e. 8-bits).

8.10.1 Input from standard-input device

A *read* from address 0xFFC yields a 16-bit word, with the actual data byte in the lower order byte. There is no use in reading the input port until the connected device has put the data there: so 0xFFD is used to read the *input status register*; the top bit (sign) of 0xFFD is set when the input data is available (DAV).

Thus, a read routine should go into a tight loop, continuously reading 0xFFD, until it goes negative; then 0xFFC can be read to get the data. Reading 0xFFC clears 0xFFD again.

8.10.2 Output to the standard-output device

Output, to 0xFFE, runs along the same lines as input. A write to 0xFFE will send the lower order byte to the standard-output device. The sign bit of 0xFFFH signifies that the device is in a *ready to receive* (RDY) state; again there is no use writing data to the output port until the device is ready to read it.

Example Write a fragment of program that will output the contents of the lower-order byte of address 500 to the standard output device mentioned in section 8.10.2.

```
testStatus:     lodd fff            /read status
                jpos testStatus     /not ready
                jzer testStatus     /not ready
      out:      lodd 500
                stod ffe            /output
```

Exercise Write a fragment of program that will read from the standard input device into 501.

Exercise Write a program that will output the contents of the lower-order bytes of addresses a0 to a9 (say, 500 to 509) to the standard output device - use the earlier example, and the previous examples on loops (chapter 6) as your building blocks.

Exercise Write a program which will:

1. Continuously read from the standard input device, *until*:
2. The number -1 (0xFFFFH) is received to signify END-OF-INPUT;
3. Send what was read to the standard output device.

Exercise Write a program which will:

1. Continuously read from the standard input device, *until*:
2. The number -1 (0xFFFFH) is received to signify END-OF-INPUT;
3. Add 1 to each number just read;
4. Send them to the standard output device.

Exercise Write a program which will:

1. Continuously read from the standard input device, *until*:
2. The number -1 (0xFFFFH) is received to signify END-OF-INPUT;
3. For each character read, check if it is in the range 'A' to 'Z', if so, make it lower-case, i.e. add 0x20 to it;
4. If it is not an upper case character, leave it alone;
5. Send it to the standard output device;

Exercise The example above:

```
testStatus:     lodd fff            /read status
                jpos testStatus     /not ready
                jzer testStatus     /not ready
      out:      lodd 500
                stod ffe            /output
```

is unsatisfactory for many purposes; what happens, for example, if the output device is broken, or is switched off, and, as a consequence *never* becomes ready; the program would stay in the tight loop and the only way to stop it would be to reset/reboot.

Change the code to count its 'not-ready' failures and, if this count ever reaches 'maxcount' (e.g. maxcount = 100) to put -1 in AC and JUMP to label 'exit'.

8.10.3 Polled I/O

The scheme of input-output outlined above is called *polled* input-output(I/O). Polled I/O is unsatisfactory for two major reasons:

- CPU time is wasted during I/O. Testing for ready status is a waste of CPU time. This cannot be solved by simple programming, it requires *interrupts*.
- A faulty I/O device can hang the system. What happens, for example, if the output device is broken, or is switched off, and, as a consequence *never* becomes ready; the program would stay in the tight loop and the only way to stop it would be to reset/reboot.

  One solution: change the code to count its *not-ready* failures and, if this count ever reaches `maxcount`, e.g. `maxcount = 100`, to put -1 in AC and JUMP to label `exit`.

A partial solution is as follows: control does not stay in the tight loop, but the CPU goes off and does other things, returning now and again to check status. But, interrupts provide the real solution.

8.11 Interrupts

[NB. Mac-1 has no interrupts — the following is modelled on interrupts on the 80X86].

As we have indicated in the previous section, it would be intolerable to have the CPU wasting its time constantly monitoring input (and output) status registers.

Consider the case of the simple case of a keyboard (GUI interfaces with a mouse present an even greater problem). In Windows and other operating systems the keyboard is read even when the computer is away running another part of the program. This is done with a special type of subroutine call – an *interrupt*.

When you hit a key on the keyboard, something like the following happens:

Hardware Actions Note: These hardware actions are invisible to the programmer; i.e. they do not need any assembly/machine code.

1. The keyboard is connected to a keyboard controller (a chip) on the motherboard. When you hit a key, two pieces of data become apparent to the controller: (i) a code identifying the key (including shift, control etc.); (ii) that a key was hit. The controller will store the code in a register;

2. The keyboard controller puts a '1' on an interrupt line on the system bus; if you look at the pin layout on a Pentium chip, you will see one pin labelled INTR (interrupt) and another NMI (non-maskable interrupt). In the discussion here, we will deal only with INTR. The difference between INTR and NMI is that NMIs will always be responded to whereas the CPU can *disable* INTRs using an instruction DI (disable interrupys), see below;

3. As soon as the CPU is prepared to handle the interrupt, it sends an acknowledge signal to the controller. Normally, during each *fetch* part of the *fetch-decode-execute cycle*, the CPU will check the interrupt line. Note: this means we have to add a little to *fetch*.

4. The keyboard controller sends a small integer (0–255), an *interrupt vector* to identify itself. There could be many interrupting devices: keyboard, mouse, disk, ethernet card, etc., each with different interrupt servicing software subprograms (drivers), so the CPU must first identify the device in order to be able to select the appropriate subprogram;

5. The CPU reads and stores the interrupt vector; in effect, the interrupt vector is the address where the address (*address of address* of the interrupt service routine (ISR) is held; ISRs are like other subprograms, their *address* is where their first instruction is held.

   Why *address of address*? The answer is simple — for flexibility. It is nice not to force OS manufacturers to use fixed addresses for ISRs. With the interrupt vector containing a fixed address that points to a variable address, we have the flexibility of changing the variable address. In fact, low memory (addresses 0–1023) is reserved for a 256 entry interrupt vector table (IVT); four bytes (32 bits) per interrupt vector.

   In other words, the interrupt vector of the keyboard never changes, but the driver software (ISR — subprogram) may be placed anywhere as long as the address in the interrupt vector table (IVT) is kept updated.

6. The CPU PUSHes the PC onto the stack. That is, a normal action before you jump to a subprogram (see CALL);

7. The CPU PUSHes the condition flags N, Z, onto the stack; in fact, in a Pentium, there is a FLAGS register and it is this that is saved on the stack. Why? Consider the following:

   ```
   lodd a1
   subd a2   /compare a1 and a2
   jzer xyz  /if equal jump to xyz, i.e. if Z flag is 1, jump
   ```

   The CPU can be interrupted just as it is about to fetch `jzer xyz`; it then goes off and services that interrupt (executes the ISR); what was in flag Z will have been overwritten, and a wrong decision to jump or not jump may occur on return from the ISR.

8. The CPU multiplies the interrupt vector by four to get an address in the IVT; the interrupt vector is a number in the range $0 \ldots 255$ and we want an address in $0 \ldots 1023$. Another way of looking at this is that each IVT entry if four bytes, hence interrupt vector 0 points to address 0, interrupt vector 1 points to address 4, interrupt vector 2 points to address 8, etc.

9. The CPU takes the address of the *interrupt service routine* from the entry in the IVT. That is, let's say the interrupt vector is 1; the CPU goes to memory cell 4, (and 5, 6 and 7) to get a 32 bit number (e.g. `0x 0070 07FB`); `0x 0070 07FB` is the actual address of the interrupt service routine;

10. The interrupt service routine is CALLed (but implicitly); i.e. in the example above, we have the equivalent of `CALL 0070 07FB`

Software Actions We are now in the interrupt service routine; i.e. this part *is* programmed.

1. The first instruction is usually DI (disable interrupts); i.e. prevent the CPU getting confused by being interrupted while it is handling an interrupt;

2. PUSH (save) all CPU registers (accumulators) onto the stack; in Mac-1, this means only the AC; however, in a Pentium, there are many accumulators. Why do this? The reason is the same as the reason given above for the FLAGS; when you return from servicing an interrupt, you must ensure that the CPU is left exactly as you left it. However, if an interrupt service routine is certain that it will not modify a register, then it need not save it;

3. Read the input port — in the normal way. If things are designed properly the device will always be ready after it has generated an interrupt; however, there is nothing to stop the ISR (interrupt service routine) first checking the status register;

4. Put the data in a buffer (memory area — an array), and maintain the buffer pointer; all this means is *put the input character somewhere where the reading program can get at it*;

5. The interrupt service routine may need to tell the keyboard controller that it has completed servicing the interrupt;

6. Restore (POP) all saved registers;

7. Execute a return-from-interrupt (IRET on an 80x86): (i) IRET restores (POP them and write their values into N and Z) FLAGS and (ii) POPs the return address; i.e. same as RET, with the additional step of restoring the FLAGs.

IRQs – Interrupt ReQuests It would be easy to say that IRQs are the same as *interrupt vectors*; that is close to the truth, but not quite. In fact, all interrupts go through an intermediate devise, a PIC (Programmable Interrupt Controller); when a device wants to interrupt, it sends its IRQ (number) to the PIC; the PIC translates IRQ to interrupt vector and passes the request to the CPU. The story then continues as above.

Transparency of Interrupts A key factor is the *transparency* of interrupts. The interrupt causes the service routine to run, but when that routine is finished, and IRET executed, the executing program should be none the wiser — except, maybe, it notices that an instruction took 20 or 30 μ-sec to run, instead of just 1 μ-sec; and, of course, there will be another character in the input buffer.

Exercise In a certain computer system the time taken for the processor to recognise and acknowledge an interrupt is 4 microsec.; it takes 10 microsec. to save the PC and flags register, ditto 10 microsec. to restore them. If the execution time for the interrupt handler instructions for the peripheral device is 70 microsecs.:

(a) what is the total time for each interrupt?

(b) estimate the highest interrupt frequency that may occur?

Assume that there are **no** other generators of interrupts.

8.12 Direct Memory Access (DMA)

In addition to the term *polled I/O*, we have the term *programmed I/O*; programmed I/O refers to the practice of the CPU reading each byte into the accumulator (LODD) and then storing it in memory (STOD). This is inefficient: (i) like *polled I/O* it keeps the CPU occupied rather inefficiently; (ii) all data must pass twice through the system bus. Programmed I/O may be used in an interrupt service routine.

Devices like disk or tape may require very rapid data transfer of data from the device to memory and vice-versa; we cannot tolerate any inefficiency.

In addition, if you look at the system diagram in Figure 8.5, you will see that both the disk (for example) and memory are connected to the system bus. Hence, there may be little requirement for the CPU to get involved in a data transfer, except to initiate it. Typically, a third device will be connected to the bus – a DMA controller – which mediates between the two data transfer devices and ensures an orderly use of the bus. In this case, the data passes through the system bus only once.

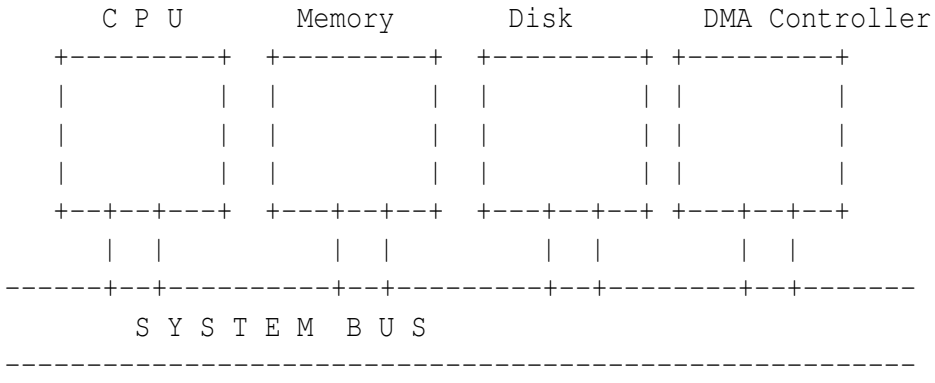


Figure 8.5: System with DMA

Recalling Mac-1a, and memory read/write: during a LODD, for example, Mac-1a would place the address to be read in MAR, and issue an RD. In a machine supporting DMA, DMA would be able to capture the bus (since I/O devices can lose data if they are kept waiting, DMA would have highest priority access to the bus). In addition, the CPU would need to use some sort of protocol to determine when data finally arrives following an RD.

When DMA is in operation, the only effect on the CPU and executing program is some slowing down, because the DMA must *steal* bus cycles — so called *cycle-stealing* — for the data transfer. There is only one bus, so traffic must take its turn. The degree of CPU slowing down will depend of how fast the DMA transfer can take place. If the DMA transfer can use the full data rate of the bus, then the CPU will stop for a while.

8.13 Addressing – General

In a computer, memory locations can hold *instructions* or *data*. In addition, as we shall see the *data* can be interpreted either as a plain *value*, e.g. 100, or as an *address* or *reference* to another data item. Those of who are familiar with C/C++ will recognise *pointers*; and Java people will recognise *references*.

In general, machine instructions usually take zero, one, or two *operand*s (e.g. in `lodd a0; a0` is the single operand; `lodd` is the *operation*;

Actually Mac-1 has no multi-operand instructions. For a start, it is an *accumulator* machine, i.e. in instructions like `addd, lodd`, the second – implicit – operand is AC, the accumulator.

Operands can be *data*, or can refer to data – i.e. address of data, or can be labels – which translate to addresses – of *instructions*, e.g. for `jumps`.

The question of *addressing* is concerned with how operands are interpreted. In the case of data the operand can be:

- A so-called *immediate* constant, e.g. LOCO 10;
- A value held in a register – no example in Mac-1;
- A value held in memory, e.g. LODD a0;
- In addition we can *qualify* the operand to say in which *mode* it should be interpreted:

  Immediate: When an operand is interpreted as an *immediate* value, e.g. LOCO 5, it is the actual value '5' that is put in the accumulator. In the case of Mac-1, the value '5' is part of the instruction: `LOCO 5; 0111 0000 0000 0101`;

  Direct: the specified name or number is the 'direct' *address* of the intended operand, e.g. LODD a0, which is the same as LODD 501 – if our usual definitions apply –, so that the contents of memory cell '501' are loaded into the accumulator. Thus, `0000 0101 0000 0001 ; lodd 501`;

  If memory cell 501 contains '7'

```
     +--------------+
501: | ... 0111     |
     +--------------+
```

  it is the '7' that gets loaded into the AC.

  Register: the number or name of a register is given, which contains the real operand; conceptually similar to *direct* except that the register name/number is substituted for memory address;

  Indirect: the specified name or number *contains* the *address of* the operand, i.e. the operand is the 'indirect' address of the true operand, e.g. let us invent LODI 501, so that the *contents of the contents of* memory cell '501' are loaded into the accumulator. Thus, `xxxx 0101 0000 0001 lodi 501`; if memory cell 501 contains '50a', and memory cell 50a contains '3', we have:

```
     +--------------+
501: | 50a          |
     +--------------+
       ...
```

```
     +--------------+
50a: | 3            |
     +--------------+
```

and it is the '3' that gets loaded into the AC.

Indexed: Lets us assume that we have an integer array stored in an array of N (= 9, say) contiguous locations starting at location 1230 (say), and we want to successively add them into the accumulator:

```
          lodd 1230
loop:     addi 1230+SI   ;ac<- [ac]+m[1230+si]
          inc  SI
          ...test for end
          jump loop
```

Of course, because Mac-1 has no index registers, I have had to invent one (SI), and an addressing mode for add: addi – add direct using SI as an index; also I had to invent an 'increment instruction. Indexing is a bit like running your finger along a table of figures. The instructions 'lodl'...'subl' introduced below are a bit like indexed – but they use a very special index register called the stack-pointer (SP).

Addressing modes look complicated, but if you are careful to analyse what a construct means – by drawing a diagram, if necessary – then there are no real pitfalls.

Also, for those who are not specialists in assembly programming, you should keep to the simple modes and only use the complex modes when they are absolutely essential.

8.14 Exercises

1. Comparison of the Mac-1 instruction set with those of other machines, e.g. Intel 80X86 (Pentium II is equivalent to 80686).

   (a) Mac-1 has instructions with, at most, one operand, on a computer like the 80X86, instructions often have two operands. Think about operations that could use 3 or more operands – write them down, with justification. Are such instructions a good idea – or bad; give a two or three point discussion.

   (b) Research and design an instruction set that has no *operands* at all, except for stack operations PUSH and POP. Write up your findings (one A4 page or less).

2. Figure 8.1 gives a list of Mac-1 instructions. Do some research on the 8086/8088/80X86 or the 68000 series and choose *six* more of your favourite instructions that you want included in the Figure. List them with description and justification – what would you use them for and how they would help

3. Describe using pictorial illustrations, and examples, the following addressing modes: (a) immediate; (b) direct, (c) indirect.

4. Given the memory values below and a one-address machine with an accumulator, what values do the following instructions load into the accumulator? Illustrate your answer with picture(s).

```
addr. 20 contains 40
addr. 30 contains 50
addr. 40 contains 60
addr. 50 contains 70

(1) load immediate 20
(2) load direct 20
(3) load indirect 20
(4) load immediate 30
(5) load direct 30
(6) load indirect 30
```

5. Calling subprograms.

   (a) Sketch the start of a subroutine that has 4 (16-bit integer) arguments passed to it. Also, show that CALL in the calling program and the few instructions preceding the CALL.

   (b) Draw a picture of the stack just before the subroutine gets down to its work.

6. Explain how an *index* register may assist in the handling of arrays; use as an example the case where you wish to add 3 to the array of 10 numbers starting at address 0x500.

7. Write a procedure 'outch(c,maxtries)' that will write a character (the first argument) to the standard output device; the second argument should be 'maxtries' – the number of 'not-readies' that will be declared a 'time-out' failure; on success, it should return 0 in the AC, on 'timeout' -1 in AC; e.g.

```
c = 0x45;
error = outch(c,1000);
```

8. Use 'outch' in the previous exercise in writing a procedure that has *four* character parameters and will write these to the standard-output device, i.e. called as: `error = write4(a,b,c,d);`.

9. Write a procedure 'ch=inch(maxtries)' that reads from the standard-input device. 'ch' is returned in AC. It 'times-out' after 'maxtries' not-readies and returns -2 in AC. If it gets END-OF-INPUT, it returns -1 in AC. END-OF-INPUT is a special character which you check for – assume that you have a variable set to its value, or, if it makes it easier, assume the special value is 0) it returns -1 in AC.

8.15 Self assessment questions

These are also *recall* type questions that appear as parts of examination questions.

1. (a) Referring as necessary to Figure 8.1, explain the instructions `push, pop`.

   (b) Consider the following program:

```
loco 29
stod a1
loco 31
stod a2  /1
```

```
lodd a1  /2
push     /3
lodd a2  /4
push     /5
pop      /6
stod a1  /7
pop      /8
stod a2  /9
```

(i) after /1 what is in a1, a2, the AC register?

(ii) after /2 what is in a1, a2, the AC register?

(iii) after /3 what is in a1, a2, the AC register, top of stack?

(iv) after /4 what is in a1, a2, the AC register, top of stack?

(v) after /5 what is in a1, a2, the AC register, top of stack?

(vi) after /6 what is in a1, a2, the AC register, top of stack?

(vii) after /7 what is in a1, a2, the AC register, top of stack?

(viii) after /8 what is in a1, a2, the AC register, top of stack?

(ix) after /9 what is in a1, a2, the AC register, top of stack?

2. Referring as necessary to Figure 8.1, and using the following programs as examples, explain the chief difference(s) between 'jump' and 'call'. In the case of 'main1' what will be in 'a1' at /1?

```
main1: loco 22
       stod a1
       call sub1
       lodd a1    /1

sub1:  loco 1
       addd a1
       stod a1
       retn

main2: loco 22
       stod a1
       jump sub1
       lodd a1    /2

sub2:  loco 1
       addd a1
       stod a1
       retn
```

3. Explain what is mean by the term *reentrant* in *reentrant subprogram*? (See section 8.8.) Why is it important that subprograms in multitasking operating systems be reentrant?

4. What is the difference between a *macro* and a subprogram? (see section 8.9). Extensive use of macros would lead to *larger*, yet *faster*? executable programs. Explain.

5. Memory mapped input in Mac-1. The input port is mapped to address 0ffcH (Hex); a read from 0ffcH (Hex) yields a 16-bit word, with the actual data byte in the lower order byte; 0ffdH is mapped to the input status register; the top bit (sign) of 0ffdH is set when the input data is available (DAV). Reading 0ffc clears 0ffd again.

   Write a fragment of Mac-1 code that will read from the standard input device into location 1000; include appropriate comments.

6. (a) Explain why programmed input-output is inappropriate in all but specialised situations.

   (b) Describe a situation in which programmed input-output would be appropriate.

7. In the context of a machine instruction, explain the difference between *immediate addressing* and *direct addressing*. In Mac-1 assembly language, give three examples of direct addressing; give one example of immediate addressing. Hint: addd, lodd, loco, stod, subd.

8. What is meant by ROM? RAM?

9. Explain the difference between ROM and RAM.

10. Give one good reason why a computer system should have some ROM memory.

11. The following describes a simple memory mapped input-output scheme: " Mac-1a uses memory-mapped I/O, whereby some memory cells are 'mapped' to i/o ports; for simplicity we assume that there are only two ports, one connected to a 'standard-input device', another connected to a 'standard-output device':

    Input: data mapped to 4092/0xFFC (lower-order byte = data byte); status mapped to 4093/FFD (sign bit set denotes 'data available').

    Output, mapped to 4094/0xFFE (lower-order byte); status 4095/0xFFF (sign bit set denotes 'ready') . . . "

    (a) Briefly, explain the principle used in both 'read' and 'write' operations.

    (b) Outline a fragment of program (using pseudo-code or assembly code - see Figure XX) that will write the contents of the lower- order byte of address 500 to the output device mentioned in the first part of the question.

    (c) Explain, briefly, why *programmed* input-output is unsatisfactory for many applications and how interrupts can provide some remedy.

12. Referring to Figure 8.1, explain the operation of the Mac-1 instructions: `call` and `retn`; you should mention the roles of the stack and stack pointer; illustrate your answer with appropriate examples / diagrams. In your answer please mention the major reasons why `call` and `retn` cannot be replaced by simple jump instructions.

13. Referring to Figure 8.5, explain how DMA may improve the efficiency of a computer system.

14. (a) Explain, briefly, the principles of *programmed* I/O, *polled* I/O, *interrupt driven* I/O, and explain why *programmed* and *polled* are unsuitable in most cases.

    (b) In a certain computer system the time taken for the processor to recognise and acknowledge an interrupt is 4 microsecs; it takes 10 micro secs to save OR restore the PC and other registers. If the execution time for the interrupt handler instructions for peripheral X is 70 micro secs,

    (i) what is the total time for each interrupt?

    (ii) estimate the highest interrupt frequency that may occur?

    Assume that there are **no** other generators of interrupts.

15. When writing cookery recipes or other instructions, how may the concept of subprogram assist: (a) the readability of the instructions; (b) the overall size of the instructions; (c) the maintainability of the instructions.

Chapter 9

Introduction to Operating Systems

9.1 Introduction

An *operating system* is a program. When you switch on a computer (a computer system), the hardware ensures that the operating system is read in to the memory (RAM). This process is called *booting* — *bootstrapping*). Then control is transferred to the operating system.

I am tempted to say that, from then on, the operating system (program) runs all the time until the the computer is switched off. However, we all know that only one program can be running on a single processor at any time; and most computers operate with just one processor. What about the applications we want to run — our word processor, or spreadsheet, our games ...

We can revise that statement to say that the *operating system* is *in charge of the computer all of the time*. To show how a simple program (the operating system) can be *in charge*, or manages, the computer, yet relinquish the processor to other programs, is one of the major objectives of this course. Note: I use the term *in charge* because I want to reserve the term *control*; when we say *transfer of control* (to a program), we mean that the instructions of this program begin to execute on the processor – and the operating system (and other programs), for the moment, slide into the background.

A possibly helpful analogy is, if we decided to have a debate in class, and I decided I was chair of the debate. If I was efficient, I'd ensure that only one person spoke at once; I'd stop the speaker when his/her time was up; I'd be fair in my scheduling of speakers; under certain rules, I'd allow interruptions for points of clarification; I'd keep notes on who has spoken.

Already, in chapter 1, we have mentioned other viewpoints: operating system as a platform for executing software, operating system as a service provider, operating system as government, etc.

Let me give some examples of operating systems and the degree of management exerted:

- MS-DOS.

  When an application program starts running, it is fully in control until it decides to finish; in addition, if asked nicely, MS-DOS and its related software (BIOS - Basic Input-Output System) provide services for running programs, e.g. saving data to file, writing text on the screen, doing the donkey work of reading the keyboard, etc .... There is nothing to stop a rogue program taking over the processor for ever. In addition, MS-DOS is a so called *single user* operating system – only one program runs at any time (fine, I realise that this is always the case, but *multitasking*

operating systems Windows NT/2000/XP and Linux/UNIX can give an illusion of being able to allow more than one program at a time to run.

In addition, MS-DOS will allow a program, itself, to access hardware (write to the screen, read a keyboard, write to a disk file, ...; Windows NT/2000/XP and Linux/UNIX will not.

- Windows NT/2000/XP, Linux/UNIX.

  These operate much more as very strictly controlling managers or chairpersons. A program is allowed control of the processor only for a limited amount of time, then, if another is waiting, it gets control – again for a limited period, etc ....

  You may ask, once a program is running, how does the OS stop it, if it has to? The answer lies in *interrupt*s. In the chairperson debate analogy, interrupts provide a brute-force method of shutting up the current speaker and handing control back to the OS.

  If a program wants to access hardware, the program must ask the operating system (Windows NT/2000/XP, Linux/UNIX) to do that for it.

  The other major contrast between MS-DOS and the Windows family, is that in MS-DOS, the default user interaction is via a *command-line* (console, teletype style) user interface, whilst in the Windows family, a GUI (graphical user-interface) is the default. The difference between command-line and GUI is much more than skin-deep – it has some quite profound implications for the programming of the OS and of application programs; later see *event-driven* programming.

- Windows 3.1, 95, 98, ME.

  Both in strictness of the management style, and in their multi-tasking capability, these are somewhere between MS-DOS and the NT/UNIX like operating systems.

9.1.1 Why Do We Need an Operating System?

Now taking the viewpoint of the operating system as a service provider or platform, we can see that the operating system fulfills three major objectives:

- handling of interaction with the hardware;
- provision of an execution environment (platform) for application programs;
- provision of a human-computer interface – which can be a simple *command line* interface like that of MS-DOS or Windows CMD, or the fancy GUI interface of Windows; in most contexts, the user-interface is called *shell* – a shell that, for the user, surrounds the O.S.

In some senses, modern operating systems are like the ease of use equipment found in modern motor cars. Both do their best to provide a layer between the user (or user programs) and the underlying hardware. Take for example an automatic gearbox. For most drivers that's a great help. For others, an automatic gearbox may be a great help, but in certain circumstances they may find it annoying not being able to change gear themselves. It's the same with operating systems — they make life easy for you, but they also restrict you; moreover, the OS's way of doing things may incur a small power overhead, i.e. they may cause the application to run a little slower — just like the automatic gearbox.

Most modern operating systems provide easy to use ways for programs to access hardware, for example the graphics screen. Most sensible programmers like this; on the other hand, some games programmers,

simply because they think they can make the graphics operate faster, would be prepared to write programs that directly access the graphics card.

Generally, as operating systems get more sophisticated, the more facilities they provide; on the other hand, the more restrictive they get. Since the operating system often really does know best, this state of affairs leads to greater reliability — less chance of the 'blue screen of death' for which Windows 3.1, 95, 98 were infamous.

If an operating system is multi-user, then the operating system must apply further restrictions in order to provide a private working environment for each user and to ensure that, either erroneously or maliciously, users' programs do not interfere with one another or with another user's resources.

From the point of view of a user typing at the keyboard or pointing and clicking with a mouse, the operating system makes the computer *reesponsive*. From the programmer point of view, it provides several services. The user of the service can be an application program - via invocation of a service function, e.g. `printf("Hello, World!");` – or a human user – via the operating systems human-interface.

With Windows NT/2000/XP and Linux/UNIX, we are moving away from single-user operating systems to multi-tasking – where, for example, the user can set the computer to compiling a large file, while he/she starts to edit another file. Microsoft Windows-95/98 offers some multi-tasking features. Windows NT/2000/XP is truly multi-tasking.

Since the late 1960s, operating systems on large computers have been multi-user; they allow a large number of users to use a single computer at one time, with each having the illusion of being the sole user of the machine. In the multi-user case, in addition to normal operating system duties, the operating system must:

- Allocate the CPU on a fair basis, and so that users, or a single user's separate programs, do not notice much delay;
- Keep each program, or bits of them, in memory (RAM) at once – a difficult juggling trick;
- If it begins to run out out of RAM, then save some of the contents of RAM to disk and bring in more urgently needed 'stuff' from disk;
- Keep each program from interfering with another program's memory, whether maliciously or by error;
- Manage, securely, a private area of disk for each user/program;
- Perform accounting tasks (perhaps) – so that users can be charged for the use of each service.

9.1.2 Multitasking on a Single User machine?

Before proceeding, we need to resolve a possible confusion. Let's say you have a PC at home, running Windows 2000 or XP, or, for that matter, Mac OS X or Linux. That machine might only ever have one user on it. In this situation, what advantage does multitasking provide?

For a start, you (the single user) can run many applications simultaneously, particularly if some of them require little interaction from you; for example, you could be downloading a large file using a browser, running an email program listening for new mail, and typing a letter using a word processor.

Even better, let's say you decide to send an email. You start the email program. Now you must type the message; a number of possible solutions exist: (a) The email program could have had word processor program code added to it — thereby duplicating file storage and programmer effort; or, (b) You could first type the message using a separate word processor program, save that as a file, and then run the email program into which you include the message file; the latter is solution in a single tasking operating system. But in a multitasking operating system, the email program can get the operating system to run a word processor in parallel with it (the email program), and through interprocess communication, receive the typed message from the word processor.

Another related point. Later we'll see that, even for an apparently single user system, e.g. a home system, it makes sense to have multiple users. For example, Windows 2000 will always have an *Administrator* user who is responsible for (owns) the operating system files and most of the other system software files. Then you have (separate) ordinary users who are responsible for their own files. Of course, there is nothing to stop a single user using the *Administrator* account all the time. The problem with that is that, as *Administrator*, a simple mistake can damage the system, whereas, an ordinary user has not the rights to do such damage. Thus, such security can prevent not only malicious damage, but also damage due to errors.

9.2 Operating Systems – Evolution

These rough notes trace the evolution of operating systems from (a) the very beginning when computers were (expensive) laboratory instruments operated by scientists, through to (b) what is called *open shop*, where user-programmers operated the computer themselves, to (c) *operator shop*, where a skilled operator handled the direct interaction with the computer, to (d) where the beginnings of todays *operating* software were introduced, and so on to the current day.

What I want to show is that an *operating system* performs rather like an old style human operator: (i) provides an interface between the programmer/user and the computer; and (ii) manages the computer so that it runs more efficiently. This section is based on (Tanenbaum 1999) or (Tanenbaum 1990).

9.2.1 The Beginning – up to about 1954

Computers were treated as laboratory instruments and could only be programmed using binary codes (machine code). The methods of inserting the codes varied: switches, paper tape, punched cards or, at the beginning, *patched* in with wires which connected High or Low voltages to indicate '1' or '0'.

9.2.2 Open Shop

[Here we seem to be concentrating on software development as the main use of computers. Well, at the time we are discussing, there wasn't much off-the-shelf software, so much time *was* spent on software development.]

By this time (say 1959) there were compliers – like FORTRAN. Programmers would run their own programs – much like users of personal computers do nowadays. Programmers booked the machine for (e.g.) 1/2 hour at a time. Programs were prepared off-line – onto punched cards – computers were too expensive for such menial tasks.

It was up to the new user to load whatever program was required onto the machine – from cards; executable and source code were both on cards. (Tanenbaum 1999) or (Tanenbaum 1990) p. 8, 9 describes a typical session, in which the user tries out his/her latest FORTRAN (source) program. (You may be able to imagine the following scenario if you think of an organisation with a single small, slow microcomputer, with one floppy drive, no hard disk and no resident operating system.)

1. At the appointed hour, you would charm/threaten/cajole the previous user out of the computer room.

2. First, the FORTRAN compiler had to be loaded - a huge deck of cards — via the card reader; note: before this there was no software in the computer, except for a small card reader program. Such a program was sometimes called a 'bootstrap' or card-loader – these were the great-great-grand-daddy of todays operating systems; a typical card-loader occupied, maybe, 50 to 100 bytes of memory.

3. Then your new source was read in, and compiled.

   Note: if the compiler required two or more passes, then steps 2 and 3 had to be repeated for the requisite number of passes – with a separate compiler deck for each; of course, the intermediate (half-compiled) result of pass one had to be output as a card deck! All the same sort of things go on in computer systems these days, but users don't see it happening — it happens fast, and results all go to disk files.

4. If there were compiler errors, you noted them and left the computer room. The next user took over - or the computer lay idle until the next half-hour slot began. You would then book another 1/2 hour (maybe the next free was 2 days away - or, you could come in in the middle of the night), try to locate the error, and retype the offending card(s); N.B. there was no editing facility in the computer.

5. If there were no compilation errors, the machine would punch the *object* code onto cards; roughly speaking, object code is machine code, see the appropriate chapter of Computer Systems. Note: compilation, linking, etc. are discussed in a chapter at the end of these notes.

6. The next stage was to *link* and *load* the object code with the library of system subroutines – also object code. To do this, the linker program, followed by library, followed by users object code, had to be read in via the card reader.

7. If there were no linker errors (unlikely, for they could be plentiful!) you ended up with a program in the computers memory, ready to execute.

8. Now, you could either *run* the program – by putting the start address into the PC register, via switches, or punch out the executable program – the equivalent of an `.exe` file.

9. If the program ran correctly (and, you didn't run out of time) you left to celebrate. Otherwise, you noted the error – very difficult, if the program just crashed. Big installations had line printers, which allowed printing of a full *dump* of the memory – in Hex or Octal – at the time of crashing. If you had no line printer, you noted the contents of a few memory locations using the console switches and lights - you could dial any memory location using switches and examine the contents on a light panel (one switch for every address bit and one light for every data bit). Or, in desperation, you could try to *patch* the machine code to try to get it to do something.

Clearly, much time was wasted:

- Loading programs;
- Idle time, if a user did not take an allocated time slot;
- Users scratching their heads wondering what was wrong with a program;
- etc. . . . – but no computer games!

All this on a machine that cost a few million pounds.

> Side-bar. In 1960, an average programmer salary was about 2000 Euro per annum, a computer cost around 2,000,000 Euro: ratio person annual salary to computer capital cost, 1 to 1000. Today, we have programmer salary (say) 40,000 Euro; for a powerful PC (say) 1000 Euro: ratio 40 to 1. Thus, compared to 1960, computers are $\frac{1}{40000}$ times the cost; or programmers are 30,000 times more expensive!

9.2.3 Operator Shop

The next stage was to hire an *operator*. The operator performed all interaction with the computer. Users submitted 'jobs' on cards, and received the results back on line printer paper.

The operator was more skilled and quicker at operating the computer. Also, similar jobs could be *batch*ed, e.g. all the FORTRAN compilations – allowing the FORTRAN compiler program to be loaded only once, followed by all the linking, etc.

Furthermore, the operator could assert *priority*:

- To a user who was prepared to pay more;
- To a small job, or one which used an already loaded program;

Also, the operator could halt jobs that ran too long, either through a program error, or having been deliberately programmed to do so by a greedy programmer.

These are all features that can be seen in modern operating systems.

9.2.4 Off-line Input-Output and Resident Monitor

Reading cards (slowly) and printing them (slowly) was not much of a job for a multi-million dollar computer! Magnetic tape is much faster to read and to write than are cards. It is more efficient to read the cards on a cheaper *satellite* machine, and create a magnetic tape, that contained many jobs, and get the expensive machine to work its way through the tape. In fact the expensive computer could have two tape decks – one to replace the card input, one to replace the card output. Also the system software (compiler, linker/loader) could be kept on a tape.

Most significant, however, from an operating systems point of view, was the addition of a small *resident monitor* program which remained in memory all the time and which ran each job in turn. This was the great-grandfather of today's OSes. The term *operating system* crept in because of the similarity of the resident monitor's role to that of a human operator.

The previous instructions to the operator (in English) now had to be coded for the *resident monitor*. These instructions had to be punched on cards — a typical convention was to precede resident monitor commands with an asterisk, '*', to distinguish them from source program statements, or data; i.e. the resident monitor had a rudimentary *command language* interpreter. A typical resident monitor was IBM's FORTRAN Monitor System, FMS, that ran on the IBM 709.

9.2.5 Spooling Systems and Schedulers

Early resident monitor systems used *polled input-output* (PIO), see chapter 8, to read from and write to their magnetic tapes; thus, when they needed input, or needed to output, the CPU gave up all other activity and attended to the tape. PIO is very inefficient, most of the time is wasted waiting for *status-ready* and things like that.

Therefore, the next elaboration was to add software to the resident monitor to allow it, simultaneously, to read from tape, do CPU processing, and, perhaps, write to an output tape or printer – sort of *multiprocessing*. This required *interrupt* driven I/O.

Eventually, disks were added to the main machine. Now, the off-line tape preparation satellite machines would be dispensed with, since the card jobs could be read-in on the main machine (using interrupts) and placed on disk to await execution. In addition, running jobs could write to disk files (faster), with the disk files being printed fairly independently.

A primitive *scheduler* was added to the resident monitor:

- When a job was read-in from cards (or tape), its name and characteristics were placed in a queue;
- The scheduler was run at the end of each job, to select the next job from the queue and set it running.

The human operators tasks are now limited to:

- Mounting tapes – for large data files, too big for cards – only very privileged users were allowed to keep data on the disk;
- Restarting the resident monitor when it crashed, or a user program crashed it;
- Loading cards;
- Unloading printer paper.

This mode of running was called *spooling*: **s**imultaneous **p**eripheral **o**perations **o**n **l**ine – **spool**. Spooling systems are the grandparents of modern OSes.

9.2.6 Multiprogramming/Multitasking

Nevertheless, spooling doesn't totally remove I/O waiting. Consider a program which is doing a stock update, and the stock files is on magnetic tape: very often the CPU has to issue a *read-from-tape* and can do nothing until the data arrives. The job is then said to be *I/O bound* – no amount of interrupt handling can help in this case.

Multiprogramming/multitasking comes to the rescue. If the memory is big enough — or the jobs small enough:

- You can place a number of them in memory at once;
- When one becomes I/O bound, transfer to another; when that becomes I/O bound transfer to another (maybe the original one), ...

This scheduler is a good deal more complicated than the spooling one. CPU time is one of the main **resources** that jobs are in competition for – implementing policies for sharing such resources is a big part of an operating system's task.

Multiprogramming also introduces competition for memory space. Also, the resident monitor and other programs must be protected from malicious or erroneous programs that try to write to parts of memory not owned by them.

What we have just described is called *batch* multiprogramming: jobs (*processes*) go onto a queue, work their way to the top of the queue, get started, run until I/O bound, get stopped temporarily and another process runs for a while ... until, finally, the process is completed. The term *process* is used to mean an executing program that performs a job; more about processes in later chapters.

We contrast *batch* operation with *interactive*. With batch, it doesn't matter too much how long a job waits on the queue, or how much time it waits one it has become I/O bound, so long as it completes in a reasonable time. However, with *interactive* operation, you have an impatient human being waiting to see results.

At this stage the resident monitor has become a true *operating system*, albeit still *batch*.

In addition to queuing and running processes, the OS provides services to the process via *service call*s. Processes can no longer access input-output devices directly – they must – in the terminology of multi-level machines – execute an OS machine level instruction that does that for them.

Of course, two processes may require the same resource at the same time, e.g. tape, printer; CPU time and memory are also resources that are in heavy demand. The business of granting resources in a fair way is not a trivial one.

9.2.7 Interactive Multiprogramming and Time-sharing

The batch multiprogramming described above was fine for the computers – but not for people – if programmers can be considered people, which, in organisations, is rare! In developing a program you could spot an error five seconds after getting your batch job back from the machine, and have to wait a day to get another try. Programmers needed quicker response.

Interactive multiprogramming or *time-sharing* was the solution. Instead of interacting, indirectly, via cards and printout, each user has an on-line interactive terminal. They also have access to the disk – perhaps their own directory – to store programs and data between sessions.

The whole basis of time-sharing is that there may be 5, 10 or 100 users connected to a single computer – with one CPU – with each of them getting the impression that she/he is the sole occupant. Of course, if all these users are running heavily CPU intensive programs (number crunching), the impression of having a machine to yourself quickly subsides. However, that will rarely be the case; at any instant, out of 20 connected users, 7 might be editing (requires little or no CPU time), 10 might be thinking about what to do next, and only 3 doing anything that really exercises the CPU — e.g. compiling or running programs.

Incidentally, operating systems have ways of discouraging heavy CPU and memory users from peak interactive times. For these users, a form of *batch* service is offered. The user submits a job to a batch queue much the same way as before - except the job is specified (1) interactively (2) in a disk file, and (3) the inputs and outputs also use disk files. The batch queue is treated differently from the queue associated with the interactive users - it is probably only given a chance at night, or when the interactive use gets very slack.

Massachusetts Institute of Technology (MIT) were the first to demonstrate a time-sharing system – about 1960; that was CTSS. It ran on a converted IBM 7094. But, on machines whose hardware was meant to run one program/user at a time, time-sharing was not easy — see later.

Multics The next step was MIT, Bell Labs. and General Electric co-operating in producing MULTICS (MULTiplexed Information and Computing Service). MULTICS was way ahead of its time. For the companies involved, maybe too far ahead, for it was costly to produce, and was not commercially successful.

But, MULTICS was enormously influential on the technical development of operating systems. MULTICS begat UNIX. Scratch the surface of most modern operating systems and you will find some MULTICS or UNIX influence.

Minicomputers Up to 1961 computers were a bit like electricity generating plants — they took up whole buildings or, at least, floors or rooms. They cost $millions.

In 1961 Digital Equipment Corporation (DEC) introduced the PDP-1. The PDP-1 had 4K of 18-bit words. It cost a mere $ 120,000. For some jobs it was nearly as fast as an IBM 7094 – the 7094 cost $2 million). Personal computing was on the horizon.

A whole family of PDPs, so-called *mini-computers* were developed, leading to the PDP-8 – the first true personal, or, at least, laboratory computer – and the PDP-11.

UNIX After Bell Labs. dropped out of the MULTICS project, Ken Thompson, one of the MULTICS team at Bell Labs., wrote a single user version of MULTICS for a PDP-7. It was written in assembly language. It was called UNIX – as a pun on Multics.

The PDP-7 had 4K memory! Yet this early UNIX featured:

- A file system;

- A process control mechanism;
- File utilities;
- A command interpreter.

Next step: UNIX was ported onto a PDP-11/20 and later a PDP-11/45.

But, besides its power and elegance, the next step was to ensure UNIX's fame and fortune. Dennis Ritchie joined the team and developed the C programming language. Around 1972, a new version of UNIX was written, almost entirely in C. Now UNIX could be ported onto any machine that had a C compiler.

9.3 History of Computers and Operating Systems — Summary

9.3.1 1st Generation (1945-1955): Vacuum Tubes and Plugboards

- Plugboards;
- Vacuum Tubes;
- Machine language;
- The computer is the eighth wonder of the world!

9.3.2 2nd Generation 1955-1965): Transistors and Batch Systems

- Transistors: greater reliability, smaller, less power, faster than vacuum tubes – all the factors you want in a computer;
- PDP-1 from DEC — interactive computing, cheap(ish) computing;
- PDP-8 from DEC — personal computing, cheap computing.

9.3.3 3rd Generation (1965-1980): ICs, Multiprogramming and Timesharing

- People becoming more important than the computer;
- OS written in a high-level language.

9.3.4 4th Generation (1980-1991): PCs and Distributed Computing

- Ordinary people using computers;
- computers become a consumer item;
- Need for *user-friendly* operating systems and software – people are no longer prepared to go to college for three years just to learn how to use a computer;
- Networks of PCs replace mainframes.

9.3.5 The World Wide Web, PCs everywhere (1991-)

- World-wide-web;
- PCs everywhere;
- Left as exercise for student.

9.4 Exercises

1. Identify two major points of contrast between MS-DOS and Windows NT/2000/XP (and Linux/UNIX). Give brief description of the implications of each.
2. Referring to the previous question, can you think why some game programs, that will run on Windows 98, will not run on NT/2000/XP? A related point: why, compared to MS-DOS and earlier Windows, is it less likely that computer running NT/2000/XP will completely 'freeze', requiring rebooting or complete restart?
3. More advanced OSes like Windows NT/2000/XP (and Linux/UNIX) have more power, but also more responsibility; explain/discuss.
4. There is a price (& I don't mean monetary price) to pay for the enhanced management and service capabilities of NT/2000/XP (and Linux/UNIX); discuss. Hint: memory, processor speed. On what sort of computer system does this price start to become intolerable.
5. Briefly, state *what an operating system does*. [5 marks]

   General point: a three hour exam. lasts 180 minutes; let's cut that back to 150 minutes to account for reading questions, checking at the end, revising answers ... Thus, 1.5 minutes per mark.

   A question like this could be answered by giving five significant points very briefly; or, two/three points with more substantial explanation.

   For five marks, in terms of writing, we probably require no more than five or so sentences – where each sentence expresses a single relevant idea/point.
6. In the early history of computer systems, how could an *operator* increase the efficiency of use of a computer system.
7. In the early history of computer systems, how could an *offline input-output* increase the efficiency of use of a computer system.
8. In the early history of computer systems, how could a *resident monitor* increase the efficiency of use of a computer system.
9. In the context of *spooling* or, more generally, of operating systems, explain the most basic purpose of a *scheduler*.
10. Briefly, explain a possible relationship between a computer *operator* (a person) and *operating system* (software).

Chapter 10

From Magnets to File Systems

10.1 Introduction

This chapter gives and introduction to magnetic disks and file systems. First we introduce analogue magnetic recording (on a tape); then playback; then digital recording/playback on a tape; then we move to disks. Disks are an important advance over tapes, because disks have random access, whereas tapes are sequential access, i.e. if you want to read something at the end of a tape, you have to read the whole tape until you get there.

Next we discuss the need for a *file system* – a way of structuring and organising the data on a disk. Finally, we discuss some details of MS-DOS, Windows and UNIX file systems.

There are three principal reasons we need magnetic storage:

- for *non volatile* storage; computer memory is *volatile*, it decays when the power is switched off, we need a way of storing data and programs between uses of the computer;
- we need a medium for exchanging data and programs;
- we need a medium for storing data collections that have got too large to store in available memory.

10.2 Magnetic Recording

If you bring a magnet in contact with a steel pin, the pin will become magnetically aligned, i.e. the pin will behave like a small magnet – how it is aligned (North-South, or S-N, will depend on how it was brought in contact with the magnet; the strength of its magnetism will depend on the strength of the original magnet.

For magnetic recording, we need some way of producing a magnet that varies the strength and alignment of its magnetism. Thus, *electromagnetism*. When you pass electrical current through a wire coil wound into a cylindrical shape, the coil behaves as a magnet, one end becoming as North, the other as South; the more current the greater the strength of the magnet. Incidentally, if you put a certain type of iron or other material (ferrite) in the middle of the coil, the strength of the magnetism (the *magnetic field* is stronger than if you have just air – all this means is that you need less current.

Now we have the basis for magnetic recording. Let's extend the steel pin in length, using, say, a steel piano wire, and run the wire in close proximity to a coil; the current in the coil varies according to the strength of, say, a sound. Thus, at any point on the wire, we have an impression of the sound level as that point passed the coil: *magnetic recording*. This was roughly how Valdemar Poulsen recorded and reproduced sound at the Paris Exposition in 1900, (White 1984).

How was the sound reproduced, i.e. replayed? In exactly the opposite manner to recording. When a magnet is moved past a conductor, a current is induced in the conductor; when a magnet is moved past a conductor wound in a coil, a lot of current is induced in the coil. So, we have the coil already (with its piece of iron in the middle), we also have the magnet(s) – the length of steel piano wire; so, to replay, you just run the piano wire past the coil at the same speed as you did when you were recording, and out of the coil comes a current that varies according to the strength of the magnetism on the piano wire (which, in turn, varies according to the strength of the original magnetism produced by the coil, and this depended on the current in the coil, which depended on the loudness of sound . . .). You then take that current, amplify it and send the amplified signal to a speaker, and you have *analogue* sound recording.

If you want digital recording, you simply arrange, for example, that N-S magnetism (e.g. negative current in the coil), corresponds to a '0' and S-N magnetism corresponds to a '1'. Or something like that – the details are unimportant. Thus, we could record (*write*) and playback (*read*) a stream of bits.

The same coil, the *head*, could be used for reading and writing.

Magnetic Tapes Early magnetic sound recording (up to about 1950) used steel wire. Next advance was a plastic tape with magnetic material painted onto it. In digital tapes, it was arranged to record nine (9) tracks across a $\frac{1}{4}$ inch tape. Thus, you could fit a byte, plus a parity bit (see chapter 2) across the tape.

Exercise. (a) Assuming no gaps, work out the capacity of a 2400 ft. magnetic tape, which uses 6250 b.p.i. (bits per inch) density. (b) If your tape unit could read at 1 foot per second, calculate the data rate in bytes per second. (c) Reading at 1 foot per second, how long would it take to read the whole tape – or find something that might be at the end?

Problems with Magnetic Tapes Because they are *sequential access*, for large collections of data (files), and for large numbers of files, access to the data on them is slow.

What you would like is something close to the *random access* of memory: when accessing memory it doesn't matter whether you want to access memory address 100, or 100,000,000, it takes the same (small) amount of time.

A magnetic drum is one answer – where you have a large rotating cylindrical drum with lots of *fixed* read/write heads, one corresponding to each track of bits. Drums were important at one stage, in fact, on some computers in the 1950s they *were* main memory, backed up with tape on one hand, and some fast (RAM like) memory on the other.

10.3 Magnetic Disks

In 1955, IBM invented a magnetic disk (the RAMAC: fifty 24 inch platters mounted on a vertical shaft, rotating at 1200 rpm, 100 bpi along track, and 20 tracks per inch, leading to an overall capacity of 5 Megabytes), (White 1984). Actually, 5 Megabytes was huge for 1955, and many smaller disks, usually

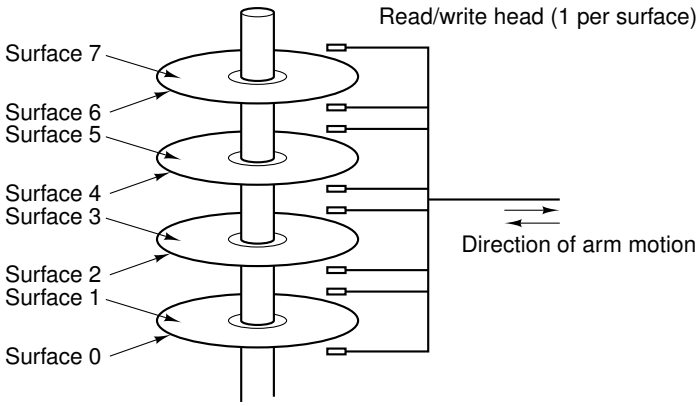


Figure 10.1: Disk with Four Platters

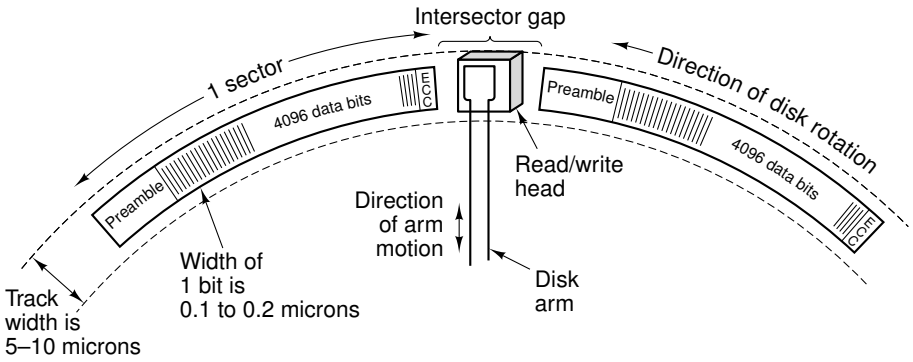


Figure 10.2: Two Sectors on a Disk Track

with one platter and two heads, were prevalent until the 1980s. Nowadays, of course, it's hard to find a disk with less capacity than 10 Gigabytes.

But the principle is the same: many platters (disks painted with magnetic material – like tapes), one read/write head for each side of each *platter*, many *tracks* from the outside of the platter to the inside (leaving plenty of room for the drive shaft); when you have multiple platters, the tracks above and below one another (on the same diameter) form a *cylinder*. We talk of cylinders because the multiple read/write heads move in unison – for a 10 platter disk (two heads per platter), when head 5 moves to track, so do heads 0, 1, . . . 18, 19.

Most of the remainder of this chapter is from: (Tanenbaum 2001) and (Mueller 2001).

10.3.1 Cylinders, Heads, Sectors

When addressing a disk, we need the following address components: *cylinder*, *head*, and *sector*, see Figures 10.1 and 10.2 (from (Tanenbaum 1999).

Sectors are subdivisions of tracks and are need for the following reason: when you read a disk, you must read the whole of the requested (cylinder, head, sector). If there were no sectors you'd have to read the whole track – which, for a typical 30GByte disk, might be as much as 500 KBytes; in other words, the smallest a file could ever be is 500 KB – which would be wasteful, and the smallest unit of information you could ever read is 500 KB, when, say, you might only want to read the next five characters (5 bytes). Generally, a sector contains 512 bytes. For a typical 30 GByte disk, you might have up to 1000 sectors per track.

That's about it: we have a way of writing bytes to a non-volatile medium, we have a way of reading that data back.

When the OS wants to write, it uses a subprogram (a driver) to get the disk controller to write, for example, the 512 bytes starting at e.g. memory address 1025 (to 1536) to, e.g., the *sector* whose *disk address* is cylinder 25, head 2, sector 55.

Thus, a *sector*, typically 512 bytes, is the smallest unit of information that can be written or read; if you want to write/read three bytes, you simply have to indicate that the other 499 are zeros or some other value corresponding to 'blank'. Thus, a file can never occupy less than 512 bytes.

10.3.2 Linear Sector Numbering

It is much easier for everyone involved if we number sectors from 1 to maximum-number — and leave it up to the the disk controller (part of the disk unit), to translate that into (cylinder, head, sector) and that is what is done.

10.3.3 Blocks and Clusters

Actually, an OS or its *file system* may choose to work in *blocks* (Microsoft call them *clusters* or *allocation units*) of one, two, four, eight, 16, ... sectors. In the case of, for example four, no file could occupy less than four sectors or 2K bytes.

From the point of view of the operating system a cluster is the smallest unit of data on a disk.

Small disks, such as floppy disks, use a cluster size of one or two sectors.

FAT16 and FAT32 file systems use cluster sizes of up to 64 sectors (32,768 bytes).

Typically, when the OS (or its file system) talks to a disk driver, it will talk in terms of cluster numbers: write to cluster 10, rather than write to sectors 641–705; and it definitely doesn't talk in terms of (cylinder, head, sector).

10.3.4 Partitions

In general, there is no real reason why the whole of a disk (from sector 1 to *maximum-sectors*, or correspondingly from cluster 1 to *maximum-clusters*) can be treated as a whole. However, with the onset of very large capacity disks, at least two factors lead to the further subdivision of the disk into *partitions*; also called *logical disks* or *volumes*. These are:

- the limited size of the number used to a identify the cluster; if we have a 4 GByte disk, and 32 Kbytes per cluster, the last cluster is numbered 128K – that would take 17 bits. Some systems, e.g. FAT16, use only 16 bits – that is why FAT16 can handle only 2 GByte at a time;
- you might want to put more than one operating system on the disk.

10.3.5 Master Partition Boot Record, or Master Boot Record (MBR)

In order to cater for partitions, we need some sort of directory of partitions at some agreed point on the disk; the word *boot* is here because the bootstrapping program in ROM has to have somewhere obvious to look for an operating system program. Usually, the MBR is contained in the first sector of the disk.

10.3.6 Primary, Extended Partitions

MS-DOS and Windows further separate partitions into *primary* and *extended*; MS-DOS and Windows do their best to ensure that you have just one primary partition; once you have an extended partition you can further subdivide that into other logical partitions; thus in FAT16, with a 10 GByte disk, you might have a 2 GByte primary partition, (C:), and a 8 GBytes extended partition, divided into four 2 GByte logical partitions, D:, E:, F:, and G:.

10.3.7 Disk Sectors, Clusters, Files, etc. — Summary

- Users (humans) and application software treat disks as collections of files, i.e. when they want to read from or write to a disk, they use a filename such as `file2.txt`;
- The operating system takes that filename and, see section 10.6, translates that into a *cluster* identifier (number);
- Next, the operating system passes that cluster number to disk driver software which in turn translates the cluster number into a *linear sector number*;
- Finally, that linear sector number is sent to the disk controller (out on the disk unit itself) which now translates it into *(head, cylinder, sector)*.

10.4 Time to Read and Write — Latency

There are three types of *latency* (delay) that affect the read/write time:

- seek latency; the time to locate the correct cylinder; we need to add *head settling time* to this;
- rotational latency; the time to bring the desired sector under the read/write head;
- transfer latency; the time for those sectors with data to be scanned.

Worked Example on Disk Capacity, Latencies, etc A hard disk has the following characteristics. Rotation rate: 7,200 revs per minute. Number of sectors per track (assume fixed): 400. Number of platters 6. Number of heads 12. Number of cylinders 17,000. Average seek time – time for head servomechanism to wake up and to move to the desired cylinder: 10 millisecs; time to move between adjacent cylinders: 1 millisec.

(a) Compute the capacity of the disk (bytes).

```
Capacity = 12 (heads) x 17,000 (cylinders) x 400 (sectors) x 512 bytes

           (Here, I'm trying to show you how to calculate by hand;
            Beware of punching everything into a calculator -- it is
            easy to lose a '0' or two! But if you do it out roughly
            by hand, then you can check the calculator result. Also, in
            an exam. or in an assessed assignment like this, if you
            simply write down an answer, you lose marks.)

         = 12 x 17 x 4 x 512 x 10^5 bytes

         = 12 x 17 x 2 x 1024 x 10^5 bytes

         = 12 x 1.7 x 2 x 1024 x 10^6 bytes

         ~= 12 x 1.7 x 2  Gigabytes

         ~= 40.8 GBytes

The exact answer is 41.78 x 10^9 bytes = 38.9 GBytes
```

(b) Compute the maximum rotational latency. Answer:

7,200 rpm is 7,200/60 revs per sec. = 120 revs per sec.

Maximum rotational latency occurs when the sector you want to read has just passed by as you decide to read it, i.e. time for one revolution.

```
      1 revolution takes 1/120 secs. = 8.333 x 10^-3 secs
                                     = 8.333 millisecs.
```

Therefore, the maximum rotational latency is 8.333 millisecs.

I.e. the delay caused, if the sector you wanted has just passed when you decided to read.

(c) Compute the average rotational latency– which is the average of the minimum (0) and the maximum (see b). Answer:

Minimum latency is 0, maximum is 8.333 ms, therefore

average rotational latency is $((0+8.333)/2)$ *millisecs.* $= 4.17$ *millisecs.*

(d) Compute the maximum transfer rate in MB per second, i.e. when a head is reading from a sector, the maximum transfer rate is the rate at which bytes (bits) are passing under the head. [You can read only one head at a time]. Answer:

Maximum transfer rate is calculated by calculating the rate at which the data is passing by under the head, i.e. this is the maximum rate that the data can be read into the head. Note that only one head can be reading at a time. We have (a) revs per second; (b) data per rev. (or track); hence, we can calculate data per second (what we want) by multiplying the two.

```
Maximum transfer rate = 400 x 512 x 7,200/60  bytes per sec.
                      = 4 x 12 x 512 x 10^3 bytes per sec.
                      = 4 x 6 x 10^3 Kbytes per sec.

                      = 24 MB per sec.
```

(e) Assuming one cluster of eight (8 sectors) all on the same cylinder are to be read, compute the expected (average) read time; suggestion, base your answer on: (i) the average seek latency; (ii) the average rotational latency; transfer latency. Answer:

```
Transfer latency = time to read 8 sectors.

Time to read 400 sectors = 8.3 ms
Time to read 1 sector = 8.3/400 ms
Time to read 8 sectors = 8 x 8.3/400 ms

                       = 0.166 ms

Average time = av. seek latency + av. rot. latency + transfer latency

             = 10 ms (given) + 4.17 (see (c)) + 0.166 ms

             = 14.3 ms
```

Note: seek latency is by far the greater and all steps possible (e.g. defragmentation and use of contiguous allocation) should be taken to minimise its effects.

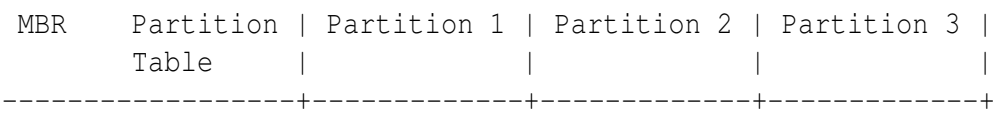
10.5 File Systems

- The cylinder, head, sector model is far too complicated for humans;
- And even the linear sector or cluster model;
- Need structure and organisation – how do you organise your papers at home?
- A directory is just data – a list of files;
- Root directory is located near the beginning of a partitions;
- And the remaining directories are just (specially marked) files

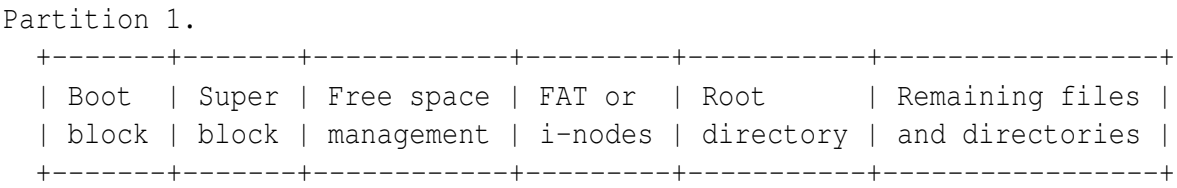
For more information, see (Tanenbaum 2001), chapter 6, and (Mueller 2001), chapter 25.

Let's summarise what we have so far. The remainder of this section is from (Tanenbaum 2001).

The disk is a linear sequence of sectors (typically 512 bytes). The first sector contains the MBR (Master Boot Record – but also contains Partition details, so maybe better called MPBR, see above). Then we have the remainder of the disk allocated to separate partitions (there may be just one partition – as in floppy disks). This state of affairs may be depicted as follows, for three partitions.



A typical partition layout is shown below.



Boot Block If the partition is bootable, this block contains the boot code – with possible pointers to code elsewhere.

Super Block Contains information about, for example, the file system used.

Free space management Contains information about blocks in the partition that remain unused. May be stored as a bit-map – with one bit for every block in the partition.

File allocation data See section 10.6.

Root directory, and other files and directories In general a directory is just another file, see section 10.10, but the root directory may be special in that its size may need to be known in advance of any use of the partition. Remaining files and directories are merely pointed to by the entries in the root directory, or by directories below that.

10.6 Implementation of a File System

The chief problem in implementing a file system amounts to keeping a record of which blocks go with which file (note again that directories, except for possibly the root directory, are just files) and in which order.

10.6.1 Contiguous Allocation

When a file is created, it is allocated a contiguous number of blocks; for example, let us say that we are using a file system with 2 KByte blocks. We want to create a file `ch1.txt` which contains 26 KBytes; assume that blocks up to number 50 are already used. We allocate 25 blocks, 51 to 75, to `ch1.txt`.

Advantages Contiguous allocation is simple to implement. The directory entry for a file, e.g. `ch1.txt`, needs to record for no more than *start block* (51) and *size* (25).

Contiguous allocation has good read/write performance. Once the heads have been moved to the beginning of the file, they need to move again only in small steps – or not at all for smaller files.

Disadvantages However, contiguous allocation suffers from one major drawback: *disk fragmentation*. For example, assume we have just 200 blocks partition, allocated as follows:

```
                Blocks
Boot, super etc. 1 to 50
ch1.txt          51 to 75    (50 Kbytes)
ch2.txt          76 to 100   (50 Kbytes)
ch3.txt          101 to 150  (100 KBytes)
ch4.txt          151 to 160  (20 Kbytes)
ch5.txt          161 to 190  (60 KBytes)
                 191 to 200 is still free
```

If we want to create `ch6.txt` and it is, for example, 30 Kbytes, we cannot. However, let us say that we have deleted `ch4.txt`; we now have 40 KBytes free (blocks 151 to 160 and 191 to 200) but they are not contiguous, and so we are still in trouble. let us delete `ch1.txt`; now we are fine, we can put `ch6.txt` in blocks 51 to 65.

However, now we have free blocks 66–75, 151–160, 191–200, a total of 60 KBytes, but they are usable only for smaller files: 20 KBytes or 30 KBytes.

An additional problem with contiguous allocation is that when a file is created, it may not be known how large it will end up. E.g. when you open a file `program1.bas` in an editor, you have no idea how large it will become.

Contiguous allocation is used in CD-ROM file systems; here the sizes of files are known in advance and there is no deleting, rewriting.

10.6.2 Linked List Allocation

Each block contains a number which points to the next block. Thus, two files, fileA, fileB. Some special number, e.g. -1, signifies 'this is the last block', see Figure 10.3.

Advantages For Linked List Allocation, the directory entry needs to record just the start block; the next block can be found by reading the *next-block* number.

Disadvantages

- The file becomes *sequential access*; often in reading/writing a file, you need *random access*. In sequential access, reading something near the end of a file requires reading everything up to it (at least reading every *next-block* number, and reading one number can take almost as much time as the whole block).

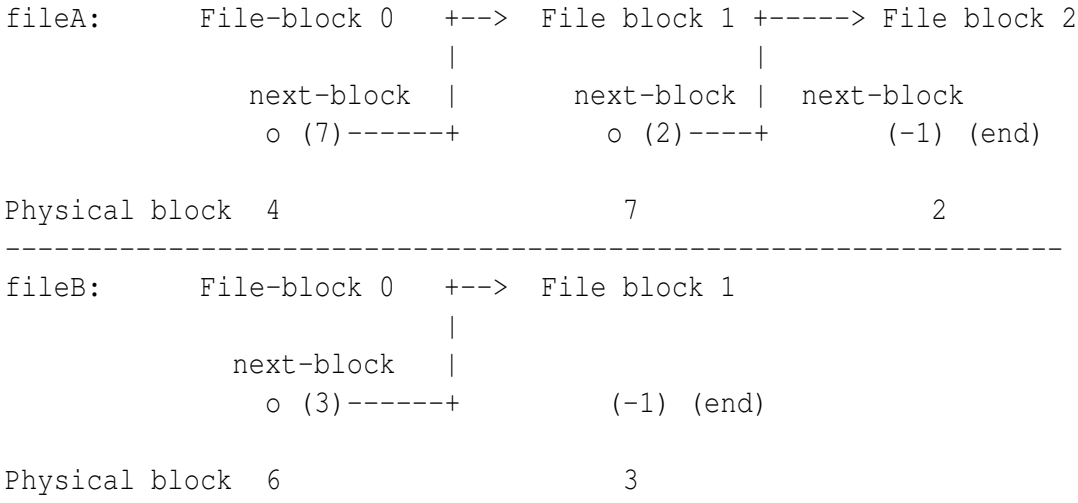


Figure 10.3: Example of Linked List Block Allocation

FileA: blocks 4, 7, 2. FileB: blocks 6, 3.

Physical block	Next-block	
0		
1		
2	-1	(end of fileA)
3	-1	(end of fileB)
4	7	(start of fileA)
5		
6	3	(start of fileB)

Figure 10.4: Example FAT Method of Block Allocation

- A number of bytes (e.g. two or four) is wasted on the *next-block* number. Not that the waste of two or four bytes is a great problem, however, it is often convenient to be able to read/write blocks of data that are exact multiples of 512 bytes.

10.6.3 File Allocation Table (FAT)

The File Allocation Table method is just another way of storing the Linked List. There is a table with an entry for each physical block that points to the *next-block*. Thus, for the examples fileA, fileB above, a FAT is shown in Figure 10.4.

Chains are terminated with a special number, e.g. -1.

File Allocation tables are stored in a special part of the partition and read into memory when the partition is accessed.

Advantage If the FAT is read into memory, it becomes possible to do fast random-access — i.e. with no necessity for intermediate disk accesses.

Disadvantage The FAT takes up space in memory. If we have a 20 GB partition, and 1 K blocks, we need a FAT with 20 Million entries; this could not be handled by a 16-bit FAT (0–65,535) (e.g. Windows FAT16 file system). With a 32 bit FAT (e.g. Windows FAT32 file system), each entry would be 4 bytes, i.e. 80 MB in total.

10.6.4 I-Nodes

Here a special data structure (data block), an *i-node* (index node) stores information about the file.

The i-node stores: file attributes; address (physical block number) of file block 0; address of block 1; . . . ; address of block $n-1$; address to another block of addresses (used only for files larger than n blocks. UNIX uses i-nodes.

10.7 Miscellaneous

10.7.1 Bad spots

When formatting a partition, the formatting program usually writes data to each sector, then reads the data back, checking against what was written. If the check indicates an error, there may be a *bad spot* on the disk, i.e. the magnetic paint may have a little flaw. What to do? Throw the disk away?

One way of handling the problem is to group all the bad sectors and associate them with a *bad spot* hidden file. Then they never get in the way again. These days, the disk controller (on the disk unit) handles the problem; it keeps a list of all bad sectors and ensures they are avoided when it translates linear sector numbers to (head, cylinder, sector).

10.7.2 Disk fragmentation

We have already mentioned disk fragmentation in the context of *contiguous allocation*. Well, even with FAT, disk fragmentation can cause performance problems.

When you start with a new disk (or new partition), the blocks allocated to a file are consecutive. This means that, once the head is placed at the beginning of the file, reading/writing requires a minimum of head movement. However, as time goes on, and the partition fills up, and files get deleted, then a single file may find itself strewn all across the disk – with maybe a head movement for every block. In such a case file accesses (read or write) could become slow.

Thus, defragmentation programs. These read a file into memory, delete the fragmented blocks, find a contiguous set of blocks, and write the file to that. Obviously, it requires a bit of juggling, but it can be done.

10.7.3 Windows FAT File Systems

MS-DOS and Windows 3.1, 95, 98 and ME used a FAT file system: FAT12, FAT16 and most recently FAT32 – where the number refers to the number of bits in the File Allocation Table Entry.

Exercise. (a) FAT16 (the original FAT for hard-disks) uses a 16 bit FAT entry. The maximum cluster/block/allocation unit size it can use is 64 sectors, or 32,768 bytes; what is the maximum partition size it can handle?

(b) Do the same for FAT32, again assuming 32 KByte blocks. What is 2 TB?

(c) Can you think of any problem associated with such large block sizes?

10.8 NT File System (NTFS)

NTFS is the preferred file system for Windows NT, 2000 and XP(?). NTFS is based on a data structure called the MFT (master file table); the MFT is a much extended version of a FAT, (Mueller 2001).

When you create an NTFS partition, the following 10 system files are created:

- $mft – holds the MFT;
- $mftmirr – a mirror of $mft, for recovery purposes;
- $badclus – contains all bad blocks;
- $bitmap – cluster availability stored as a bitmap;
- $boot – a bootstrap program, if the partition is bootable;
- $attrdef – attribute definitions table;
- $logfile – log file, contains a log of file transactions; for recovery purposes;
- $quota – quota table; disk quota table for users;
- $upcase – table giving conversion for filenames to UNICODE;
- $volume – contains volume information, e.g. name an size.

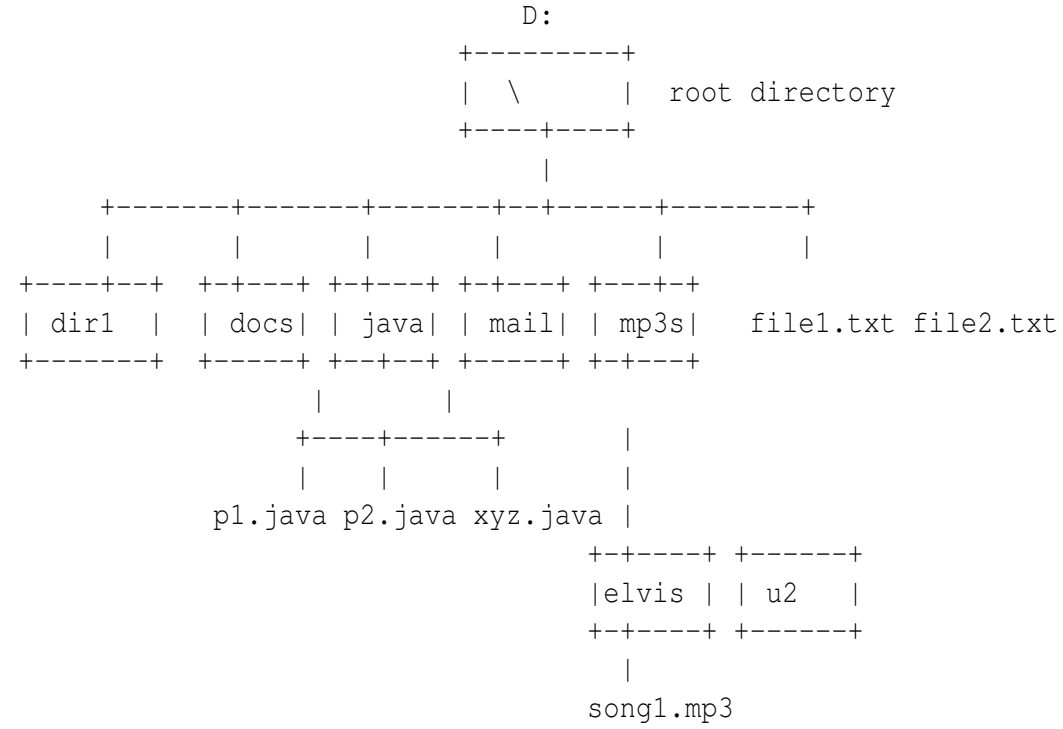
10.9 Directories

A directory is just a file. In the directory file we need a chunk of information about each file and where to find it. Typical information in a directory entry is:

- name;
- time and date created;
- attributes;
- starting physical block number (if using FAT);
- size.

10.10 Hierarchical File System

Windows, MS-DOS and UNIX have *hierarchical* file systems. The example below shows a `D:` of your hard disk (I say `D:` because it wouldn't be typical of a `C:`, which would have Windows specific directories such as `windows`, `My Documents`, etc.).




```
 '\' or in UNIX '/' has two meanings:
- a leading \ means the 'root' directory
- subsequent \s are separators for components of 'pathname';
```

Equally, it could be `A:`, a floppy disk, or a Zip drive or CD-ROM. In addition, there are *pretend* partitions (e.g. `X:`, `Y:` accessible across the network on servers. All present the same hierarchical view to the user.

This is a *tree* structure; in other words, any directory (folder) may contain true files (these are like *leaves* on a tree, or other directories (like *branches* on a tree. *In computer science the root of a tree is always at the top*!

Each directory may contain an unlimited number of files or other directories (i.e. what we can roughly call the width of the tree is unlimited); also, the depth of the tree (the number of times directories may branch into other directories) is unlimited.

I hope you will agree that this tree-like structure is an invaluable way of organising information. Like some early file systems (*), you could put all your files in `\`, but chaos will soon reign.

(*) From memory, MS-DOS 1.0 had a single directory structure.

In addition, the Windows GUI doesn't really emphasise the hierarchical structure. Many users could be led to work with very few folders, e.g. `My Documents`, `Programs`, etc.

There are other ways of displaying hierarchical file structures. You'll see these when using Windows.

In Windows, we have multiple partitions (also called *volumes* or *drives* : A:, C:, D:, etc. In UNIX everything is attached to a single root. Note: don't let the term *drive* confuse you; it means logical drive. In a floppy disk, `A:`, one disk drive corresponds exactly to one partition, but as you know, a typical hard drive may have many partitions, i.e. one physical disk drive may correspond to many logical drives.

10.10.1 Directory Navigation

For more information on some of these commands, see section 11.3.

The format DOS commands is: `command-name [/options] [arguments]`

For example: `dir /w X:\mp3s\elvis`

`X:> cd \mp3s\elvis\` attaches you to the `elvis` subdir. of `mp3s`

Assume you are in `elvis` directory.

`X:> cd ..` moves up one level in directory hierarchy; i.e. now in `mp3s`.

`..` refers to the parent of the current directory; `.` is a synonym for current directory; thus:

`X:> cd .` (does nothing)

If you are in `\mp3s\elvis\` :

`X:> mkdir kingcrol` (creates a sub-directory)

`X:> rmdir kingcrol` (deletes that directory - cannot use plain `del`)

Other commands:

- `del`, `erase`: delete a file;
- `copy` *file-from file-to*;

10.10.2 Paths and Path-Names

In general, anywhere you can use a *file-name*, you can use a *path-name*. Thus, if we are in directory `X:\mp3s\elvis`, we can delete file `song1.mp3` in any of the following ways:

```
del song1.mp3

del \mp3s\elvis\song1.mp3        # full path name

del x:\mp3s\elvis\song1.mp3      # path name, including volume

del .\song1.mp3                  # current-dir\filename
```

If you were in `X:\mp3s`, you could do the same by:

```
del elvis\song1.mp3

del \mp3s\elvis\song1.mp3        # full path name -- always okay

del x:\mp3s\elvis\song1.mp3      # path name, including volume

del .\elvis\song1.mp3            # current-dir\dir\filename
```

If you were in `X:\vb`, you could do the same by:

```
del \mp3s\elvis\song1.mp3        # full path name -- always okay

del x:\mp3s\elvis\song1.mp3      # path name, including volume

del ..\mp3s\elvis\song1.mp3      # parent-dir\sub-dir\dir\filename
```

Of course, giving the full path can become tiresome and error prone. Assume that you are using the command line interface and that you are writing and compiling Java programs. Assume that you are working in the directory `D:\java\`. You create the program `p2.java` using `edit`: `edit p2.java`. How does the OS know where to find `edit`? Say it is in `C:\Windows\`. The answer is that the OS will have a list of places it knows to look in. This list is called `PATH`.

Let's go further. You've had to download the Java compiler. It's installed in `C:\j2sdk1.4.0_01\bin\javac`.

Now that you created `p2.java`, you want to compile it. Therefore, type `javac p2.java`; no go, OS complains that it cannot find a program called `javac`. Now what you have to do is type `C:\j2sdk1.4.0_01\bin\javac p2.java`; and that works fine. But, it's tedious and error prone.

You could connect to `C:\j2sdk1.4.0_01\bin\` and type `javac C:\java\p2.java`, and that would work after a fashion, but the results would be placed in directory `C:\j2sdk1.4.0_01\bin\`, not what you want.

The answer is to add `C:\j2sdk1.4.0_01\bin\` to the `PATH` (the set of directories where the shell is to look for programs it must execute.

In Windows 98, ME, this is done via `AUTOEXEC.BAT`. If you look at it, you will find a line something like:

```
SET PATH = C:\WINDOWS;C:\WINDOWS\COMMAND
```

What you need to do is add a line:

```
SET PATH = %PATH%;c:\j2sdk1.4.0_01\bin\
```

this says, *add* `c:\j2sdk1.4.0_01\bin\` *to the current PATH*. Equally effectively, you can change the original line to:

```
SET PATH = C:\WINDOWS;C:\WINDOWS\COMMAND;c:\j2sdk1.4.0_01\bin\+
```

When either of these changes is made (and you've rebooted), when confronted with a command or program name that isn't one of it's own (see chapter 11, the Windows shell first looks in the current directory, then in `C:\WINDOWS`, next `C:\WINDOWS\COMMAND` and finally in `c:\j2sdk1.4.0_01\bin\`.

Editing `PATH` in Windows 2000 is a little different; we'll see how to do that in a practical.

10.10.3 Command History

When using the command-line interface, quite often you end up repeatedly using the same commands, e.g.

```
edit p2.java
javac p2.java
-- errors
edit p2.java
javac p2.java ...
```

In Windows and Linux, *command history* can save a lot of typing. Using the *arrow* keys, you can scroll back through previous commands. When you find the one you want, you simply type *return* and it gets executed. Or, you can edit it.

In Windows 2000 and Linux, command history is activated by default; in Windows 98, you must activate it by running program `DOSKEY`.

10.10.4 Auto-completion

Again, assuming that you are using the command-line interface. Let's say you are working with files with long filenames, e.g. `File-with-a-very-long-name-1`, `File-with-a-very-long-name-2`, `File-with-an-even-longer-name-`. Typing these is tedious and error prone. In Windows 2000 and Linux, *auto-completion* allows you to type, `F<tab>`; the `<tab>` signals to the shell (command-line interpreter) to have a go at guessing which file you want; in the case that we have the three files given above, the shell will auto-complete up as far as `File-with-a` and then wait for another clue; if you fill in `n` and type `<tab>` again, as in `File-with-an<tab>`, the shell will go on to auto-complete `File-with-an-even-longer-name-`.

In Windows 2000, auto-completion is not activated by default; it must be activated by editing the Registry; we may do some Registry editing in a practical.

10.11 File Security and Permissions

In MS-DOS and Windows 3.1, 95, 98, ME, *any* user of a machine had access to *any* file and so could read, write (including delete!), and execute any file. Since these systems were primarily single user, that was fine (sort of) — but problems could be caused by users deleting essential system files.

In Windows 2000 (NTFS only), as in UNIX/Linux, we have the concept of file *permissions*. Essentially, you can specify who can access files. Typically, there are *three* sets of permissions: (a) those of the *owner* (the user, e.g. `jbloggs`, who created it) of the file; (b) a *group*, e.g. if `jbloggs` is part of a group `ITSos1` and file `osnotes.txt` is associated with not only owner `jcampbell` but also with group `ITSos1` and you, `pmurphy` are associated with that group, the you have*read* permission for the file; finally, there is (c) *world* (everyone). If file `osnotes.txt` has world read permission set, then anyone, provided they can gain access to the machine, can read it.

Windows 2000 provides the same security measures for other operating systems *objects*, e.g. the Registry.

You can verify Windows 2000 security and permissions by attempting, on a laboratory machine, to delete Windows files! (I think — don't quote me.)

We will examine this matter when we come to installing Windows 2000 and later creating users. The installer will have installed Windows 2000 system files as user `Administrator`. Later, when we create user `jbloggs`, we will find that `jbloggs` can interfere with system files. In general, this is a good thing. Even if you are installing Windows 2000 on a home/single user machine, it is a good idea to have an `Administrator` user to do system stuff, and then an ordinary user (e.g. `jbloggs`) to do everyday stuff. Maybe the benefits of this scheme are not obvious; well discuss it in class and in practicals.

10.11.1 Unix-Linux Access Permissions

I include the following, from Linux, as an actual example. The language used in NTFS is slightly different, but deep down, the principles are similar.

Here is part of the directory (using `ls -l`) from the `/home/jc/cprogs` directory.

```
-rwxr-xr-x   1 jc       others      12798 Mar  7 13:29 hello
-rw-r--r--   1 jc       others        218 Mar  6 11:14 hello.c
-rw-r--r--   1 jc       others         96 Mar 11 19:22 sc2
-rwxr-xr-x   1 jc       others        227 Mar 11 19:35 scmenu
^    ^       ^ ^        ^            ^     ^            ^
|    |       | |        |            |     |            |
|    |       | |        |            |     |            +- name
|    |       | |        |            |     |
|    |       | |        |            |     +- date last modified
|    |       | |        |            |
|    |       | |        |           + size (bytes)
|    |       | |        |
|    |       | |        +- group that has group access to file
|    |       | |
|    |       | +- name of the owner
|    |       |
|    |       +- number of links to the file (usually 1)
|    |
|    +- permissions (see below)
|
+- type of file: '-' file, 'd' directory
```

The permissions (nine characters) define the access permissions for three types of user, and for three types of use. The characters refer to, in order from left to right:

```
    owner                   group     all
    Read, Write, eXecute    R W X     R W X
```

Hence, `hello` (an executable):

```
-rwxr-xr-x   1 jc       others      12798 Mar  7 13:29 hello
```

- can be Read, Written-to, and eXecuted by owner (jc);
- can be Read or eXecuted by the *group*;
- can be Read or eXecuted by *all* (everyone).

On the other hand, `hello.c` (the source file):

is the same, except it cannot be eXecuted by anyone (which makes sense – it wouldn't run).

```
-rw-r--r--   1 jc       others        218 Mar  6 11:14 hello.c
```

If the file is `-r--r--r--`, then nobody can alter or delete it.

10.12 Memory Hierarchy, Need for Files and all that

In an ideal world, computers would have an infinite (well, let's say the amount of memory addressable by 64 bits) and all possible programs and all possible data would be in memory. But we are not yet in this ideal world. memory is finite, and access is slow enough to sometimes be a problem.

As we will see in chapter 13 computers try to get as many as possible programs into memory at once. Actually this problem is very similar to that of a single program (and its data) which is larger than the physical memory available. The trick (*virtual memory*, discussed in chapter 13) is to have as much as possible of the program in main memory, with the rest on disk, but with software in the operating system, and some hardware, which allows swapping between memory and disk in a way that interferes as little as possible with the running of the program.

In addition we mention another trick (*cache memory* which allows us to mix very fast memory (close to the CPU) with slower main memory, to yield overall speedier memory access.

> Aside (from (Tanenbaum 2001) section 12.6, pp. 894–895). 32 bits, allowing us to address 2^{32} bytes/words (4.3 GBytes/GWords) once seemed infinite. However, we now have disks capable of holding 100GB. 64 bit addressing is the next big thing (soon to come in Windows 2000 and Linux). With 32 bits worth of memory, if you give a byte to everyone in the world, you haven't enough to go around. But 2^{64} is about 2×10^{19}, we have 3GB per person (in the world). Viewed another way, with 64-bit addressing, you could produce objects at the rate of 100 MB per second for 5000 years before you would run out of addresses (for *each and every* byte). You could store 1 billion full length digitised two hour movies — each digitised to 4 GB each.

Returning to our ideal world, what we would like is:

- infinite amount of memory (i.e. we can hardly imagine a situation when we will run out of addressing space;
- memory allowing very fast access – just like a CPU register;
- memory which is non-volatile, i.e. we can switch off a machine, and when we switch on again, the operating system, programs, and data are in the same state as when we switched off.

We cannot have all this, so compromises are necessary: we can have a small amount of very fast memory; a middling amount of medium speed memory; a lot of slow access storage on disk – which has the added advantage of being non-volatile, but has the disadvantage that it cannot be directly addressed; and, finally, we can have an lots and lots of storage on tapes and other archival media; this latter storage has the disadvantage of being very slow (perhaps requiring the intervention of a person to load the tape or other media).

Figure 10.5 (from (Tanenbaum 1999)) shows the five level memory hierarchy. Figure 10.6 shows the same memory hierarchy analysed according to capacity (size), speed of access and cost.

Data that are currently accessed most frequently should be in the highest level of storage hierarchy. Each offers about an order of magnitude or more faster access than its lower neighbour.

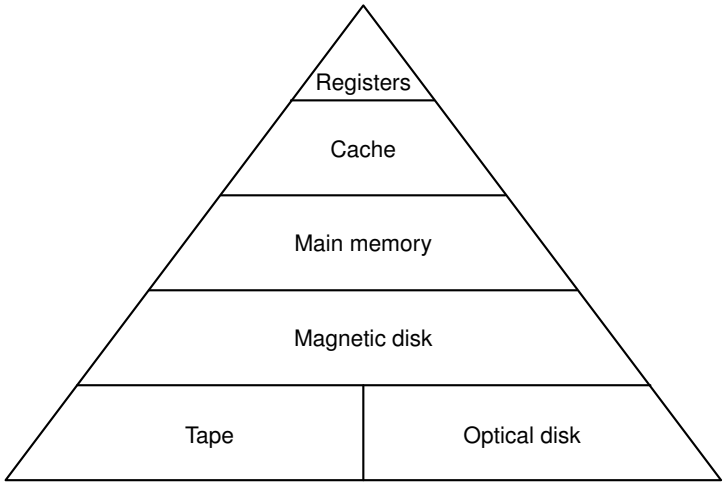


Figure 10.5: Five level memory hierarchy

	Capacity	Access time	Cost per bit
CPU Registers	1--100 bytes	1 CPU cycle	$20x10^-5 per byte (*5)
CPU Cache	256K--2MB	1--2 cycles	$1 per 0.1 MB (*4)
Main Memory	64MB--1GB	10 nanosec. 10 cycles(*1)	$1 per 10 MB
Disk	10GB--100GB	10 ms or 10x10^6 cyc.	$5 per GB
Offline Archive	up to 500 GB per tape	>10 mins (600 sec.)(*2)	$2 per GB (*3)

(*1) based on 100 MHz memory, 1 GHz CPU
(*2) based on time for operator to find and load media
(*3) based on $1.50 for a 650 MB CD-ROM
(*4) equals $1 x 10^-5 per byte
(*5) based on $500 for a recent (2001) Pentium containing 50,000,000 transistors, i.e. $1 x 10^-5 per transistor; assume two transistors per bit, plus four for interface (20 per byte) gives $20 x 10^-5 per byte, or 20 times more costly than cache.

Figure 10.6: Memory Hierarchy Analysed by Capacity, Speed, Cost.

10.13 Self assessment questions

1. Make sure you understand how to navigate a hierarchical file system.
2. Assuming you start with a fresh partition, draw the file/directory hierarchy that exists after *1, *2, ...

```
mkdir os1  *1

cd os1
mkdir notes *2

cd notes *3

edit ch1.txt  (creates file ch1.txt)
...
edit ch2.txt  (creates file ch1.txt)  *3

cd \
mkdir compsys  *4

mkdir notes2
cd notes2
edit pract1.txt  *5

copy pract1.txt \os1\notes\x.txt  *6

cd \
rmdir compsys *7

del notes2\pract1.txt *8
```

10.14 Self assessment questions

1. Describe three principal reasons for adding magnetic disk storage to a computer.
2. Using an appropriate diagram, give a brief description of the layout of data on a typical multiple platter magnetic disk. Ideally, you should mention the terms: track, cylinder, head, sector, block (or cluster, allocation unit).
3. Describe two principal reasons why you might wish partition a disk into more then one partition.
4. When you read/write a disk, describe three delays (latencies) that make up the total access time.
5. Contiguous allocation leads to fast read/write. Why? Explain in terms of latency.
6. In the context of latency (hint: seek latency), discuss why you might want to perform *defragmentation* on your computer system.

7. In a simple implementation of *Linked List Allocation*, each block (allocation unit) in a file contains a field which points to the next block (or is the last block, contains some number such as -1, which indicates end-of-file. Give two disadvantages of this simple implementation of Linked List Allocation.

8. In a simple implementation of *Linked List Allocation*, each block (allocation unit) in a file contains a field which points to the next block (or is the last block, contains some number such as -1, which indicates *end-of-file*.

   (a) Describe how the File Allocation Table (FAT) implementation of Linked List Allocation.

   (b) Describe one advantage, and one disadvantage, over the simple implementation, of the FAT method.

   (c) Describe the principal differences between FAT16 and FAT32 file systems used by Windows.

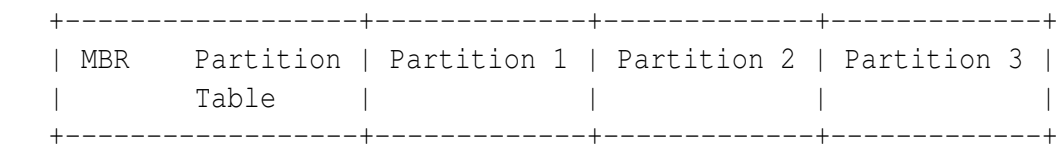
   (d) FAT16 (the original FAT for hard-disks) uses a 16 bit FAT entry. The maximum cluster/block/allocation unit size it can use is 64 sectors, or 32,768 bytes; what is the maximum partition size it can handle?

   (e) Do the same for FAT32, again assuming 32 KByte blocks. What is 2 TB?

   (f) Can you think of any problem associated with such large block sizes?

9. In the so-called FAT16 file system used by some versions of MS-Windows, the maximum possible size of a file is 2 Gigabytes. Explain. Hints: sector size is 512 bytes, block size?, size of binary numbers used for FAT entries? largest FAT entry possible?

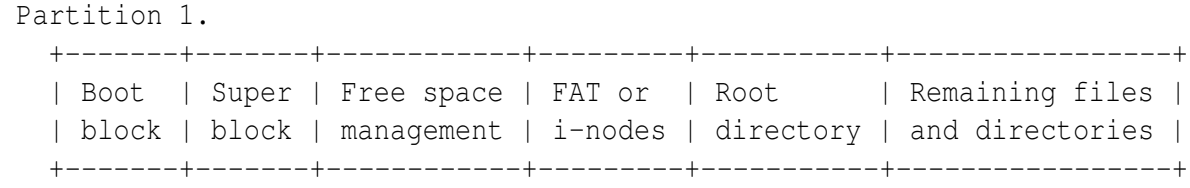
10. The following shows the layout of a disk which has been partitioned into multiple partitions.



   (a) Why might it be necessary to partition a disk into multiple partitions?

   (b) What is the purpose of the section *MBR and Partition Table*?

   (c) A typical partition layout is shown below. Briefly indicate the purpose of any *three* of the parts given.



Chapter 11

Brief Case Study – MS-DOS

11.1 Introduction

MS-DOS stands for Microsoft Disk Operating System. MS-DOS version 1.0 was developed for the original IBM PC which was launched in August 1981. The IBM PC used a 16-bit microprocessor from Intel called the 8088; in fact, the 8088 was a slightly cut-down version of the Intel 8086.

During lectures, I may give additional handouts on MS-DOS..

11.1.1 History

The 8086 was the beginning of the computer family we now know as the Pentium. The family developed as follows: 8086 (16-bits); 80186 – 8086 with some additions to make it easier to use for data communications; 80286 – a faster 8086 with additional addressing range – could address 16 Megabytes, and with some features to allow multitasking; 80386 – the start of true 32-bit computing, actually, the only advance since the '386 has been some *go faster* parts; the 80386 was faster, but most significantly its CPU was 32-bit, and it allowed 32-bit addressing (could now address 4 Gigabytes of memory); 80486 – 80386 with some cache memory on the chip, and the floating point arithmetic unit also on the chip; 80586 (called the Pentium) . . . You know the rest.

We better mention what came before the 8086. In 1971, Intel developed the first microprocessor, the 4-bit Intel 4004; soon after came the 4040. Then in 1973, came the 8-bit Intel 8008, followed shortly after by the Intel 8080. The 8080 brought about the idea of a *personal computer*. Later, a company called Zilog developed a more powerful copy of the 8080, the Z80.

Some reasonably powerful (for the day) microcomputer systems were built around the Z80 and 8080. A small operating system was developed for the 8080/Z80 – Digital Research's CP/M (Control Program for Microprocessors). When the 16-bit 8086 came on the scene in 1979, initially, industrial inertia prevented anyone leaving the Z80–CP/M bandwagon. It took a company with the strength of IBM to introduce 16-bits to the personal computer arena.

Digital Research had started a development of a 16-bit version of CP/M – CP/M-86. When in late 1980, IBM decided to develop the PC, they initially didn't think too much about the operating system; eventually, the started to look around. They got in touch with Bill Gates at Microsoft, who had developed a successful BASIC interpreter for the 8080. Gates agreed to supply a BASIC compiler. But after some thought, IBM realised they needed some semblance of an operating system too. Gates suggested Digital

Research, but CP/M-86 was away behind schedule and, we assume, Digital Research had no idea of what was about to happen to the world of computing.

So IBM go back to Gates — we're in trouble, please help us!. Incidentally, Microsoft had been selling a version of UNIX under licence. UNIX might have done the job, only it needed a hard disk and 100 Kbytes or more of memory. The IBM PC had just a single 160K floppy disk, and 64 K of memory. Gates found a local Seattle company (Seattle Computer Products) who had developed a crude operating system for the 8086 (86-DOS). Seattle Computer Products agreed to sell 86-DOS to Microsoft; Microsoft enhanced 86-DOS to produce MS-DOS. MS-DOS shipped with the first IBM PCs in August 1981.

I have a very nice video which goes through all this history and background — please remind me if you want it shown.

11.1.2 Structure

MS-DOS has a structure along the lines of Figure 11.1. As you can see, MS-DOS allows application programs free rein to directly access the ROM BIOS or even the hardware itself. This freedom is not present in Windows 2000 (or XP) or UNIX/Linux.

The memory layout that exists after booting has completed is shown in Figure 11.2. The total memory addressable by the 8086, and MS-DOS, is 1 Megabyte (20 bits – 16 bits plus 4 additional bits added by a bit of jiggery-pokery). Of this 1 MB, IBM decided to keep from 640K and above for special use. At the time, this wasn't a big deal; the first PCs had 64K of RAM. It was 1985 before PCs started to have the full 640K.

11.2 Processes in MS-DOS

When, at the MS-DOS prompt `A:\`, you type a command like `copy file1.txt file2.txt` (see section 11.3, the following happens:

1. The shell (command interpreter) reads the command;
2. The shell looks to see if `copy` is one of those commands it already has in the kernel;
3. `copy` is in the kernel, so the shell causes a jump to the `copy` part of the kernel; that runs, and when it is done, the kernel passes control back to the shell;
4. If the command is something like `format c:` (not in the kernel), the shell has a look on the current disk directory for a file called `format.exe`;
5. If `format.exe` is there, the shell loads the file into the area just above the shell (marked `program` in Figure 11.2; the shell then causes a jump to the `program` part of memory; that runs, and when it is done, it passes control back to the shell;
6. The same is true for, e.g. `edit file1.txt`; or whatever;
7. The same is also true for a program that you may have written yourself, e.g. `MYPROG.EXE`;
8. The same is also true for a *batch* file like `autoexec.bat` which contains, not a program, but a list of commands.

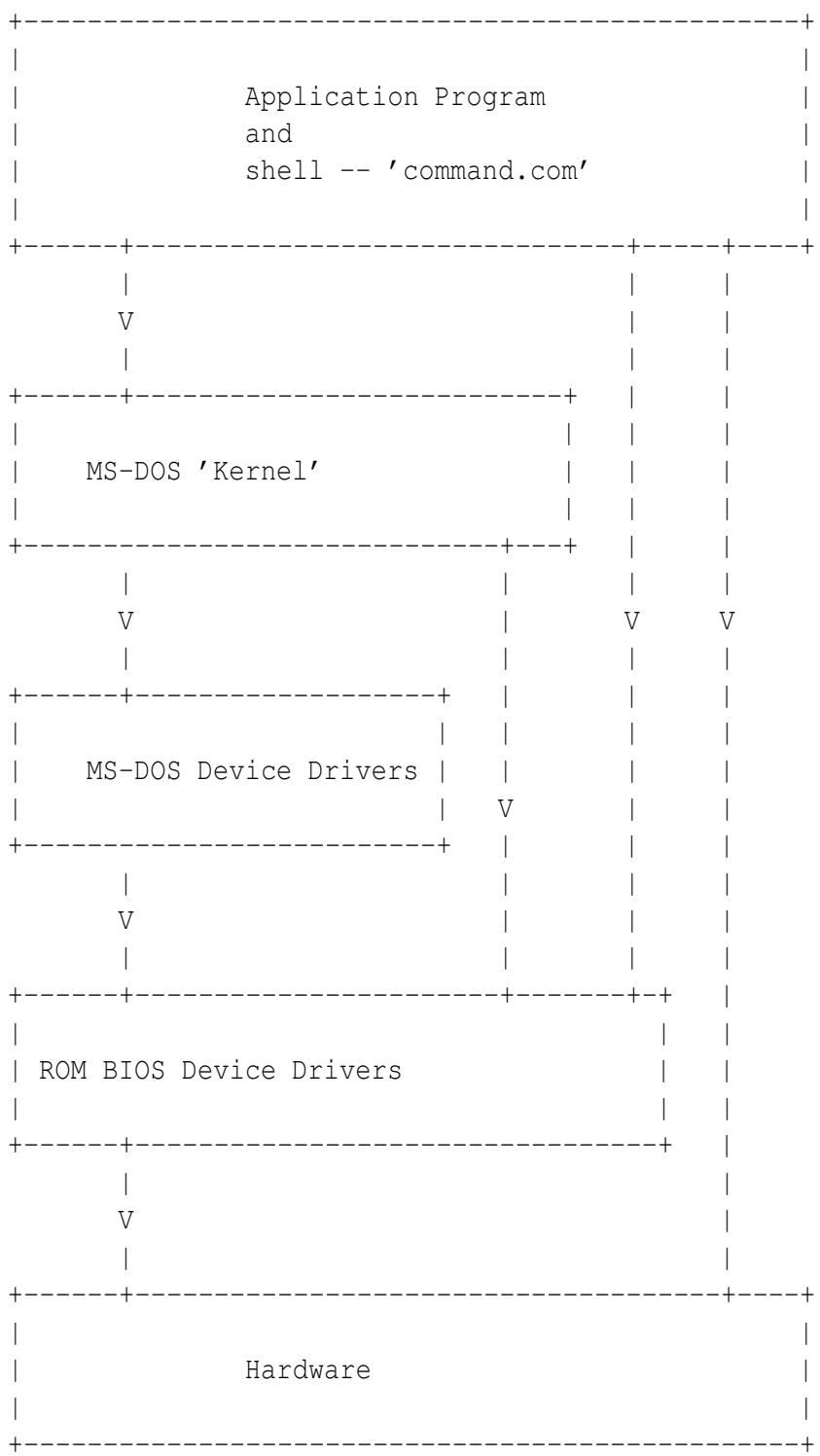


Figure 11.1: Interface Structure of MS-DOS

Figure 11.2: MS-DOS Memory Layout after Booting

Thus, roughly speaking, in MS-DOS the life cycle of a process is as follows:

- The shell waits for a command;
- A program is loaded;
- The program becomes a process when the shell causes a jump to its program area;
- The process runs until it is done;
- The last thing a process does is pass control back to the shell or kernel; and the cycle starts again;

11.3 Some MS-DOS Commands

From: http://www.angelfire.com/geek/dosprompt/commands.html

```
CD     Changes the current directory (folder.)

Syntax:
cd [drive:][path]
cd ..  (Back up 1 level in the directory structure.)
cd \  (Change to the root directory.)

Example:
cd c:\windows\command
  ------------------------------------------------------------------------

COPY    Copy files to another location.

Syntax:
copy [source][destination][/switch]

Examples:
copy c:\autoexec.bat a:\
copy c:\autoexec.bat c:\backup
copy c:\files\help.doc a:\backup
copy c:\files\help.doc a:\backup /v

Switches:
/v  Verifies that the files were copied correctly.

Using Wildcards:
The copy command can be used in conjunction with the  "?"  and  "*"
wildcards.  The asterisk wildcard  "*"  can be used in place of either the
file name,  file extension,  or both.  The question mark wildcard  "?"  can
be used in place of individual characters within a file name,  file
extension,  or both.

Notes:
```

The destination directory must exist before copying files. The copy command will not create a new directory.
I.E. "copy c:\file.txt a:\backup" will only work if the "backup" directory already exists on the A: drive.
To copy files in subdirectories, use the XCOPY command.

DATE Lets you to change the date displayed by your computer.

Syntax:
date [mm][dd][yy]

Examples:
date 07-19-01
date 07-19-2001

DEL Deletes files.

Syntax:
del [drive:][path][filename][/switch]

Example:
del c:\oldfile.txt
del c:\oldfile.txt /p
del c:\junk\oldfile.txt

Switches:
/p Prompts for confirmation before deleting the file.

Notes:
If you are in the directory from which you are deleting files, you do not have to include the path information within the delete command.

DIR Displays a list of a directory's files and sub-directories.

Syntax:
dir [drive:][path][filename]

Examples:
dir c:\
dir c:\windows
dir c:\windows*.exe

Switches:
For a full listing of the DIR command's switches and attributes, type "DIR /?" at the DOS prompt.

Using Wildcards:
The DIR command can be used in conjunction with the "?" and "*"

wildcards. The asterisk wildcard "*" can be used in place of either the file name, file extension, or both. The question mark wildcard "?" can be used in place of individual characters within a file name, file extension, or both.

Notes:
Many of the DIR commands can be specified from within the AUTOEXEC.BAT file, via the "set dircmd" environment table. This will prevent having to re-type the switches and attributes each time.

DOSKEY Starts Doskey.

DOSKEY a TSR (Terminate-and-stay-resident) program which recalls MS-DOS commands, creates macros, and more. Doskey allows the user to cycle through previously typed DOS commands by using the up and down arrow keys. Doskey should be loaded via the AUTOEXEC.BAT file.

EDIT Starts MS-DOS Editor, a full-screen ASCII text editor with drop-down menus.

Syntax:
edit [drive:][path][filename]

Examples:
edit
edit c:\autoexec.bat
edit c:\files\hello.txt

FDISK Displays a series of menus to help you partition your hard drive.

Caution: Deleting or changing a partition deletes all of the data stored on that partition!

Syntax:
fdisk

FORMAT Formats a hard or floppy disk.

Syntax:
format [drive:][/switch]

Examples:
format a:
format a: /q /u

Switches:
/q Quick format. Only for previously formatted disks.
/u Unconditional format. The disk cannot be unformatted.

/f:size Specifies the size of the floppy disk to format. Rarely needed.
/s Copies the system files IO.SYS, MSDOS.SYS and COMMAND.COM to your newly formatted disk.

MKDIR (or MD)   Make a directory.

Syntax:
mkdir [drive:][path]

Examples:
mkdir c:\test
mkdir c:\test\junk

MORE   Used in conjunction with other commands (such as MEM) to display information one screen at a time.

Examples:
type bigfile.txt |more

Notes: The symbol preceeding the MORE command is the "pipe" symbol.

PATH   Sets a search path for executable files.

The PATH statement is not used outside of the AUTOEXEC.BAT file.

Syntax:
path [drive:]path[;..]

Examples:
path=c:\;c:\dos\;c:\windows\;
path=c:\;c:\windows;c:\windows\command\;c:\windows\system\;

RENAME (REN), (MOVE does the same) Allows you to change the name of a file.

Syntax:
ren [drive:][path][oldfilename][newfilename]

Examples:
ren c:\autoexec.bat c:\keepme.txt
ren c:\junk\test.doc c:\junk\test.txt

RMDIR (RD for short) Removes (deletes) a directory.

Syntax:
rmdir [drive:][path]

Examples:
rmdir c:\junk

Notes: Directories must be empty, and contain no subdirectories, hidden, or system files before you can delete them with this command.

TIME Displays the time, or allows you to set your computer's clock.

Examples:
time 04:11p
time 08:30:00a

XCOPY

Copies files (except hidden or system files) and directories, including subdirectories. Using this command, you may copy a directory, all it's subdirectories, and all files contained therein.

Syntax:
xcopy [source][destination][/switches]

Examples:
xcopy c:\files*.* c:\backup
xcopy c:\windows*.* /s c:\backup

Switches:
/s Copy directories and subdirectories (unless empty.)
/e Copy subdirectories, even if empty.
/v Verify that each file is copied correctly.

Using Wildcards:
The xcopy command can be used in conjunction with the "?" and "*" wildcards. The asterisk wildcard "*" can be used in place of either the file name, file extension, or both. The question mark wildcard "?" can be used in place of individual characters within a file name, file extension, or both.

Chapter 12

Multitasking and Process Management

12.1 Introduction

In Chapter 9 we saw how the *operation* of computers, by operators, in the sense of *driving* them or controlling the, evolved along a path: (a) users operating them, (b) to operators, and finally, (c) through various sophistications of *operating system* software to (d) multitasking/multiprogramming operating systems.

Also, along the way, we have noted that operating systems currently sold with *personal computers*, i.e. mostly used in *single user* mode, are, in fact, quite sophisticated multitasking operating systems; i.e. Windows 2000, Windows XP, Linux, even Windows ME and 98/95.

Note that *single user* is no longer an issue: a so-called single user computer may be running many tasks at once: typing into a word-processor, surfing the web, playing mp3 files, running an email client, ... In addition, we note that the software at the heart of Windows 2000 Professional and Windows 2000 Server is the same software; the only difference is that certain additional features are enabled when you purchase a Server licence.

Also, whether a machine is interactive, or batch operated, or a mixture, has little impact on the overall requirement for multitasking, though the choice of scheduling algorithm may depend on whether interactive or batch.

First, we give a demonstration of the effect of multitasking.

Most of this chapter is taken from (Tanenbaum 2001).

12.2 An Example of the Advantages of Multitasking

Figure 12.1 shows the execution of three identical jobs, under two different operating system regimes.

Assume that the jobs/processes/programs are in memory to start with, that they do some computation (taking 10 millisec) which results in the requirement to read a particular record from a disk file (taking 10 millisec), finally they do some more computation (again 10 millisec). Viewed as executions of single programs this may not appear too useful - where do the results go to - but, in fact, the jobs are quite typical of what would be encountered by an operating system during the execution lifetime of three *processes*.

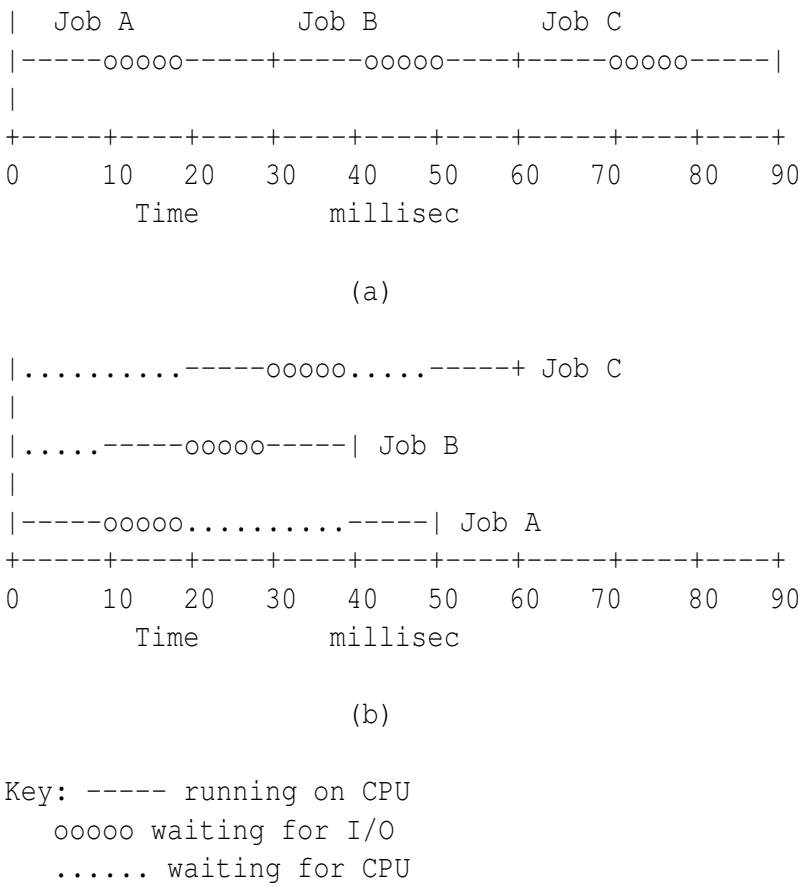


Figure 12.1: Illustration of multitasking (a) single user; (b) multiprogrammed

Figure 12.1(a) shows how the jobs would have to run on a single user system like MS-DOS. The computer is *I/O blocked* during each disk input – regardless of whether the input is by programmed I/O, interrupts, or DMA (see earlier chapters).

Figure 12.1(b) shows the effect of multitasking; when one process become IO bound, the CPU can do processing for another. Thus, a 90 millisec of computer usage can be reduced to 60; add more jobs and you get a greater increase in efficiency.

12.3 Multitasking on a Single User machine?

In addition to the advantages of multitasking given in the previous section, we also, in the introductory chapter 9.1.2, have outlined the rationale for providing multitasking on a machine that will never have more than one user at a time logged in, namely:

- The single user can run many applications simultaneously;
- Multiple applications may cooperate with one another; this means that can put together quite powerful applications by combining separate programs/processes;

In addition, we pointed out the benefits of having multiple user accounts on an apparently single user system, in particular the advantages of having an *Administrator* account which is the only account allowed to modify system software. This security measure can prevent not only malicious damage, but also damage due to errors.

12.4 Components of an Operating System

The work that a modern multitasking operating system performs is a good deal more complex than that of, for example, a simple resident monitor – or even a single user OS like MS-DOS.

The following functions are required:

1. Scheduling. Allowing jobs to enter the queue; then switching the processor from one process to another;
2. Resource allocation. Allocation of the computer's resources to processes: memory, IO devices, CPU time;
3. Resource protection. Ensuring that no process can access any resource which has not been allocated to it; for example: one process should not be able to read the memory allocated to another; nor read another's disk files;
4. Provision of IO services; UNIX and Windows 2000 not only provide IO via system calls (to system subprograms), but bar processes from dealing directly with IO devices;
5. Interrupt handling;
6. Provision of a file system; i.e. provide a set of system calls to do things like: `openANewFile("file1.txt");write("This is some text");` to a file, `closeFile()`, etc.;

   The list of functions just given are often provided by the *kernel* – the central part of the operating system.
7. Analysing and decoding user commands, and converting these into program names. This last function is usually provided by a part of the operating system called the *shell*; early shells were command line (e.g. MS-DOS, `prompt> copy file1 file2`. But, as we know, most interaction these days is via a graphical user interface (GUI).

We now look briefly at a selection of topics that pertain to these requirements.

12.5 Processes

12.5.1 Process Life cycle

The concept of a process is central to multitasking. A rough definition of the term *process* is: *a program in execution*. People have difficulty comprehending the difference between a program and a process. A program is simply a list of instructions; it has no *life*. Consider the cookery recipe example

During its life, a process goes through a series of discrete states, see section 12.2:

- Running – it is currently running on the CPU;
- Ready – it could use the CPU if the CPU were available;

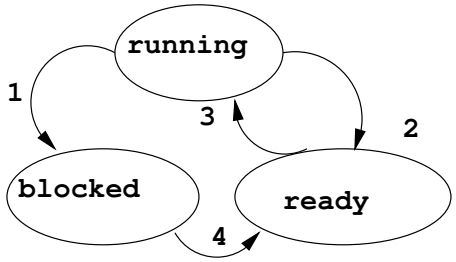


Figure 12.2: Process States: a process can be running, blocked, or ready

- Blocked – a process enters the *blocked* state when it requests some resource that cannot be supplied immediately, e.g. it issues a disk read, but a disk read takes a long time and there is no point in that process waiting twiddling its thumbs while the data is read. The process remains in that state until the blocking need becomes satisfied, at which point it becomes *ready*.

When a job is admitted to the system, a corresponding process is created and placed at the back of the queue and it is marked *ready*. It gradually moves to the top of the ready list – see scheduling below; when the CPU becomes available the process makes a state transition from *ready* to *running*.

To prevent any one process, call it process A, monopolising the system – i.e. to provide *time-sharing* it must be possible possible to stop the process after it has consumed its allocated amount of CPU time. This is done via an interrupting clock (hardware) which allows a specified time quantum – a *time-slot* to pass; if process A is still using the CPU the clock generates an interrupt, causing the system to regain control. Process A now makes the state transition back to *ready*, i.e. A is put back in the ready queue.

The next process, process B, on the ready queue is now put into the *running* state; and so on ....

If a running process initiates an I/O operation before its time-slot expires then the process is forced to release the CPU; it enters the *blocked* state and releases the CPU.

If a process is in the blocked state and the event for which it is waiting happens, then the process makes the transition from blocked to ready.

12.5.2 Process Control Block (PCB)

When a process makes the transition from running to either blocked or ready (in which case it relinquishes the CPU), it is necessary to store some details, for example: just as in a subprogram call, where it has got to (the program counter (PC)), but more. The *process control block (PCB)* is a record containing important current information about the process, for example: current state of the process, unique identification of the process, the process's priority, open files & IO devices, use of main memory and backing store, other resources held, copies of the registers (see chapter 7).

12.5.3 Context Switch

When the CPU switches from one process to another, this is called a *context switch*. A context switch is a bit like a subprogram call, but requires more housekeeping effort, in particular maintenance of the PCB. The CPU effort required to make a context switch is one of the factors involved in choosing a scheduling algorithm, see section 12.7: context switching can take a lot of processing time and if you are not careful, the CPU can spend all its time on context switching.

On the other hand, since there must be a context switch every time the kernel gains control, it does not usually cost much extra for the kernel to make a decision and allow a different process to run.

12.5.4 Thread versus Process

The term *thread* refers to the 'thread' of control, i.e. how control weaves its way through memory as a program executes.

You may have multiple *threads* within one process. For example, in a word processor, you might have: (i) a thread handling input; (ii) one handling formatting for the screen; (iii) one handling spell-checking; (iv) one handling periodic saves to file; etc. But each operates on the same memory space. In contrast, separate processes have separate memory spaces.

Because they do not have separate memory spaces, and because switching between them causes less kernel processing overhead than for processes, threads are also called *lightweight processes*.

12.6 The Kernel

All of the operations involving processes are controlled by a portion of the OS called its *kernel*. The kernel represents only a small portion of the code of the entire OS but it is intensively used.

One very important function of the kernel is interrupt processing. In large multi-user systems, the kernel is subjected to a constant barrage of interrupts. Clearly, rapid response is essential: to keep resources well utilised, and to provide acceptable response times for interactive users.

In addition, while dealing with an interrupt, the kernel disables other interrupts. Thus, to prevent interrupts being disabled for long periods of time, the kernel is designed to do the minimum amount of processing possible on each interrupt and to pass the remaining processing to an appropriate system process.

The kernel provides:

- interrupt handling;
- process creation & destruction;
- process state switching and scheduling;
- interprocess communication;
- manipulation of PCB;
- support of IO activities;
- support of memory allocation & deallocation;
- support of the file system;
- support of system accounting functions.

12.6.1 Kernel Privileges

The kernel must have *privileges* that allow it to do execute instructions that normal processes cannot access, e.g. disable interrupts, access IO devices. Privilege is usually conferred by the (hardware) mechanism of CPU mode (either *privileged* or *user*) which is stored in the CPU hardware, and is only changeable by a process in *privileged* mode, i.e. the kernel. User level processes, e.g. processes such as a running Word program, run in *user* mode.

In the original Intel 8086/88 and and to a greater extent in the 80286, there was limited support for privileged modes; however, in the 80386 and after, modes are provided that support kernel privileges and thus full multitasking.

As we have mentioned above, one of the major activities of the kernel is interrupt servicing. Let us concentrate on one important type of interrupt, namely, the timing interrupt that the kernel uses to switch to the next process when it is performing multitasking.

The key to each interrupt is its *interrupt vector* (or, equivalently, its IRQ). The interrupt vector gives the address of the interrupt service subprogram. Now, obviously, from the point of view of a malicious process that might wish to grab control from the OS, that interrupt service routine (and the interrupt vector) are the key. If the malicious process can overwrite the interrupt vector (to point to its own program area) and can overwrite the interrupt service routine to do whatever it (the malicious process) wants — for example, jump to its code — then the OS has lost control. Being able to overwrite an interrupt vector, or the kernel code, is like being able to change the door locks on the bank and the safe to your own.

Consequently, the key to the OS retaining control is to ensure that no process can write to any part of the OS's memory, including the interrupt vector area.

In fact, this problem is the same problem as stopping separate processes from interfering with one another's memory area. So we will discuss both under the same heading.

12.6.2 Memory Protection

The challenge is to stop processes interfering with one another's memory area and with the OS's memory area. This is easily handled by having, in hardware, but only writable by the OS (in priviledged mode) a *base register* and a *limit register*.

Take the case of a process, A, of size 16 MB (0x1000000) loaded at 32M, in Figure 12.3. The *base register* contains 32M; and the *limit register* 48M - 1. Now, the hardware, or some mixture of hardware and the kernel will allow process A (size 16 MB) to access physical memory only between 32M and 48M - 1. Addresses outside that range will cause a violation which will be reported to the kernel and the process will be halted. For those of you who use Linux or UNIX, this is what causes a *segmentation fault*.

Then whenever process B (size 6M and loaded at 48M) is running, the base and limit registers contain, respectively, 48M and 54M -1.

This means that we are not only protecting the kernel and the interrupt vector area, but also protecting one process's memory area from other processes.

The values to be placed in the base and limit registers are other things to be stored in a process's process control block.

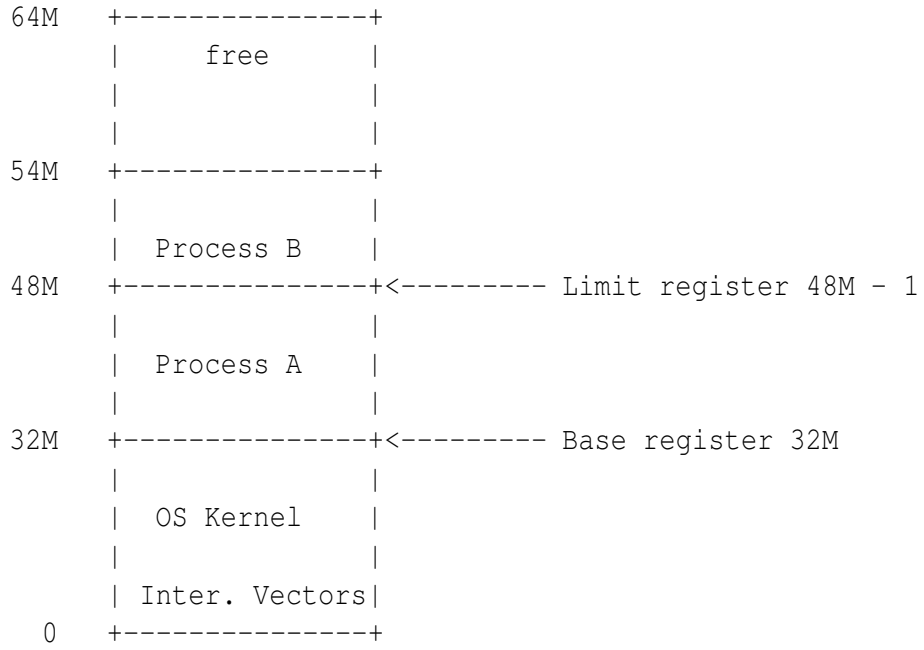


Figure 12.3: Operating System plus two processes; process A is running

Incidentally, overwriting kernel memory with your own code is one way of *cracking* an operating system and getting control of a machine. And once you are in privileged mode, you can do anything! One method of overwriting kernel memory is by overflowing the stack area of a subprogram (see Computer Systems Chapter 7). If you want further information on the basic principle, remind me in a lecture.

12.7 Scheduling

Scheduling is the process of switching the CPU between the currently competing processes.

There are several competing goals that scheduling policies aim to fulfill. One goal is *good throughput*, i.e. getting through as many jobs as possible. One way of fulfilling this goal is to minimize the number of process switches which, in turn, minimizes the amount of housekeeping involved in context switches (see section 12.5.3.

However, another goal is responsive service – give an interactive user the impression that he/she has sole use of the machine.

It is not difficult to see that, in an interactive system, these two goals may conflict.

Before discussing further objectives of scheduling algorithms, we need to identify three broad categories of computer systems.

12.7.1 Batch, Interactive, Real-time

Batch Systems The job is submitted via a disk file, no other interaction is required; the output is another file, plus maybe some printout. Examples: processing of a payroll, daily accounting jobs in a bank.

Interactive Systems Need no description.

Real-time Systems Real-time computer systems are characterised by the urgency with which the must respond. A typical real-time system is one which controls a manufacturing process; for example, liquid A is heated to 90 degrees, kept at that temperature for 10 minutes, then the heater is switched off and the liquid allowed to stand for 5 minutes, it is then mixed with liquid B and allowed to stand until the temperature reaches 45 degrees, then ...

The point of *real-time* is that there are likely to be severe consequences if liquid A is held at 90 degrees for more (or less) than 10 minutes; or, if the point at which the temperature reaches 45 degrees is missed – for example, the liquid could solidify and destroy itself and the machine.

Examples such as those above could be called *hard* real-time tasks.

An example of a *soft* real-time task would be an audio player on a PC or across a network; you don't like breaks, but it isn't the end of the world if you get one.

12.7.2 Preemptive versus Non-preemptive Scheduling

Preemptive scheduling means that the kernel can interrupt the execution of a process before it has completed. *Non-preemptive* scheduling picks a process and allows it to run to completion.

Clearly, preemption is necessary in interactive systems. However, each preemption requires a context switch, see section 12.5.3. In a batch system it may be possible to avoid the cost of context switching and use a non-preemptive scheduler; on the other hand, what about an erroneous or malicious process that attempts to run forever?

Essentially, nearly all interactive and real-time systems need to use *preemptive* scheduling. Current Windows, UNIX, and Linux use preemptive scheduling.

12.7.3 Cooperative versus Preemptive Scheduling in Windows

MS-DOS had no scheduling to speak of. You typed: `prog` and `prog.exe` started and ran to completion.

Windows, up to Windows 3.1, was little more than a GUI pasted on top of MS-DOS. Windows 3.1 had *cooperative* multitasking; cooperative multitasking was also use in Macintosh operating systems, up to the current one, OS X. Cooperative multitasking, as the name suggests, requires the cooperation of the process; at frequent intervals, the process, via a system call, should return control to the kernel – thereby asking "kernel, if you want, you can have the CPU, and give it to another process if you wish".

This required disciplined (and correct, and difficult) programming: the programmer had to ensure that the system call was included at points in the program – somewhere just the right frequency of system calls would occur. Too few calls and the process cold monopolise the system; too many, and the time would be wasted on context switches.

Of course, it is not difficult to imagine the problems caused by erroneous or malicious systems. In addition, application programmers had to think about scheduling – something that should be far from their minds.

Windows 95 introduced preemptive scheduling. However, backwards software compatibility, the great bugbear of all Microsoft and Intel systems, meant that it had to support software using the cooperative method as well. Hence, in 95, and 98 and Me, you cannot really say that the kernel is in full control.

Note: in some discussions of cooperative versus preemptive scheduling in Windows, confusion is caused by the mention of 16-bit versus 32-bit applications. 16-bit applications refer to applications developed

for older versions of Windows or DOS; more likely than not they assume cooperative multitasking, or none at all (MS-DOS). However, the issue is not in the addressing used (16-bit or 32-bit), but in the scheduling mechanism.

12.7.4 Goals of Scheduling Algorithms

The following identifies some goals of a scheduling algorithm/policy according to the type of system.

All systems
1. Fairness – all processes should be treated equitably;
2. Predictability – regardless of the load on the system, it should be possible to estimate how long the job will take to complete;
3. Enforce priorities;
4. Avoid indefinite postponement – as a process waits for a resource its priority should grow (so-called aging);
5. Degrade gracefully under heavy loads;
6. Balance – keep all parts of the system busy; processes using underutilized resources should be favoured; give preference to processes holding key resources.

Batch systems
1. Throughput – complete as many jobs as possible per unit time;
2. Turnaround – minimise time between submission of a job and its completion;

Interactive systems
1. Response times – maximize the number of interactive users receiving acceptable response times;
2. Be predictable – a given job should run in about the same amount of time regardless of the load on the system;
3. Give better service to processes exhibiting desirable behaviour, e.g. low paging rates – see next chapter;

Real-time systems
1. Meet deadlines;
2. Be predictable and degrade gracefully, e.g. if the going is getting really tough, retreat into a safety mode.

The next section outlines some scheduling algorithms.

12.8 Scheduling Algorithms

In the lecture, we'll discuss the analogy of a bank teller deciding how to most efficiently serve a ques of customers in a bank. I know they all use *first come first served*, but there would be better ways!

12.8.1 First Come, First Served (FCFS)

Under FCFS, the scheduler runs each process until it either terminates or leaves because of IO or other resource blockage. Processes arriving in the queue wait in the order that they arrive.

FCFS is *non-preemptive*, so a process is never blocked unless by some action of its own, e.g. requesting a service (resource) that takes some time to provide.

FCFS is usually unsatisfactory for interactive systems because of the way that it favours long processes — like the person in the bank queue in front of you with the weeks takings from a shop!

12.8.2 Round Robin (RR)

The intention is to provide good response ratios for short as well as long processes. This policy services (allocates CPU to) a process for a single time slot (of length = q seconds), where q = between 1/60 and 1; if the process is not finished after q seconds, it is interrupted at the rear of the ready queue & will wait for its turn again. New arrivals enter the ready queue at the rear.

RR can be tuned by adjusting q. If q is so high that it exceeds the overall time requirements for all processes, RR becomes the same as FCFS. As q tends to 0, process switching happens more frequently and eventually context switches occupy all available time. Thus q should be set small enough so that RR is fair but high enough so that the amount of time spent on context switching is reasonable.

Obviously, RR is *preemptive*.

12.8.3 Shortest Process Next (SPN)

SPN (like FCFS) is non-preemptive. It tries to improve response for short processes over FCFS. But, it requires explicit information about the service time requirements for each process. The process with the shortest time requirement is chosen.

One problem with SPN is that long processes wait a long time or suffer starvation. Short processes are treated very favourably.

Another problem with SPN is that there has to be some way of estimating the time requirement; this may be based on previous performance.

12.8.4 Shortest Remaining Time (SRT)

Shortest Remaining Time (SRT) is a preemptive version of SPN. Like round robin, it uses a time sharing scheme, but may not schedule the processes in exact order that they are in the queue. The scheduler uses not only the estimate of total time requirement, but decrements time requirement as the process runs for a time quantum.

12.8.5 Priority

Priority scheduling requires that each process is assigned a priority. At each scheduling event, the process queue is sorted according to priority, and the process of highest priority chosen.

Forms of priority scheduling can be used in interactive systems, real-time systems, and in batch systems. In many real-time systems, it is necessary to have some priority mechanism.

In some cases, priorities may be updated dynamically.

12.8.6 Conclusion

- Preemption is costly but may be worth it.
- In round-robin the time-slot should be large enough so that preemption cost does not become excessive, e.g.. if a process switch costs 100 microseconds, the time slot, q, should be about 100 times as long.
- Some policies are more expensive than others to implement. FCFS requires only a queue;
- OSes sometimes employ hybrid policies, with RR for short processes and some other method for long.
- Memory management, see next chapter, affects scheduling. Processes that need a substantial amount of memory are often given larger time-slots so that they can get more work done while they are occupying main memory.

12.9 Self assessment questions

1. The Figure below shows how three jobs would run on a single user system like MS-DOS. Explain how multitasking can increase the throughput of the system (i.e. allow the jobs to be run in a shorter time).

```
|  Job A          Job B          Job C
|-----ooooo-----+-----ooooo----+-----ooooo-----|
|
+-----+----+----+----+----+----+-----+----+----+
0     10   20   30   40   50   60   70    80   90
        Time        millisec

Key: ----- running on CPU
   ooooo waiting for I/O
```

2. The concept of a *process* is central to multitasking operating systems. A process *state transition diagram* is shown in Figure 12.2.

   Give a brief explanation of each of the three states along with a brief explanation of typical circumstances that would cause a process to make each of the transitions 1, 2, 3, and 4.

3. Identify *five* objectives of a good scheduling algorithm; assume that the operating system is aimed primarily at interactive operation (as opposed to batch or real-time operation).

4. Windows 2000 (and NT), UNIX and Linux use *pre-emptive* multitasking. However, Windows 3.1 used *cooperative multitasking*. Distinguish between the two.

5. Explain the term *context switch*; how does the *cost* of a context switch affect the choice of scheduling algorithm.

6. From the point of view of *scheduling*, briefly explain the different requirements imposed by the following types of system: (a) batch, (b) interactive, (c) real-time.

7. Explain the difference between *preemptive* and *non-preemptive* scheduling.

8. (a) Explain how *round-robin* preemptive scheduling works.

   (b) Explain the need to compromise in choosing the time slot used in *round-robin*. Hint: context switch, response time.

   (c) In a purely *batch* operated computer system, explain why *round-robin* scheduling may be inappropriate.

Chapter 13

Memory Management

13.1 Introduction

Let us take a multitasking system, running on a system with 64 MB memory; the kernel requires 32 MB, and we have three processes, ProcessA which needs 16-MB, ProcessB which needs 6-MB, and ProcessC which needs 4-MB. We can fit all into the available memory at once, see Figure 13.1. Hence, multitasking is only a matter of switching control between the three processes and the kernel.

Relocation One problem is relocation. You may wonder how a program knows where it will be located in memory. Frequently in the sections on assembly programming, we assumed that a program would start at 0x100, with the area 0–0xff kept for system data, and that all addresses are related to 0. The solution is that programs are *relocatable* and, in fact, work on a *logical* address space rather than a *physical* (real) address space.

If our program for ProcessA in Figure 13.1 is to be relocated to such that its addressing starts at 32M instead of 0 (i.e. it is now loaded at 32 MB + 0x100) (0x2000100) then all addresses need to have 0x2000000 added to them.

This is all easily handled by having a *base register* that is loaded each time the process runs. In the case of the example above, the base register is loaded with 0x2000000 and all memory accesses have their addresses this value added to their logical address (e.g. 0x105) to get the physical address (e.g. 0x2000105).

Recall from section 12.6.2 that the operating system, with help from hardware, must stop processes from interfering with one another's memory area. This is easily handled by having, in addition to a relocation *base register*, a *size register* or a *limit register*.

The Big Problem – OS and Processes cannot all fit into memory The big problem, however, is when the OS and the current processes (even if there is just one of them!) will not all fit into memory at once.

In fact, it's rather confusing to think about memory management and process management together — they are quite separate activities. Therefore, from now on, I'm going to forget about multiple processes (multitasking) and consider just the problem of the OS and one process requiring more memory than is available.

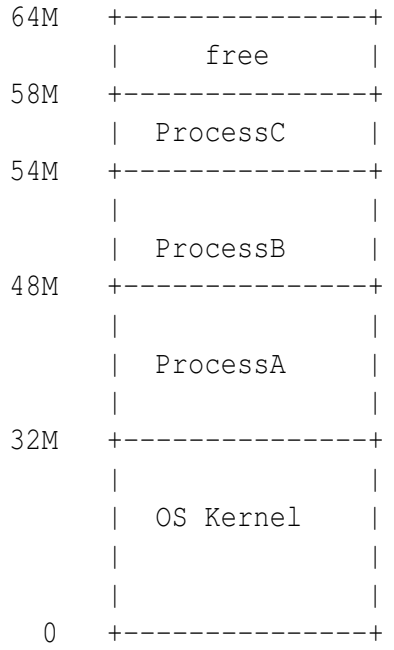


Figure 13.1: Operating System Kernel plus three processes

13.2 Virtual Memory

The OS must keep as many processes as possible in memory at once. As mentioned above, this problem is very similar to that of a single program (and its data) which is larger than the physical memory available. The objective of *virtual memory* is to have as much as possible of the program in main memory, with the rest on disk, but with software in the operating system, and some hardware, which allows swapping between memory and disk in a way that interferes as little as possible with the running of the program.

A guiding principle is that the more frequently data are accessed, the faster the access should be. If swapping in and out of main storage becomes the dominant activity, then we refer to this situation as *thrashing*.

But please note: *virtual memory management is largely independent of process management*; the switching between processes that is done by a scheduler may sometime be referred to as swapping, but that's an entirely different from virtual memory swapping.

The relationship between each layer and the one below it can be considered as similar to that between cache and main memory, or between virtual memory stored on disk and virtual memory stored in the actual physical memory. If we need to access an item from a higher level then access is significantly slower. We call such a situation a cache/virtual memory *miss*. In contrast a cache/virtual memory *hit* is when we find the data we need in the faster memory and need not turn to the associated slower memory.

Physical memory is the hardware memory on the machine, usually starting at physical address 0. Certain locations may be reserved for special purposes; the kernel may reside in in the low addresses. The rest of physical store may be partitioned into pieces for the processes in the ready list. Each process may have its own *virtual memory* – the memory perceived by the process. As far as the process is concerned all address references refer to virtual space. Each process' virtual memory is limited only by the machine address size.

However, we emphasise again that the issue of memory management is largely independent of processes and whether we have have multitasking. Thus, virtual memory can also be very useful even in a single

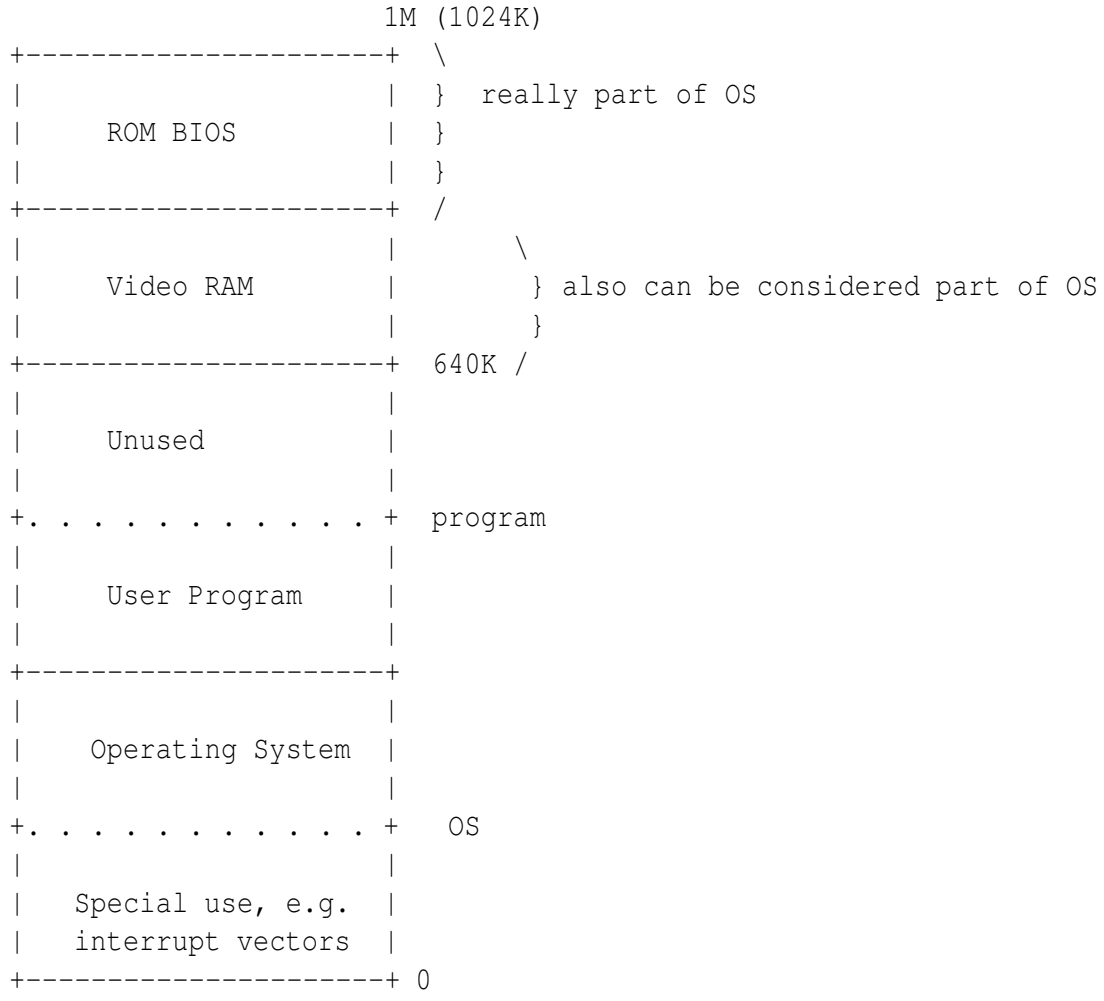


Figure 13.2: Single Process System

user system – if the computer has a larger address space than physical memory, and if programs need a large amount of memory; compare *overlaying* – in which the programmer takes responsibility for segmenting the program into chunks that will fit into the main memory.

13.2.1 Working Without Virtual Memory

The simplest situation is that of a single process system (e.g. MS-DOS), shown in Figure 13.2. We show the operating system divided into four parts.

We also show the constraints (OS + program = 640K or less) imposed by the 16-bit (really 20-bit) addressing used by the Intel 8086; but, again, these do not affect the principle of what we are discussing.

The main point of Figure 13.2 however, the main point is that we have memory divided between: (a) the operating system; (b) the single program; and (c) unused. When the program exceeds available memory (note that it cannot stray into 640K and above territory. Why? – there are at least two reasons), you either try to make the program smaller, or you rewrite it with *overlaying*.

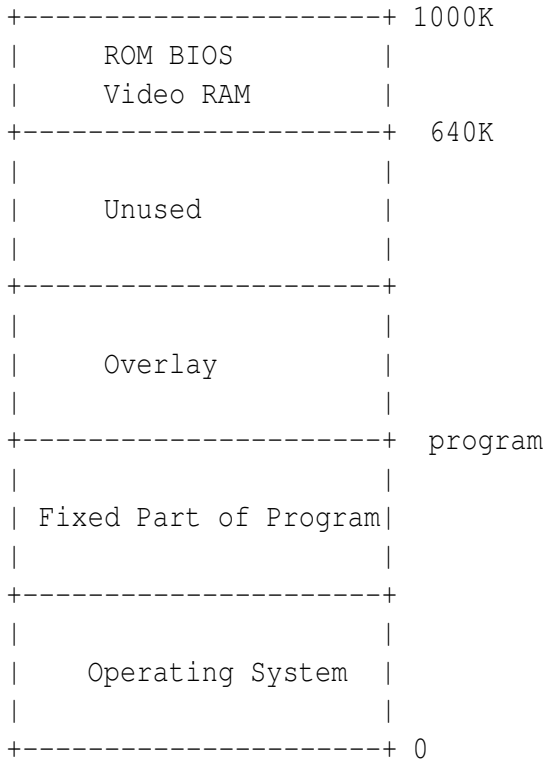


Figure 13.3: Single Process System

13.2.2 Overlaying

Figure 13.3 shows the situation with *overlaying*. Here the programmer divides the program into: (a) a *fixed* part; (b) two or more overlay parts.

The idea is best described by an example. lets say we have a program which does payroll. There are even major parts:

(0) initialise;

(1) read in the time sheet data, pay rates, income tax and other deductions data;

(2) compute hours worked by each employee;

(3) compute pay for each employee;

(4) compute deductions;

(5) print pay slips;

(6) write pay data to file.

Let us say that the system has been analysed and designed and that programmers have found that the program is too large to fit into program memory all at once, but that it can be split into seven roughly equal modules (overlays) corresponding to the parts given (obviously, it is highly unlikely that overlay sections would coincide exactly with functional sections, . . . this is merely an illustrative example.

We have also a *fixed* (main) part that contains mainly the data area – we must have a common data area that stays in memory all the time.

How does the program run? First we load the main/fixed program part; next, we request overlay 0; that initilises data areas etc. Now we request overlay 1, which get written *over* overlay 2; overlay 2 is executed; then overlay 3, etc.

This works fine, but it is a big nuisance for programmers to have to artificially divide programs. Virtual memory comes to the rescue.

13.3 Paged Virtual Memory

Note: what is covered here is difficult to write about. But it's really simple. In class, I'll draw diagrams on the board and do all sorts of antics and arm waving to make it simple!

13.3.1 Introduction

The most common virtual memory technique is paging. Paging is designed to be transparent to the process — like all virtual memory. (There is another form of virtual memory — *segmented* virtual memory, but we do not cover that.)

The program and data of each process are regarded by the OS as being partitioned into a number of *pages* of equal size. The computer's *physical* memory is similarly divided into a number of pages. Pages are allocated among processes by the OS. At any time, each process will have a some pages in memory, while the remainder is in a *swap file* (or, in Linux, the swap partition).

The real point behind paging is that, whenever a process needs to access a location that is in a page that is in secondary memory, the page is fetched into main memory, invisibly to the process, and the access takes place, almost as if the datum had already been in memory. Thus, the process thinks that it has *all* of its virtual memory in main memory.

Let me emphasise again that the remainder of this discussion can proceed as if we are only concerned with a *single* process, which happens to be using a larger amount of (virtual) memory than can fit in physical memory. In addition, we shall simplify the arithmetic by assuming a page sizes of 1000 because it is easier to write the examples and I don't have to depend on people knowing 1024, 2048, 3072, 4096, ...

The process's virtual memory is divided into pages. Page 0 refers to locations 0 to 999, page 1: 1000 to 1999, and so on, ...

We are going to imagine a system with just 52,000 bytes of memory — not very realistic in this day and age, but it is quite adequate to demonstrate the basic principles. Also, we imagine that the kernel takes up 50,000 bytes of that. That leaves 2,000 bytes of actual memory for user processes.

First of all recall what we said about a simple OS like MS-DOS. The kernel loads the program and data, passes control to the program/process; the process runs, ... If there is not enough memory space for the process, it's tough luck, it cannot be executed. The situation in Figure 13.4(a) shows a small 2000 byte process loaded at 50,000 (above the OS); everything is fine — the process fits in the available memory.

We discusses *overlaying* as a solution that used to be used; in this case, the programmer splits the program into chunks that are small enough to fit into memory (one at a time). Usually, there would be

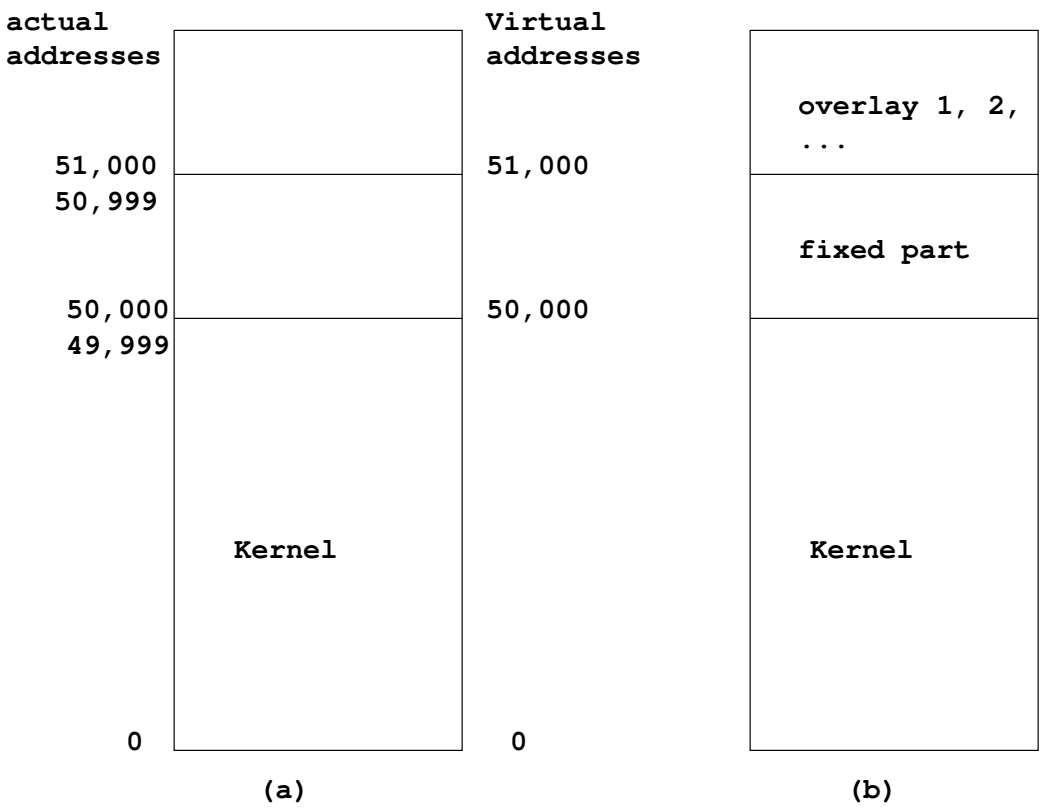


Figure 13.4: (a) Memory map of a process in a non-memory managed OS; (b) Using overlaying.

a fixed part (mainly the data) that would sit in memory all the time, and successive chunks (overlays) would be loaded into the remaining memory — one at a time as they are needed. Figure 13.4(b) shows the memory map in the case of overlaying.

As I have said, the principles of memory management in general, and virtual memory in particular, can best be understood if we ignore multitasking — we are going to concentrate on the task of executing a process whose overall memory requirement is greater than what is available.

13.3.2 Virtual Memory

The principles behind *virtual memory* are as follows:

1. Load as much as possible of the process into actual/physical memory;
2. Keep a copy of the complete process (its memory image) in a disk file; this is called the *swap* file. (In Windows 2000 it's a file, in Unix, it's a complete partition, but the difference is insignificant.)
3. The virtual memory manager (in the kernel) organises the process's (virtual) memory into chunks called *pages*; pages are normally 512, 1024, 2048 or 4096 bytes — see main notes for a discussion of page size.

   Some pages are *in* main memory, others are not — but *all* are in the swap file.
4. If in 1., only part of the process could be loaded, the process may run fine for a while. In this situation, the memory manager has nothing to do. All needs to be done is that the hardware that supports virtual memory management does appropriate address translation;

However, at some stage, the process will try to access a memory address that is not in physical memory. This is called a *page fault*. When a page fault occurs, the memory manager must read in from the swap file the page that is needed. Note: this is not a happy situation for a process; as we discussed, reading a disk file takes a minimum of 10-millisec. while reading memory takes maybe only 10-nanosec.

Which page to read in is obvious. However, where to put it — which of the currently *in memory* pages to replace? That is the subject of *page replacement policy* — see main notes.

13.3.3 Paged Virtual Memory — some examples

We are going to look at the performance of paged virtual memory on a system with just two (2) 1,000 byte pages available for user programs. In the first example, we imagine the following sequence of memory accesses:

```
5051, 8997, 5052, 8998, 5053, 8999, 5054, 9000, 5055, 9001
```

Incidentally, such a set of memory accesses could result from the execution of a program like:

```
address    instruction
5051       lodd 8997
5052       stod 8998
5053       lodd 8999         etc.
```

We are going to assume that, when it is loading the process, the kernel can see enough in the executable file to be able to choose the most appropriate pages to load into memory. This is a bit of a fudge, but what I am interested in is how paged virtual memory works once the process is up and running — the details of the startup are hard to explain and irrelevant for the purposes of this introduction.

Don't forget that a full copy of the process will be created in a swap file.

Just after the process has started, the memory map is as in Figure 13.5(a) — the pages corresponding to 5,000–5,999 (page 5) and 8,000–8,999 (page 8) are loaded into the two available pages.

Everything runs fine until we must access `9,000` — this cause a *page fault*. A swap must take place. Page 9 (9,000–9,999) of the process must be read in from the swap file. Where to put it? I.e. what to *swap out*? This depends on the *page replacement policy*. (You will need to have a quick look ahead at section 13.5).

If the page replacement policy is *least recently used*, then page 5 has just been used, and page 8 is the least recently used. This results in a memory map as shown in Figure 13.5(b). We get to the end without further page faults. Ignoring startup, we get by with a total of *one* page fault.

On the other hand, if *first-in first-out* is the policy, things remain the same up until we access 9000, i.e. as in Figure 13.5(a). With first-in first-out, page 5 was the first page read in (assume that anyway) and it is it which is replaced by page 9, see Figure 13.6(c) (next page).

Now what happens when we access `5055`? Page 5 is no longer in, so it must be swapped in — replacing page 8, see Figure 13.6(d).

Ignoring startup, we encounter a total of *two* page faults.

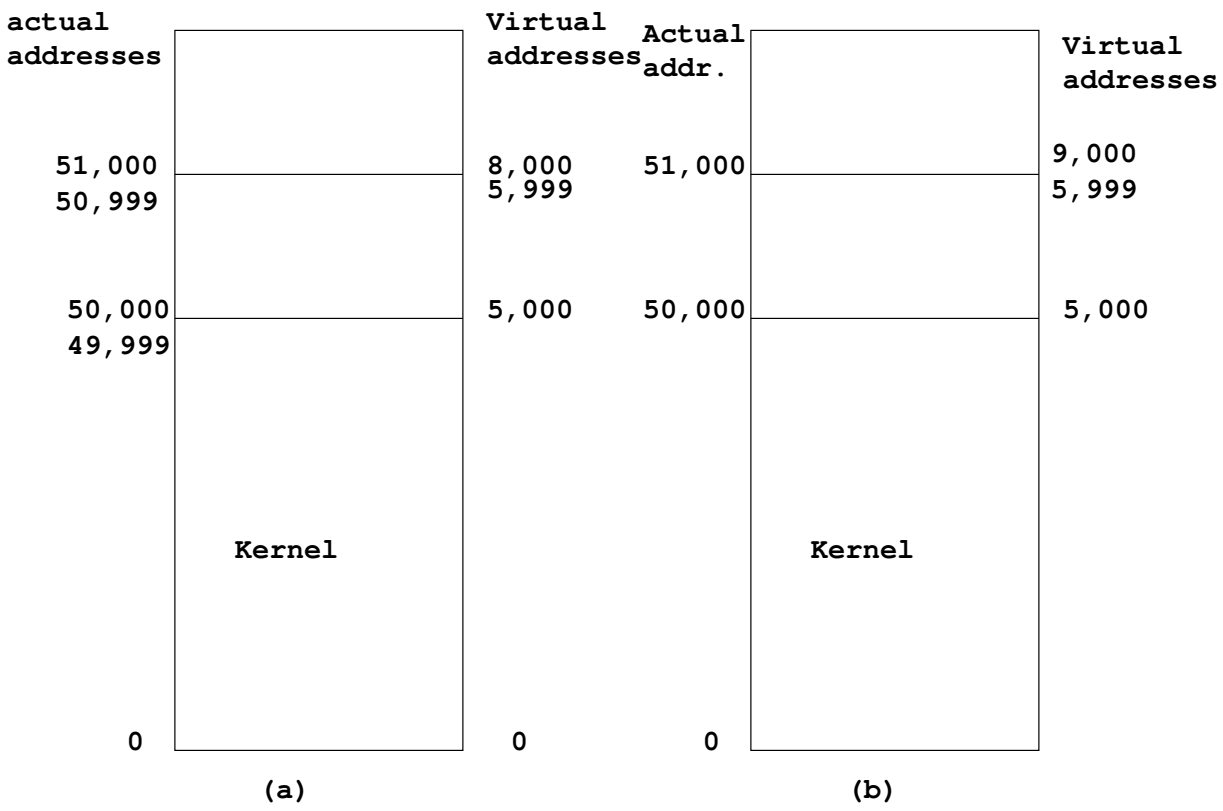


Figure 13.5: (a) Memory map showing pages 5 and 8 in memory; (b) when page 9 replaces page 8.

We have just demonstrated a situation in which *least recently used* is superior or *first-in first-out*. In general this will be the case. So why does Windows 2000 use *first-in first-out*? Answer: *first-in first-out* is easier to implement; and if you have plenty of memory, the poorer performance of *first-in first-out* will not be so noticeable.

Another example. Let us now look at an example that produces many page faults; memory accesses are:

```
5051, 8997, 5052, 9997, 5053, 10997, 5054, 11997, 5055, 12997
```

Such a set of memory accesses could result from the execution of a program like:

```
address    instruction
5051       lodd 8997
5052       stod 9997
5053       lodd 10997         etc...
```

As I will explain in class, a program like this is quite possible — and it is quite possible for a simple mistake make the earlier program turn into this one.

At the the beginning the situation is as in Figure 13.5(a); and when `9,997` is accessed the situation is as in Figure 13.5(b); already we have a page fault. Then when `10,997` is accessed, we have Figure 13.7(a) and when `11,997` is accessed, we have Figure 13.7(b). And we have another page fault when `12,997` is accessed.

Ignoring startup, we encounter a total of *four* page faults. If the whole process was like this, we would experience *thrashing*.

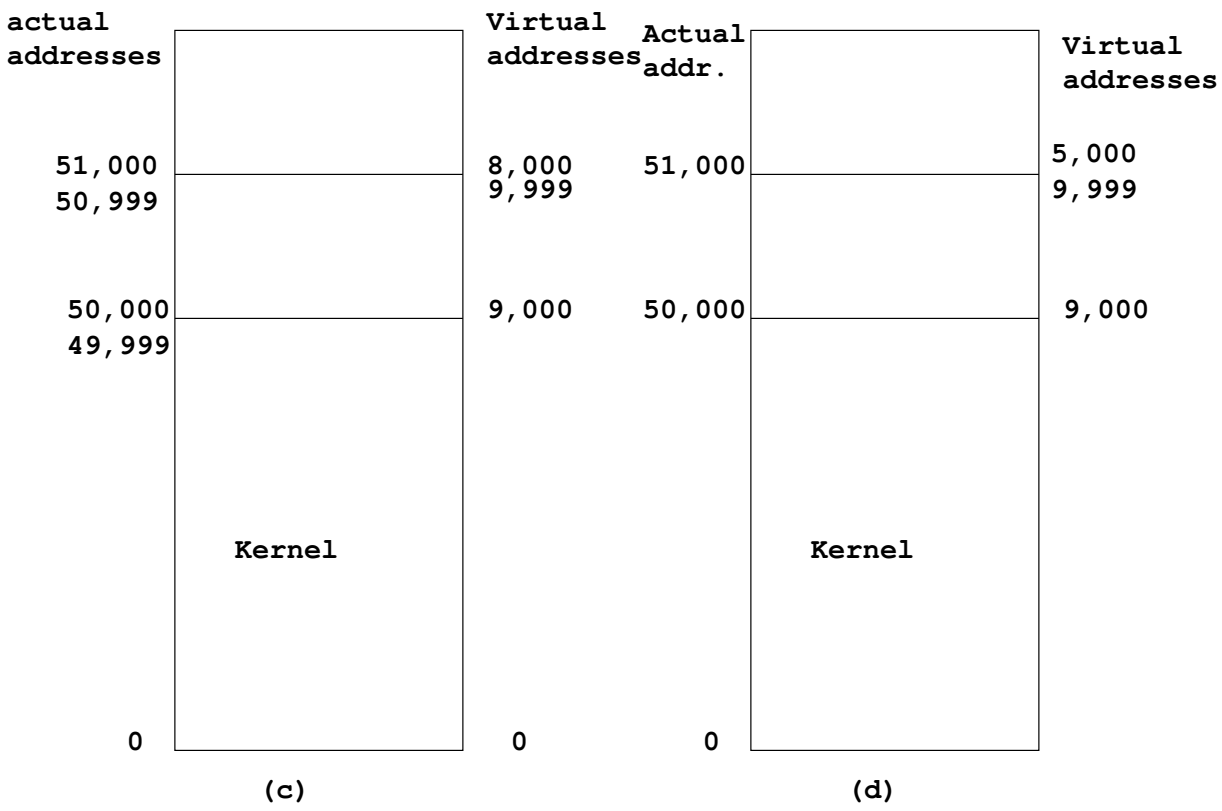


Figure 13.6: (c) Page 9 replaces page 5 (first-in); (d) Page 5 replaces page 8.

How does *first-in first-out* suit this process? We'll have a look in class.

In an examination, in answering a question like 'assuming least recently used page replacement policy, identify the page faults that result in the following sequence of memory accesses ...', I'd not expect the detail given above. Something like the following would do:

```
page fault                    *            *            *            *
          5051, 8997, 5052, 9997, 5053, 10997, 5054, 11997, 5055, 12997
page in     5     8           9           10           11           12
page out                      8            9           10           11
```

13.4 Choice of Page Size

The choice of page size determined at the design stage is a trade- off between various considerations:

- Since the smallest part of the the swap file (or partition) is a *sector* (512 bytes on all systems that I know of), then it would be wasteful of disk access time to use any smaller that 512 bytes; so 512 is a lower bound;
- Given sectors of 512 bytes, pages size should be a multiple of 512 bytes, with 512 minimum; Windows 2000 defaults to 4096 bytes (I do not think you can changes that);
- As we have discussed in connection with file systems, it may take little more time to read eight or 16 contiguous sectors than to read one, so, from the point of view of disk read time, it makes sense to make the page size as large as possible;

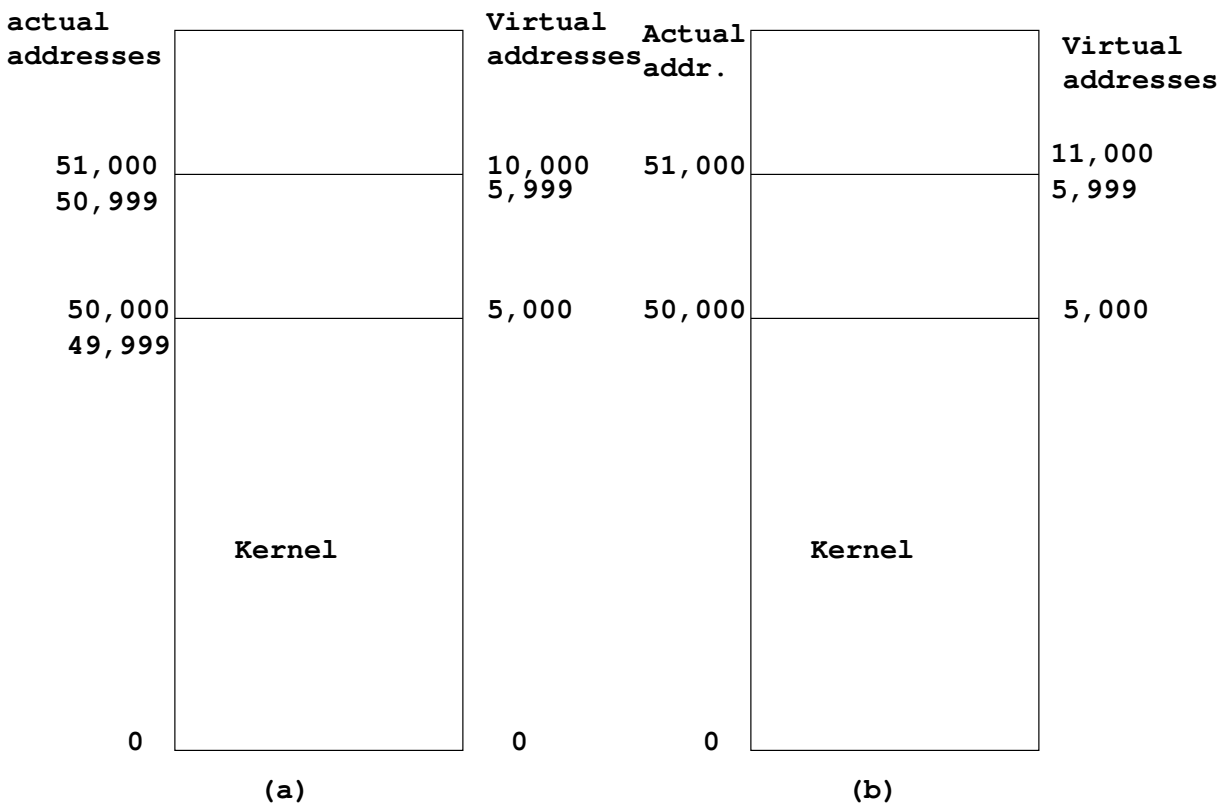


Figure 13.7: (a) Page 10 replaces page 9; (d) Page 11 replaces page 10.

- However, when there is limited physical memory, large pages may be a waste; for example, if a program has just read address 2023 there is a good chance that it will next read 2024, and 2025 and right on up to 3K. But there is diminishing chance that the sequence will ever reach 4K; hence, a 2K page size would be wasting 1K of physical memory per page. I.e. waste is least with small pages;
- Another way of stating the previous point is that page fault rate is likely to be lower with small pages, assuming that a fixed amount of physical memory is available;
- Pages have to be kept track of — using tables; the table overhead space is least with large pages, since page tables have fewer entries;
- Efficiency of transfer between main memory and disk store – best with large pages, since large chunks of data take about the same amount of time to transfer as small amounts.

13.5 Page Replacement Policies

The virtual memory system must keep some pages in physical memory and other pages in the swap file/partition — unless virtual memory requirements are less than the available physical memory.

When a page fault occurs a decision must be made: the process that suffered the page fault cannot continue until the appropriate page is swapped in.

A page replacement policy/algorithm is used to decide which page frame to vacate i.e. swap out its page.

The main goal can be stated as: pages likely to be used soon or used a lot in the future should be kept in; this will minimize the number of page faults.

Thrashing. If page faults and swapping reaches a level at which all processes present are constantly swapping, then little or no progress can be made. If this persists, the kernel is spending all its time transferring pages between swap and physical memory — a condition known as *thrashing*.

13.5.1 First in, First out (FIFO)

Also called *longest resident*. With *first-in-first-out*, when a page is needed, the page that has been in memory for the longest time is chosen. The rationale is that a page that has only recently been swapped in will have a higher probability of being used again soon. However a frequently used page still gets swapped out when it gets to be old enough, even though it will have to be brought in again immediately. This was demonstrated in the example above. Windows 2000 uses first-in-first out.

13.5.2 Least Recently Used (LRU)

Least recently used page replacement policy is based on the assumption that the page reference pattern in the recent past is a mirror of the pattern in the near future. Pages that have been accessed recently are likely to continue to be accessed and ought to be kept in physical memory. It is expensive to implement properly, since in order to find the page least recently used, either a list of pages must be maintained sorted in use order, or each page must be marked with the time it was last accessed. This may be why Windows 2000 uses *first in first out* instead.

13.6 Locality of Reference

Paged virtual memory works best if the same area of memory is accessed all the time – in this case we are accessing the same page, and no swapping need to be done.

Next best is to move slowly through memory: 0, 1, 2, . . . , 998, 999, 1000 (new page), 1001, . . . , 1998, 1999, 2000 (new page); i.e only one page fault per 1000 accesses.

This principle is called *locality of access*, and applies to data memory equally as to program instruction memory.

13.6.1 Impact of Process Management

See *context switching*, last chapter. Clearly switching between processes can have a significant impact on paging performance, just like an unruly program which jumps all over the memory space and disobeys *locality of reference*.

13.7 Cache memory

We'll discuss this in class.

13.8 Self assessment questions

1. Briefly describe how a *virtual memory* system can provide a program with access to instruction and data storage limited only by the addressing range of the processor; i.e. the virtual memory accessible may be much greater than available *physical memory*.

2. A program, running on a computer system which uses paged virtual memory, uses the following sequence of memory addresses:

   ```
   51 1076 52 3974 2342 53 1511 3975 54 2782 3976 3123
   ```

   Assuming a page size of 1000, that the program is allocated *two* page frames, and that the page replacement algorithm is *least-recently-used*, calculate how many page faults will occur during execution of the program.

3. A computer system uses *paged virtual memory*, and a *least recently-used* page replacement policy – i.e. if a page needs to be brought into memory, the *least recently used* is overwritten. The page size is 1000 bytes. There is physical space for only *two* pages (2 x 1000 bytes). Calculate the number of page faults for the sequences of address accesses given in (i) and (ii); hence, or otherwise, give a general guideline on the preferable ordering of memory accesses for systems with *paged virtual memory* or memory *caching*.

   ```
   (i) 30, 1001, 31, 2001, 32, 3001, 33, 4001, 34, 5001, 35, 6001;

   (ii) 30, 1001, 31, 1002, 32, 1003, 33, 1004, 34, 1005, 35, 1006.
   ```

4. Repeat the previous question for first in first out page replacement policy.

5. The following lists elements of the so-called *memory hierarchy*:

   a CPU registers;

   b CPU cache memory;

   c main memory;

   d magnetic disk;

   e offline archive.

6. Discuss this hierarchy by comparing the elements under the headings: (i) capacity (size), (ii) speed of access (read, write), (iii) cost per bit. Suggestion: A table containing comparative figures, together with a brief commentary, would suffice.

7. In the context of multitasking system memory management, explain the meaning of the term *relocation*.

8. In a multitasking memory management, in the context of *relocation* explain the purpose of the *base register*; illustrate your answer with an example.

9. In the context of multitasking memory management, explain how the combination the *base register* and *limit register* may be used to prevent a process from accessing the memory of other processes or the kernel; illustrate your answer with an example.

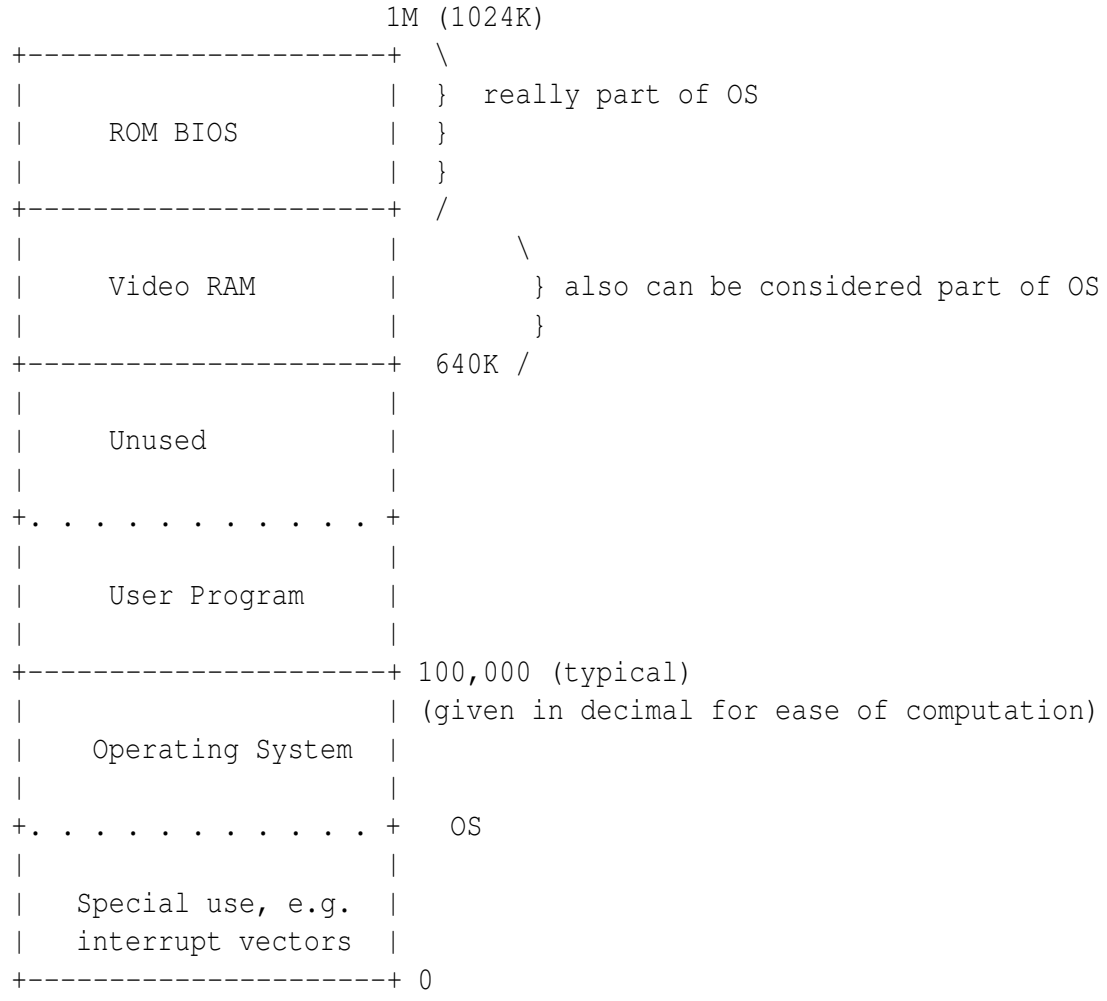


Figure 13.8: Memory map

10. Figure 13.8 shows a memory map of a single tasking operating system such as MS-DOS. There is a program 'prog.exe+ on your hard disk. Explain what happens when someone executes 'prog.exe' by typing: 'prog'. Assume `prog.exe` is of size 40,256 bytes. Suggestion: decoding command ... reading program into memory ... memory map when in memory ... starting program ... when program is finished ...

11. Referring to the Figure 13.8, if a program is larger than then the 540K memory available, explain how *overlaying* could be used to allow the program to run in available memory.

12. In a *paged virtual memory* discuss the choice of page size.

13. In a multitasking operating system, how may process management affect the performance of paged virtual memory.

14. In the context of operating systems, explain, briefly *three* of: (i) kernel; (ii) process; (iii) process control block; (iv) virtual memory, (v) command interpreter (shell); (vi) file systems.

Chapter 14

Miscellaneous Items

This chapter includes some miscellaneous topics: how sound is captured, stored, handled by a computer. During the course, I will also be providing some handouts on PC components — displays, particular buses (ISA, PCI, USB), modems, I/O interfacing.

14.1 Audio Data in Computers

Figure 14.1 shows audio data being captured by a computer system, stored, then played back via a loudspeaker.

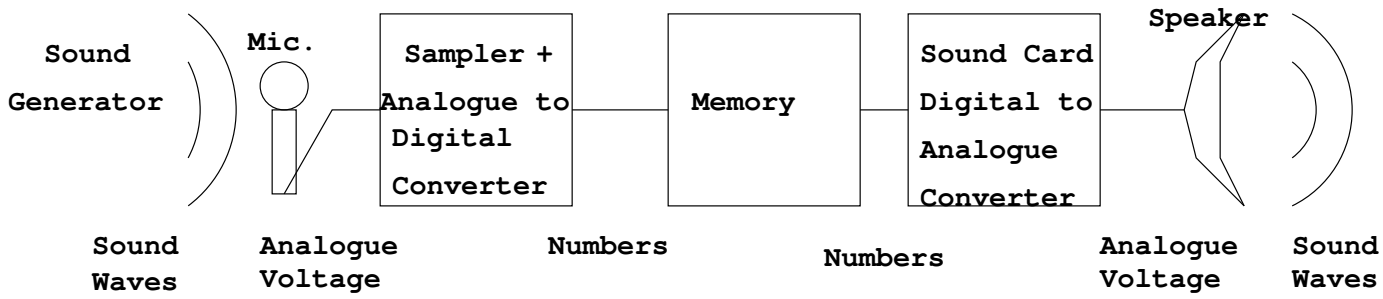


Figure 14.1: Audio data on a computer.

Sound Sound is produced by air vibrating; by that we mean rapid changes in air pressure. If you go to a really loud disco, you will be able to feel the loudspeakers pushing the air at you.

When the string of a guitar is plucked, or the string of a violin bowed, the string vibrates. That, in turn, causes the air to vibrate. Those vibrations, just like the waves when you drop a stone in a lake, travel outwards. Eventually, if you are nearby, the vibrations reach your eardrum and cause the sensation of sound.

If the vibrations are very slow, slower than about 20 vibrations per second, the eardrum will not receive any sensation – but, like in the disco, you may feel you body being shaken by them.

Likewise, if the vibrations are faster than about 16,000 times per second, you will not hear them – but dogs and other animals will.

Human voices produce sound by vibrating the vocal chords. Wind instruments produce sound by air vibrating in the instrument tube. The rate of vibration can be governed by the length of the tube and this is done by lengthening and shortening the tube – as in a trombone, or by pressing your fingers over holes in the tube – as in a flute or whistle.

Loudspeakers Loudspeakers (or headphones) are electrically controlled sources of vibration or pressure. If you pass a voltage wave (vibrating voltage) to a loudspeaker, the core of the loudspeaker vibrates in sympathy with the voltage. When the loudspeaker core pushes out, you get positive pressure – the air is pushed at you; when the core maves back, you get negative pressure – the air is sucked away from you.

Microphones Microphones are like loudspeakers in reverse; in fact, you can use loudspeakers or headphones as microphones. When the air near a microphone vibrates (i.e. there is sound), the microphone produces a voltage at its electrical outputs.

Therefore, we can convert from sound to electricity (voltage) using a microphone. And we can convert from electricity to sound using a microphone.

Analogue versus Digital Radios, TVs and cassette tape recorders handle sound as electricity (voltage and current); this is called analogue – the voltage is analogous to the sound. If you drew a graph of the intensity of the air pressure that causes the sound, and another graph of the intensity of the electrical voltage, the two graphs would be the same shape.

Computers use numbers (digits – digital) instead of voltage. Thus, when we need to get sound into a computer, we have to convert the pressure (sound, vibrations) into numbers.

When you want to get sound into a computer, first you convert the sound to electrical voltage using a microphone. Next, read the voltage at rapid intervals (about 44,000 times per second for Compact Disc (CD)) and convert the reading into a number. This conversion into numbers is done by an electronic circuit called an Analogue-to-Digital Converter (ADC). Now, the sound can be stored on a computer, and on its disk, and in its memory, and copied onto CD.

Eventually, when you want to hear the digital sounds again, we must convert from numbers back to voltages, and then, using a loudspeaker, back to air pressure.

The conversion from numbers to voltage is done by an electronic circuit called a Digital-to-Analogue Converter (DAC). Most of a Sound Blaster card is taken up with its Digital-to-Analogue Converter. A Sound Blaster also has Analogue-to-Digital Converter, to which you connect a microphone or other input source.

Electronic (or Electroacoustic) Music When we produce music for a CD we do the following: (a) cause air vibrations by a variety of means (voices, strings, wind instruments); (b) convert those vibrations to electrical voltages; (c) convert these voltages to numbers; (d) copy the numbers onto the CD. And then when we want to hear the sound, we: (e) convert the numbers back to voltages; (f) convert the voltages back to sound.

But computers can produce numbers on their own. And they can be programmed to produce strings of numbers that correspond to certain sounds; this includes sounds that are hard to produce using ordinary instruments. So, we can use the computer to jump straight to (c) and cut out the conventional sound production altogether.

Electronic Instruments What if you want to produce an electronic piano? Let's say it has 40 keys, numbered k1 to k40.

Now you take a real piano with 40 keys; you take each key in turn, press it, and record the sound produced; these days, the recording is more likely to be digital (numbers), but the same principle applieds to analogue (voltage) recording. You store the sound for all 40 keys.

Now, you connect a computer to an electronic piano keyboard; when you press the key k20 say, the keyboard indicates to the computer that k20 has been pressed. The computer looks for the sound that we recorded for k20 on the real piano and sends that to the equivalent of a Sound Blaster card and to an Digital-to-Analogue and then to loudspeakers or headphones.

Magnetic Recording See Operating Systems, chapter on disks and file systems.

Data Compression If you sample sounds at CD rates (44,100 Hz in stereo and use 16 bits per sample), you get 88,200 numbers per second, or 176,400 bytes. That's about 10.6 Megabytes per minute, or about 635 Megabytes per hour (this figures – a CD holds about 650 Megabytes of data).

That's an awful lot of data. How could it be reduced, without detracting from the quality? (If you decrease the sampling frequency, the sound will become soft and dull – like listening to Medium Waveband radio or at the end of a telephone; if you decrease to one byte or less, you will get other annoyances.)

The answer is **data compression**.

There are a number of principles by which we can compress data – i.e. store or transmit as near as possible the same information, only using less bits. There are two categories of data compression:

- Lossy. Used for audio (e.g. MP3) and images, where the data are eventually going to be converted into an analogue form to be listened to or viewed by a person. Here, loss of the odd bit may not make any difference.
- Lossless. Used where the data are, and always will remain, digital; where each bit is crucial. E.g. the ASCII string 74 68 65 20 63 61 74 20 73 63 61 20 6F 6E 20 74 68 65 20 6D 61 74 2E reads: `"the cat sat on the mat".`

  If I decide to remove all the least-significant bits (reasonable for lossy, but not here), I get: 74 68 64 20 62 60 74 20 72 62 60 20 6E 6E 20 74 68 64 20 6C 60 74 2E, which reads: `"thd b't rb' nn thd l't."` Not much use!

Some of the principles by which we can compress data are:

- Redundancy. For example:

  `TH_R_ _S _NLY _N_ W_Y T_ F_LL _N TH_ V_W_LS _N TH_S S_NT_NC_.`

  You use redundancy all the time when texting on mobile phones. Can be lossless.
- Use less bits for the more common parts of the message. For example, 'e' is very common in English text, 'z' and 'q' vey uncommon. If you were to use a two bit code for 'e' and similarly small code sizes for 'a', 't', etc. and 16 bits for 'z' and 'q', you would achieve considerable savings. This is the basis of Huffman coding. Huffman coding can achieve compression ratios of around 50% for English text. Lossless.

- Perceptual coding. Work out what parts of sounds (and pictures) are most noticed by humans – gives these larger codes; give small codes to parts where the humans will never notice. Lossy.
- Differential coding. Let's say the following sequence of numbers appear in a digital recording of a voice: 32,001 32,015, 32,020 32,010 32,005 .... As it is, we require 16 bits per number. However, let's send the first, and send only differences for the remainder: 32,001 14 5 -10 -5. For the differences, we can get away with five bits. Can be lossless.

Recall that we started off at 176,400 bytes per second, that is 1.4 Mbits per second. MP3 compression can compress music to 128Kbits per second (around 10%) and sound very tolerable.

14.2 Compilation, Interpretation and all that

14.2.1 Introduction

This section explains the steps required to create an executable (runnable) program using a high level language. The examples used are from C++; however, what is said here remains true for a great many programming languages – with minor differences. I use some rather exaggerated programming in places – so that I can show certain features of the process.

Integrated Development Environments (IDEs) hide all the details, but, under the bonnet, they contain all the components mentioned here.

The lifecycle of a program goes through the following stages:

Creation of source code This is done using a text editor.

Compilation Source code to **object** code.

Linking Building an executable program from object code *building blocks*; these building blocks may be bits of program provided by the programmer or from a library provided by a third party.

Loading for Execution We have already mentioned this in chapter 7.

In addition, I briefly mention assembly languages and interpreted languages.

For additional information, see the excellent dictionary of computing at `http://foldoc.ic.ac.uk/`.

14.2.2 Creation of Program Source Code

The first thing to be done is typing in the source code using a plain text editor; *source* means the original code written by the programmer.

In this example, I have used subprograms (in C++ called functions) to demonstrate my points. It is good practice to separate a program into subprograms, but some of the use or subprograms here may be a little artificial.

In addition, I have prepared my source code in separate files. Again, for this little example, separate files may seem a artificial; on the other hand, when programs get beyond a certain size and complexity, it is essential to use this sort of modularity of construction.

The main program – mathmod.cpp

```
//------- mathmod.cpp ------------------------
// j.g.c. 2/2/97 -- from math2.cpp
// demo of separate modules
//-------------------------------------------
#include <iostream>
#include "funs.h"

int main()
{
  float x,y,z;
  int i,j,k;

  cout<< "enter first int:";
  i=getInt();
  cout<< "enter second int:";
  j=getInt();
  k=i+j;
  cout<< i<<", "<< j<< ", "<< k<< endl;

  cout<< "enter first float:";   x=getFloat();
  cout<< "enter second float:";  y=getFloat();
  z=addf(x,y);
  cout<< x<<", "<< y<< ", "<< z<< endl;

  return 0;
}
```

The subprograms – funs.cpp

```
//------- funs.cpp ----------------------------
// j.g.c. 2/2/97 -- functions for mathmod.cpp
// demo of separate modules
//-------------------------------------------
#include "funs.h"

int getInt(void)
{
  int i;
  cin>> i;
  return i;
}

float getFloat()
{
  float f;
  cin>> f;
  return f;
```

```
}

float addf(float a, float b)
{
  float c=a+b;
  return c;
}
```

14.2.3 Compiling

Source programs like `mathmod.cpp` `funs.cpp` etc. are *not* directly executable.

The next stage of creating an executable is to *compile* `mathmod.cpp` into *object* code. In Linux/UNIX this is done as follows, but for `g++`, you can substitute the command name of whatever compiler is on your machine.

```
g++ -c mathmod.cpp
```

This compiles and puts the object code into a file `mathmod.o`. This object code is essentially *machine* code but it is only a *building block*; we need more building blocks, namely the subprograms, and code contained in *libraries*.

```
g++ -c funs.cpp
```

compiles `funs.cpp` and produces `funs.o`.

14.2.4 Assembling

It is possible to write programs in assembly language. In this case, you don't compile, but **assemble**. However, the output is the same as that for a compiler, i.e. object code.

Note: some compilers compile to assembly code, and then use an assembler to get to object code. But whether they go from source to object, or go source - assembly - object makes no real difference to the principle.

The main point about assembly language is that it is merely a symbolic version of (numeric) machine language. Hence, assembly (translation) is very easy – as you have seen.

14.2.5 Linking

As I have said, `mathmod.o` and `funs.o` are merely building blocks.

The next stage is to *link* the object code in `mathmod.o` and `funs.o` with appropriate library code – put the building blocks together:

```
g++ -o mathprog mathmod.o funs.o
```

This produces the executable file `mathprog`. You don't need to tell it to get code from the libraries – it knows that.

Even though we use `g++`, `g++` actually invokes a command called `ld` to do the linking-loading – *loading* refers to loading of library code. Here, the extraction of the code for `cin.get` etc. from the *library* is kept implicit; however, a library is nothing more than an object file, with appropriate indexes to each function.

14.2.6 Execution

Finally, to execute `mathprog`, you type;

```
mathprog
```

This reads the contents of `mathprog` into memory, and starts execution at an appropriate start address.

14.2.7 Compiling & Linking – Summary

First compile to object code.

```
g++ -c mathmod.cpp       # produces mathmod.o
g++ -c funs.cpp          # produces funs.o
```

Now link.

```
g++ -o mathprog mathmod.o funs.o
```

And, you can do all this in one line:

```
g++ -o mathprog mathmod.cpp funs.cpp
```

Figure 14.2 summarises the process.

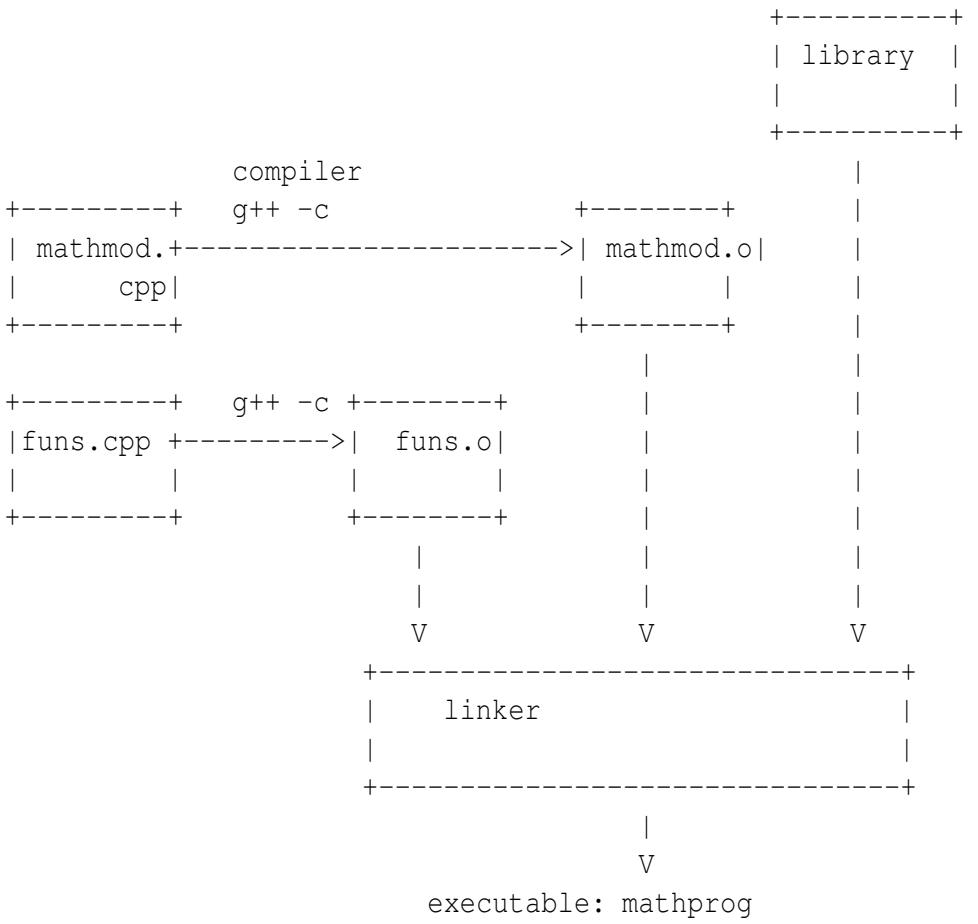


Figure 14.2: Compiling and Linking to Produce an Executable.

14.2.8 Static versus Shared Libraries

In linking `funs.o` the code for `addf`, `getInt`, and `getFloat` is linked *statically*, i.e. the object code for each of the functions is copied into the file `mathprog`. For common functions like `cin.get` this can become wasteful – consider the case of (a) `funs.cpp` is a very large set of subprograms handling

communication with the web; (b) a great many programs with this code duplicated in them; (c) on a multitasking system, and great many processes running, each with a duplicate copy of `funs.o` loaded into memory. Wasteful.

Hence *shared* libraries, in which the linker inserts just a *pointer* to shared code that is already in the operating system.

In Windows, we have **Dynamic Link Libraries (DLLs)**. Let's assume `funs.cpp` contains a large set of subprograms to do with games programming. Since only a small percentage of users play games that need the these subprograms, you don't want to burden the operating system with the code. On the other hand, you don't want to have every games program carrying around the extra 5 Magabytes (e.g.) of `funs.o`. Solution: create a separate file called `funs.dll` which is a cross between object and executable. Now, if any program needs `funs.dll` it can load it (dynamically).

14.2.9 Interpreted Languages

The chief difference between a **compiler** and an **interpreter** is as follows. A compiler translates from one language **source** to some other **object** language. On the other hand, an interpreter takes a program and executes it.

When you create a Visual Basic program, you get a choice: you can compile and create an `.exe` file much as described above.

If you run the VB program in interpreted mode, then something quite different happens. There is an *interpreter* program (let's call it `vbinterp`) which reads the *source* and executes it directly. `vbinterp` has its own **fetch-decode-execute** cycle running (recall chapter 5 of Computer Systems).

We have something like the following, grossly simplified, assuming that we are dealing only with instructions of the form `result = num1 operation num2 .`

```
int operand1, operand2, result;
String line;

f= open(prog1.vba);

while(NOT end-of-file(f))

 line = readLine(f); /*fetch next line*/

 decode 'line' to produce:
     - operation
     - operand1, operand 2

 /* execute */

 if(operation=='+')result= operand1 + operand2;
 else if(operation=='-')result= operand1 - operand2;
 else if(operation=='/')result= operand1 / operand2;
 else if(operation=='*')result= operand1 * operand2;
 else if(operation=='F')result= fred(operand1, operand2);
 else /* etc */
```

```
  assign 'res' to actual result variable.

  and go back for more ...
 }
```

14.2.10 Java — Compiler AND Interpreter

Java is interpreted, but it's interpretation is a little different from that described for Basic. Java programs go through a compiler **and** an interpreter. When you compile a Java program (e.g. `prog1.java`), you produce a file `prog1.class` which contains Java **byte-code**. Java **byte-code** is sort-of like machine code, which is interpreted by a Java interpreter program called Java Virtual Machine(JVM).

So, we have a Java compiler, which produces the byte-code; then we have the Java interpreter program, the JVM, which applies its version of the *fetch-decode-execute cycle*; see chapter 6 for a discussion of a *hardware* CPU version of the *fetch-decode-execute cycle*. Of course, the Java Virtual Machine is software.

Since the JVM is software, any computer can execute Java byte-code as long as they have the JVM interpreter program.

From the point of view of Operating Systems, you could think of the JVM as another *layer* on top of the operating system, and below the applications. Indeed, the JVM provides *operating-system-like* facilities, like support for concurrency (multitasking).

Another *operating-system-like* facility provided by the JVM is its treatment of *applets*. Applets are Java byte-code (executable) programs meant primarly for downloading over the Internet by web-browsers and executed directly by the browser running the JVM. Now, this would normally be a recipe for disaster, and you would fear all sorts of viruses and malicious programs. However, rather like an operating system with its *privileged* and *user* modes, the JVM has a special mode for executing applets. For example, applets may not access disk files.

14.2.11 Java Enterprise Edition (J2EE) and .NET

A JVMs runs on a single machine. In these days of the Internet, people and enterprises often want to do something called *distributed computing*, i.e. a set of cooperating processes, (see Operating Systems course), executing on separate machines connected via a network (such as the Internet). This is all the easier if the processes are all running on JVMs. Add a bit of Internet glue and you have J2EE.

Where are Microsoft in all this? When Java (developed and *owned* by Sun Microsystems) became successful, Microsoft did not like it. They attempted to create an extended Java of their own; but Sun forbad them. Then came the success of Java distributed computing. So Microsoft designed their own version of J2EE — called .NET. C# (pronounced "C-sharp") is Microsoft's attempt at Java. Like Java, C# is compiled to a sort of byte-code, and executed on a virtual machine. The is a .NET version of VisualBasic.

14.2.12 Ultimately, All Programs are Interpreted

If you think about it, in computing, interpreters (in general) crop up everywhere. In addition to the examples we have mentioned:

- The operating system shell (e.g. Windows CMD) fetches and decodes your commands, and then gets the kernel to do the executing;
- The hardware processor itself is an interpreter; in fact, if you look at the *microprogram* that runs Mac-1, you will clearly see that the program performs a *fetch-decode-execute* cycle. And even if the processor is controlled by hardware rather than a microprogram, then the hardware will, in its own way, be performing interpretation.

14.3 Graphic User Interfaces

14.3.1 Introduction

Already, we have experienced two sorts of *user interface* provided by operating systems:

Command line As provided by MS-DOS.

Graphic User Interface (GUI) As provided by Windows.

The purpose of this section is to explain some of the differences in the implementation of the two.

14.3.2 Command Line Interface

Command line interaction, as in MS-DOS proceeds as follows:

```
begin
     Prompt user;
     Read command;       /* fetch (shell)*/
     Decode command;     /* decode (shell) */
     Do processing;      /* execute (kernel) */
     /* this may involve further prompt, input, process */
     Display result;
end;
```

And if you look at a typical program (to be executed), you will see code of the form. In other words, in the program, as in the operating system, everything follows a predictable sequence.

```
begin
     Prompt user;
     Read input ;
     Process input;
     Display result;
end;
```

14.3.3 Graphic User Interface

A little thought will indicate that the *fetch decode execute* scheme described above will not work for a GUI:

- What do you prompt? the user can do any of a large range of actions; the user is in charge, not the program;
- What input device do you read? Mouse, keyboard, other pointing device?
- What do you expect to read? A number? A character string?
- etc . . .

The basic structure of a Windows and Macintosh application program is as follows:

```
begin
  Initialise Desktop;
  While (not Done)
   begin
    GetNextEvent(eventMask, theEvent);
    HandleEvent(theEvent);
   end
  CleanUp;
 end
```

`HandleEvent` looks something like the following:

```
if(theEvent==0) do something;
else if(theEvent==1) do something-else;
else if(theEvent==2) do another-thing;
etc., ...
```

The important point is this: HandleEvent (the application part) is driven by the event, not the other way round.

This can allow an interface to be *modeless*: the program identifies the mode from the input. For example, in Microsoft Word, you may type a character, click on the mouse to move the cursor, click on a button, . . . – endless possibilities.

What is happening behind the scenes is important. The operating system – via interrupts – captures *events* and maintains a queue of them. These events are responded to by `GetNextEvent(eventMask, theEvent); HandleEvent(theEvent);`.

In fact, when you write a Windows program, you will be forced to start with a *framework* like that above. The main programming will be in providing subprograms to handle events 0, 1, 2, ...

In contrast with a command line regime, the sequence of actions is musch less predictable.

14.4 Interprocess Communication

This chapter may, as necessary, be expanded during lectures.

- Need for processes to send signals to one another;
- Need for process synchronisation;
- Pipes, e.g. in Windows commmand line, you can sort a file into another and then print:

  ```
  sort file1.txt > sorted.txt
  print sorted.txt
  ```

  Using a *pipe* (`|`), this can be replaced by:

  ```
  sort file1.txt | print
  ```

- Named pipes or FIFOs. If the processes are not connected as `sort, print` above, a named pipe can be used.
- UNIX domain *sockets*; the nice thing about *socket*s is that a simple extension of the concept gives us a method for processes to communciate across a network; i.e. the processes need not be on the same machine;
- Message queues;
- Semaphores;
- Shared memory.

Bibliography

BBC (1992). *The Dream Machine*, BBC Publications.

Brookshear, J. (1999). *Computer Science, an overview*, 6th edn, Addison Wesley.

Dick, D. (2002). *The PC Support Handbook: Configuration and Systems Guide*, Drumbreck Publishing.

Ferry, G. (2003). *A Computer Called LEO: Lyons Tea Shops and the World's First Office Computer*, Fourth Estate.

Flynn, I. & McHoes, A. (2001). *Understanding Operating Systems*, 3rd edn, Brooks/Cole.

Harris, J. (2002). *Operating Systems (Schaum's Outlines)*, McGraw-Hill.

Hennessy, J. & Patterson, D. (2002). *Computer Architecture - A Quantitative Approach 2nd ed.*, 3rd edn, Morgan Kaufmann.

Hillis, W. (1999). *The Pattern on the Stone*, Weidenfeld and Nicolson.

Horowitz, P. & Hill, D. (1989). *The Art of Electronics*, 2nd edn, Cambridge University Press.

Knuth, D. (1997). *The Art of Computer Programming, Volume 2*, 3rd edn, Addison Wesley.

Meyers, M. (2001). *All in One A+ Certification*, 3rd edn, Osborne/McGraw-Hill.

Mueller, S. (2001). *Upgrading and Repairing PCs*, 13th edn, Que.

Patterson, D. & Hennessy, J. L. (1994). *Computer Organisation and Design - The Hardware/Software Interface*, Morgan Kaufmann.

Petzold, C. (2000). *Code: the hidden language of computer hardware and software*, Microsoft.

Ritchie, C. (1997). *Operating Systems: incorporating UNIX and Windows*, 3rd edn, Letts.

Silberschatz, A., Galvin, P. & Gagne, G. (2002). *Operating System Concepts*, 6th edn, Addison Wesley.

Stallings, W. (2000a). *Computer Organisation and Architecture*, 5th edn, Prentice Hall.

Stallings, W. (2000b). *Operating Systems*, Prentice Hall.

Stewart, J. & Scales, L. (2000). *Windows 2000 Foundations*, Coriolos.

Sybex (2000). *Windows 2000 Complete*, Sybex.

Tanenbaum, A. (1990). *Structured Computer Organisation*, 3rd edn, Prentice Hall.

Tanenbaum, A. (1999). *Structured Computer Organisation*, 4th edn, Prentice Hall.

Tanenbaum, A. (2001). *Modern Operating Systems*, 2nd edn, Prentice Hall.

Thompson, R. & Thompson, B. (2000). *PC Hardware in a Nutshell*, O'Reilly.

Tulloch, M. (2001). *Windows 2000 Adminintration in a Nutshell*, O'Reilly.

Wallace, R. (2000). *MCSE Training Kit Microsoft Windows 2000 Professional*, Microsoft.

White, R. (1984). *Introduction to Magnetic Recording*, IEEE Press.