Worked solutions to selected problems

CHAPTER 2

2.1 (a) $[1000110101]_2 = 1 \times 2^9 + 0 \times 2^8 + 0 \times 2^7 + 0 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$
$= 512 + 32 + 16 + 4 + 1$
$= [565]_{10}$

(b) $[0.1000110101]_2 = 1 \times 2^{-1} + 0 \times 2^{-2} + 0 \times 2^{-3} + 0 \times 2^{-4} + 1 \times 2^{-5} + 1 \times 2^{-6} + 0 \times 2^{-7} + 1 \times 2^{-8} + 0 \times 2^{-9} + 1 \times 2^{-10}$
$= 0.5 + 0.03125 + 0.015625 + 3.90625 \times 10^{-3} + 9.765625 \times 10^{-4}$
$= [0.551\,757\,812\,5]_{10}$

2.2 To convert a decimal integer into binary the algorithm is to repeatedly divide it by two. The sequence of remainders written last (the MSB) to first (the LSB) is the resulting binary number. For a fractional number the algorithm is to multiply the fraction repeatedly when the sequence of carries over the decimal point form the binary number.

(a) Integer

2) 21345 (1
2) 10672 (0
2) 5336 (0
2) 2668 (0
2) 1334 (0
2) 667 (1
2) 333 (1
2) 166 (0
2) 83 (1
2) 41 (1

(b) Fraction

0.673
× 2
1.346
× 2
0.692
× 2
1.384
× 2
0.768
× 2
1.536

2) 20 (0
2) 10 (0
2) 5 (1
2) 2 (1
1 (0

$= [0101001101100001]_2$

Breaking this into groups of four, starting from the right, gives:

0101 0011 0110 0001

or $[\ 5 \quad 3 \quad 6 \quad 1\ ]_{16}$

2
1.072
2
0.144
2
0.288
2
0.576

$= [0.1010110]_2$

Breaking into groups of four, working right from the binary point gives:

0.1010 1100

or $[0.AC]_{16}$

2.3 $[3FDA104E]_{16}$

Convert each digit from hexadecimal to binary:

0011 1111 1101 1010 0001 0000 0100 0111

Now divide into groups of three starting at the right-hand side:

00 111 111 110 110 100 001 000 001 000 111

Interpret each group as an octal number giving:

$[0\ 7\ 7\ 6\ 6\ 4\ 1\ 0\ 1\ 0\ 7]_8$

2.4 (a) $[1220]_4$; (b) $[0.11233]_4$.

2.5

	(a) Signed magnitude	(b) 2's complement	(c) Offset binary
+63	00111111	00111111	10111111
−125	11111101	10000011	00000011
−0.125	10010000	11100000	01000000
	↑ Binary point	↑ Binary point	↑ Binary point

2.6 The received number comprises the data to be checked together with remainder, both of which could have become erroneous during transmission. The check is to divide the complete number by the constant $(x^5 + x^3 + x + 1)$:

```
101010) 1010111111100 | 1000011
        101010
            111111
            101010
             101010
             101010
             000000
```

The remainder is zero, indicating that the data 10101111 is correct.

2.7 (a) $11011.1101 \times 2^2 = 0.110111101 \times 2^7$

The IEEE P754 single-precision format is a 32-bit grouping interpreted as:

$$(-1)^s 2^{e-127}(1 \cdot f)$$

where s is the sign digit, e is an 8-bit biased exponent and f is a 23-bit unsigned integer.

For the above number, $e - 127 = 7$ and $e = 134 = [10000111]_2$

The floating-point representation is then:

0 10000111 11011110100000000000000

Table S1 Problem 3.2(a).

(i)

Location (Hex)	*Instruction (Hex)*	*Mnemonic*	*Comment*
00	28 06	LDM 06	Fetch loop constant to modifier
01	0009	CLR	Clear accumulator
→02	58 FF/M	STA FF/M	Store accumulator in modified address
03	80 03	JMPM 03	Jump if modifier zero to own location, i.e. stop
04	0008	INCM	Add +1 to modifier
05	80 02	JMP 02	Jump to start of loop
06	FF61		Loop constant (2's complement of 9F)

(ii)

Location (Hex)	*Instruction (Hex)*	*Mnemonic*	*Comment*
→00	0009	CLR	Clear accumulator
01	58 10/I	STA 10/I	Store accumulator in indirect address
02	50 08	LDA 08	Fetch operand address
03	48 09	SUB 09	Subtract 00F, i.e. last location cleared
04	98	JMP0 04	Jump to self if accumulator zero, i.e. stop
05	40 0A	ADD 0A	Add 0100, i.e. 00FF + 1 (incrementing by 1)
06	58 08	STA 08	Replace modified address
07	80 00	JMP 00	Jump back to start of loop
08	60		Operand address
09	00FF		Loop constant
0A	0100		Increment constant

Table S2 Problem 3.2(b).

(i)

Location (Hex)	*Instruction (Hex)*	*Mnemonic*	*Comment*
00	28 08	LDM 08	Fetch loop constant to modifier
01	0009	CLR	Clear accumulator
→02	40 07	ADD 07	Add +1 to accumulator
03	58 FF/M	STA FF/M	Store accumulator in modified address
04	A0 04	JMPM 04	Jump to self if zero, i.e. stop
05	0008	INCM	Add +1 to modifier
06	80 02	JMP 02	Jump back to start of loop
07	1		Constant +1
08	FF61		Loop constant (2's complement of 9F)

(ii)

Location (Hex)	*Instruction (Hex)*	*Mnemonic*	*Comment*
→00	50 0A	LDA 0A	Fetch count to accumulator
01	40 0B	ADD 0B	Add +1
02	58 0A	STA 0A	Replace in memory
03	58 0C/I	STA 0C/I	Store count in indirect address
04	50 0C	LDA 0C	Fetch operand address
05	48 0D	SUB 0D	Test for end of loop
06	98 06	JMP0 06	Jump to self if zero
07	40 0E	ADD 0E	Increment operand address by +1
08	58 0C	STA 0C	Replace in memory
09	80 00	JMP 00	Jump back to start of loop
0A	( )		Accumulating number
0B	1		Constant +1
0C	60		Operand address
0D	00FF		Loop constant
0E	0100		Increment constant

(b) $1.01110101 \times 2^{-12} = 0.101110101 \times 2^{-11}$
and $e - 127 = 11$ giving $e = 138$ and the floating-point representation as:

0 10001011 10111010100000000000000

(c) $-0.110101011 \times 2^{6}$ gives

1 10000110 11010101100000000000000

(d) $-0.000001101 \times 2^{-1}$ gives

1 01111001 11010000000000000000000

Table S3 Problem 3.3.

Location (Hex)	*Instruction (Hex)*	*Mnemonic*	*Comment*
00	28 10	LDM 10	
→01	50 A0	LDA A0/M	
02	98 06	JMP0 06	
→03	A0 03	JMPM 03	Test for zero loop
04	0008	INCM	
05	80 01	JMP 01	
→06	0005	EXAM	
07	50 11	STA 11	
08	0005	EXAM	
09	50 11	LDA 11	
0A	40 12	ADD 12	Unpacking of address of zero location
0B	58 13/I	STA 13/I	and storing in table
0C	50 13	LDA 13	
0D	40 14	ADD 14	
0E	58 13	STA 13	
0F	80 03	JMP 03	
10	FFC0		Test loop constant, −0040
11	( )		Temporary store
12	A0		
13	A1		Address table constants
14	1		

CHAPTER 3

3.2 Machine-code programs for this problem are given in Tables S1 and S2.

3.3 A typical program for this problem is shown in Table S3. Note that in this program we are handicapped by having only one modifier register and not being able to increment a memory location directly. The 'test for zero' loop employs the modifier register to increase the speed of operation of the program and the 'unpacking' of the address is performed using indirect addressing. An alternative approach is to use the indirect address function for the test loop, since the actual address could then be held in the indirectly addressed location.

3.4 In this program, given in Table S4, we can take advantage of the fact that the same modifier constant can be used; therefore we can employ a modified instruction for both transfers.

3.7 This program is shown in Table S5. The basic logic is to extract the 4-bit BCD digits using the AND function and then to multiply by

Table S4 Problem 3.4.

Location (Hex)	*Instruction (Hex)*	*Mnemonic*	*Comment*
00	28 0A	LDM 0A	
→01	50 80/M	LDA 80/M	
02	58 0A	STA 0A	
03	50 F0/M	LDA F0/M	Exchange instructions
04	58 80/M	STA 80/M	
05	50 0A	LDA 0A	
06	58 F0/M	STA F0/M	
07	A0 07	JMPM 07	
08	0008	INCM	Loop test
09	80 01	JMP 01	
0A	FFE0		Loop count constant
0B	( )		Temporary store

Table S5 Problem 3.7.

Location (Hex)	*Instruction (Hex)*	*Mnemonic*	*Comment*
00	28 0E	LDM 0E	Fetch constant to modifier
01	50 0F	LDA 0F	
02	98 06	JMP0 06	
→03	50 0F	LDA 0F	Fetch BCD word to accumulator
04	13 04	SRL 4	Shift right four places
05	58 0F	STA 0F	Store shifted word
06	60 11	AND 11	Extract BCD digit
07	40 10	ADD 10	Add contents of temporary store
08	A0 08	JMPM 08	Stop with binary word in accumulator
09	0008	INCM	
0A	A8 12	MUL 12	Multiply by binary 10, result in X-reg
0B	0006	EXAX	Exchange accumulator with X-register
0C	58 10	STA 10	
0D	80 03	JMP 03	
0E	FFFD		Count constant −3
0F	( )		BCD word
10	0000		Temporary storage
11	000F		AND constant binary 15
12	000A		Binary 10

binary 10; the result is stored away and added to the next BCD digit, which is multiplied by binary 10, and so on.

Note that multiplying two integers $\times 2^{-15}$ results in the product being placed in the least significant end of the X-register (the sign digit of the X-register is not normally used).

Table S6 Problem 3.8.

Location (Hex)	*Instruction (Hex)*	*Mnemonic*	*Comment*
00	50 04	LDA 04	Fetch a to accumulator
01	B8 05	XOR 05	Form a ⊕ b
02	B8 06	XOR 06	Form a ⊕ b ⊕ c
03	0000	STOP	
04	( )	a	
05	( )	b	
06	( )	c	

Table S7 Problem 3.10.

Location (Hex)	*Instruction (Hex)*	*Mnemonic*
00	28 0A	LDM 0A
→01	50 0C/IM	LDA 0C/IM
02	58 0B	STA 0B
03	50 0D/IM	LDA 0D/IM
04	58 0C/IM	STA 0C/IM
05	50 0B	LDA 0B
06	58 0D/IM	STA 0D/IM
07	A0 07	JMPM 07
08	0008	INCM
09	80 01	JMP 01
0A	FFED	
0B	( )	
0C	0280	
0D	0180	

3.8 This is a trivial problem if the original expression is manipulated correctly; for example,

$$S = c(\bar{a}\bar{b} + ab) + \bar{c}(\bar{a}b + a\bar{b})$$
$$S = c\bar{z} + \bar{c}z, \text{ where } z = \bar{a}b + a\bar{b} = a \oplus b$$
$$S = c \oplus z = c \oplus a \oplus b$$

A suitable program is shown in Table S6.

3.10 The main difference with this program, shown in Table S7, compared to that of problem 3.4 is that we need to use indirect addressing because the available address bits have been exceeded: that is, 0160 → 0180 (sector 1) and 02D0 → 02F0 (sector 2). Since we

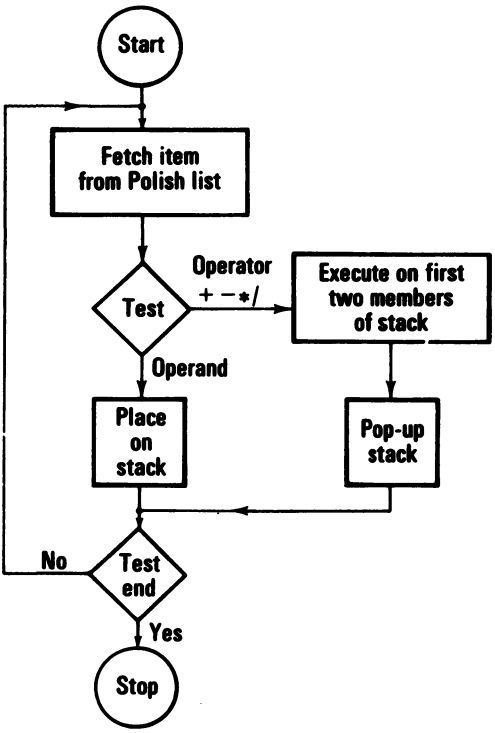
Table S8 Problem 3.11.

Label	*Instruction*		*Comment*
	ORG	0000	Locates program at 0000
C1	EQU	n	
C2	EQU	−k	
C3	EQU	00FF	Constants
C4	EQU	0001	
C5	EQU	0000	
	LDM	C2	Load modifier with $-k$
	CLA		Clear accumulator
	ADD	C1	Add n to accumulator
	STA	MEMREF	Store n in MEMREF
START	LDA	MEMREF/I	Load contents of address n into accumulator
	AND	C3	AND with 00FF
	OUTA		Output to I/O register
	LDA	MEMREF/I	Load contents of address n into accumulator
	SRL	8	Shift right 8 places
	OUTA		Output to I/O register
	LDA	MEMREF	Load address n into accumulator
	ADD	C4	Increment accumulator by +1
	STA	MEMREF	Store in MEMREF
	INCM		Increment modifier register by +1
LOOP	JMPM	LOOP	If modifier register equals zero loop, i.e. stop
	JMP	START	Jump to START
	END		Pseudo-operation to signify end of code
MEMREF	RMB	2	Reserve two memory bytes, i.e. 1-word

are working in sector 0 we may specify directly within the sector, but locations outside the sector must be specified indirectly. This means including two extra 'pointer' locations; the addresses in these locations may be modified in the usual way by setting the modifier bit in the instruction to 1. Note that when the modifier and the indirect address bits are both set to 1, it is the content of the indirect address that is modified.

3.11 The assembler code program is shown in Table S8. Note that the **EQU** pseudo-operation is liberally used to set up constants. This is good practice since it permits easy changes to these to be made at a later date without having to modify any of the code below. For example, in this case the values of n and k are not defined absolutely here. **MEMREF** is used as a temporary store for the current address of the memory location to be output to the I/O register and all such space is allocated after the **END** pseudo operation. The **AND** function is used as a masking function to enable only the least significant byte to be meaningful.

(a)



(b)

STACK	A	End
	B	STACK
	A * B	End
	C	STACK+2
	D	STACK+3
	C−D	STACK+2
	E	STACK+5
	E/C−D	STACK+2
STACK+8	E/(C − D) + (A * B)	End

Figure S1 Problem 3.12.

Table S9 Problem 3.12.

Label	*Instruction*	*Comment*
	ORG 0000	Locates program at 0000
C1	EQU AE	Address of first word of Polish list
C2	EQU 0D	Terminating character
C3	EQU 0015	Operator constants
C4	EQU 0002	
C5	EQU 0001	
START	CLR	Clear modifier
	EXAM	
J1	LDA C1/M	Fetch first word from Polish list
	SUB C2	Test for operator/operand/END operator
	JMPN TABLE_1	
SELF	JMP0 SELF	Stop, end of Polish list
	ADD C2	Reconstitute operand
	STA TEMP	Store temporarily
	LDA ST+2	Push down stack
	STA ST+3	
	LDA ST+1	
	STA ST+2	
	LDA ST	
	STA ST+1	
	LDA TEMP	Place operand on stack
	STA ST	
B1	INCM	Increment modifier register
	JMP J1	Jump table to appropriate arithmetic conditions
TABLE_1	ADD C3	
	JMP0 J2	
	ADD C4	
	JMP0 J3	
	ADD C4	
	JMP0 J4	
	ADD C5	
	JMP0 J5	
J2	LDA ST+1	Divide
	DIV ST	
B2	STA ST	
	JMP J6	
J3	LDA ST+1	Subtract
	SUB ST	
	JMP B2	
J4	LDA ST+1	Add
	ADD ST	
	JMP B2	
J5	LDA ST+1	Multiply
	MUL ST	
	EXAX	
	JMP B2	
J6	LDA ST+2	Pop-up stack
	STA ST+1	
	LDA ST+3	
	STA ST+2	
	JMP B1	
	END	
STACK	RMB 8	Operand stack of depth four
TEMP	RMB 2	Temporary store

Table S10 Problem 3.12.

Label	*Instruction*	*Comment*
	ORG 0000	Locates program at 0000
SPACE	EQU F0	Address of start of stack
C1	EQU AE	Address of first word of Polish list
C2	EQU 0001	Constant +1
	CLR EXAM	Clear modifier
	LDA C1/M STA SPACE/M	Fetch first word from Polish list Store on stack and push down
	INCM	
	LDA C1/M	
	STA SPACE/M	
	INCM	
	EXAM SUB C2 EXAM LDA SPACE/M	Read off top of stack
	EXAM SUB C2 EXAM LDA SPACE/M EXAM SUB C2 EXAM ADD SPACE/M STA SPACE/M INCM	Add and push up stack (note this gives B + A since B at top of list)

3.12 The Polish notation (see Chapter 3, reference [5]) is well suited to stack operation, and places the operators after, rather than between, the operands. We assume initially that the Polish list consists of fixed maximum-length expressions, located from AE Hex upwards and terminated with the marker @. In this case it would be very simple to construct a push-down/pop-up store, or stack, of finite depth to perform the evaluation. Figure S1(a) shows a typical flowchart for such a routine, and a simple assembler code program is given in Table S9. In this program the stack has been programmed by physically moving the contents of the memory locations; this is not the best way since, in general, it is always better to manipulate addresses rather than words.

An alternative approach is shown in Table S10. In this case the modifier register acts as a **pointer** (the same results could be achieved by indirect addressing) pointing to the address of the top member of the stack. Note that only the 'top' of the stack is moved

during the stacking and unstacking processes. In this case it is also necessary to make special provisions to ensure that the arithmetic operations are performed in the correct order (A * B) since the normal application of this technique gives B * A if B is at the top of the stack. It is still necessary with this type of stack to know its precise size beforehand; a perfectly general stack for handling Polish lists of any size may be constructed. In this case the operations of push-down and pop-up become simply a matter of altering list pointers, as shown in Figure S1(b).

If at the start of the process a block of continuous words is allocated in memory for the list cells, sooner or later these will become exhausted. Moreover, the memory will contain deleted cells as a result of amending the lists. In order to ensure maximum utilization an **available space list** is created (using a stack principle) which is employed to note the addresses of all free cells. When a new cell is required it is obtained from the available space list and, conversely, when no longer required it is returned to the list. Note that the top of the list is continually changing, and is given by the pointer obtained from the available space list, in this particular case taken in sequence from **STACK** to **STACK+8**.

CHAPTER 4

4.1 A possible microprogram for the LINK instruction is given in Table S11. Alternative techniques could include placing the link address in special modifier registers, or using memory locations with prewired addresses. Neither of these techniques, however, allows 'nesting' of the subroutines, and a better approach would be to use a push-down/push-up stack register with modification facilities provided on the top register.

Table S11 Problem 4.1.

Next address	*Micro-operation*	*Comment*
01 … 06		;fetch instruction
21	E_0,M_i	;content of control register to memory register
22	C_{ad},R/W	;instruction address bits to c-bus
23	WAIT	;wait for end of memory write cycle
24	C_{oa},X_i,+1lsb	;increment instruction address by +1
25	AU_o,E_i,END	;transfer incremented address to control register

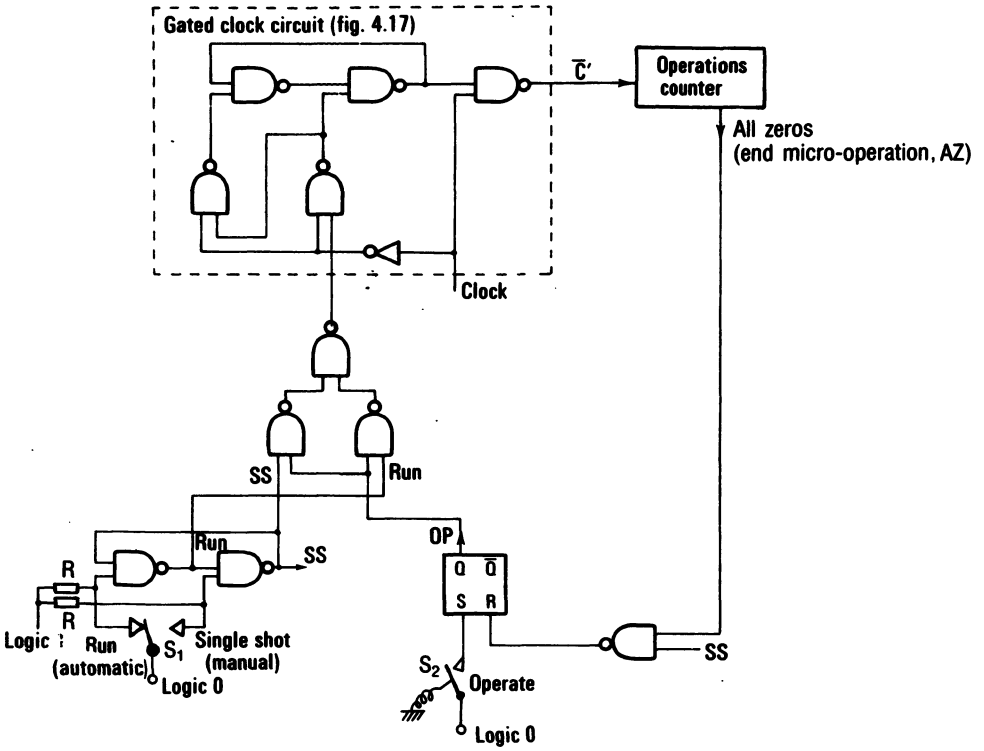


Figure S2 Problem 4.4.

4.4 A typical circuit to perform this function is shown in Figure S2. To operate the computer in the automatic mode switch S_1 is put to 'run' and the operate key depressed; to stop the machine S_1 is changed over to the single-shot position. The gated clock circuit ensures that, though there may be a momentary pause in the clock pulses, only complete pulses reach the counter. Note that all the switches must be buffered (by a flip-flop) to eliminate contact bounce.

When working in the manual mode it is necessary to depress the operate key (which is spring-loaded) in order to initiate the execution of an instruction. The operate key sets the OP flip-flop, which is reset in turn by the micro-operation END. Note that when the operate key is depressed in the single-shot mode the flip-flop has both the S and the R inputs equal to 1: that is, the 'indeterminate' condition. However, the effect of this is to put both the Q and $\bar{Q}$ outputs to

Table S12 Problem 4.5.

Next address	*Micro-operation*	*Comment*
27	C_{ad},R/W	;Fetch multiplier
28	WAIT	
29	M_o,MD_i	;Multiplier to modifier register
30	A_c,M_i	;Use M-register to hold Multiplicand
31	A_c,$K_4\bar{K}_3\bar{K}_2\bar{K}_1\bar{K}_0$	;Clear accumulator, set K-counter to 16
32	BRC($MD^0 = 0$) 35	;Test multiplier value
33	A_o,X_i,M_{au}	;Add multiplicand to partial product
34	AU_o,A_i	
35	A_o,MD_i,R	;Shift accumulator and modifier one place right
36	K_d,BRC(K = 0) 32	;Decrement K-counter, branch if non-zero
37	MD_o,X_i,END	;Transfer least significant part of product to the X-register

logic 1 (using NAND logic) which will allow the machine to operate, thereby changing the counter and causing AZ to go to 0; this assumes of course that the machine responds before the key is released.

4.5 The register set of our primitive computer does not really contain enough registers to be able to undertake multiplication and division efficiently. The modifier register can be used to store the multiplier and has to have associated with it a bit-test facility for MD^0 (bit position 0). The multiplication algorithm used is simply to test the least significant bit of the multiplier and add the multiplicand to the partial product, held in the accumulator, or not, depending whether it is a 1 or 0 respectively, and then to shift the partial product right one place. In our case the LSB of the accumulator is assumed to move into the MSB of the modifier: i.e. the modifier is an extension of the accumulator and both shift together. The memory register is used to hold the multiplicand. A microprogram is shown in Table S12.

CHAPTER 5

5.1 (a) Arithmetic using sign and magnitude numbers is very simple for multiplication and division (it is only necessary to compare signs: if different, the result is negative, if the same, the result is positive). However, addition and subtraction becomes a complicated process. The magnitude of their operands themselves must be compared as well as the signs:

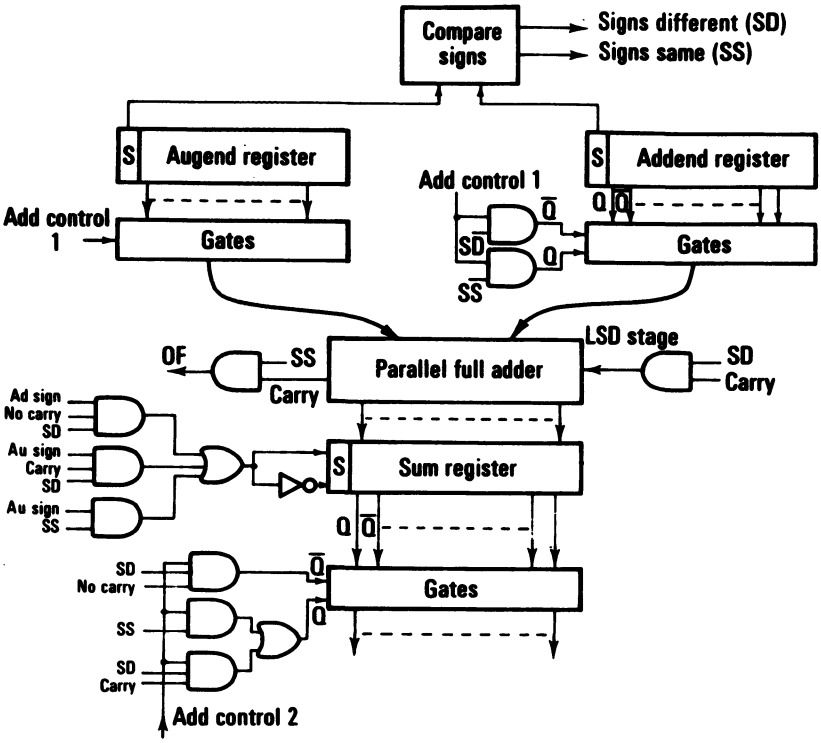


Figure S3 Problem 5.1.

1. Signs the same: add magnitudes, duplicate sign in result.
2. Signs different: (i) *Augend larger*, add 1's complement of addend to augend (with end around carry) result given sign of augend. (ii) *Addend larger* (or equal), add 1's complement of addend to augend, complement result, result given sign of addend.

To implement the algorithm in hardware it is necessary to be able to compare the augend and addend for the >, <, or equal relationships using, for example, the circuit shown in Figure 5.36. A conventional parallel full adder can be used, with the overflow (that is, carry from MS stage) being fed back to the carry input of the LS stage (to give an end-around carry). The complemented result can be obtained by gating out the $\bar{Q}$ outputs of the sum register in place of the usual Q outputs. Similarly the $\bar{Q}$ register outputs can be gated to the full-adder circuits to obtain the complement of the addend. However, it may be shown that an end-around carry only occurs if the augend is larger than the addend; consequently, the final carry output may be used as a control signal thus allowing the comparator

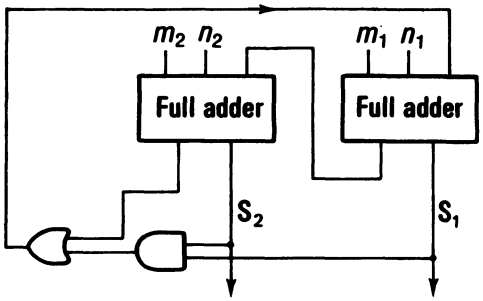


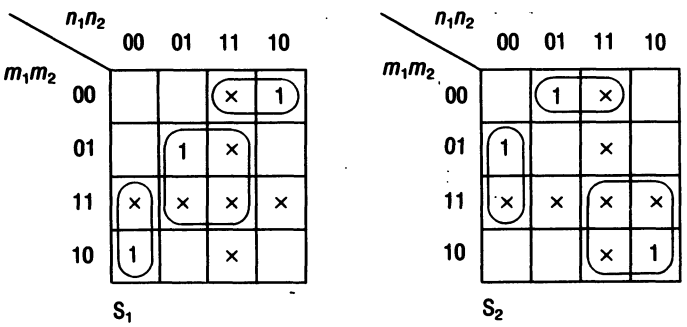
Figure S4 Problem 5.3.

Table S13 Problem 5.3.

(a) Mod 3 addition

$.m_1m_2$ +	0 00	1 01	2 10	n_1n_2
0 00	00	01	10	
1 01	01	10	00	
2 10	10	00	01	

(b) K-Maps



to be dispensed with. If this technique is used special precautions must be taken with the overflow resulting from the addition of two like sign numbers. Subtraction may be performed by changing the sign of the subtrahend. A suggested block diagram for a sign and magnitude arithmetic unit is shown in Figure S3.

(b) The 1's complemented arithmetic unit is quite straightforward, the arithmetic operations being performed in much the same way as for the normal 2's complemented system, with the exception that the end-around carry scheme must be incorporated.

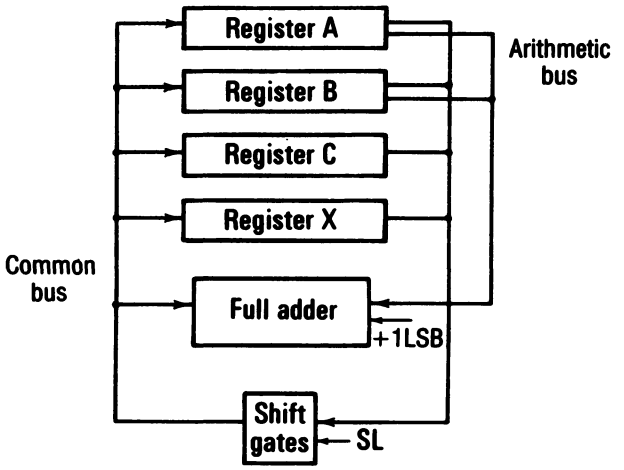


Figure S5 Problem 5.4.

Table S14 Problem 5.4.

Next address	*Micro-operation*	*Comment*
27	RA_o,X_i,R,RB_{au}	;RA to X-reg. shifted one place right, output of RB taken to input of arithmetic unit
28	AU_o,RA_i	;Output of AU to RA
29	$RA_o,X_i,+1lsb$	;Complemented output RA to X-register
30	AU_o,RC_i,END	;Output of AU to RC

5.3 The design tables for this problem, treated as a switching network, are shown in Table S13; note that only mod 3 numbers (0, 1, 2) are allowed. The equations, shown below, may be implemented in the usual way:

$$S_1 = \bar{m}_1\bar{m}_2n_1 + m_2n_2 + m_1\bar{n}_1\bar{n}_2$$
$$S_2 = \bar{m}_1\bar{m}_2n_2 + m_1n_1 + m_2\bar{n}_1\bar{n}_2$$

An alternative design approach is to use binary full adders and to correct the resulting binary sum by adding 1 into the least significant stage. This may be done by detecting the overflow (carry-out) and $S_1S_2 = 1$ conditions; a logic diagram is shown in Figure S4.

5.4 We assume that the registers (RA, RB and RC) are connected to a common parallel highway system, complete with shifting gates, similar to that discussed in the main text; a typical logic diagram is shown in Figure S5, and a suitable microprogram in Table S14. Note

that the same symbolism as used in the text is employed for the microinstructions: that is, RA_0 puts the output of register RA to the common highway; $\overline{RA}_0$ puts the complemented outputs to the highway, etc.

5.6 (a) The first part of this problem is fairly straightforward, and a suggested flowchart is shown in Figure S6. The main difficulties are caused through a lack of double-length mode operation (this becomes even more apparent when the microprogram is considered later). Consequently, it is necessary to program the double-length shift operation required for multiplication. This is done by extracting the least significant digit of the most significant half of the partial product, before shifting, and inserting it in the most significant digit position of the least significant half of the partial product, *after* shifting. Note also the formation of a code word (TAG) from the inspection of the sign digits of the multiplicand and multiplier; the codeword is used in the program to determine the appropriate correction operations.

Table S15 Problem 5.6.

Next address	*Micro-operation*	*Comment*
21	C_{ad},R/W	
22	A_o,X_i	;transfer M'plicand to X-reg
23	A_c,$K_4\bar{K}_3\bar{K}_2\bar{K}_1K_0$	;clear Acc. set K-counter to $K_4\bar{K}_3\bar{K}_2\bar{K}_1K_0$
24	WAIT	
25	M_o,MD_i,MD_o^{15},G_i	;transfer M'plier to Modifier reg., set flag G
26	BRC($MD^0 = 0$) 29	
27	A_o,M_i,M_{au}	;add M'plicand to most significant half of
28	AU_0,A_i,A_o^0,F_i	partial product, Set flag F
29	A_o,M_i,R	;shift P.P right
30	M_o,A_i	
31	BRC($X^{15} = 0$) 33	
32	A_o^{15},A_i^{14}	;arithmetic shift
33	MD_o,M_i,R	;shift multiplier right
34	M_o,MD_i	
35	F_o,MD_i^{15}	;set MSB of M'plier to flag F
36	K,K_d,BRC($K \neq 0$) 26	;decrement K-counter and Jump back
37	G,BRC($G = 0$) 42	
38	X_o,M_i,$\bar{M}_{au}$,+1lsb	
39	AU_o,X_i	;flag G set add 2's complement of
40	A_o,M_i,M_{au}	M'plicand to MSPP
41	AU_o,A_i	
42	MD_o,X_i,END	;transfer LSPP to X-register

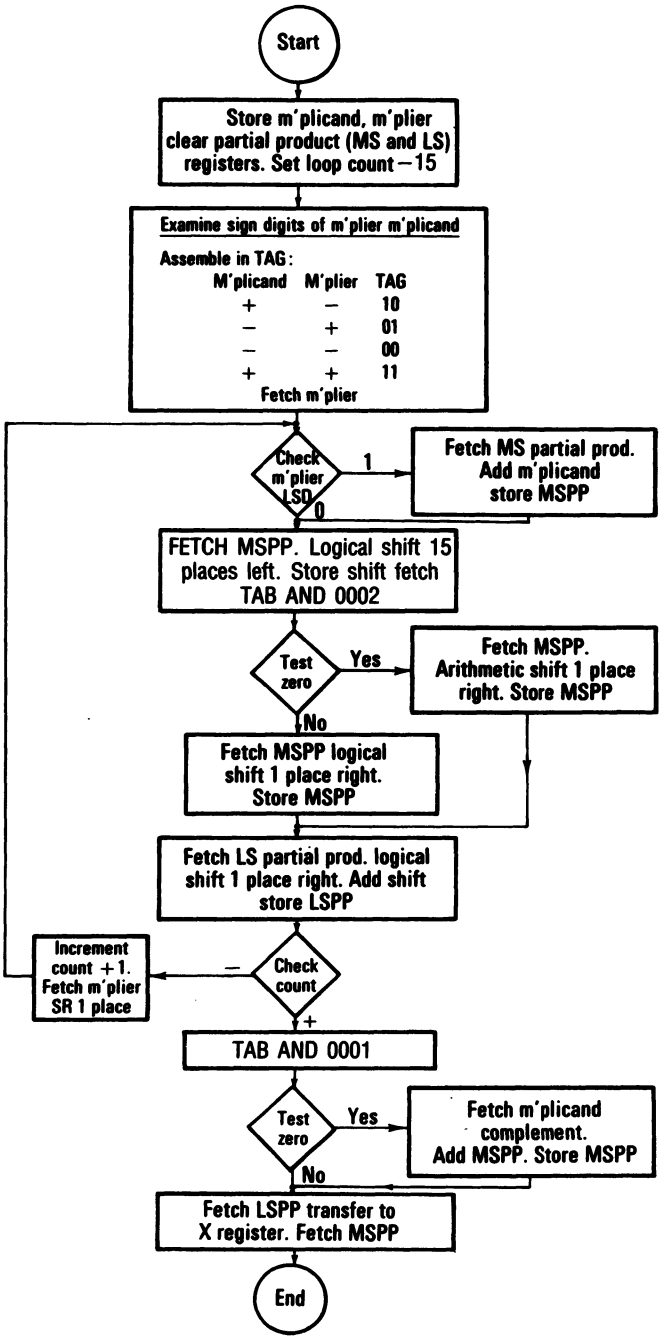


Figure S6 Problem 5.6(b).

(b) To implement the algorithm in hardware presents a number of problems. Strictly speaking the machine structure is too primitive (due mainly to an insufficient number of registers) to incorporate multiplication/division algorithms effectively. In this example it is necessary to use the modifier register as part of the multiplication logic; this limits the operation of the machine since the programmer must store away the contents of the modifier register (which must be retrieved later) before executing a multiplication instruction. This function could also be included as part of the microprogram, by initially writing away the contents of the modifier (at micro-operation level) into a predetermined location using a wired address technique. A suitable microprogram for the multiplication algorithm is shown in Table S15, and follows a similar logic to the flowchart given in Figure S6.

The initial micro-operations are concerned with loading the registers; the multiplier is held in the modifier register, and the X-register contains the multiplicand. Partial products are accumulated in the accumulator (most significant part only) which is initially cleared to zero; register M is left free and is used as a temporary buffer store. The least significant part of the partial product is inserted bit by bit into the most significant end of the modifier register during the double-length shift operations. Note that it is necessary to invent some new micro-orders concerned primarily with bit manipulations. These are represented by $A_i^n A_o^j$ which refers to input or output from a particular bit stage (n, j) of a register: for example, $A_o^{14} A_i^{15}$ sets the bit contained in bit 14 into b15 (MSB) of register A; this operation is required for the arithmetic right shift function. In addition it is necessary to be able to set **flags** or flip-flops to store certain transitory conditions: for example the sign digit of the multiplier (in the example flag G is used for this purpose). Flag F is used to store the least significant digit of the partial product; this is required during the right shifting of the multiplier. The most significant digit of the multiplicand (held in the X-register) is treated as a condition (X_{MSD}) in the usual way.

5.8 The control logic for the multiplier follows conventional lines and is based on the microprogram principle. For example, the decoded output of the multiplier compare logic would be used in conjunction with the microinstruction sequence to generate the necessary switching waveforms. The control unit must perform the following operations.

1. Load multiplicand and multiplier registers. Clear input register. Set cycle counter to n (number of bits).

2. Examine decoder outputs from multiplier compare logic:

   $\bar{M}_2M_3$: generate normal and gate signals
   M_2: generate left shift and gate signals

   (At this stage the contents of the input register would be added to the contents of the multiplicand register.)
3. Transfer contents of partial product register to input register.
4. Examine decoder outputs:

   if $M_2M_3 \neq 1$ jump to 6
   $M_2M_3 = 1$ generate normal and gate signals

5. Transfer partial product register to input register.
6. Generate: multiplicand shift two places left, multiplier shift two places right.
7. Check end of multiplication instruction

   $n = 0$ stop
   $n \neq 0$ decrement cycle counter, jump to 2

Note that two counters are required:

(a) An N-bit cycle counter (where $n = 2^N$) capable of being preset to n (generally by setting to the all 1s condition) and counting down for each input pulse. The count-down requirement is easily achieved by inverting the literals of the normal count-up flip-flop terms. For example, in the 3-bit binary counter of Figure 4.7(d) the input terms would become:

$$J_A = K_A = 1; \quad J_B = K_B = \bar{A}; \quad J_C = K_C = \bar{A}\bar{B}$$

(b) A 3-bit clocked control counter capable of being reset to allow jump operations.

The logical circuits required for register shifting, gating, etc., are shown in Figure S7.

5.10 Assume the normal computer structure described in the text.

(a) The mask or AND operation may be performed using special gating logic ($n \times 3$ input gates) connecting A and B to the common bus. It is necessary to form A.B transferring the result first to the X-register, and then back to the A-register.
(b) The inclusive OR operation may be performed in an analogous manner to that described in (a) above for the AND function. The exclusive OR operation may be performed very easily by generating a micro-operation which suppresses the carries in the full-adder unit (effectively performing modulo 2 addition only).
(c) To generate the 2's complement of register A it is first necessary

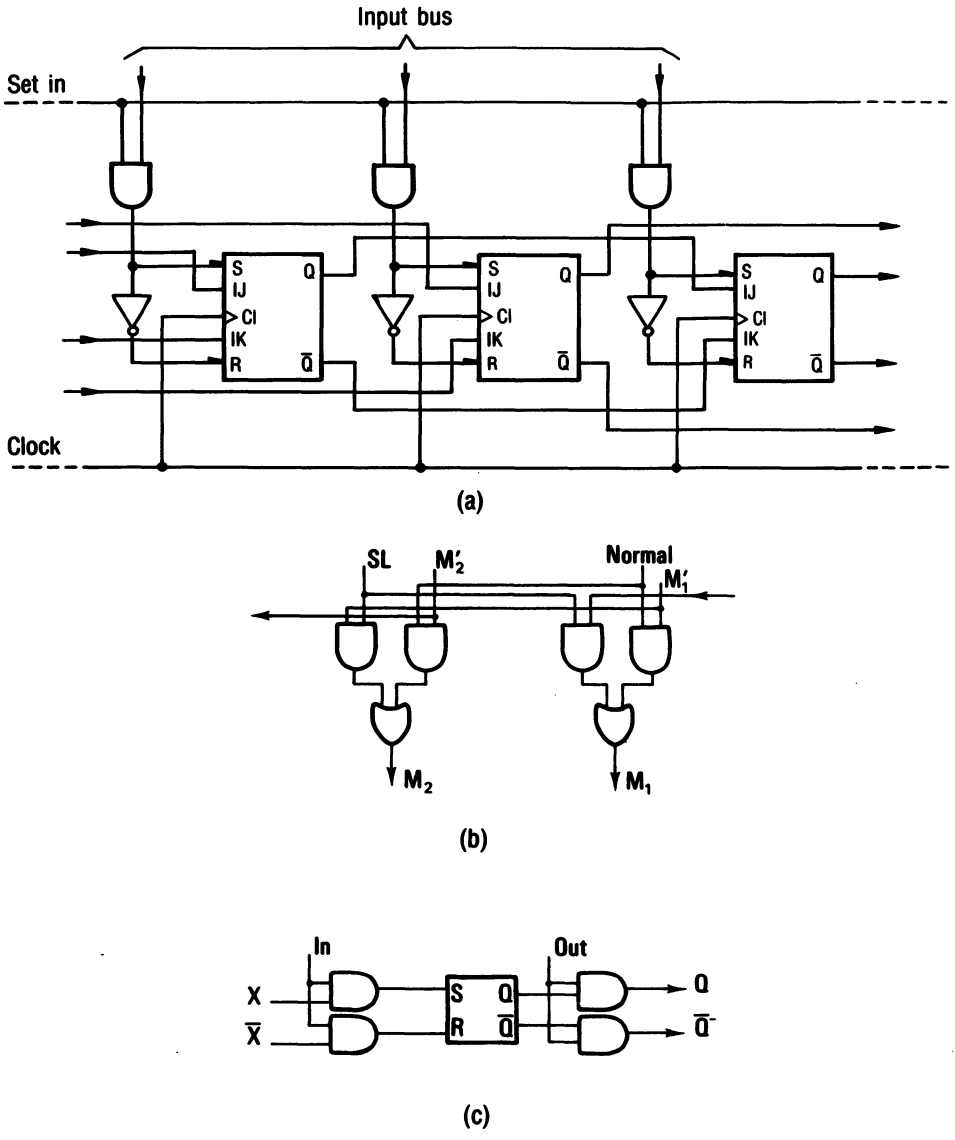


Figure S7 Problem 5.8: (a) register shifting two places right; (b) gates shifting one place left; (c) buffer register and gates.

to find the inverse. This may be done by implementing the micro-operation 'Output complementary outputs of A to common bus' ($\bar{A}_o$) which effectively interchanges the Q and $\bar{Q}$ outputs on the common bus. It is now a simple matter to form the 2's complement by using the micro-operations:

$$\bar{A}_o, X_i, + 1LSB$$
$$AU_o, A_i$$

which has the effect of forming the complement of register A in the X-register, adding +1, and then putting the result back into the A-register.

(d) The NAND function can be performed either by using direct NAND gating to the common bus or by using the AND function followed by the micro-operations:

$$\bar{A}_o X_i$$
$$X_o A_i$$

CHAPTER 6

6.2 Eight 4K × 8 RAM chips and two 16K × 8 ROM chips are required to satisfy the memory requirement of this microcomputer. The chip enables of the memory chips can be used to select them and since ten are required in total, four address lines (most significant) will need to be allocated for this purpose. In practice the most significant address line, A15, can be used to select either ROM or RAM. Address lines A⟨14:12⟩ can be decoded by a 3-to-8-line decoder enabled by A15 (low for assertion) to select one of the eight RAM chips, and A⟨15:14⟩ decoded directly to select one of the two ROM chips. A block diagram of this arrangement is shown in Figure S8(a). An alternative arrangement using standard cell macros would be to use one 3-to-8-line decoder to select the RAM chips and a 2-to-4-line decoder for the ROMs. The associated memory map is given in Figure S8(b).

6.3 The efficiency of recording is directly related to the space occupied on the disk by the bit-magnetization state. It is, then, inversely proportional to the number of flux changes per bit and so the solution requires the number of flux transitions per bit to be evaluated. These are best seen by sketching the waveforms for each of the recording modes, covering all of the possible bit-transitions for each case, i.e. $0 \rightarrow 1$, $1 \rightarrow 1$, $1 \rightarrow 0$, $0 \rightarrow 0$ and averaging the result. Figure S9 shows these and tabulates the efficiencies normalized to the NRZ mode.

6.4 In the worst case of randomly stored data the table-look-up program is required to search for any bit combination within the word, and each word must be examined in turn; a typical program is shown in Table S16. If we assume a single address machine with an average instruction time of 1 μs, the maximum time to search through the

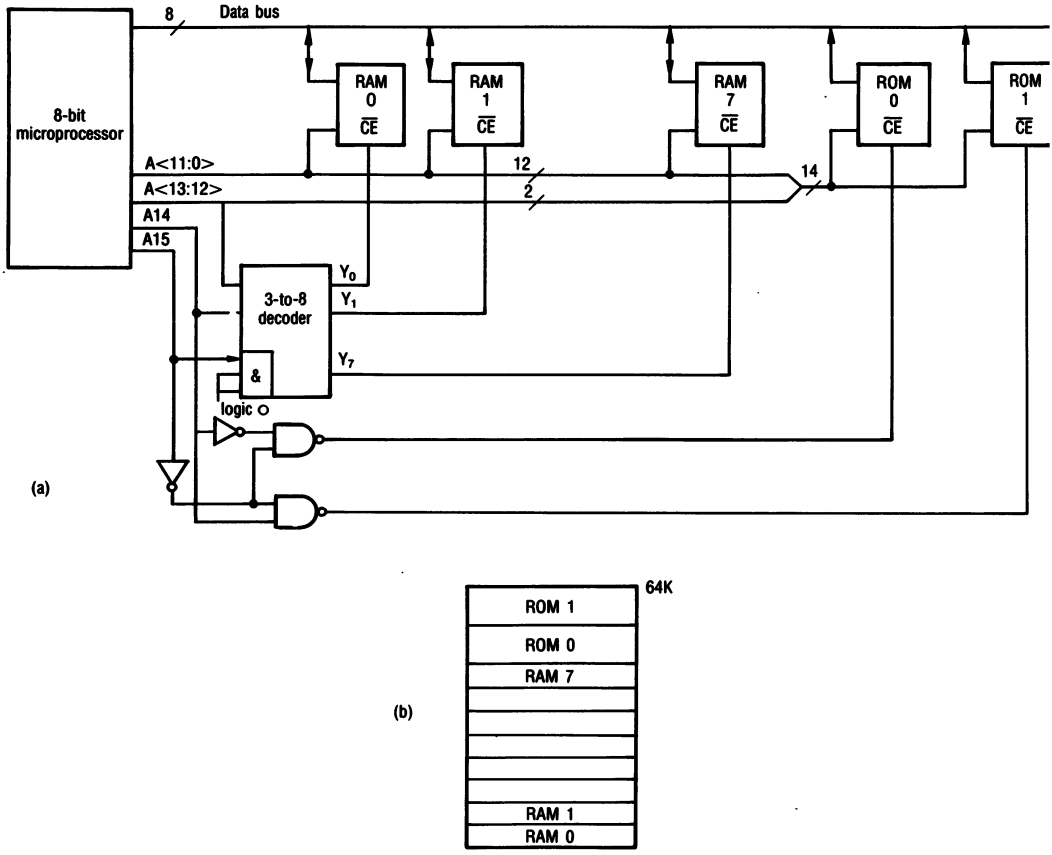


Figure S8 Problem 6.2: (a) microcomputer memory block diagram; (b) memory map.

table is $1 + (6 \times 65536 \times 1) \approx 400$ ms which on average gives a 200 ms search time. A special table-look-up instruction can be provided by combining the instructions

```
AND     CONS
SUB     VARS
JMP0
INCM
JMP     BACK
```

The instruction would need to be a specified address-type instruction, the address specifying the mask constant, with the tag word being stored immediately after it. Allocating a mnemonic **TLU** for this

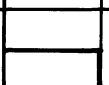
Recording mode	Bit pattern				Flux changes	Normalized efficiency
	0	1	1	0		
RZ					4	50%
NRZ					2	100%
NRZI					2	100%
PM					6	33%
FM					6	33%

Figure S9 Problem 6.3.

Table S16 Problem 6.4.

Label	*Op-code*	*Operand*	*Comment*
	LDM	(−65536)	;Fetch count constant to modifier
BACK	LDA	WORD/M	;Fetch word to accumulator
	AND	CONS	;AND with mask constant
	SUB	VARS	;Subtract required tag-bits
	JMP0	FND	;Test accumulator, if zero jump to FND
	INCM		;Increment modifier register
	JMP	BACK	
FND	LDA	WORD/M	
	STOP		
CONS	(	)	;Mask constant
VARS	(	)	;Tag-bits

instruction and assuming that it has the same average execution time, the table-look-up program now becomes

```
LDM     (−65536)
LDA     WORD/M
TLU     CONS
```

```
LDA     WORD/M
STOP
```

The number of memory locations required for the program has been reduced from 11 to 7 and the maximum table-look-up time to 1 + (1 × 65536 × 1) + 2 ≈ 66 ms. Thus the use of a special instruction has effectively reduced the search time by a factor of 6. However, there is a practical limit to the improvement of speed that can be achieved since it will depend on how well the microprogram can be optimized. Moreover in practice the search times could be improved by suitable structuring of the stored data and accessing the table by means of a generated address pointer. The most efficient technique for table look-up, however, is to use an associative memory (see Figure S10). The memory must be capable of matching to any part of the word: hence it must be possible to examine all bits. To operate the memory the desired pattern (tag bits) must first be set into the descriptor register and the match indicators of each word set to 1.

All words in the memory are searched simultaneously on a bit-by-bit basis. The first access (read cycle) is to bit 15 in all words, and the outputs are compared with bit 15 of the desired tag. The match indicators of those words in which bit 15 equals the tag are left set to 1, all mismatches causing the indicators to be set to 0. The final result is that only those words which contain bits identical with the specified descriptor have their match indicators set to 1. The address of the

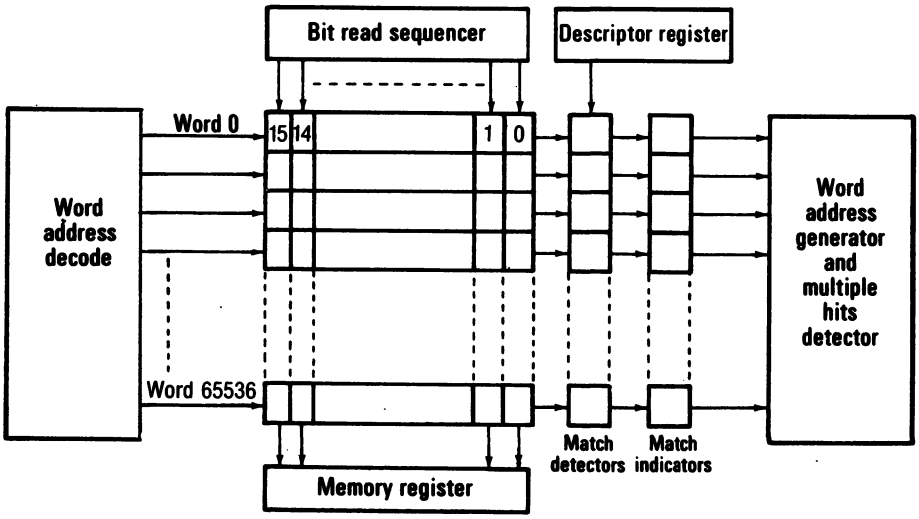


Figure S10 Problem 6.4.

word must now be obtained, and the required word read into the M-register in the usual way. Note that n read operations are required, where n is the number of bits in the word. If we assume a 500 ns access time the complete search takes 8 μs. A special instruction is required of the form 'Search for word specified in address, contents to be placed in the accumulator'. After the basic search operation is completed, the microprogram must access the associative memory and transfer the required word to the accumulator. The complete table-look-up routine then consists of one instruction with an execution time of the order of 10 μs.

6.5 (a) For direct mapping the main memory address is made up of the **tag** field and the **index** field, a total of 19 bits in this case (2^{19} = 512K). The index field is used to address the cache memory and requires 11 bits to do so, giving a tag field length of eight bits. The cache memory is partitioned into blocks so that the index field comprises a **block number** field together with a **word address** field to identify the required word in that block. The block size is 16 so that the word field is four bits. The total number of blocks available is, then, 128 ($=2^7$).

(b) The hit ratio is defined as:

$$\text{Hit ratio} = \text{number of hits/total number of accesses}$$

or in other words, for a hit ratio$|_r$ of 0.9, 90% of the total memory accesses are successfully read from the cache and 10% from the main memory, giving:

$$\begin{aligned}\text{Average access time} &= (0.1 \times 150 + 0.9 \times 30)\,\text{ns} \\ &= 42\,\text{ns}\end{aligned}$$

(c) Because a write-through method is used, every write will take a main memory cycle time. Thus the overall hit ratio for read and write cycles is reduced and is given by:

$$\begin{aligned}\text{Hit ratio}|_{r+w} &= (0.9 \times 0.75) \\ &= 0.675\end{aligned}$$

CHAPTER 7

7.1 The DMA mode of operation may be implemented using the form of microprogram shown in Table S17. It is assumed that multiple DMAs are allowed (on a hardwired priority basis) with each peripheral device setting up a particular bit of a DMA request register (DMAR). At the conclusion of each computer instruction the DMAR register

Table S17 Problem 7.1.

Next address	*Micro-operation*	*Comment*
21	**BRC(DMAR = 0) END**	;test for DMA request
22	**$DMAC_{ad}$, R/W**	;output of encoder to Address Bus
23	**WAIT**	
24	**M_oC_i**	
25	**IO_o,M_i**	;write away data word
26	**C_{ad},R/W**	
27	**C_o,X_i,+1lsb,+1wc**	;add +1 to word count and address
28	**C_i,AU_o**	
29	**BRC(A < 0) 25**	;check word count, if zero set DMAF
30	**DMAF,END**	

must be examined for zero contents (using the condition DMAR = 0) to determine if a DMA request has been made. A dedicated memory location (DML) holds the word count and address of the location into which the first word is to be transferred of the device requiring DMA service. The address of this control word is obtained by encoding the contents of the DMAR, the outputs of this encoder being gated directly to the address bus using a new micro-operation $DMAC_{ad}$. The peripheral equipment must be program-initiated, and the contents of the DML control word primed (with the number of words, $-n$, and the start address of the data block) before the DMA mode of operation takes place. Both parts of the control word must be incremented by 1 in the microprogram for each word transfer. When the word count goes to zero a flag is set (DMAF) indicating to the external device that the data transfers have been completed.

7.3 To allow for the fact that two keys may be pressed inadvertently, a two-character buffer store should suffice. It can also be assumed that a character will remain on the parallel output lines of the VDU interface long enough for data transfer to take place. It is also assumed that the characters must be parity-checked and then packed, two to a word, in the main computer memory. The input message must be preceded and terminated by suitable control signals, say @ and *. The erase symbol is used to erase the character immediately preceding it.

Characters are input on an interrupt basis, that is each time a key is pressed an interrupt is sent to the CPU. The interrupt procedure may be assumed to be a standard one as described in the text and includes the usual supervisory program (see Figure 3.9). The input program must, of course, be initiated from the main program.

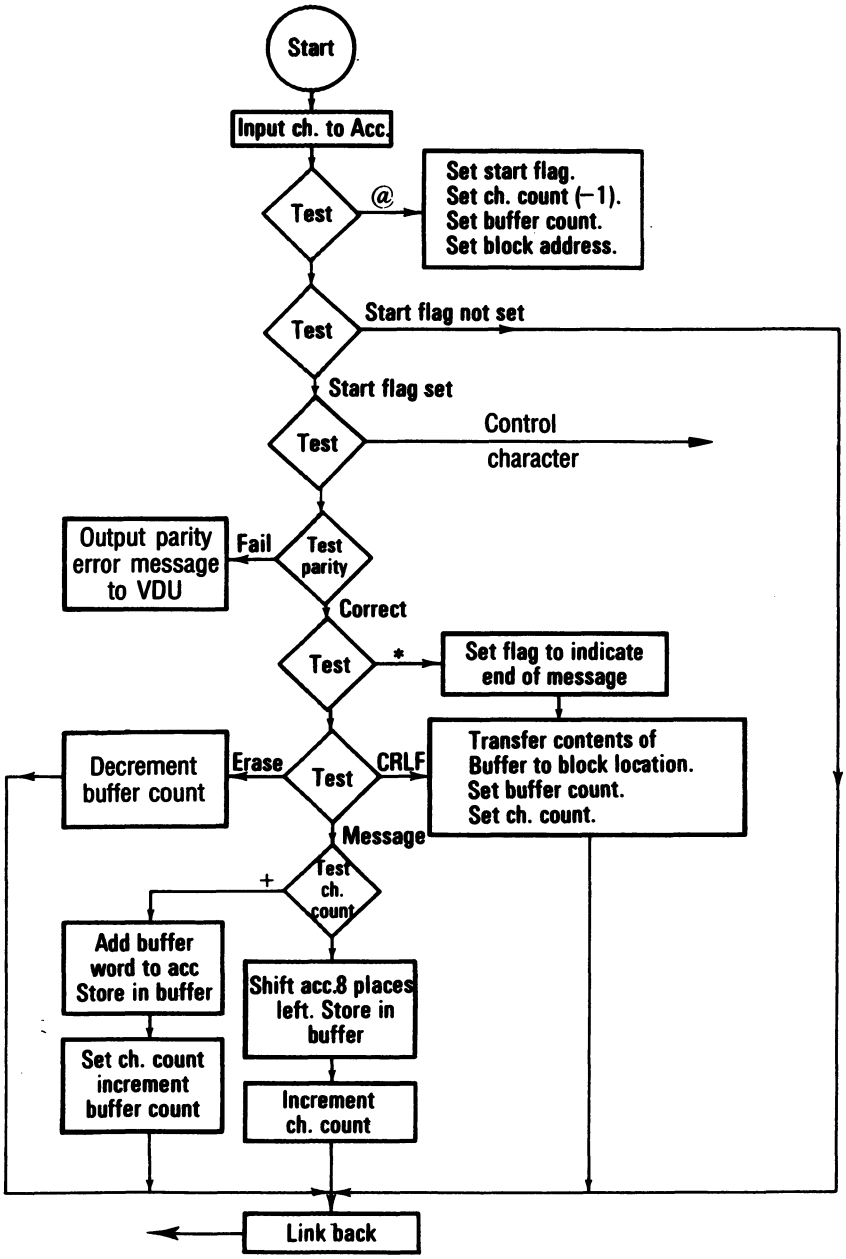


Figure S11 Problem 7.3.

A flowchart for the character input process is shown in Figure S11. Note that the character input is held in a buffer store until CRLF is received, when the entire line is stored away into the reserved memory locations (starting at the specified block address). If an erase symbol is received the buffer count is decremented by one allowing the character to be erased to be overwritten in the buffer store. The start of the message is indicated by the @ character which is used to set a start flag; this is to allow, for example, the message to always start on a new line. Control characters are tested for and if found cause a branch to a service routine to analyse and act accordingly.

The program presented in this example is considerably over-simplified, since multiple key depressions and the generation of specific control characters are ignored.

7.4 In order to transmit digital data over normal voice telephone links the binary output of the computer is modulated before transmission. At the receiving end of the line the signal must be correspondingly demodulated back into binary form; this is normally done using a modulator–demodulator unit known as a **modem**. Various methods of modulation are possible and data transmission can be either synchronous or asynchronous. Chapter 9 should be referred to for further details.

Asynchronous working is best for this problem and the 3.3 ms bit-rate allows each message bit to be registered into a single flip-flop stage and input a bit at a time on an interrupt basis. Using this technique it is possible to have up to 16 separate channels working into a single word buffer. Though this technique requires the minimum of external equipment, the software load (assembly and checking of the message) is considerable. Alternatively, the message bits may be clocked serially into a shift register and then transferred as a parallel word directly into the CPU. Thus if each character message is 12 bits long (including start/stop and parity checks) one complete word can be assembled every 39.6 ms. An interrupt signal to the CPU can be initiated by the stop signal with the data transfer occurring at the beginning of the next message start signal; alternatively, a timing gap or dummy message could be allowed between each actual message.

One of the major decisions that has to be made by the computer systems engineer is the allocation of tasks between hardware and software. In this case we are primarily concerned with the assembly and checking functions. The final decision depends on cost, speed, and availability of equipment. For example, one solution would be to use a computer dedicated to the assembly and decoding (including channel multiplexing) of all incoming messages. Alternatively, for

Table S18 Problem 7.4.

Next address	*Micro-operation*	*Comment*
21	M_c,A_o,X_i	;clear M-register, transfer accumulator to X-register
22	M_6,M_4,M_2,M_0	;set vector into M-register
23	I_m,A_i	;AND M-register with accumulator
24	BRC(PC=0) 26	;even parity check on 8 LSBs of accumulator
25	K_1	
26	M_c,X_o,A_i	
27	M_6,M_4,M_2,M_1	;parity fails, set 2^0 in K-register
28	I_m,A_i	
29	BRC(PC=0) 31	
30	K_2	
31	M_c,X_o,A_i	
32	M_6,M_5,M_4,M_3	;parity fails, set 2^1 in K-register
33	I_m,A_i	
34	BRC(PC=0) 36	
35	K_3	;parity fails, set 2^2 in K-register
36	M_c,X_o,A_i	
37	A_o,X_i,R	;right shift to lose check bit
38	X_o,A_i,R	;lose second check bit
39	A_o,MD_i,R	;move LSB to Modifier
40	A_o,X_i,R	;lose third check bit
41	X_o,A_i	
42	MD_o,A_i,L	;restore LSB to form 4-bit message word
43	K_{ou},MD_i,END	;error bits in modifier

common protocols off-the-shelf interface devices could be used and non-standard cases could be implemented in an ASIC.

Table S18 shows a microprogrammed routine to decode a 7-bit Hamming code held in the least significant end of the accumulator. Note the need to incorporate special micro-operations and logic to perform a parity check on the contents of the accumulator and to set individual bits of the M-register to 1. Moreover, provision must also be made to output the contents of the K-register to the common bus and to set up a parity condition.

CHAPTER 9

9.2 (a) **EtherNet**: The format has 26 bytes of control information, operates at 10 Mbps and waits for an acknowledgement of the receipt of data before the next transmission. If we assume the propagation delay between station A and station B to be ts, then the interval between transmissions of new data is $2t$, provided the data is always

received correctly. The maximum message length is 1500 bytes, so that only two packets are required in this example. For packet 1 (60 bytes of data) the transmission time is

$$(86 \times 8)/10 \times 10^6 = 69\,\mu s$$

For packet 2 (1500 bytes) the transmission time is

$$(1526 \times 8)/10 \times 10^6 = 1221\,\mu s$$

Total transmission time is then $(69 + 1221 + 4t) = (1290 + 4t)\,\mu s$.
(b) **Cambridge ring**: Here the frame size is 38 bits of which 22 are control and 16 data. The ring normally operates at 10 Mbps and the packet has to circulate back to the source to indicate correct data transmission. Let this time be $2t$ so as to allow a fair comparison with the other network types. For packet 1, 30 frames will have to be transmitted and for packet 2, 750, giving a total of 780 frames. The transmission time per frame is $38/10 \times 10^6 = 3.8\,\mu s$, and the total transmission time is $(780 \times 3.8 + 1560t) = (2964 + 1560t)\,\mu s$.
(c) **IBM Token ring**: The data field of this network is variable and so only two packets are required. Each packet has 21 control bytes associated with it and the ring typically operates at 4 Mbps. Ring-propagation delay is again assumed to be $2t$. For packet 1 the transmission time is

$$(81 \times 8)/4 \times 10^6 = 162\,\mu s$$

For packet 2 the transmission time is

$$(1521 \times 8)/4 \times 10^6 = 3042\,\mu s$$

The total time for transmission is, then, $(3204 + 4t)\,\mu s$.

9.3 The Cambridge ring operates on the insertion mode principle so that the buffer register of the transmitting station becomes part of the ring and lengthens it effectively by one station. At 10 Mbps a 50 μs propagation delay is equivalent to 500 bit-times, or 500/38 = 13 packets. The maximum number of stations that can be connected at any one time is, then, 14.

9.4 (a) The HDLC bit-string has to be checked from left to right and a zero inserted after every five consecutive 1s. The start and stop fields are not part of the given bit-stream and are excluded from this algorithm.
Writing the given bit-stream in binary we have

F	B	F	F	4	E	F	2	8	0	0
1111	1011	1111	1111	0100	1110	1111	0010	1000	0000	0000
	↑	↑	↑							

Inserting the bit-stuffing zeros,

111 1100 1111 1011 1110 0100 1110 1111 0010 1000 0000 0000

adding the start and stop fields,

01111110 111 1100 1111 1011 1110 0100
1110 1111 0010 1000 0000 0000 01111110

writing this as a Hex number, right justified gives the transmitted bit-stream, 3F7CFBE4EF28007E.

(b) In the given bit-stream, FBFF4EF2800:

FB is the address byte
FF is the control field
4EF is data and 2800 are the CRC bits

The CRC will have been correctly computed if there is zero remainder when the above bit-stream is divided modulo-2 (refer to worked solution of problem 2.6) by the constant 101011. The arithmetic is not shown here, but zero remainder does result.
(c) The control field bits are FF, or 11111111 and the two leading 1s identify the frame as an unnumbered frame.

CHAPTER 10

10.1 In the case of Figure 10.24 there are three input terminals and therefore eight different tests, designated t_0–t_7. The 16 faults f_1–f_{16} are all the possible s-a-0 and s-a-1 faults occurring on the eight connections C1–C8 and are referred to in the usual way as C1/0, C1/1, etc. Thus the fault matrix consists of 8 rows and 17 columns including the correct output; this is shown in Table S19. Examining the derived G_D matrix, shown in Table S20, we see that the essential tests are t_3 (testing f_5, that is, C3/0), t_1 (for f_8) and t_2 (for f_6 and f_{10}). These essential tests will cover all the faults except f_1 and f_{11}, which may be covered by including test t_7. A possible test set (in fact the minimal) is thus t_1, t_2, t_3, and t_7, which in terms of input/output values becomes:

001/1, 010/1, 011/0, 111/1

10.3 The reliability of the RAM blocks is given by:

$$[1 - (1 - r_2)(1 - r_2)] = 1 - (1 - r_2)^2$$

The reliability of the 2-out-of-3 disk system may be calculated in the simplified way using a truth-table format:

Table S19 Problem 10.1.

		f_0	f_1 *C1/0*	f_2 *C1/1*	f_3 *C2/0*	f_4 *C2/1*	f_5 *C3/0*	f_6 *C3/1*	f_7 *C4/0*	f_8 *C4/1*	f_9 *C5/0*	f_{10} *C5/1*	f_{11} *C6/0*	f_{12} *C6/1*	f_{13} *C7/0*	f_{14} *C7/1*	f_{15} *C8/0*	f_{16} *C8/1*
000	t_0	1	1	0	1	1	1	1	1	1	1	1	1	1	0	1	0	1
001	t_1	1	1	0	1	0	1	1	1	0	1	1	1	1	0	1	0	1
010	t_2	1	1	1	1	1	1	0	1	1	1	0	1	1	0	1	0	1
011	t_3	0	0	1	1	0	1	0	1	0	1	0	0	1	0	1	0	1
100	t_4	0	1	0	0	1	0	0	1	0	1	0	0	1	0	1	0	1
101	t_5	0	1	0	0	1	0	0	1	0	1	0	0	1	0	1	0	1
110	t_6	1	1	1	0	1	1	1	1	1	1	1	0	1	1	1	0	1
111	t_7	1	0	1	0	1	1	1	1	1	1	1	0	1	1	1	0	1

Table S20 Problem 10.1.

	(f_0f_1)	(f_0f_2)	(f_0f_3)	(f_0f_4)	(f_0f_5)	(f_0f_6)	(f_0f_7)	(f_0f_8)	(f_0f_9)	(f_0f_{10})	(f_0f_{11})	(f_0f_{12})	(f_0f_{13})	(f_0f_{14})	(f_0f_{15})	(f_0f_{16})
t_0		1											1		1	
t_1		1		1				1					1		1	
t_2						1				1			1		1	
t_3		1	1		1		1		1			1		1		1
t_4	1			1			1		1			1		1		1
t_5	1			1			1		1			1		1		1
t_6			1								1				1	
t_7	1		1								1				1	

D_1	D_2	D_3	R
0	1	1	$(1 - r_4)(r_4)(r_4) = r_4^2 - r_4^3$
1	0	1	$(r_4)(1 - r_4)(r_4) = r_4^2 - r_4^3$
1	1	0	$(r_4)(r_4)(1 - r_4) = r_4^2 - r_4^3$
1	1	1	$(r_4)(r_4)(r_4) = r_4^3$

The overall reliability is the sum of the individual reliabilities:

$$R_D = 3r_4^2 - 2r_4^3$$

The system reliability is given by the expression:

$$\begin{aligned} R_s &= [1 - (1 - r_2)^2]r_1r_2[r_4^2(3 - 2r_4)]r_5 \\ &= [1 - (1 - 0.92)^2]0.95 \times 0.85[(0.8)^2(3 - 2 \times 0.80)]0.90 \end{aligned}$$

Thus

$$R_s = 0.61$$

The mean time between failures may be calculated using the expression:

$$R_s = e^{-\lambda t} = e^{-\lambda 1000} = e^{-1000/M}$$

where λ = failure rate, M = mean time between failures, and t is the mission time.

Expressing $e^{-\lambda 1000}$ as a series expansion we have:

$$R_s = 1 - \lambda 1000 + \frac{(\lambda 1000)^2}{2} + \cdots$$

The failure rate λ is very small so the expression approximates to:

$$R_s = 1 - \lambda 1000$$

Now

$$M = \frac{1000}{1 - R_5} = 1/\lambda$$

Thus

$$M = \frac{1000}{0.39} = 2564 \text{ hours}$$

Therefore the mean time between failures is 2564 hours.

Logic symbols

APPENDIX 1

A1 INTRODUCTION

MIL-STD-806B (US Department of Defense) symbols (see section A4) are used extensively for describing logic circuits both in textbooks and papers and in the graphics of schematic capture in CAD packages. However, with the introduction of the ASIC and the general move towards customized LSI and with it the use of cell libraries, the traditional discrete component logic symbols are at too low a level to represent complex logic elements.

Internationally, Working Group 2 of the IEC Technical Committee TC-3 has consolidated all the work that has taken place on developments of logic symbols, and in the USA the IEEE has produced a standard containing all of the IEC work which is known as IEEE Std 91-1984.

This standard represents a new symbolic language which extends, rather than making obsolete, the use of the well-used MIL-STD-806B symbols, and the two can exist comfortably together as can be seen from their use in this and other texts.

It is not the intention to cover the whole of the IEEE standard in this appendix, but rather to introduce the principles associated with it, so as to encourage its wider acceptance by design engineers.

A2 SYMBOL STRUCTURE

A symbol for an element is made up from a combination of outlines, together with one or more qualifying symbols (see Figure A1 and Table A1). The function of these qualifying symbols is to define what logical operations are taking place at the inputs and outputs. The outlines of elements may be abutted or embedded. When a circuit has one or more inputs that are common, or where control is to be exercised over all

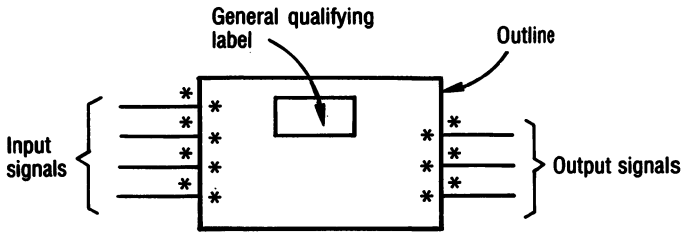


Figure A1 Symbol representation: * denotes possible position for qualifying symbol.

Table A1 General qualifying symbols (subset).

Symbol	*Description*
&	AND gate or function
>1	OR gate function, i.e. at least one active input is needed to activate the output
=1	Exclusive OR, i.e. one and only one input must be active to activate the output
=	Logic identity. All inputs must be at the same state
2k	An even number of inputs must be active
2k + 1	An odd number of inputs must be active
1	The one input must be active
▷	A buffer or element with more than the usual output capacity
⊿	Schmitt trigger or element with hysteresis
X/Y	Coder, code converter, e.g. DEC/BIN, BIN/HEX
MUX	Multiplexer/data selector
DMUX or DX	Demultiplexer
Σ	Adder
P–Q	Subtracter
CPG	Look-ahead carry generator
π	Multiplier
COMP	Magnitude comparator
ALU	Arithmetic logic unit
I = 0	Element powers up cleared to the 0 state
I = 1	Element powers up set to the 1 state
Φ	Highly complex function; 'black box' symbol with limited detail shown under special rules

of the inputs, a common control block is used of the shape shown in Figure A2.

Symbols appear both outside and inside the outline. Table A2 shows the input and output qualifying symbols which are *outside* the outline, while Table A3 shows a small subset of those which can be used *inside* the outline.

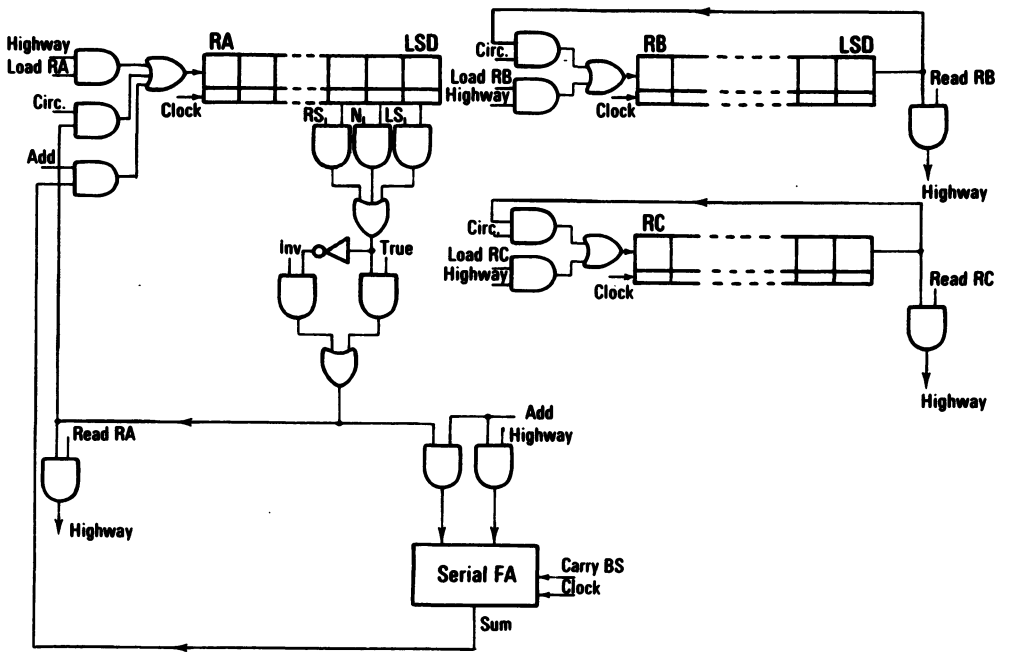
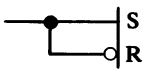


Figure A2 Common control block.

Table A2 Qualifying symbols for inputs and outputs (subset).

Symbol	*Description*
	Logic negation at input. External 0 produces internal 1
	Logic negation at output. Internal 1 produces external 0
	Active low input. Equivalent to in positive logic
	Active low output. Equivalent to in positive logic
	Dynamic inputs active on indicated transition
c	Input for analogue signals (on an analogue symbol)
#	Input for digital signals (on an analogue symbol)

Table A3 Symbols inside the outline (subset).

Symbol	*Description*
▽⊢	Three-state output
▷⊢	Output with more than usual output capability (symbol in direction of signal flow)
EN	Enable input
J,K,R,S	Usual meanings associated with flip-flops: e.g. R = reset to 0, S = set to 1
⊣⊢	Postponed output (of a pulse-triggered flip-flop). The output changes when the input initiating the change returns to its initial state
⊣▷T	Toggle input causes internal state of output to change to its complement
⊣D	Data input to a storage element equivalent to: 
CT=9⊢	The output is active if the content of the register is as indicated
⊣CT=15	The contents-setting input, when active, causes the register to be set to that value

A3 DEPENDENCY

This is a feature of the IEEE standard which raises the level of complexity that can be described by a logic element symbol. It permits the relationship between inputs or outputs, or inputs and outputs, to be logically defined without showing all of the elements and interconnections involved. Eleven types of dependency have been defined (see Table A4), but for the purposes of use in the diagrams of this book only the principles involved within a subset of these will be described, the reader being referred to the full IEEE Standard for a complete description.

The general labelling rules for applying dependency are as follows.

1. Label the input (or output) affect*ing* other inputs or outputs with the standard letter symbol (as in Table A4) indicating the relationship involved, followed by an identifying number.
2. Label each input (or output) affect*ed* by the standard letter symbol input (or output) with the same identifying number.
3. If it is the complement of the affect*ing* input that is to be used with another input then a bar is placed over *that* input and *not* the affecting input.
4. If two affect*ing* inputs (outputs) have the same letter and the same identifying number, they form an OR relationship with each other.

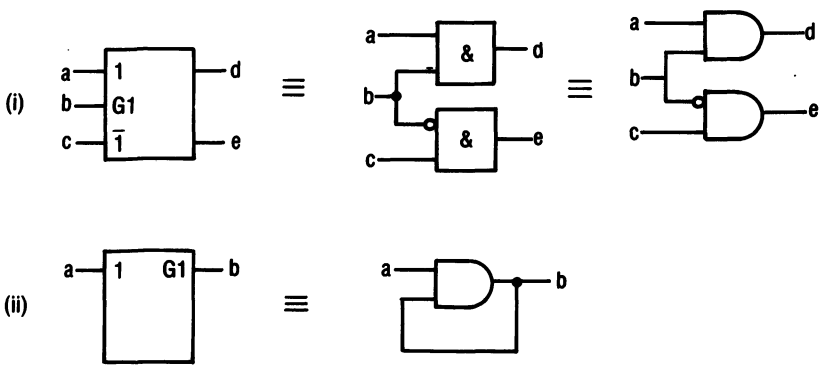
Table A4 Dependency types.

Dependency type	*Qualifying symbol*
Address	A
AND	G
Control	C
Enable	EN
Negate (Exclusive-OR)	N
Interconnection	Z
Mode	M
OR	V
Reset	R
Set	S
Transmission	X

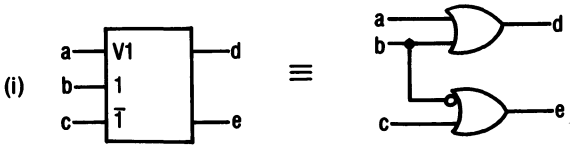
5. If the affect*ed* input (output) requires a label to denote its function, i.e. G, this label is *prefixed* by the identifying number of the affecting input. If there is more than one affecting input the identifying numbers of these are prefixed in the same way, and separated by commas.
6. For multiple affecting inputs in (5) above the order of logical implementation is taken as being from left to right.
7. In cases where ambiguity may arise through the use of numbers, another character may be used in their place, i.e. where the labels denoting the functions associated with an affected input are themselves numbers, such as the outputs of a coder.

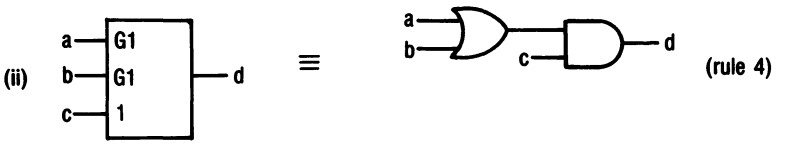
Some examples are given in sections A3.1 to A3.6.

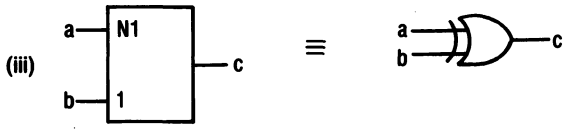
A3.1 AND (G)



A3.2 OR (V) and Exclusive-OR (N)

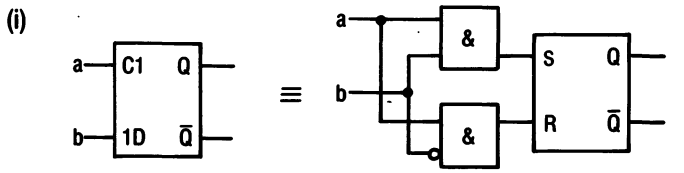




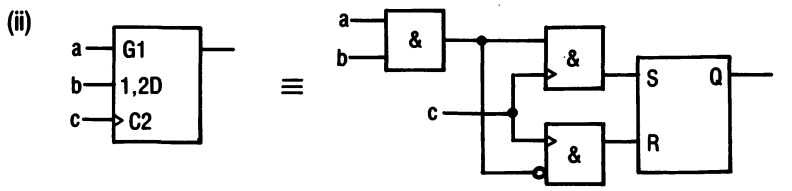


A3.3 Control (C)

Control inputs are used to enable (disable) the data inputs of storage elements. Dynamic inputs are shown with the appropriate symbol.



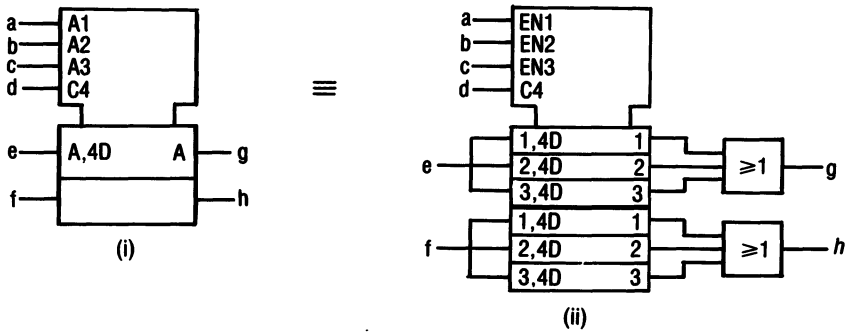
The D-type flip-flop is shown as an SR type driven from a single input and its complement.



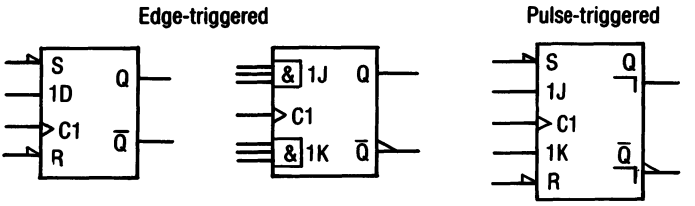
The dynamic input c affects the input b after it has been ANDed with input a, control being exercised as an AND function.

A3.4 Address (A)

This provides a symbolic representation of those elements, such as memory, which use address control inputs to select specified sections of an array. The output of the array is the OR function of corresponding elements of selected sections. As an example; a 3-word by 2-bit memory is shown below in (i) with an expansion of the functionality in (ii). From the latter it can be seen that A1 selects both bits of word 1, and if selected the values of the inputs on e and f can then be clocked in under control of input d, which is a control input. Similarly A2 and A3 select words 2 and 3 respectively. (i) is really a shorthand notation of (ii), and although it does not display the array structure of the memory, is concise and correct in its description of the input and control functions.



A3.5 Bistable elements



A3.6 Coders

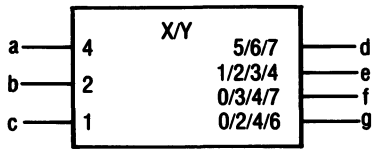
The general symbol for a code or coder is shown as the general qualifying symbol, for example Y/Z, X/Hex, etc., and gives some indication of the code and the relationship between inputs and outputs. The internal logic relationship is further indicated by labelling the inputs and outputs as follows. Either

1. label the inputs with numbers such that the internal value equals the sum of the weights associated with those inputs in their 1-state; or
2. replace Y by an appropriate indication of the input code and label the inputs with characters which refer to this code.

The output relationships with the internal logic value are indicated by either

1. labelling each output with a list of numbers (using '/' as a separator) representing those internal values that lead to the 1-state of the output; or
2. replacing Z by an appropriate indication of the output code and label the outputs with characters which refer to this code.

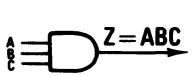
Alternatively the general symbol for a coder may be used together with a truth table showing the input/output relationship. This is the recommended way of showing a programmed PROM.

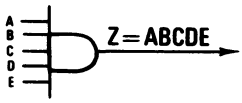


Inputs			Outputs			
a	b	c	d	e	f	g
0	0	0	0	0	1	1
0	0	1	0	1	0	0
0	1	0	0	1	0	1
0	1	1	0	1	1	0
1	0	0	0	1	1	1
1	0	1	1	0	0	0
1	1	0	1	0	0	1
1	1	1	1	0	1	0

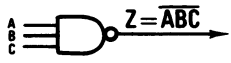
A4 MIL-STD-806B LOGIC SYMBOLS

A4. 1 AND gates

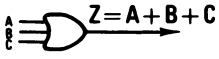


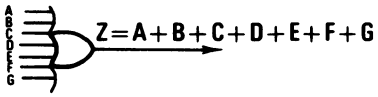


A4. 2 NAND gates

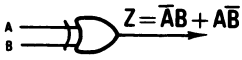


A4 3 OR gates

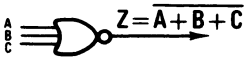




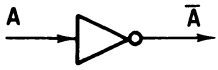
A4 4 Exclusive OR gates



A4 5 NOR gates



A4 6 Inverter amplifiers



Glossary of abbreviations

APPENDIX 2

AD	address strobe
ADB	ASP data buffer
ADCCP	Advanced Data Communications Control Procedure
ALU	arithmetic and logic unit
APE	associative processing element
ASCII	American Standard Code for Information Interchange
ASIC	application-specific integrated circuit
ASM	algorithmic state machine
ASP	associative string processor
ATE	automatic test equipment; automated test environment
ATM	asynchronous transfer mode
BCD	binary-coded decimal
BER	bit error rate
BiCMOS	Bipolar CMOS
BIST	built-in self-test
BSC	Binary Synchronous Communications
CA	collision avoidance
CAD	computer-aided design
CAFS	content-addressable file system
CASE	computer-aided software evaluation; computer-aided system evaluation
CCD	charge-coupled device
CCITT	Consultative Committee for International Telegraph and Telephone
CD	collision detection
CISC	complex instruction set computer
CML	current-mode logic
CMOS	complementary metal oxide semiconductor
CPU	central processor unit
CRC	cyclic redundancy check

CSMA/CD	carrier sense multiple access-collision detection
DAP	distributed array processor
DART	dual asynchronous receiver transmitter
DD	data dictionary
DDCMP	Digital Data Communication Message Protocol
DFD	data flow diagram
DIL	dual-in-line
DMA	direct memory access
DML	dedicated memory location
DMR	double mode redundancy
DS	data strobe
DSP	digital signal processor
DTL	diode-transistor logic
DXF	Drawing Interchange Format
E^2ROM	electrically erasable programmable read-only memory
ECL	emitter-coupled logic
EEPROM	electrically erasable programmable read-only memory
EMPL	Extensible MicroProgramming Language
EPLD	electrically programmable logic device
EPROM	erasable programmable read-only memory
ERD	entity relationship diagram
FET	field-effect transistor
FIR	finite impulse response
FLOPS	floating-point operations per second
FM	frequency modulation
GCR	group code recording
HDL	hardware description language
HDLC	high-level data link control
I^2L	integrated injection logic
IPSS	International Packet Switched Service
ISDN	Integrated Services Digital Network
ISP	Instruction Set Processor
ISPS	Instruction Set Processor Specification
JLCC	J-leaded chip carrier
LAN	local area network
LAPB	link access procedure-balanced
LCC	leadless chip carrier
LED	light-emitting diode
LFSR	linear feedback shift register
LLC	logical link control
LRU	least recently used
LSB	least significant bit
LSI	large-scale integration
MAC	medium access control

MAL	minimal average latency
MAN	metropolitan area network
MFM	modified frequency modulation
MIMD	multiple instruction multiple data
MIPS	millions of instructions per second
MISD	multiple instruction single data
MOPS	millions of operations per second
MOS	metal oxide semiconductor
MSB	most significant bit
MSI	medium-scale integration
MTBF	mean time between failures
MTRF	mean time to repair a fault
MTTF	mean time to failure
NRZ	non-return to zero
NRZ1	non-return to zero one
OSI	Open Systems Interconnection
PAL	programmable array logic
PC	personal computer
PCB	printed circuit board
PCM	pulse code modulation
PDX	primary data exchanger
PE	processing element
PGA	pin grid array
PLA	programmable logic array
PM	phase modulation
PROM	programmable read-only memory
PSS	Packet Switchstream Service
RAM	random-access memory
RISC	reduced instruction set computer
RISP	reduced intrusion scan path
RLL	run length limited
ROM	read-only memory
RTL	resistor–transistor logic
RZ	return to zero
SDLC	synchronous data link control
SIMD	single instruction multiple data
SIMPL	Simple Identity MicroProgramming Language
SISD	single instruction single data
SP	stack pointer
STD	state transition diagram
TMR	triple mode redundancy
TTL	transistor–transistor logic
UART	universal asynchronous receiver transmitter
VDM	Vienna Development Model

VDU	visual display unit
VGA	video graphics array
VHDL	VHSIC hardware description language
VHSIC	very high-level silicon integrated circuit
VLSI	very large-scale integration
WF	weighting factor
WORM	write once read many
ZM	zero modulation

Index