Appendix 1 Introduction to logic circuit minimisation using Karnaugh map methods

A1.1 Introduction

When analysing a logic problem the result may be a long and complex logic expression. Before trying to implement the expression it would be desirable to try to simplify the expression. The process of simplification is called *minimisation*. It is possible to use algorithmic rules to carry out the minimisation process but it can be difficult to identify the best approach to achieve the optimal solution. Karnaugh maps are a way of achieving an optimal result using a graphical approach.

A1.2 Two variable expressions

Minimising two variable problems appears a relatively straightforward process. Generating the truth table for the expression will result in the characteristic output pattern for a basic gate. Consider the two expressions

$F_1 =$ (Y AND $\overline{Z}$) OR ($\overline{Y}$ AND Z)

$F_2 =$ (Y AND Z) OR ($\overline{Y}$ AND Z)

The truth tables for these two expressions are given in Table A1.1. The truth table for F_1 has generated the output of an XOR function (see Figure 3.2). F_2 however does not produce output that can be matched with any of the basic gates in Figure 3.2 and thus would not appear to be reducible.

We will now try mapping the output of the truth table on a Karnaugh map. The map is a 2×2 grid. The first column represents $\overline{Y}$; the second column represents Y. The first row represents $\overline{Z}$; the second row represents Z. Figure A1.1 demonstrates this layout. Data from the truth table for a function can be transcribed onto the Karnaugh map. The result for $Y = Z = 0$ is placed in the top left box. The result for $Y = 1$,

Table A1.1 Truth tables for (a) F_1 = (Y AND $\overline{Z}$) OR ($\overline{Y}$ AND Z) and (b) F_2 = (Y AND Z) OR ($\overline{Y}$ AND Z)

Y	Z	F_1
0	0	0
0	1	1
1	0	1
1	1	0

(a)

Y	Z	F_2
0	0	0
0	1	1
1	0	0
1	1	1

(b)

Figure A1.1 Two variable Karnaugh map

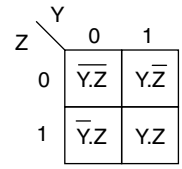
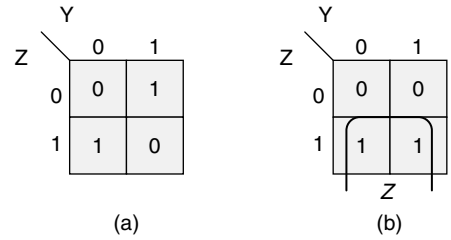


Figure A1.2 Karnaugh maps for (a) (Y AND $\overline{Z}$) OR ($\overline{Y}$ AND Z) and (b) (Y AND Z) OR ($\overline{Y}$ AND Z)



Z = 0 is placed in the top right box and so on. The resulting Karnaugh map for the two functions F_1 and F_2 above are shown in Figure A1.2.

The diagonal pattern in Figure A1.2(a) is typical of an XOR function. The pattern in Figure A1.2(b) shows two adjacent 1 values. It is this sort of pattern, which we are interested in. What this pattern demonstrated is that if Z = 1, then it does not matter what the value of Y is, the output will be 1. We can now rewrite F_1 and F_2 as

$$F_1 = Y \text{ XOR } Z$$
$$F_2 = Z$$

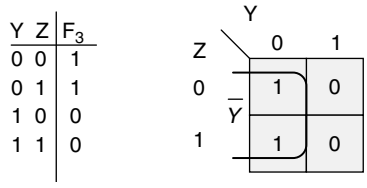
Let us now consider a third function

$$F_3 = (\overline{Y} \text{ AND } \overline{Z}) \text{ OR } (\overline{Y} \text{ AND } Z)$$

The truth table and Karnaugh map for this function are shown in Figure A1.3.
From the Karnaugh map it can be seen that the output of this function will be 1 if Y = 0, irrespective of the value of Z. The output is therefore the inverse of Y, or $\overline{Y}$. F_3 can thus be rewritten as

$$F_3 = \overline{Y}.$$

Figure A1.3 Truth table and Karnaugh map for F_3 = ($\overline{Y}$ AND $\overline{Z}$) OR ($\overline{Y}$ AND Z)



A1.3 Three variable expressions

This technique can be extended to cover three variable expressions. The layout of the Karnaugh map for these expressions is shown in Figure A1.4. It can be seen from the Karnaugh map in Figure A1.4 that the top row represents Z and the bottom row is $\overline{Z}$. The left half of the map represents $\overline{X}$ and the right half is X. The middle two columns represent Y and the outside two columns represent $\overline{Y}$. The Karnaugh maps in Figure A1.5 illustrate the pattern we would expect for F_4 = X and F_5 = $\overline{Y}$ AND Z. In Figure A1.5(a) the four cells containing 1 all fall within the area where X = 1. Two of the cells are in the region where Z = 1 and two in the region where Z = 0. Similarly two are in the region where Y = 1 and two where Y = 0. All four cells therefore only have X = 1 in common. This means that the expression produces a TRUE result whenever X is TRUE. Note that in Figure A1.5(b) the cells of the right-most column are thought to wrap round and be adjacent to the left-most column. It is thus possible to group the bottom-left cell with the bottom-right cell. The two cells in which there is a 1 have Y = 0 and Z = 1 in common and thus the expression will return TRUE if Y is FALSE and Z is TRUE. As an AND gate will return TRUE (1) only when all its inputs are TRUE then the expression will return TRUE if $\overline{Y}$ is TRUE and Z is TRUE. Figure A1.5(c) illustrated the pattern for F_6 = X OR ($\overline{Y}$ AND Z). The expression X OR ($\overline{Y}$ AND Z) is formed by taking the expression for each of the groupings (which will be made up of a number of terms connected by ANDs) and connecting them via ORs. This type of expression is known as *sum-of-products* form. The reason for this name is obvious if you remember that AND can be replaced by '.' and OR by '+'.

The objective when grouping the 1's together is to create the biggest group of adjacent 1's as possible. In Figure A1.5(c) it can be seen that the bottom-left cell is counted twice. This does not matter; the important things are that every cell containing a 1 is grouped at least once and that it is included in the biggest group possible. It would be possible to group the right half of the map together giving X. The 1 in the bottom-right cell would then be $\overline{X}$ AND $\overline{Y}$ AND Z. The final expression would then be X OR ($\overline{X}$ AND $\overline{Y}$ AND Z) and thus we would have an extra superfluous term.

Figure A1.4 Three variable Karnaugh map

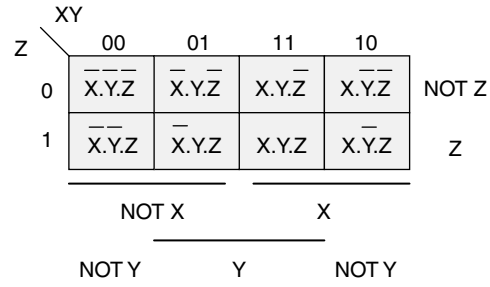


Figure A1.5 Karnaugh maps for (a) $F_4 = X$, (b) $F_5 = \overline{Y}$ AND Z and (c) $F_6 = X$ OR ($\overline{Y}$ AND Z)

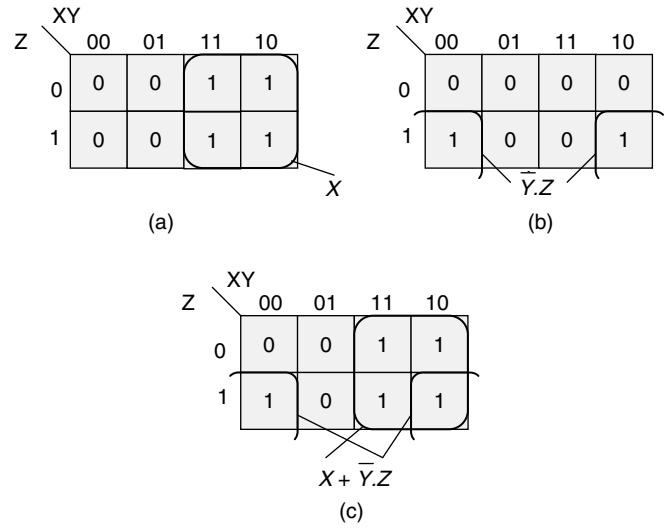
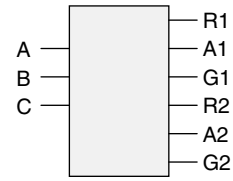


Figure A1.6 Traffic light controller



A1.3.1 Three variable example

A traffic signal normally consists of a group of three lamps, a top red lamp, a middle amber lamp and a bottom green lamp and we call this group a *traffic light*. A basic set of traffic lights consist of two pairs of traffic lights at right angles. Let us consider the pair controlling travel in a north–south direction as set 1 and the pair controlling traffic in an east–west direction as set 2. Initially the two red lamps (R1 and R2) will be on. R2 will then remain on while set 1 lamps go through the following cycle. The amber lamp (A1) is turned on. This will be followed by R1 and A1 being turned off and then the green lamp (G1) will be turned on. After a period of time, A1 will be turned on and G1 turned off, then R1 will be turned on and A1 turned off. Set 2 lamps will now go through the same sequence with R1 remaining on. The lamps actually pass through eight states before repeating the pattern. A circuit for controlling the lamps will have three inputs to indicate the current state and an output for each of the six lamps (see Figure A1.6).

The eight states through which the traffic lights pass can be shown on a truth table. A one output for any light indicates that the light is on and a zero output indicates that the light is off. The truth table is shown in Table A1.2.

From this truth table an expression can be determined for each of the lights by mapping the output of each. The Karnaugh maps in Figure A1.7 show the Karnaugh maps for each of the lights in set 1 along with the minimal form resulting from an analysis of the maps. Once we have the minimised form of the expression, the logic circuit, as in Figure A1.8, can be drawn.

Table A1.2 Truth table for traffic light controller

A	*B*	*C*	*R1*	*A1*	*G1*	*R2*	*A2*	*G2*
0	0	0	1	0	0	1	0	0
0	0	1	1	1	0	1	0	0
0	1	0	0	0	1	1	0	0
0	1	1	0	1	0	1	0	0
1	0	0	1	0	0	1	0	0
1	0	1	1	0	0	1	1	0
1	1	0	1	0	0	0	0	1
1	1	1	1	0	0	0	1	0

Figure A1.7 Karnaugh maps for traffic light set 1

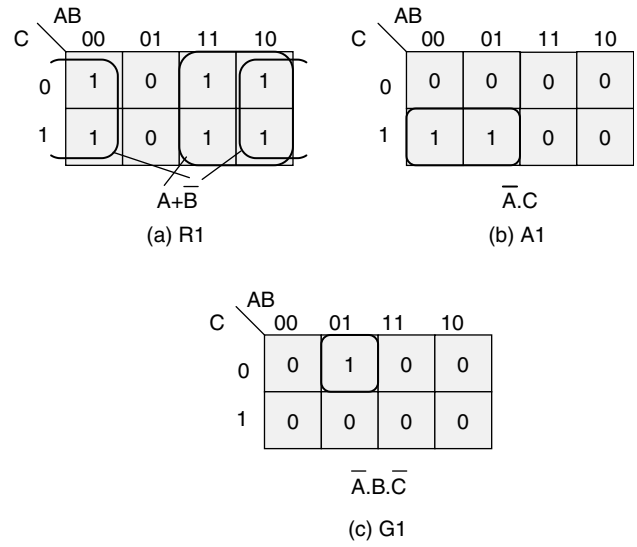


Figure A1.8 Logic circuit for traffic light set 1

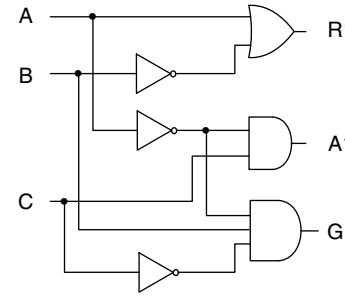


Figure A1.9 Four variable Karnaugh map

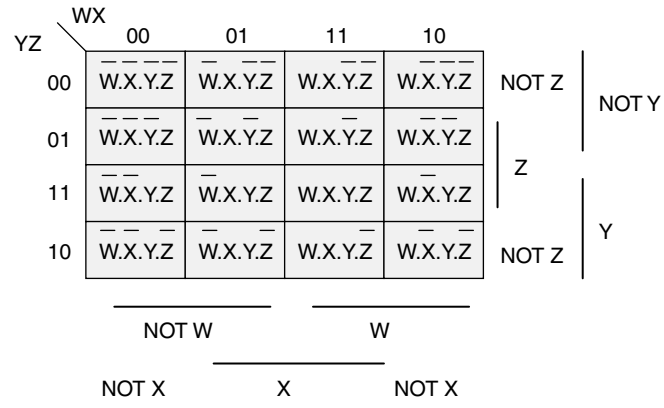
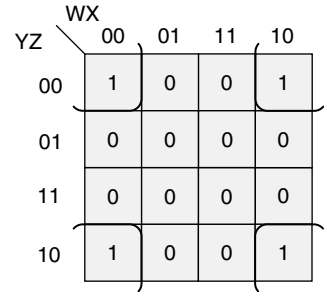


Figure A1.10 Karnaugh map for $\overline{X}$ AND $\overline{Z}$



A1.4 Four variable expressions

The layout illustrated in Figure A1.9 demonstrates the Karnaugh map for logic expressions consisting of four variables. The principle of transferring the result from a truth table to the Karnaugh map still applies. The objective is to group all the cells containing 1 together in the biggest clusters available still applies. When we were considering three variable Karnaugh maps, it was observed that the left-most column was considered to wrap around and to be adjacent to the right-most column. In the case of a four variable Karnaugh map the top row is also considered to wrap around and to be adjacent to the bottom row. This means that the four corners are adjacent. From Figure A1.10 we can see that the four corners represent $\overline{X}$ AND $\overline{Z}$.

A1.4.1 Four variable example including don't care conditions

Figure A1.11 illustrates a 7-segment display. These displays are used to display the ten digits. Each of the segments is referred to by a letter as shown in Figure A1.11.
A 7-segment display controller will need seven outputs (1 for each segment) and four inputs. Four inputs are sufficient to select between 16 states but there are only ten in the 7-segment display. If there were only three inputs however, only eight states could be used. The truth table for the segment *a* output from the controller is shown in Table A1.3. A one in the result column indicates that the segment is on and a zero indicates that the segment

Figure A1.11 7-segment display

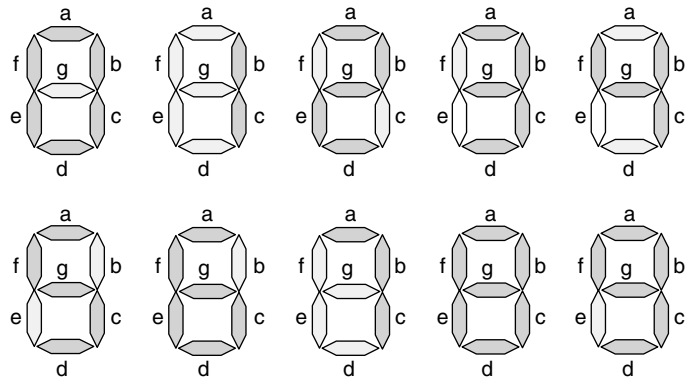
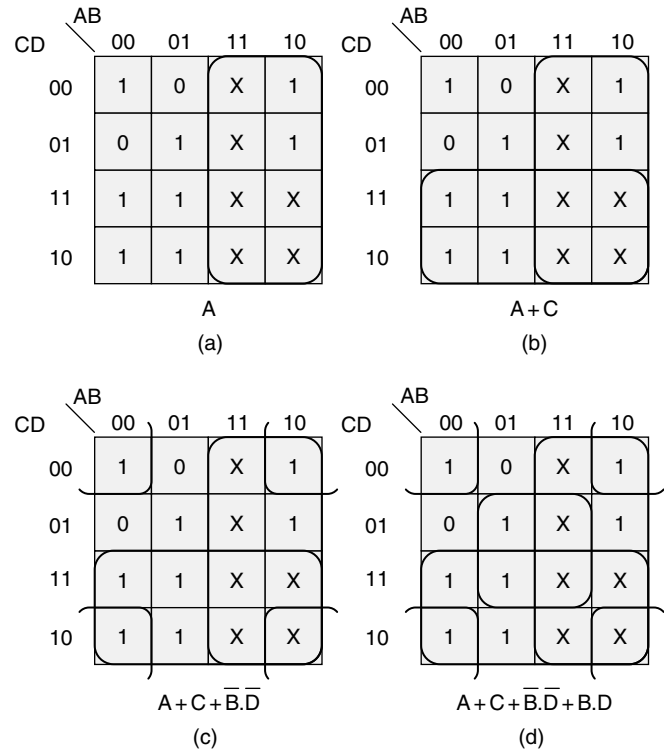


Table A1.3 Truth table for output *a* of a 7-segment display controller

A	*B*	*C*	*D*	*a*
0	0	0	0	1
0	0	0	1	0
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	X
1	0	1	1	X
1	1	0	0	X
1	1	0	1	X
1	1	1	0	X
1	1	1	1	X

Figure A1.12 Karnaugh map for segment *a* of a 7-segment display controller



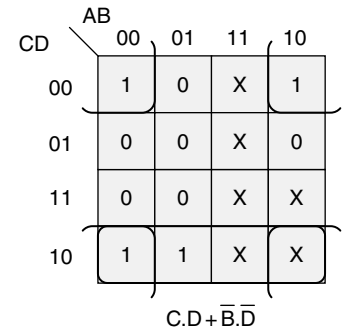
is off. As the final six states are not used, an X is used to indicate that we don't care what the output is. This comes in useful when we come to determine the minimised form for the logic expression for each output.

Figure A1.12 shows how the cells containing a 1 are grouped together to construct the minimised form. A1.12(a) shows how all the don't care cells can be grouped with the two cells in the top-right corner to give the term A. The four cells in the bottom-left corner could possibly be grouped together to give $\overline{A}$ AND C but it is more efficient to group them with four of the don't care conditions, as shown in Figure A1.12(b), to give the term C. That leaves two single cells. The one in the top-left corner can be combined with the four corners, as illustrated in Figure A1.10, to give $\overline{B}$ AND $\overline{D}$ (Figure A1.12(c)). This means that the cell in the bottom-right corner is used three times. This is not a problem as any cell can be used as many times as necessary. The final cell with a 1 in it can be grouped with the other four cells in the middle of the map, as shown in Figure A1.12(d), to give B AND D. It may appear desirable to make a group of six by including the middle two cells of the bottom row. This will not enable us to form a term that includes all six of the desirable cells but exclude the second cell in the top row. The final expression for segment *a* is therefore A OR C OR ($\overline{B}$ AND $\overline{D}$) OR (B AND D). The brackets in the previous expression are superfluous but make it easier to read.

In generating a minimal expression for segment *a*, all the don't care cells were used. It is not necessary to use all the don't care cells provided a minimal expression incorporating all the cells containing 1 is formed. Figure A1.13 shows a truth table and Karnaugh map for the *e* segment. In this case two of the six don't care cells are used. The expression for segment *e* is thus (C AND $\overline{D}$) OR ($\overline{B}$ AND $\overline{D}$).

Figure A1.13 Truth table and Karnaugh map for segment *e* of a 7-segment display controller

A	B	C	D	*e*
0	0	0	0	1
0	0	0	1	0
0	0	1	0	1
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	1
0	1	1	1	0
1	0	0	0	1
1	0	0	1	0
1	0	1	0	X
1	0	1	1	X
1	1	0	0	X
1	1	0	1	X
1	1	1	0	X
1	1	1	1	X



EXERCISES

1 Produce the Karnaugh maps and minimised expressions for traffic light set 2 in Section A1.3.1 and thus complete the logic circuit for the traffic light controller.

2 Draw up a complete truth table for the 7-segment display controller in Section A1.4.1. From the truth table, produce a complete set of Karnaugh maps and thus the minimised logic expressions for the seven outputs and finally a complete logic circuit for the controller.

Appendix 2
Introduction to debug

A2.1 Introduction

Debug is a utility that has been available to users of MS-DOS systems since the very early versions. Its original function was to offer a tool that would enable developers to identify and fix errors in programs. It has very limited facilities but can be found in the Windows operating systems of today. We will discuss enough of the facilities of debug so that we can write a few simple programs that will give us an understanding of the 80x8s programmer's model and the nature of low-level programming. Assembly language programming and checking are covered in material listed in the references at the end of this book. Rather than listing all the commands and discussing them in detail, this appendix will go through the steps involved in entering and debugging a simple program. Each command will be explained as it is encountered.

When reading through this appendix it must be remembered that all values used are in hexadecimal notation. It should also be noted that MS-DOS is not case sensitive, that is, upper and lower case letters are the same to MS-DOS. In order to make the text easier to read, the commands in the text of this appendix are written as upper case letters. The example provided in the figures however, use lower case letters. In Section A2.3, for example, the command R is exactly the same as r in Figure A2.3.

A2.2 Running debug

As debug is an MS-DOS utility, it is necessary to run it from an MS-DOS window within windows. This can be done by going to the *Run* option in the *Start* menu and entering *debug* in the file text box. This will start debug immediately. Alternatively you can enter *command* (Windows 9x) or *cmd* (Windows 2000) in the text window within the Run utility or select an MS-DOS prompt option in the start menu, if there is one, or click on the MS-DOS icon, if there is one, to start an MS-DOS session. When the MS-DOS window is open you can then enter *debug* (see Figure A2.1). When debug has started the only visible sign that the utility is running as that a simple '-' prompt appears. Debug is now awaiting a command.

Figure A2.1 Starting debug

```
Microsoft(R) Windows 98
   (C)Copyright Microsoft Corp 1981-1999.

C:\WINDOWS>debug
-
```

Figure A2.2 Debug commands

```
-?
assemble     A [address]
compare      C range address
dump         D [range]
enter        E address [list]
fill         F range list
go           G [=address] [addresses]
hex          H value1 value2
input        I port
load         L [address] [drive] [firstsector] [number]
move         M range address
name         N [pathname] [arglist]
output       O port byte
proceed      P [=address] [number]
quit         Q
register     R [register]
search       S range list
trace        T [=address] [value]
unassemble   U [range]
write        W [address] [drive] [firstsector] [number]
allocate expanded memory        XA [#pages]
deallocate expanded memory      XD [handle]
map expanded memory pages       XM [Lpage] [Ppage] [handle]
display expanded memory status  XS
-
```

A2.3 The *?* command

To see a list of all the commands available, the *?* command is useful. It will give all the commands and a brief indication of their syntax. It will not however give an explanation of the function of each command! (This is an old, free command line utility – what do you expect?) (See Figure A2.2.)

A2.4 The R command

The register command (*R*) displays the content of all the CPU registers (see Figure A2.3). The first row of the output shows the value of each of the general-purpose registers. The second row shows the value of each of the segment registers, the instruction pointer and the flags register. The flags register is shown as a series of eight mnemonics representing the value of each of the bits in the flag register. A translation of the mnemonics is shown in Table A2.1. The third row of the *R* command display shows (on the left side) the address of the next instruction to be

Figure A2.3 The *R* command

```
-r
AX=0000 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000  SI=0000  DI=0000
DS=18A1 ES=18A1 SS=18A1 CS=18A1 IP=0100 NV UP EI PL NZ NA PO NC
18A1 :0100 00A0A000       ADD     [BX+SI+00A0],AH        DS:00A0=00
-r ip
IP 0100
:0200
-r f
NV UP EI PL NZ NA PO NC  -zr
-r
AX=0000 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000  SI=0000  DI=0000
DS=18A1 ES=18A1 SS=18A1 CS=18A1 IP=0200 NV UP EI PL ZR NA PO NC
18A1 :0200 A0A700         MOV     AL,[00A7]              DS:00A7=00
-
```

Table A2.1 Flag register values

	OF	*DF*	*IF*	*SF*	*ZF*	*AC*	*PF*	*CF*
1	OV	DN	EI	NG	ZR	AC	PE	CY
0	NV	UP	DI	PL	NZ	NA	PO	NC

executed, its machine code and the assembly language equivalent. The address is in the form *CS:IP* where *CS* is the code segment (CS) register value and *IP* is the instruction pointer (IP) register value. Figure A2.3 provides an example of the use of this command. It should be noted that at this stage we have not entered any code into the computer. The information relating to the next instruction is going to be random as any data or code may have been placed in the area of memory we are currently displaying. It will also be impossible to determine the CS value in advance. The figure is thus purely illustrative and if you were to try this yourself the information is unlikely to be identical. In Figure A2.3 the address of the next instruction is *18A1:0100*. Debug interprets the memory content at this location as a 4-byte *add* instruction and displays a disassembled form of the instruction. On the right-hand side of the screen is the text *DS:00A0 = 00*. This shows that the data at the address specified by the operand in the instruction is *00*.

(1) Changing register values using the R command

Figure A2.3 demonstrates how it is possible to change register values via the *R* command. In the figure the IP value is changed from *0100* to *0200*. The command entered by the programmer was *r ip*. Debug responded by displaying the value of the IP register then a ':' prompt so that a new value could be entered.

(2) Changing flag values using the R command

Figure A2.3 then goes on to show that the *r f* command will cause debug to display the value of all the flags and then allow the user to enter a new flag value. In the Figure A2.3 the zero flag goes from *0* (NZ) to *1* (ZR).

Figure A2.4 The *A* command

```
-a 0100
18A1:0100 db 8
18A1 :0101 db 36
18A1 :0102 db 0
18A1 :0103 mov al,[0100]
18A1 :0106 add al,[0101]
18A1 :010A mov [0102],al
18A1 :010D
-
```

Figure A2.5 Tracing through a program

```
-t=0103

AX=0008 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000  SI=0000  DI=0000
DS=18A1 ES=18A1 SS=18A1 CS=18A1 IP=0106 NV UP EI PL ZR NA PO NC
18A1:0106 02060101      ADD     AL,[0101]               DS:0101=36
-t

AX=003E BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000  SI=0000  DI=0000
DS=1BEB ES=1BEB SS=1BEB CS=1BEB IP=010A NV UP EI PL NZ NA PO NC
18A1:010A A20201        MOV     [0102],AL               DS:0102=3E
-t

AX=003E BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000  SI=0000  DI=0000
DS=1BEB ES=1BEB SS=1BEB CS=1BEB IP=010D NV UP EI PL NZ NA PO NC
18A1:010D 0000          ADD     [BX+SI],AL              DS:0000=CD
-
```

A2.5 The *A* command

The assemble command (*A*) allows a programmer to enter a program using assembly language mnemonics. Figure A2.4 shows a short program to add two 8-bit memory resident numbers and place the result back in memory. The command used was *a 0100*, which tells debug where to start storing the program within the code segment. MS-DOS programs require a 100-byte *program segment prefix* (PSP) at the start of each program so the default start for any code loaded would be *CS:0100*. If the start address is omitted then the code will be stored starting from the IP value.

If no instruction is entered for a particular address, such as location *010D* in Figure A2.4 then the programmer will be returned to the debug prompt.

A2.6 The *T* command

The trace command (*T*) allows a single instruction to be executed. This is often referred to as *single-stepping* through a program. This enables the programmer to study the effects of each instruction. Figure A2.5 shows the effect of stepping through the example program. The first of the trace commands has the format $t = 0103$. This tells debug to trace the instruction at offset *0103* within the code segment. The first three lines of the program entered were data and should not be traced. Debug will

Figure A2.6 Displaying memory locations

```
-d 0100 l10
1BEB:0100 08 36 3E A0 00 01 02 06-01 01 A2 02 01 00 00 EB .6>.............
-
```

not know that they are data and will try to interpret them as instructions. After the *T* command the registers are displayed so you can see what happened. In this case the AX value changed from *0000* to *0008*. (Remember that AL is the least significant eight bits of AX.) The next two instructions do not need an address as they will trace the instruction determined by IP and IP is incremented after each instruction. Note that the final instruction should be storing the value produced in AL (3E), which is done by writing it to memory. The effect of this instruction cannot be seen by just studying the registers.

A2.7 The *D* command

The display or dump command (*D*) enables the programmer to view the contents of memory. The *D* command is generally used with at least one argument. A single argument would tell debug from where to start displaying the contents of memory. A second argument indicates to debug the number of bytes to display (see Figure A2.6). This second argument needs to be preceded by an *l* for length. If no argument is provided then a block of 128 bytes will be displayed.

In Figure A2.6 16 (10_{16}) locations are displayed starting at location *0100*. Each line displayed by the *D* command will end with the ASCII equivalent of each memory location. If there is not a printable ASCII character corresponding to that value then a '.' is displayed. In the example only the locations containing the values *36* and *3E* have ASCII equivalent characters. These characters are *6*>.

A2.8 The *U* command

It is possible to display the program at a particular location in memory using the unassemble command (*U*). Some thought needs to be used when applying this command. The first example in the use of this command in Figure A2.7 shows the *U* command unassemblying the program from location *0100*. Unfortunately the first three bytes are data, which are now being interpreted as instructions so the results are not valid. The second example *u 0103 010A* will unassemble the code only, as the data for this program is outside this range.

A2.9 The *G* command

Single-stepping through a long program will be very tedious. An alternative way of executing a piece of code, which you are fairly certain is correct, is by using the go command (*G*). This will execute the program from the current IP location or from an address preceded by an '=' until a *breakpoint*. A breakpoint is an address at which the execution of the program will finish or pause. Up to 10 breakpoints can be specified. The example in Figure A2.8 specifies *0100* as the start address and *010D* as the end address or breakpoint.

Figure A2.7 Unassemble example

```
-u 0100
1BEB:0100 08363EA0      OR      [A03E],DH
1BEB:0104 0001          ADD     [BX+DI],AL
1BEB:0106 02060101      ADD     AL,[0101]
1BEB:010A A20201        MOV     [0102],AL
1BEB:010D 0000          ADD     [BX+SI],AL
1BEB:010F EB13          JMP     0124
1BEB:0111 57            PUSH    DI
1BEB:0112 26            ES:
1BEB:0113 8B3E92DE      MOV     DI,[DE92]
1BEB:0117 26            ES:
1BEB:0118 894DFE        MOV     [DI-02],CX
1BEB:011B 5F            POP     DI
1BEB:011C 3400          XOR     AL,00
1BEB:011E DA1B          FICOMP  DWORD PTR [BP+DI]
-u 0103 010a
1BEB:0103 A00001        MOV     AL,[0100]
1BEB:0106 02060101      ADD     AL,[0101]
1BEB:010A A20201        MOV     [0102],AL
-
```

Figure A2.8 Execute to a breakpoint

```
-g=0103 010d

AX=003E BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000  SI=0000  DI=0000
DS=18A1 ES=18A1 SS=18A1 CS=18A1 IP=010D NV UP EI PL NZ NA PO NC
18A1:010D 0000          ADD     [BX+SI],AL              DS:0000=CD
-
```

Figure A2.9 The *E* command

```
-e 0100 'anything'
-d 0100
18A1:0100 61 6E 79 74 68 69 6E 67-01 01 A2 02 01 00 00 EB anything........
18A1:0110 13 57 26 8B 3E 92 DE 26-89 4D FE 5F 34 00 DA 1B .W&.>..&.M._4...
18A1:0120 04 E8 5A FA 0E 1F E9 F5-B9 E8 40 FE 81 3E 1A 04 ..Z.......@..>..
18A1:0130 00 F0 75 06 C7 06 1A 04-FF FF E9 A7 B6 BF A1 89 ..u.............
18A1:0140 C7 06 8E D3 01 00 E8 E4-03 73 22 0B C0 75 3C E8 .........s"..u<.
18A1:0150 AC 0B 86 F2 89 0E 07 83-89 16 09 83 BA F5 82 E8 ................
18A1:0160 BD 2F E8 FA 00 73 06 0B-C0 74 1F EB 1E 8B 0E BF ./...s...t......
18A1:0170 E1 8A 36 C1 E1 8A 16 C2-E1 51 52 E8 61 04 5A 59 ..6......QR.a.ZY
-e 0110 61 6e 79 74 68 69 6e 67
-d 0100 120
18A1:0100 61 6E 79 74 68 69 6E 67-01 01 A2 02 01 00 00 EB anything........
18A1:0110 61 6E 79 74 68 69 6E 67-89 4D FE 5F 34 00 DA 1B anything.M._4...
-
```

A2.10 The *E* command

The enter command (*E*) is used to enter data. The *A* command is used to enter code so the start address is an offset within the code segment. The *E* command is for data and thus the address argument is an offset within the data segment.

Figure A2.9 shows two ways of entering the text data *anything* into memory. The first shows that debug can accept a literal and store it into consecutive areas of memory. The second shows the programmer entering the data as separate character codes. Data can be entered into the code segment if a full address is given that is *CS:0100*. Debug will then replace CS with the current code segment register value.

A2.11 The *P* command and invoking operating system functions

In order to correctly demonstrate the use of the proceed command (*P*), we need a more extensive example program. The *P* command is used to execute instructions such as the *int* instruction. The *int* instruction causes a software interrupt or *trap*. This type of instruction is used to invoke operating system functions. The example in Figure A2.10 is a program, which will use an operating system function to read a line of text from the keyboard and another function to write the text back to the screen. A third operating system function will terminate the program.

Note that the characters appearing after a ';' are comments and do not need to be entered as part of the program. They are supplied to give a brief indication of what each part of the program is for. The data between *0100* and *0112* are required for the function that will receive the data from the keyboard. The first byte (0100) indicates the maximum number of characters that the program is willing to accept. The next byte (0101) will hold the number of characters actually input. The following 16 bytes will hold the characters input. To accept 16 characters from the user a seventeenth byte is required by the program to hold an end of string character. This character will be a *Carriage Return* (CR) (ASCII code 0D) character.

MS-DOS makes many of its I/O routines available via interrupt number 21_{16}. To differentiate between the different functions, an argument is passed to the interrupt handler via the AH register. To cause the *buffered input from console with echo* function to be invoked AH must contain the value *0A* when *int 21* is executed. In addition, DX must contain the address of the start of the parameter list that is the

Figure A2.10 Extended example

```
0100 db 10          ;max. no character to input
0101 db 0           ;actual no. characters input
0102 db 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
                    ;input buffer
0112 db 0           ;allow for end-of-string

;read buffered input from console with echo
0113 mov ah,0a      ;int 21 function no. oa
0115 mov dx,0100    ;start of param. list
0118 int 21         ;invoke os routine

;place a $ at end of string
011a mov cl,24      ;character 24 = $
011c mov al,[0101]
                    ;al holds number of characters
011f cbw            ;convert al byte to ax word
0120 mov bx,0102    ;initialise bx to start of string
0123 add bx,ax      ;bx now points to end of string
0125 mov [bx],cl    ;place $ at end of string

;output string to console
0127 mov dx,0102    ;dx holds start of string
012a mov ah,09      ;int 21 function no. 09
012c int 21         ;invoke os routine

;terminate program
012e int 20
```

Figure A2.11 Inputting an input string

```
-t=0113

AX=0A08  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=18A1  ES=18A1  SS=18A1  CS=18A1  IP=0115   NV UP EI NG NZ NA PE CY
18A1:0115 BA0001        MOV     DX,0100
-t

AX=0A08  BX=0000  CX=0000  DX=0100  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=18A1  ES=18A1  SS=18A1  CS=18A1  IP=0118   NV UP EI NG NZ NA PE CY
18A1:0118 CD21          INT     21
-p
text input
AX=0A0D  BX=0000  CX=0000  DX=0100  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=18A1  ES=18A1  SS=18A1  CS=18A1  IP=011A   NV UP EI NG NZ NA PE CY
18A1:011A B124          MOV     CL,24
-d 0100 120
1BEB:0100 10 0A 74 65 78 74 20 69-6E 70 75 74 0D 00 00 00 ..text input....
1BEB:0110 00 00 00 B4 0A BA 00 01-CD 21 B1 24 A0 01 01 98 .........!.$....
-
```

maximum number of characters, actual number of characters and input buffer. The code between *0113* and *0119* will cause an input string to be received by the program.

Figure A2.11 shows the first two instructions of the program being traced and then the *P* command being used to execute the *int* command without having to step through it. A *D* command is then used to confirm the contents of the input buffer. The string entered by the user was *text input*. From the dump, it is possible to determine from the value in *0101* that this string is ten characters long. You can also see that a eleventh character is added. This is the *0D* character or *CR* character which is not included in the character count.

The next part of the program needs to replace the *0D* character by a *$* character (ASCII code 24). This is because the routine which outputs strings to the console requires the string to be terminated by a *$*. The instruction at location *011A* places a *$* character in the CL register. AL is then loaded with the length of the string. Because we wish to add this to a 16-bit register we extend AL to the whole of AX by using a *convert binary to word* (CBW) instruction (011F). BX is loaded with the start address of the string and then the value in AX is added to it to give the address of the end of the string. The *$* character is then stored at that location (0125). Figure A2.12 shows the final part of the program being traced with dumps immediately before and after the *$* character is stored in memory, to show that the *$* has been placed at the correct location.

Now that the string to be output is properly prepared, the routine to output it can be invoked. This is another *int 21* routine but this time the argument in AH must be *09* (012A). It is also necessary for *DX* to hold the address of the start of the string (0127). Figure A2.13 shows this section of the program being traced and the *int 21* function being executed using a proceed command. There is then another proceed command to execute the *int 20* instruction. *int 20* invokes an operating system routine which terminates the execution of the program.

A2.12 The *Q* command

The final essential command is the quit command (*Q*). This terminates the debug session.

Figure A2.12 Inserting a string termination character

```
AX=0A0D  BX=0000  CX=0024  DX=0100  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=18A1  ES=18A1  SS=18A1  CS=18A1  IP=011C   NV UP EI NG NZ NA PE CY
18A1:011C A00101        MOV     AL,[0101]                   DS:0101=0A
-t

AX=0A0A  BX=0000  CX=0024  DX=0100  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=18A1  ES=18A1  SS=18A1  CS=18A1  IP=011F   NV UP EI NG NZ NA PE CY
18A1:011F 98            CBW
-t

AX=000A  BX=0000  CX=0024  DX=0100  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=18A1  ES=18A1  SS=18A1  CS=18A1  IP=0120   NV UP EI NG NZ NA PE CY
18A1:0120 BB0201        MOV     BX,0102
-t

AX=000A  BX=0102  CX=0024  DX=0100  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=18A1  ES=18A1  SS=18A1  CS=18A1  IP=0123   NV UP EI NG NZ NA PE CY
18A1:0123 89C3          ADD     BX,AX
-t

AX=000A  BX=010C  CX=0024  DX=0100  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=18A1  ES=18A1  SS=18A1  CS=18A1  IP=0125   NV UP EI NG NZ NA PE CY
18A1:0125 880F          MOV     [BX],CL                     DS:000A=4F
-d 0100 110
18A1:0100 10 0A 74 65 78 74 20 69-6E 70 75 74 0D 00 00 00 ..text input....
-t

AX=000A  BX=010C  CX=0024  DX=0100  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=18A1  ES=18A1  SS=18A1  CS=18A1  IP=0127   NV UP EI NG NZ NA PE CY
18A1:0127 BA0201        MOV     DX,0102
-d 0100 110
18A1:0100 10 0A 74 65 78 74 20 69-6E 70 75 74 24 00 00 00 ..text input$...
-
```

Figure A2.13 Outputting a string and terminating the program

```
-t

AX=000A BX=010C CX=0024 DX=0102 SP=FFEE BP=0000  SI=0000  DI=0000
DS=18A1 ES=18A1 SS=18A1 CS=18A1 IP=012A NV UP EI PL NZ NA PE NC
18A1:012A B409          MOV     AH,09
-t

AX=090A BX=010C CX=0024 DX=0102 SP=FFEE BP=0000  SI=0000  DI=0000
DS=18A1 ES=18A1 SS=18A1 CS=18A1 IP=012C NV UP EI PL NZ NA PE NC
18A1:012C CD21          INT     21
-p
text input
AX=0924 BX=010C CX=0024 DX=0102 SP=FFEE BP=0000  SI=0000  DI=0000
DS=18A1 ES=18A1 SS=18A1 CS=18A1 IP=012E NV UP EI PL NZ NA PE NC
18A1:012E CD20          INT     20
-p

Program terminated normally
-
```

EXERCISES

1 Amend the program in Figure A2.10 so that it will prompt the user with the message 'Input a name': before the user is required to enter a string.

2 Take the program from Exercise 1 and amend it so that when the name is output it is reversed. See Figure 5.16 to see how this might be achieved.

Appendix 3
ASCII and extended ASCII tables

As computer systems developed, different manufacturers used different codes to represent the different types of data to be represented within the computer system. This was found to be a problem when it then became necessary to share data among systems. One of the first standards to be developed to represent data was the American Standard Code for Information Interchange developed by the American National Standards Institute (ANSI). This is a 7-bit code to represent character data. This enables up to 128 characters to be represented. The first 32 codes are used for transmission and device control codes. This leaves 96 codes to represent printable characters.

A 7-bit code was used as the standard as this meant that the code plus a parity bit could be stored within a single byte. Ninety six character codes were thought sufficient when the standard was introduced as applications at the time were rather limited and also computers were being developed in countries where English was the first language. It soon became clear that the 7-bit code was insufficient and an extended version was developed based on 8-bit codes; dispensing with the parity bit. The first 128 codes were unchanged. Initially there was no widely accepted standard for the new 128 codes. Work on standardising the extended character sets was carried out in the late 1980s through Xerox's work on codes for Chinese character sets and Apples work on developing a file exchange standard. The Unicode Consortium evolved from this work. Version 1 of the standard was published in 1991. The current version is version 3 and the consortium has over 50 corporate members.

There are a number of code charts developed by Unicode. The Basic Latin chart is compatible with the 7-bit ASCII chart. This is then extended via the Latin-1 supplement also known as ISO 8859-1. The two charts combine to form a standard form of extended ASCII chart using 8-bit codes. This chart is shown in Figure A3.1. The small square boxes in Figure A3.1(a) indicate where non-printing control codes appear. The first 32 control codes are separately listed in Figure A3.1(b). To determine the code for a particular character, use the row number as the least significant 4 bits, or nibble, and the column number as the most significant nibble. The character 'A', for example, would be code 41.

These charts would obviously be of little use when trying to convey information in non-latin alphabets. The most important work that the Unicode Consortium has undertaken is to develop a fully international character set. To achieve the

Figure A3.1 ASCII chart

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	□	□		0	@	P	`	p	□	□		°	À	Ð	à	ð
1	□	□	!	1	A	Q	a	q	□	□	¡	±	Á	Ñ	á	ñ
2	□	□	"	2	B	R	b	r	□	□	¢	²	Â	Ò	â	ò
3	□	□	#	3	C	S	c	s	□	□	£	³	Ã	Ó	ã	ó
4	□	□	$	4	D	T	d	t	□	□	¤	´	Ä	Ô	ä	ô
5	□	□	%	5	E	U	e	u	□	□	¥	µ	Å	Õ	å	õ
6	□	□	&	6	F	V	f	v	□	□	¦	¶	Æ	Ö	æ	ö
7	□	□	'	7	G	W	g	w	□	□	§	·	Ç	×	ç	÷
8	□	□	(	8	H	X	h	x	□	□	¨	¸	È	Ø	è	ø
9	□	□	)	9	I	Y	i	y	□	□	©	¹	É	Ù	é	ù
A	□	□	*	:	J	Z	j	z	□	□	ª	º	Ê	Ú	ê	ú
B	□	□	+	;	K	[	k	{	□	□	«	»	Ë	Û	ë	û
C	□	□	,	<	L	\	l	|	□	□	¬	¼	Ì	Ü	ì	ü
D	□	□	-	=	M	]	m	}	□	□	-	½	Í	Ý	í	ý
E	□	□	.	>	N	^	n	~	□	□	®	¾	Î	Þ	î	þ
F	□	□	/	?	O	_	o	□	□	□	¯	¿	Ï	ß	ï	ÿ

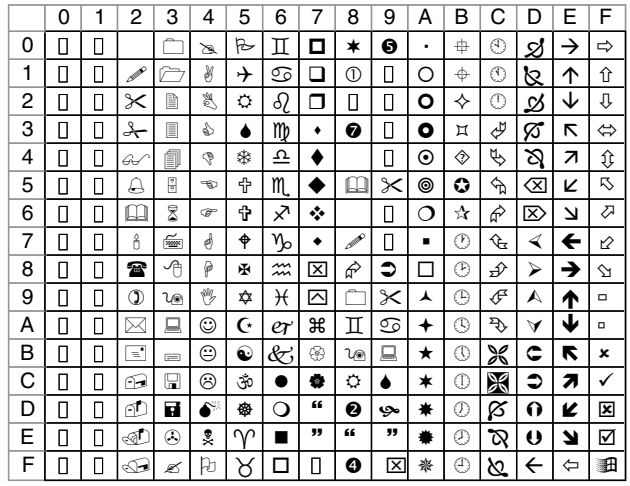
(a) Extended ASCII chart

	1	2
0	NUL (Null)	DLE (Data Link Escape)
1	SOH (Start or Heading)	DC1 (Device Control 1)
2	STX (Start of Text)	DC2 (Device Control 2)
3	ETX (End of Text)	DC3 (Device Control 3)
4	EOT (End of Transmission)	DC4 (Device Control 4)
5	ENQ (Enquire)	NAK (Negative Acknowledge)
6	ACK (Acknowledge)	SYN (Synchronous Idle)
7	BEL (Bell)	ETB (End of Transmission Block)
8	BS (Backspace)	CAN (Cancel)
9	HT (Horizontal Tab)	EM (End of Medium)
A	LF (Line Feed)	SUB (Substitute)
B	VT (Vertical Tab)	ESC (Escape)
C	FF (Form Feed)	FS (File Seperator)
D	CR (Carriage Return)	GS (Group Seperator)
E	SO (Shift Out)	RS (Record Seperator)
F	SI (Shift In)	US (Unit Seperator)

(b) Control characters

international version it was necessary to move to a 16-bit code. Although this was a great help in supporting most of the languages of the world, the 64000 codes available are insufficient for a truly international standard and so 24- and 32-bit codes are also used. There are now almost 100000 characters represented in Unicode version 3.2. Not all these characters are textual characters; other symbols are also represented. Codes 2700–27BF are used for some useful symbols and fall within the Dingbats code chart. A similar set of symbols can be found on most PCs. If a Dingbats font is not available there will almost certainly be a Wingdings font (see Figure A3.2). Wingdings has a similar set of characters to Dingbats but is not a direct replacement. Unlike Dingbats, Wingdings uses a set of codes which are reserved by Unicode for private use. This means that there is no guarantee that an application on a different system will display a similar character or symbol.

Figure A3.2 Wingdings



Appendix 4
The 80x86 family of processors

A4.1 Introduction

The Pentium processor, familiar in today's Personal Computers (PCs), can trace its roots back to the Intel 4004 microprocessor of 1971. This was a *4-bit processor*; that is, it had 4-bit registers and external data bus. In 1972 the 8008, was introduced which had an 8-bit bus. This was followed in 1974 by the 8080, which was faster than its predecessor.

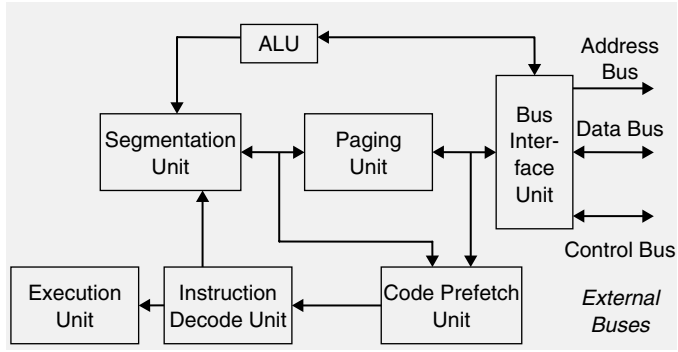
By the end of the 1970s it became necessary to move to 16-bit architectures. Some of Intel's competitors, such as Motorola, radically redesigned their CPUs but Intel decided to evolve their 16-bit design from their 8-bit CPU. This meant it was easier for software developers to produce products to run on their new CPUs. This has been a policy that Intel has followed ever since. The first 16-bit processor from Intel was the 8086. Code written for this processor will run on Pentium processors available now. This is good for migration from one generation of the architecture to the next but can adversely affect performance, as compromises need to be made in the design.

The 8086 had 16-bit registers, a 16-bit data bus and a 20-bit address bus. This gave the processor the ability to address 1 MByte of memory. (Note that the memory unit addressed was 8 bits long despite a 16-bit data bus. This enables 8-bit data operations as well as 16-bit operations to be executed. In the case of 16-bit operations, the addressed byte and the following byte form a single 16-bit data item.) This processor was followed in 1979 by the more cost-effective 8088 that had an 8-bit data bus. A block diagram of the 8086/8 CPU is given in Figure 5.3. The only difference between the 8088 and 8086 internally was that the 8088 was limited to a 4-byte instruction queue. A description of the registers in the bus interface unit and Execution unit are given in Section A4.1.

TQ A4.1 The 8080 had a 16-bit address bus, so what was the maximum amount of memory it could address?

To be able to address 1 MByte of memory the 8086 used *segmentation*. The 8086 had four 16-bit segment registers, any one of which may form the most significant 16 bits of a 20-bit address (the least significant four bits being filled with zeros).

Figure A4.1 Simplified Intel 386 pipelined architecture showing 32-bit internal data paths



This provides the start address of a 64 KByte segment to which is added a 16-bit offset to form the effective address. This technique is discussed further in Section 5.1.3.

In 1982 the 80186/80188 was introduced. This was basically an 8086/8 with a number of functions that had been performed by support devices brought onto the CPU chip. These functions include a clock generator, direct memory accessing (DMA) and interrupt controllers, programmable timers and local bus controllers. This *integrated* CPU underwent its own development independently of the rest of the family.

The successor to the 8086 is the 80286 processor introduced in 1982. This CPU introduced a 24-bit address bus and 1 GByte of *virtual memory*. Descriptors provide 24-bit base addresses and memory can be swapped to and from virtual memory on a segment basis.

The Intel 386 processor (the initial 80 seems to have been dropped at this stage) introduced in 1985 had 32-bit registers and buses. It maintained the segment addressing of the 80286 but could also support a 'linear' 4 GByte address space (segmentation unnecessary). An application transparent paged virtual memory system was introduced offering a virtual memory size of 64 TBytes. The 386 had six parallel stages that form a sort of *macro pipeline* to improve performance. The stages were the bus interface unit, code pre-fetch unit, instruction decode unit, execution unit, segment unit and paging unit (see Figure A4.1).

The Intel 486, introduced in 1989, added 5-stage pipelines to the instruction decode and execution units. An 8 KByte on-board *first level cache* was included along with a Floating Point Unit (FPU), which had previously been a separate chip (80x87). Later versions of this CPU included support for power management. This was necessary to support the growing demand for battery-powered devices. The Pentium processor, introduced in 1993, added a second execution pipeline to achieve what Intel called *superscalar* performance (two pipelines operating in parallel). There were now two on-board caches in the *Harvard architecture* style (see Section 4.3.2). *Branch prediction* was added to increase performance regards looping constructs. Internal data paths of 128 and 256 bits were added to speed-up internal operations. An Advanced Programmable Interrupt Controller (APIC) was added to ease the construction of dual processor systems. Later versions of the CPU included multimedia extensions (MMX). These extensions included additional instructions and 64-bit registers that allowed a single instruction to be executed on multiple integer data items packed into the MMX registers. This form of execution is referred to as single instruction, multiple data (SIMD) and is particularly useful in multimedia applications where the same operation needs to be performed on many items of data.

In 1995 Intel modified the underlying architecture of their Pentium processors. This new micro-architecture was referred to as P6. The processors in this family were the Pentium Pro, Pentium II, Pentium II Xeon, Celeron, Pentium III and Pentium III Xeon. The Pentium Pro had enhanced parallel features that included three parallel instruction pipelines, five parallel execution pipelines, multi-ported cache and a 64-bit external data bus that could handle multiple overlapped requests. The address bus was extended to 36 bits. The Pentium II processor was the first P6 CPU to include MMX technology and came in a new form of a single edge contact cartridge (SECC). The Pentium II Xeon was intended for higher performance systems and thus included support for higher speed buses and multi-processor design. The Celeron, on the other hand, was designed for the budget market and to keep costs down used a plastic pin grid array (PPGA) and less on-board cache. The Pentium III processor introduced Streaming SIMD Extensions (SSE). The SSE added a new set of 128-bit registers (known collectively as the XMM registers) and the ability to perform SIMD operations on packed single precision floating-point values to the MMX technology.

For the recently introduced Pentium 4 the P6 architecture has been reworked to improve performance and scalability principally through supporting greater parallelism and is now called the NetBurst micro-architecture. The Pentium 4 also includes an enhanced form of SSE (SSE2), which has a new enhanced instruction set, 128-bit SIMD integer arithmetic and 128-bit floating-point operation. The bus speeds have been improved by up to three times that of the Pentium III.

A4.2 The programmer's model

In the introduction we were considering the physical aspects of the 80x86. What we will do now is consider the processor from the programmer's perspective. The execution environment of a Pentium 4 processor consists of

(a) Basic program execution registers

- eight 32-bit general purpose registers
- six 16-bit segment registers
- one 32-bit Flag register
- one 32-bit instruction pointer

(b) Floating point registers

- eight 80-bit floating point registers
- 16-bit control/status and tag registers

(c) MMX registers

- eight 64-bit registers

(d) SSE and SSE2 registers

- eight 128-bit XMM registers
- one 32-bit MXCSR register

Details of the six-segment registers are given in Table A4.1. If this is compared to the table of segment registers available in the 8086 you will notice the addition of two registers which have no specific function but can be used to increase the quantity of memory available to the program at any one time. The segment registers were introduced in the 8086 CPU to enable addresses to be formed from 16-bit values to be placed on a 20-bit address bus as discussed earlier.

Table A4.1 P4 segment registers

CS	Code Segment start address of segment containing executable code
DS	Data Segment start address of segment containing program data
SS	Stack Segment start address of segment containing a stack data structure
ES	Extra Segment start address of a general segment
FS	General purpose segment register
GS	Start addresses segments of no default function

Table A4.2 P4 general-purpose registers

EAX	accumulator for operands and results data
EBX	points to data in the DS segment
ECX	counter for string and loop operations
EDX	I/O pointer
ESI	pointer to data in the segment pointed to by the DS register; source pointer for string operations
EDI	pointer to data (or destination) in the segment pointed to by the ES register; destination pointer for string operations
ESP	stack pointer
EBP	pointer to data on the stack

Although the eight general-purpose registers can be used to store any data or address they also have a special function in some instructions. The registers and their special functionality are listed in Table A4.2. This table should be compared with the table of 8086 general-purpose registers (Table 5.1). These registers can also be used as 16-bit registers to maintain 8086 compatibility. The 16-bit registers can be divided into two 8-bit registers. Table A4.3 lists the registers and illustrates how they are divided up into smaller units.

Table A4.3 Alternate general-purpose register names

31 16	*15* 8	*7* 0	*16-bit*	*32-bit*
	AH	AL	AX	EAX
	BH	BL	BX	EBX
	CH	CL	CX	ECX
	DH	DL	DX	EDX
	SP			ESP
	BP			EBP
	SI			ESI
	DI			EDI

Appendix 5
IEEE 754 floating point format

In Section 2.7, floating point (FP) representation was considered in a general way. In this appendix we will look at the standard format IEEE 754 for FP numbers.

A FP number such as $-0.100100101 \times 2^{+1001}$, has four components we need to consider; the mantissa (0.100100101), the sign of the mantissa (−), the exponent (1001) and the sign of the exponent (+). As the numbers represented within the computer system will always be binary the base (2) can be ignored.

Let us first consider the exponent and its sign. The IEEE 754 standard uses a method of representing the exponent using a *bias*. If the number of bits used to represent the exponent is *e* then the bias will be $2^{e-1} - 1$. If eight bits are to be used to hold the exponent then the bias will be $2^7 - 1$ or 127_{10}. The *biased exponent* is determined by adding the true exponent to the bias. As the smallest value, which can be represented in eight bits, is zero, and the largest is 255, then the range of true exponents is −127 to 128.

Let us now consider the mantissa. This will always be in the form 0.1mmm … . where m is a bit of the mantissa. It is thus unnecessary to store the 0.1; we only need the following bits. When we come to do the arithmetic then the 0.1 can be reinstated. If we have an 8-bit mantissa and we wish to store a FP number with a mantissa of 0.100100101 then the value stored will be 00100101. The sign of the mantissa will always be held in the most significant bit (1 for negative, 0 for positive). Note that, unlike the way negative integers are represented, neither the mantissa nor exponent uses two's complement notation. The exponent is biased and the mantissa is sign and magnitude.

IEEE 754 defines a 32-bit format and a 64-bit format for FP numbers. In the 32-bit format there is an 8-bit biased exponent and in the 64-bit format there is an 11-bit biased exponent. The layout of the two forms is shown in Figure A5.1. Figure A5.2 shows how the number $-0.100100101 \times 2^{+1001}$, will be stored in IEEE 754 32-bit format. Note that the exponent is 9_{10}. Add this true exponent to the bias (127) giving 136_{10} or 10001000_2.

Figure A5.1 IEEE 754 (a) 32-bit and (b) 64-bit floating-point standard

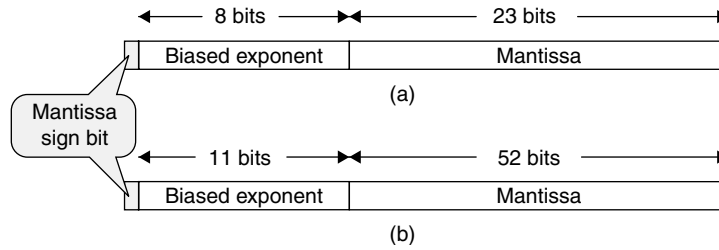


Figure A5.2 IEEE 754 example

$-0.100100101 \times 2^{1001}$ floating point notation

1 10001000 00100101000000000000000 IEEE 754 format

Acronyms

ACL	Access Control List
ADSL	Asymmetric Digital Subscriber Line
ALU	Arithmetic and Logic Unit
ANSI	American National Standards Institute
APIC	Advanced Programmable Interrupt Controller
ASCII	American Standard Code for Information Interchange
ATM	Asynchronous Transfer Mode
AUI	Access Unit Interface
BCD	Binary Coded Decimal
BIOS	Basic Input–Output System
CD-R	Compact Disk-Recordable
CD-ROM	Compact Disk-Read Only Memory
CD-RW	Compact Disk -ReWritable
CISC	Complex Instruction Set Computer
CMOS	Complementary Metal Oxide Semiconductor
COM	Communications
CPI	Clock cycles Per Instruction
CPU	Central Processing Unit
CRT	Cathode Ray Tube
CS	Code Segment (Intel 80 x86)
CU	Control Unit
CWP	Current Window Pointer
DAT	Digital Audio Tape
DDR-RAM	Double Data Rate Random Access Memory
DES	Data Encryption Standard
DIMM	Dual In-line Memory Module
DLP	Digital Light Processor
DMA	Direct Memory Access
DNS	Domain Name Service
DOS	Disk Operating System
DRAM	Dynamic Random Access Memory
DS	Data Segment (Intel 80x86)
DVD	Digital Versatile Disk
DVD-R	Digital Versatile Disk – Recordable
DVD-ROM	Digital Versatile Disk – Read Only Memory
DVD-RW	Digital Versatile Disk – ReWritable
EDC/ECC	Error Detection Code/Error Correction Code
EEPROM	Electrically Erasable Programmable Read Only Memory
EIDE	Extended Integrated Drive Electronics
EPROM	Erasable Programmable Read Only Memory

FAT	File Allocation Table
FCFS	First-Come-First-Served
FCS	Frame Check Sequence or Frame Check Sum
FDDI	Fibre Distributed Data Interface
FIFO	First-In-First-Out
FP	Floating Point
FPU	Floating Point Unit
FR	Flag Register
FSB	Front Side Bus
FSK	Frequency Shift Keying
FTP	File Transfer Protocol
GB	GigaBytes
Gb	Gigabits
GUI	Graphical User Interface
HDA	Hard Disk Assembly
Hex	Hexadecimal
Hi-Fi	High Fidelity
HLL	High Level Language
HTTP	HyperText Transfer Protocol
IC	Integrated Circuit
IDE	Integrated Drive Electronics
IDEA	International Data Encryption Algorithm
IEEE	Institute of Electrical and Electronic Engineers
I/O	Input Output
IP	Instruction Pointer also Internet Protocol
IPC	Inter Process Communication
IRET	Interrupt Return
ISA	Industry Standard Architecture
ISDN	Integrated Services Digital Network
ISO OSI	International Standards Organisation Open Systems Inter-connect
ISP	Internet Service Provider
KB	KiloBytes
Kb	Kilobits
LAN	Local Area Network
LCD	Liquid Crystal Display
LED	Light Emitting Diode
LEP	Light Emitting Polymer
LLC	Logical Link Control
LRU	Least Recently Used
LSB	Least Significant Bit
MAC	Media Access Control
MAR	Memory Address Register
MB	MegaBytes
Mb	Megabits
MBR	Memory Buffer Register
MICR	Magnetic Ink Character Recognition
MIDI	Musical Instrument Digital Interface
MIPS	Million Instructions Per Second
MMU	Memory Management Unit
MMX	Multi-Media eXtensions
MODEM	MODulator DEModulator

MSB	Most Significant Bit
MS-DOS	Microsoft Disk Operating System
NIC	Network Interface Card
OCR	Optical Character Recognition
OLED	Organic Light Emitting Diode
OS	Operating System
PC	Program Counter or Personal Computer
PCB	Printed Circuit Board
PCB	Process Control Block
PCI	Peripheral Component Interface
PDA	Personal Data Assistant
POST	Power-On Self Test
PPGA	Plastic Pin Grid Array
PROM	Programmable Read Only Memory
RAID	Redundant Array of Independent Disks
RAM	Random Access Memory
RAS	Remote Access Service
RD-RAM	Rambus Dynamic Random Access Memory
RIMM	Rambus In-line Memory Module
RISC	Reduced Instruction Set Computer
RIU	Ring Interface Unit
ROM	Read Only Memory
RR	Round Robin
RSA	Rivest, Shamir and Aldeman
RTL	Register Transfer Language
RTN	ReTurN from subroutine
R/W	Read/Write
SCSI	Small Computer System Interface
SECC	Single Edge Contact Cartridge
SIMM	Single In-line Memory Module
SIMD	Single Instruction Multiple Data
SIPO	Serial-In-Parallel-Out
SMM	System Management Mode
SMTP	Simple Mail Transfer Protocol
SPEC	Standard Performance Evaluation Corporation
SPARC	Scalable Processor ARChitecture
SRAM	Static Random Access Memory
SS	Stack Segment (Intel 80x86)
SSE	Streaming Single instruction Stream-multiple data stream Extensions
SVGA	Super Video Graphics Array
TAS	Test And Set
TB	TeraBytes
Tb	Terabits
TCP/IP	Transmission Control Protocol/Internet Protocol
TLB	Translation Look-aside Buffer
UDP	User Datagram Protocol
UPC	Universal Product Code
URL	Universal Resource Locator
USB	Universal Serial Bus
UTP	Unshielded Twisted Pair
VLAN	Very Local Area Network

VLIW	Very Long Instruction Word
VLSI	Very Large Scale Integration
WAN	Wide Area Network
WiFi	Wireless Fidelity
WORM	Write-Once Read Many times
WREM	Write-Read-Erase Memory

References and further reading

Abel, L. IBM PC Assembly Language and Programming. 5th Ed. Prentice Hall 2001.
Crusoe Processor Product Brief. Model TM5800. Transmeta 2003. Downloadable from www.transmeta.com.
Hamacher, Vranesic and Zaky. Computer Organization. 5th Ed. McGraw-Hill 2001.
Kingston Technology, Ultimate Memory Guide. Kingston Technology 2001. Downloadable from www.kingston.com.
Lister, A.M, Eager, R.D. Fundamentals of Operating Systems. 5th Ed. Macmillan Press Ltd 1993.
Mano, M. Digital Design. 3rd Ed. Prentice Hall 2002.
MIPS R5000 Microprocessor. MIPS Technologies Technical Backgrounder. Downloadable from www.mips.com.
Simha, S. R4400 Microprocessor Product Information. MIPS Technologies 1996. Downloadable from www.mips.com.
Sparc Architecture Manual. Version 9. Downloadable from www.sparc.org.
Stallings, W. Operating Systems. 4th Ed. Prentice Hall 2002.
Tanenbaum, A. Structured Computer Organisation. Prentice Hall 1999.
White, R. How Computers Work. 6th Ed. Que 2001.

Intel publications (all downloadable from www.intel.com.):
8-bit HMOS Microprocessor. Intel 1990.
8086 16-bit HMOS Microprocessor. Intel 1990.
80186/80188 High-Integration 16-bit Microprocessors. Intel 1994.
Intel Pentium 4 Processor in the 423-pin Package data sheet. Intel 2001.
Intel Pentium 4 Processor with 512KB L2 Cache data sheet. Intel 2001.
IA-32 Intel Architecture Software Developer's Manual. Volume 1: Basic Architecture. Intel 2001.
IA-32 Intel Architecture Software Developer's Manual. Volume 2: Instruction Set Reference. Intel 2001.

Websites

www.arstechnica.com
www.crucial.com/library
www.howstuffworks.com
www.intel.com
www.mips.com/content/Products/ProductInfo
www.rambus.com/technology

www.searchstorage.com
www.sparc.org/resources.htm
www.spechbench.org
www.transmeta.com/technology/specifications/index.html
www.unicode.org
www.whatis.com
www.xs4all.nl/~matrix

Index