THE ROYAL SCHOOL OF SIGNALS

TRAINING PAMPHLET NO: **365**

DISTANCE LEARNING PACKAGE
CISM COURSE
MODULE 7 - COMPUTER SYSTEMS TECHNIQUES

Prepared by:

Technology Wing
CISM Group

Issue date: 22 September 2000

Training & Recruiting
ARMY TRAINING AND RECRUITING AGENCY

DP Bureau

This publication is for training purposes only. It is not for general external use. It is not subject to amendment and must not be quoted as an authority.

Authority: _____ (Name in blocks)

Signature: _____

Date: _____

Review Date	Review Date	Review Date
Signature	**Signature**	**Signature**

Authority: _____ (Name in blocks)

Signature: _____

Date: _____

Review Date	Review Date	Review Date
Signature	**Signature**	**Signature**

Authority: _____ (Name in blocks)

Signature: _____

Date: _____

Review Date	Review Date	Review Date
Signature	**Signature**	**Signature**

Section 1

Number Systems and Codes

CONTENTS

	Page
OBJECTIVES	1-2
Number Systems	1-3
Binary Numbers	1-4
Conversion of Binary to Decimal	1-4
Conversion of Decimal to Binary	1-5
Binary Word Size, MSB and LSB, Bit Designation	1-6
The Byte	1-6
The Nibble	1-6
Other Sizes	1-6
LSB and MSB	1-6
Bit Designation	1-6
Binary Arithmetic	1-7
Binary Addition	1-7
Binary Subtraction	1-7
Signed Binary Numbers and 2's Complement	1-9
Unsigned 8-bit Numbers	1-9
Signed 8-bit Numbers	1-9
Signed Binary Numbers and Number Range	1-10
Seeing the Sign of a Number	1-11
Negating a Binary Number using 2's Complement	1-11
2's Complement Negative Numbers and Subtraction	1-12
Fractional Binary Numbers - Fixed and Floating Point Numbers	1-14
Floating Point Numbers	1-14
The IEEE 754 Standard	1-15
Memory Store for Floating Point Numbers	1-16
The Hexadecimal Number System	1-17
Alphanumeric and ASCII Code	1-18
ASCII Character Set and Codes	1-19
Parity	1-21
Computer Codes	1-22
Binary Number Code	1-22
Computer Instruction Code	1-22
VDU Display Code	1-22
Answers to Problems and Student Examples	1-23

NUMBER SYSTEMS AND CODES

OBJECTIVES

At the end of this section the student should be able to:

a. State the 3 important characteristics of any number system.

b. Identify the characteristics of a binary number.

c. Convert binary numbers to decimal and vice-versa.

d. Add and subtract binary numbers.

e. Identify the characteristics of a signed binary number.

f. Negate Binary Numbers using the 2's complement method.

g. State the characteristics of a fractional binary number.

h. Calculate the equivalent decimal fractional number given the binary fractional number value and its characteristics.

i. Calculate the value of a binary fractional number given in IEEE 745 single precision format given the characteristics.

j. Identify the characteristics of a hexadecimal number.

k. Identify the characteristics of Alphanumeric codes in the form of ASCII code.

l. Identify the difference between even and odd parity.

m. State 4 different types of code that may be stored in a computer system.

Number Systems.

1. Any number system has 3 important characteristics:

   a. A "base number".

   b. A "digit range" or "character range". This is always from 0 to (base number - 1).

   c. A "number column weighting". This is the base number raised to the power of the column number.

   In the decimal number system, for example, we have:

   base number = 10.

   digit range = 0 to (10-1) = 0 to 9; ie any single digit in a number cannot be greater than 9. If a digit becomes greater then 9 then it causes a carry to the next column.

   number column weighting = 10^n where n is the column number.

2. If we view a decimal number, therefore, we can consider it to be a number of digits arranged in neighbouring columns as shown below:

4	3	2	1	0	column number, n
10^4	10^3	10^2	10^1	10^0	column weighting
10000	1000	100	10	1	" " "
5	0	6	2	9	decimal number

and we understand the number to read:

fifty thousand ($5*10^4$), six hundred ($6*10^2$), and twenty ($2*10^1$) nine ($9*10^0$). (Remember that any number to the power 0 is 1).

Binary Numbers.

1. For a binary number the number characteristics are:

base number = 2

digit range = 0 to 1 (ie from 0 to (base number - 1)).

column weighting = 2^n

example:

7	6	5	4	3	2	1	0	column number, n
2^7	2^6	2^5	2^4	2^3	2^2	2^1	1^0	column weighting
128	64	32	16	8	4	2	1	column weighting
1	0	0	1	1	0	1	1	binary number

2. In the example above each binary digit is either 0 or 1. Obviously no single digit can have a value greater than 1 since a value of 2 would require a carry to the next column. You will be familiar with the term "bit" which is used to refer to a binary digit; this is an abbreviation made from the 'b' of binary and the 'it' of digit.

3. Conversion of Binary to Decimal. Clearly the decimal equivalent of the binary number given above is found by multiplying each bit by its weighting. Hence the binary number 1 0 0 1 1 0 1 1 has the decimal value:

(1*128) + (0*64) + (0*32) + (1*16) + (1*8) + (0*4) + (1*2) + (1*1) = 155_{10}

Where subscript 10 means a number with a base of 10, ie a decimal number.

Problem: what is the decimal equivalent of the binary number 01001010?

4. Conversion of Decimal to Binary. Simply remember the column weightings and then, starting with the first column value which is less than the decimal number, record which bits are required to produce the number.

example: convert 97_{10} to binary form.

column weightings for an 8-bit number are:

128, 64, 32, 16, 8, 4, 2, 1

to put the number into binary form we require to generate the number 97.

There will be:

0 of the 128's since this is too big

1 of the 64's, leaving 33, ie 97 - 64

1 of the 32's, leaving 1, ie 97 - (64 + 32)

0 of the 16's, leaving 1

0 of the 8's, leaving 1

0 of the 4's, leaving 1

0 of the 2's leaving 1

1 of the 1's, leaving 0, ie 97 - (64 + 32 + 1)

we thus find that 97_{10} has the binary equivalent of 01100001.

Problem: what is the binary equivalent of the decimal number 127_{10}?

Binary Word Size, MSB and LSB, Bit Designation.

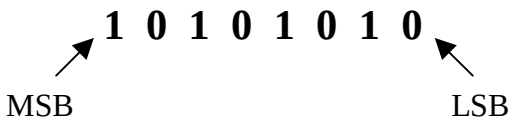
1. The term 'word' is used to refer to a binary number as a collection of 'bits' or 1's and 0's. Different logic circuits or microprocessors are designed to work with binary words of a certain size. The common sizes are:

a. The Byte. This term relates to a binary word having 8 bits and is the most common word size. Most computers still refer to memory size as the number of bytes, even though the computer itself may work with a much longer binary word. Memory devices normally have a maximum word size of 1 byte. In the examples above I have purposely used 8 bits or 1 byte.

b. The Nibble. The term Nibble refers to a word size of 4 bits. Hence 2 nibbles make 1 byte!

c. Other Sizes. All other word sizes have no special name; hence 16 bits is simply referred to as a 16 bit word, likewise 32 or 64 bits.

2. LSB and MSB.



The terms Most Significant Bit (MSB) and Least Significant Bit (LSB) refer to the bit values in a binary word which have the largest and smallest weighting, respectively. In the byte shown above the MSB has a value of '1' and the LSB has a value of '0'. Note that it is normal to write the binary word with the MSB at the left as shown above.

3. Bit Designation.

It is important to note that the number columns start at column 0, not column 1. An 8-bit number thus has a LSB which is designated Bit 0 and the MSB is Bit 7. Care should be taken when referring to individual bits in a binary number to make sure you use the correct designation.

B7	B6	B5	B4	B3	B2	B1	B0	← Bit Designation
1	1	0	1	1	1	0	0	← Binary number, LSB = 0, MSB = 1

Binary Arithmetic.

1. Binary Addition. When adding binary numbers we simply remember the rules for binary addition, which are:

$0 + 0 = 0$ and no carry

$0 + 1 = 1$ and no carry

$1 + 0 = 1$ and no carry

$1 + 1 = 0$ carry 1

example:

		7	6	5	4	3	2	1	0	
column number										
		0	0	1	1	1	1	0	0	$= 60_{10}$
	+									
		1	0	1	0	1	0	1	0	$= 170_{10}$
		1	1	1	0	0	1	1	0	$= 230_{10}$
carries from previous column			1	1	1					

explanation:

column	digits added	sum	carry
0	0 + 0	0	0
1	0 + 1	1	0
2	1 + 0	1	0
3	1 + 1	0	1 to column 4
4	1 + 0 + 1 (carry)	0	1 to column 5
5	1 + 1 + 1 (carry)	1	1 to column 6
6	0 + 0 + 1 (carry)	1	0
7	0 + 1	1	0

2. Binary Subtraction. Binary subtraction can appear complex. It turns out that it is much easier to consider subtraction as adding a negative number. We will consider this in a future section after completing some examples and reviewing positive and negative numbers.

Student Examples.

Answer the following Questions:

1. Convert the following binary numbers to decimal form:

   a. 0 0 1 0 0 1 1 0

   b. 1 0 1 0 1 0 0 1

   c. 1 1 1 1 1 1 1 1

2. Convert the following decimal numbers to binary form:

   a. 46

   b. 242

   c. 2973

3. Add the following binary numbers:

```
    1 0 0 1 1 1 0 0          1 0 0 1 1 1 0 0
+   0 1 1 1 0 0 1 0      +   0 1 1 0 1 1 0 1
  -----------------        -----------------
```

Signed Binary Numbers and 2's Complement.

1. So far we have viewed binary numbers as "unsigned", that is to say we have taken all numbers to be positive. There are many occasions in logic and computer circuits when we wish to do just that, eg when receiving character codes the concept of sign has no meaning, also when we count event occurrences we always record that an event occurred - we do not have negative events. There are, therefore many instances when we do not wish to be concerned with sign - or rather we wish to view all numbers as positive.

2. Viewed as an "unsigned" number any binary word can have a number of values represented by the whole range of codes available for the chosen word size. In the following we will consider 8-bit words or bytes since this word size is frequently used; all considerations, however, apply equally to any word size.

3. Unsigned 8-bit Numbers.

7	6	5	4	3	2	1	0	Column Number	
128	64	32	16	8	4	2	1	Column Weighting	
0	0	0	0	0	0	0	0	Smallest Number	= 0_{10}
0	1	1	1	1	1	1	1	Number Value	= 127_{10}
1	1	1	1	1	1	1	1	Largest Number	= 255_{10}

We can see that for an "unsigned", 8-bit number the number range is from 0 to 255, ie 256 different values. All values are taken to be positive, there is no concept of the negativeness of a number.

4. Signed Binary Numbers. When we wish to incorporate the idea of sign into the number we must find some way to show the positive or negative nature of the value. Normally we do this by appending positive (+) or negative (-) signs. In a computer system this could be a disadvantage since the very sign would consume additional storage space. Instead we redefine our understanding of the column weightings. We then find that we can incorporate the positive or negative nature of the number as a natural characteristic of that number; in addition, we find that there are other benefits.

5. Signed Binary Numbers and Number Range. We start by redefining the column weightings for any number; the basic rule is that the weighting value of the MSB will be negative whilst all other column weightings are taken to be positive. Shown below are examples of 8-bit signed number values starting from the most positive and working down to the most negative:

7	6	5	4	3	2	1	0	Column Number	
-128	64	32	16	8	4	2	1	Column Weighting	
0	1	1	1	1	1	1	1	Largest +ve	value = $+127_{10}$
.	.	.	.	.	.	.	.		
.	.	.	.	.	.	.	.		
0	0	1	0	1	0	0	1		value = $+41_{10}$
.	.	.	.	.	.	.	.		
.	.	.	.	.	.	.	.		
0	0	0	0	0	0	0	0		value = 0_{10}
1	1	1	1	1	1	1	1		value = -1_{10}
.	.	.	.	.	.	.	.		
.	.	.	.	.	.	.	.		
1	1	0	1	0	1	1	1		value = -41_{10}
.	.	.	.	.	.	.	.		
.	.	.	.	.	.	.	.		
1	0	0	0	0	0	0	0	Largest -ve	value = -128_{10}

Points to note in the above table:

a. All numbers follow from the normal number evaluation using column weightings.

b. The number -1 looks odd until we add all the digits with relevant column weighting, we then have -128 from the 1 in the MSB and a total of +127 from all other bits, giving a total of -128 + 127 = -1.

c. We can evaluate -41 using a similar method.

d. The largest negative number is found when there are no positive bits and only the MSB, ie -128.

e. The number range is therefore +127 through 0 to -128. Note that we still have 256 different values.

f. If we count up from -128, we will move through all the number values up to +127 in sequence.

One of the advantages of this technique is that if we add any number to its negative form (called its complement) then we get zero as we might expect.

example: from the above table +41 + (-41) is found from

```
+41  =  0 0 1 0 1 0 0 1
-41  =  1 1 0 1 0 1 1 1
sum  =  0 0 0 0 0 0 0 0
```

Note that there is a final carry out from the MSB, however if we are limited to a system using only 8 bits then this is lost and the result is 0.

6. Seeing the Sign of a Number. Probably one of the most important features of this system is that we can easily see if a value is negative or positive by simply looking at the MSB.

a. If the MSB = 0 then the number is positive.

b. If the MSB = 1 then the number is negative.

All computers and microprocessors use this fact to determine the sign of an integer (ie whole) number. Note, however, to create a negative number we do not simply put a 1 in the MSB.

Note one final feature of this system: the number zero (0) is positive! This must be so since there is a 0 in the MSB. This is important to remember in microprocessor work, eg to find if a number if > 0 it is not sufficient just to ask if the number is positive, since 0 is itself positive - more of this when we consider microprocessors.

7. Negating a Binary Number using 2's Complement Technique.

The 2's complement technique is a quick way to negate a binary number. It is called the "2's" complement since it produces the negative form, or complement, for a number written in binary or base 2 form. The rule is as follows:

a. First, write the binary number in its positive form.

b. Next, find the 1's complement, ie change all 1's to 0's and all 0's to 1's.

c. Finally, add 1 to the LSB; propagate carries as normal if they occur.

Example: Find -41_{10} as a signed 8-bit binary number.

First we write $+41_{10}$ as an 8-bit number:

$+41_{10}$ = 0 0 1 0 1 0 0 1

Next find the 1's complement:

1 1 0 1 0 1 1 0

Finally add 1 to the LSB:

$$\begin{array}{rr} & 1\,1\,0\,1\,0\,1\,1\,0 \\ + & \underline{\qquad\qquad 1} \\ & 1\,1\,0\,1\,0\,1\,1\,1 \end{array}$$

Hence -41_{10} = 1 1 0 1 0 1 1 1 in binary. You can check this in the signed number table on page 1-10.

8. 2's Complement Negative Numbers and Subtraction.

Using signed binary numbers of the form considered has the added advantage that we can more easily carry out subtraction. All we need to do is to add the negative form of the number being subtracted:

for example, the problem

40 - 26

is written instead as

40 + (-26)

These 2 will give the same result. The advantage is that in digital circuits it is very easy to add numbers and here we have turned subtraction into addition.

To complete the problem:

First write -26 in its positive form $+26_{10}$ = 0 0 0 1 1 0 1 0

Next find 1's complement = 1 1 1 0 0 1 0 1

Finally add 1 to the LSB, propagate any carry:

$$\begin{array}{rr} & 1\,1\,1\,0\,0\,1\,0\,1 \\ + & \underline{\qquad\qquad 1} \\ & 1\,1\,1\,0\,0\,1\,1\,0 \end{array}$$

Hence -26 = 1 1 1 0 0 1 1 0

Now we add this to +40:

$$\begin{array}{lll} +40 & = & 0\,0\,1\,0\,1\,0\,0\,0 \\ -26 & = & \underline{1\,1\,1\,0\,0\,1\,1\,0} \\ & & \underline{0\,0\,0\,0\,1\,1\,1\,0} \end{array}$$

Note that there is a final carry out from the MSB but, since we are using a system of 8-bit numbers, the 1 is effectively lost. The result is correct:

$$40 - 26 = 40 + (-26) = 14 = 0\,0\,0\,0\,1\,1\,1\,0$$

The process may appear rather involved but when we consider digital circuit design we will find that it is relatively easy to find the 1's complement of a number and also to add numbers together.

Student Examples.

1. Using the 2's complement technique, find the signed binary equivalent of the following -ve decimal numbers:

   a. -6, using 4-bit number.

   b. -9, using 8-bit number.

   c. -35, using 8-bit number.

   d. -108, using 8-bit number.

2. Perform the following calculations using 2's complement, 8-bit binary arithmetic. In each case check your answer by negating each result (find its 2's complement form) and remember that it is negative.

   a. 3 - 8.

   b. 40 - 60.

   c. 79 - 80.

Fractional Binary Numbers - Fixed and Floating Point Numbers.

1. Binary fractions can be written in the same manner as decimal fractions, ie using a 'binary' point, for example:

0.1010101
101.10111
1100.1010

Moving to the left from the binary point, column numbers increase from zero in the normal way: 0, 1, 2, 3 etc giving column weightings $1(2^0)$, $2(2^1)$, $4(2^2)$ etc. In a similar way, moving to the right from the binary point column numbers decrease from zero: -1, -2, -3 etc giving column weightings $0.5(2^{-1})$, $0.25(2^{-2})$, $0.125(2^{-3})$ etc.

Problem 1: using these weightings, what are the decimal equivalents of the binary numbers shown above?

2. One problem with this technique is that we need some way of saying where the binary point lies when the number is simply stored as a group of 8-bits. In the present situation we must simply decide on a convention. We may say that the binary point always lies between bits 7 and 6 of the 8-bit pattern (ie between the MSB and the next bit down).

Problem 2: using this convention, what is the decimal equivalent of the binary fractional number 1.0101101?

3. A number conforming to this convention is called a **Fixed Point Number.** Obviously there are some problems with this scheme - eg we cannot represent numbers greater than 2!

4. Floating Point Numbers. To represent the whole range of fractional numbers we use the floating point convention. Here the form of the number has 2 parts, the **MANTISSA** and the **EXPONENT.** There are many conventions, one such might be:

a. **The Mantissa.** An 8-bit unsigned fixed point number with the binary point before the MSB.

b. **The Exponent.** An 8-bit signed number which is a scaling factor. It is the power of 2 which the fixed point number is multiplied by to give the value.

hence	**number value**	=	**mantissa * $2^{exponent}$**
example 1:	floating point number	=	0 1 1 1 0 0 0 0 0 0 0 0 0 0 1 0
	mantissa	=	0 1 1 1 0 0 0 0
	exponent	=	2
hence,	the fixed point faction	=	0.25 + 0.125 + 0.0625
		=	0.4375
and	the scaling factor	=	2^2
		=	4
	floating point number value	=	0.4375 * 4
		=	1.75
example 2:	floating point number	=	1 1 0 1 0 0 0 0 1 1 1 1 1 1 1 1
	mantissa	=	1 1 0 1 0 0 0 0
	exponent	=	-1
hence	the fixed point fraction	=	0.5 + 0.25 + 0.0625
		=	0.8125
and	the scaling factor	=	2^{-1}
		=	0.5
	floating point number value	=	0.8125 * 0.5
		=	0.40625

5. The IEEE Standard. As mentioned above there can be many conventions for representing floating point numbers. If software is to be portable it is important to conform to an accepted standard and one such standard is the IEEE 754 standard for floating point representation. One representation within this standard is the **'Single Precision Standard'** which has the following features:

a. The floating point number is represented in 32 bits.

b. Bit 31 (the MSB) is a sign bit: 0 = positive, 1 = negative.

c. Bits 30 to 23 (the next 8 bits) are the exponent. This is an 8-bit unsigned integer. However, the value 127 is first subtracted to give the actual exponent value. This enables the exponent to have + ve or -ve values.

for example: an exponent of 11000000 has an actual value of 65 (192-127).

an exponent of 00010000 has an actual value of -111 (16-127).

d. Bits 22 to 0 are the mantissa. The binary point is before bit 22 and a 1 is understood to precede it.

for example: a mantissa of 01100. . . . . .

would be evaluated as 1.01100. . . . .

IEEE format:

Bit No:	31	30-23	22 21 20 . . . . . 4 3 2 1 0
Relevance:	Sign	exponent (E)	Fixed-point fraction (F)

Hence a number value = $1.F * 2^{E-127}$

with the sign indicated by the sign bit.

Also the exponent 0 is reserved to represent the number 0 and the exponent 255 to represent the number infinity.

With these rules the range of values is:

Largest:	$1.1111 \ldots 11 * 2^{254-127}$	$= 2 * 2^{127}$
		$= 3.403*10^{38}$
smallest:	$1.0000 \ldots 00 * 2^{1-127}$	$= 1 * 2^{-126}$
		$= 1.175 * 10^{-38}$

hence the range of values is: **$\pm 1.175 * 10^{-38}$ to $\pm 3.403*10^{38}$**

6. Memory Store for Floating Point Number. Since the floating point number uses 32 bits, it is clear that 4 bytes of computer memory will be required to store such a number.

The Hexadecimal Number System.

1. All computers and microprocessors use binary numbers or codes. However, when human beings wish to input binary codes it can be confusing and error prone, eg imagine working with a partner and telling him to type-in the binary code:

1 0 1 0 0 1 1 1 0 0 1 0 0 0 1 1 1 1 0 0

you would need to speak slowly and carefully to ensure your partner didn't miss some digits.

2. For this reason we normally input binary codes using a more convenient system this is the HEXADECIMAL (HEX) system. Note that inside the processor the data is still dealt with as binary 1's and 0's, using HEX provides a simpler man-machine interface.

3. Hexadecimal Numbers. The hexadecimal number system uses a base of 16, ie hex = 6, decimal 10 so hexadecimal = 16. We can determine the features of the system:

Base Number = 16

Character Range = 0 to (Base No - 1). Clearly we require 16 single, and different, characters to represent these 16 digits. We use:

0 1 2 3 4 5 6 7 8 9 A B C D E F

The great advantage of the HEX number system for our purposes is that each HEX digit can be used to represent 4 binary bits:

DECIMAL No	HEXADECIMAL No	4-BIT BINARY
0	0	0 0 0 0
1	1	0 0 0 1
2	2	0 0 1 0
3	3	0 0 1 1
4	4	0 1 0 0
5	5	0 1 0 1
6	6	0 1 1 0
7	7	0 1 1 1
8	8	1 0 0 0
9	9	1 0 0 1
10	A	1 0 1 0
11	B	1 0 1 1
12	C	1 1 0 0
13	D	1 1 0 1
14	E	1 1 1 0
15	F	1 1 1 1

Looking back at our long binary number from para 1, we see that it is much more easily expressed in hex form, all we do is to divide the binary number into groups of 4 bits starting from the LSB, then convert each group of 4 bits to its hex form:

1 0 1 0	0 1 1 1	0 0 1 0	0 0 1 1	1 1 0 0	binary
A	7	2	3	C	hex

Alphanumeric and ASCII Code.

1. To communicate between people we usually speak or write. In the latter case we use a set of standardised symbols; there are 2 principal sets of symbols which we use:

   a. The alphabet.

   b. Numbers.

2. At first sight, therefore, we appear to need to deal with 26 symbols for the alphabet and 10 symbols for numbers giving a total of 36 different symbols. However, we also require symbols for punctuation marks, upper and lower case and, when transferring information we require "control" symbols or "control codes", eg carriage return, line feed, horizontal tab etc.

3. It turns out that we can adequately accommodate all the symbols we require using a 7-bit code; this gives us 128 different symbol codes. When using this 7-bit code for alphanumeric and control codes we cannot help but store data in 8-bit bytes, consequently the eighth bit, ie bit 7, is normally zero.

4. The 7-bit codes use an agreed coding as specified in the ASCII code. The acronym ASCII stands for

   American Standard Code for Information Interchange.

The codes in the ASCII set are given on the next sheet.

5. When we type-in letters in a word processor program they will be stored in ASCII form. By consulting the data on the next page, what is represented by the hexadecimal code 46?

ASCII Character Set and Codes.

Dec	Hex	Bin	ASCII Character	Dec	Hex	Bin	ASCII Character
0	00	00000000	NUL	51	33	00110011	3
1	01	00000001	SOH	52	34	00110100	4
2	02	00000010	STX	53	35	00110101	5
3	03	00000011	ETX	54	36	00110110	6
4	04	00000100	EOT	55	37	00110111	7
5	05	00000101	ENQ	56	38	00111000	8
6	06	00000110	ACK	57	39	00111001	9
7	07	00000111	BEL	58	3A	00111010	:
8	08	00001000	BS	59	3B	00111011	;
9	09	00001001	HT	60	3C	00111100	<
10	0A	00001010	LF	61	3D	00111101	=
11	0B	00001011	VT	62	3E	00111110	>
12	0C	00001100	FF	63	3F	00111111	?
13	0D	00001101	CR	64	40	01000000	@
14	0E	00001110	SO	65	41	01000001	A
15	0F	00001111	SI	66	42	01000010	B
16	10	00010000	DLE	67	43	01000011	C
17	11	00010001	DC1	68	44	01000100	D
18	12	00010010	DC2	69	45	01000101	E
19	13	00010011	DC3	70	46	01000110	F
20	14	00010100	DC4	71	47	01000111	G
21	15	00010101	NAK	72	48	01001000	H
22	16	00010110	SYN	73	49	01001001	I
23	17	00010111	ETB	74	4A	01001010	J
24	18	00011000	CAN	75	4B	01001011	K
25	19	00011001	EMN	76	4C	01001100	L
26	1A	00011010	SUB	77	4D	01001101	M
27	1B	00011011	ESC	78	4E	01001110	N
28	1C	00011100	FS	79	4F	01001111	O
29	1D	00011101	GS	80	50	01010000	P
30	1E	00011110	RS	81	51	01010001	Q
31	1F	00011111	US	82	52	01010010	R
32	20	00100000	SPACE	83	53	01010011	S
33	21	00100001	!	84	54	01010100	T
34	22	00100010	"	85	55	01010101	U
35	23	00100011	#	86	56	01010110	V
36	24	00100100	$	87	57	01010111	W
37	25	00100101	%	88	58	01011000	X
38	26	00100110	&	89	59	01011001	Y
39	27	00100111	'	90	5A	01011010	Z
40	28	00101000	(	91	5B	01011011	[
41	29	00101001	)	92	5C	01011100	\
42	2A	00101010	*	93	5D	01011101	]
43	2B	00101011	+	94	5E	01011110	^
44	2C	00101100	´	95	5F	01011111	¯
45	2D	00101101	-	96	60	01100000	`
46	2E	00101110	.	97	61	01100001	a
47	2F	00101111	/	98	62	01100010	b
48	30	00110000	0	99	63	01100011	c
49	31	00110001	1	100	64	01100100	d
50	32	00110010	2	101	65	01100101	e

Dec	Hex	Bin	ASCII Character	Dec	Hex	Bin	ASCII Character
102	66	01100110	f	115	73	01110011	S
103	67	01100111	g	116	74	01110100	T
104	68	01101000	h	117	75	01110101	U
105	69	01101001	i	118	76	01110110	V
106	6A	01101010	j	119	77	01110111	W
107	6B	01101011	k	120	78	01111000	X
108	6C	01101100	l	121	79	01111001	Y
109	6D	01101101	m	122	7A	01111010	Z
110	6E	01101110	n	123	7B	01111011	{
111	6F	01101111	o	124	7C	01111100	|
112	70	01110000	p	125	7D	01111101	}
113	71	01110001	q	126	7E	01111110	~
114	72	01110010	r	127	7F	01111111	DEL

ASCII Control Characters

Hex	ASCII Character	Meaning	Keyboard Entry
00	NUL	Null	Ctrl-@
01	SOH	Start of Heading	Ctrl-A
02	STX	Start of Text	Ctrl-B
03	ETX	End of Text	Ctrl-C
04	EOT	End of Transmission	Ctrl-D
05	ENQ	Enquiry	Ctrl-E
06	ACK	Acknowledge	Ctrl-F
07	BEL	Bell	Ctrl-G
08	BS	Back Space	Ctrl-H
09	HT	Horizontal Tabulation	Ctrl-I
0A	LF	Line Feed	Ctrl-J
0B	VT	Vertical Tabulation	Ctrl-K
0C	FF	Form Feed	Ctrl-L
0D	CR	Carriage Return	Ctrl-M
0E	SO	Shift Out	Ctrl-N
0F	SI	Shift In	Ctrl-O
10	DLE	Data Link Escape	Ctrl-P
11	DC1	Device Control One	Ctrl-Q
12	DC2	Device Control Two	Ctrl-R
13	DC3	Device Control Three	Ctrl-S
14	DC4	Device Control Four	Ctrl-T
15	NAK	Negative Acknowledge	Ctrl-U
16	SYN	Synchronous Idle	Ctrl-V
17	ETB	End of Transmission	Ctrl-W
18	CAN	Cancel	Ctrl-X
19	EM	End of Medium	Ctrl-Y
1A	SUB	Substitute	Ctrl-Z
1B	ESC	Escape	Ctrl-[
1C	FS	File Separator	Ctrl-\
1D	GS	Group Separator	Ctrl-]
1E	RS	Record Separator	Ctrl-^
1F	US	Unit Separator	Ctrl-_

Note: Keyboard entries, eg Ctrl-A, means hold down the Ctrl key whilst pressing key A.

PARITY

1. When we transmit data we send it in 'packets' or words. Frequently the chosen word size is 8 bits or 1 byte since the transmitted word often represents an ASCII character. To allow for identification of errors, which may be introduced during transmission, we arrange to send an extra bit with each word. The extra bit is called a PARITY bit and it is chosen to be either a '1' or a '0' for each word depending on the type of parity we use and the number of 1's within the data word.

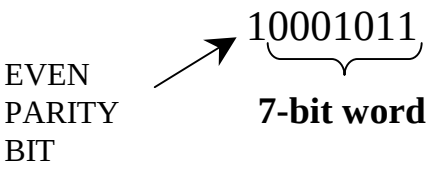
2. There are 2 options: either EVEN parity, or ODD oarity. The differences are as follows:

   a. **EVEN Parity**. When using even parity we arrange that the extra bit for each word is chosen so that it produces, *in total*, an even number of 1's when the parity bit is included. For example, if we have a 7-bit word (ASCII characters have 7-bit codes):

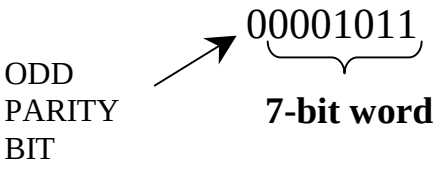
   $$\underbrace{0001011}$$

   7-bit word

   then, using EVEN parity we require to generate an extra '1' for the parity bit to give, overall, an even number of 1's. This is frequently placed in the MSB position so that the transmitted word, including parity bit, would appear as:

   b. **ODD Parity**. When using odd parity we arrange that the extra bit for each word is chosen so that it produces, *in total*, an odd number of 1's when the parity bit is included. For the example above using odd parity, we now require to generate an extra '0' for the parity bit to give, overall, an odd number of 1's in the word, including the parity bit. Once again the extra bit is frequently placed in the MSB position so that the transmitted word, including parity bit, would now appear as:

3. At the receiver, of course, we must arrange to check the number of 1's in each 8-bit binary word, and if it differs from the chosen parity we know that the data has been corrupted. We might then have an arrangement where we ask the transmitter to send the last word again before sending more data.

COMPUTER CODES

A computer is a general purpose processor which is provided with memory to store data. The memory may be viewed as a number of locations and at each location we can store or hold a single byte, ie 8-bits, of data. It is important to understand that the same 8 binary digits can be interpreted by the computer in different ways depending on the way the data is being used; for example the code 00111110 (ie 3E hex) may be interpreted as:

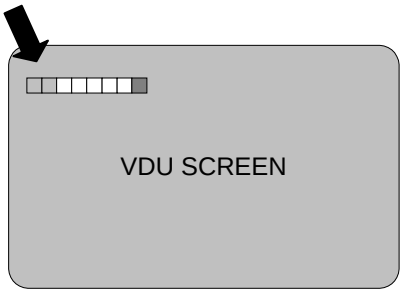
a. Binary Number Code. In one application the computer may interpret the 8 bits simply as a binary number. Thus 3E = 00111110 has the decimal value 62.

b. Computer Instruction Code. Elsewhere in the computer memory it might be interpreted as an instruction. For example, a computer which uses a Z80 microprocessor would interpret the code 3E as an instruction to:

"Load the A register with a number"

c. VDU Display Code. If the code is held in memory in the area indicating graphics output to the screen then the logic 1's in the code would indicate where the screen is to be brightened up:

3E = 00111110



d. ASCII Code. If the code is, say, part of the text code in a wordprocessor data store area, it would be interpreted as the character '>'.

Answers to Problems and Student Examples.

Page 1-4, Problem: $01001010 = 74_{10}$.

Page 1-5, Problem: $127_{10} = 01111111$.

Page 1-8, Student Examples:

1. Decimal form of binary numbers:

   a. 00100110 = 38
   b. 10101001 = 169
   c. 11111111 = 255

2. Binary form of decimal numbers:

   a. 46 = 00101110
   b. 242 = 11110010
   c. 2973 = 101110011101

3. Adding Binary Numbers:

	1 0 0 1 1 1 0 0	1 0 0 1 1 1 0 0
+		+
	0 1 1 1 0 0 1 0	0 1 1 0 1 1 0 1
	1 0 0 0 0 1 1 1 0	1 0 0 0 0 1 0 0 1
carries	1 1 1 1	1 1 1 1 1 1

Page 1-13, Student Examples:

1. Signed binary equivalent of negative decimal number:

   a. -6 = 1010 (using 4-bit number)
   b. -9 = 11110111 (8-bit number)
   c. -35 = 11011101 (8-bit number)
   d. -108 = 10010100 (8-bit number)

2. Calculations using 2's complement:

   a. 3 - 8:

+8	= 00001000	
1's complement	= 11110111	
add 1 to LSB	= 11111000	= -8

+3	= 00000011	
+		
-8	= 11111000	
	11111011	

Thus the answer is 3 - 8 = 11111011

We can see that 11111011 = -5 by negating this number by 2's complement:

number = 11111011
1's comp = 00000100
add 1 = 00000101

The negated number is +5 so the number must be -5. As an alternative we could check the value by evaluating the number using column weightings:

(1*(-128))+(1*64)+(1*32)+(1*16)+(1*8)+(1*2)+(1*1) = -5

b. 40 - 60 = 11101100
c. 79 - 80 = 11111111

Page 14, problem 1:

0.1010101 = 0.5 + 0.125 + 1.03125 + 0.0078125

= 0.6640625

101.10111 = 4 + 1 + 0.5 + 0.125 + 0.0625 + 0.03125

= 5.71875

1100.1010 = 8 + 4 + 0.5 + 0.125

= 12.625

Page 14, problem 2:

1.0101101 = 1 + 0.25 + 0.0625 + 0.03125 + 0.0078125

= 1.3515625

Page 18, para 5:

The ASCII character having hex code 46 is F.

Section 2

Basic Logic
Definitions, Symbols and Circuits

CONTENTS

	Page
OBJECTIVES	2-2
Introduction	2-3
Some Basic Definitions	
Logic	2-4
True and False States	2-4
Positive and Negative Logic	2-5
Combinational Logic and Sequential Logic	2-6
Example of a Combinational Logic Circuit	2-7
The Truth Table	2-7
Conclusions	2-9
Logic Gates and Circuits	
Introduction	2-10
The NOT Function	2-10
The AND Function	2-11
The OR Function	2-12
The NAND Function	2-13
The NOR Function	2-14
The EXCLUSIVE OR (EOR) Function	2-15
Uses for the Exclusive OR Gate	2-16
Non Equivalence Gate	2-16
Binary Adder	2-16
Binary Word Inversion under Logic Control	2-17
EOR Configuration	2-18
Conclusions	2-19
Student Examples	2-20
Answers to Problems and Student Examples	2-21

BASIC LOGIC: DEFINITIONS, SYMBOLS AND CIRCUITS

OBJECTIVES

On completion of the material in this section the student should be able to:

a. State how the logic TRUE and FALSE states are represented in written documents.

b. Define the terms POSITIVE and NEGATIVE logic.

c. Explain the difference between COMBINATIONAL and SEQUENTIAL logic.

d. Draw up a TRUTH TABLE for given inputs and stated output requirements.

e. Identify the 3 basic logic functions as the NOT, AND and OR functions.

f. Identify the common mixed logic gates: NAND, NOR and EOR.

g. For each of the basic logic functions and mixed logic gates identify:

   (1) The Gate Symbol.

   (2) The Truth Table.

   (3) The Boolean representation.

h. State an additional advantage of NAND or NOR gates.

i. Using the EOR gate, explain the working of the following common circuits:

   (1) The Non-Equivalence Gate.

   (2) The Modulo-2 adder and Half Adder.

   (3) The Binary Word Inversion Circuit.

INTRODUCTION

Microprocessors, memory devices and various computer system peripheral devices rely for their operation on the principles of digital components and logic processes. It is therefore necessary to appreciate some of the 'basics' of logic and digital components so that we can appreciate better their function. This section introduces some basic logic and digital component principles for this purpose. Whilst you will not be expected to design digital circuits it is important that you appreciate the functionality of some of the basic components. In particular we will cover:

a. Basic digital/logic gates.

b. Simple combinational logic.

c. Simple sequential logic, including in particular the basic counter and register.

SOME BASIC DEFINITIONS

Logic.

1. Logic is a branch of mathematics which deals with problems in reasoning which can have results that are expressed as either TRUE or FALSE; there can be no in-between possibilities.

2. In realising circuits which can respond in this logical way we use circuit 'gates' whose inputs can replicate the TRUE or FALSE conditions and whose outputs at any time can be either TRUE or FALSE.

3. Clearly we must have some circuit analogue of these TRUE/FALSE states; for this we have a number of choices, for example:

Representation	True State	False State
Switch	Closed	Open
Current	Flow	No Flow
Lamp	On	Off
Voltage	High	Low

Note, in the above it is up to us to define what we mean by TRUE and FALSE. We could thus choose to have the states defined the other way around. Once defined, however, we must stick with the definitions.

4. Most frequently we represent the logic 'states' as voltage levels and logic circuits are designed to output a high or low level in response to combinations of high and/or low levels at their input/s.

TRUE and FALSE States.

1. To represent the TRUE logic state we normally write the state as a logic '1'. To represent the FALSE state we represent the state as a logic '0'.

2. Since our logic can only assume one of 2 possible states we say that it is a BINARY system. Clearly our binary arithmetic will be applicable to this arrangement.

Positive and Negative Logic.

1. Having decided that our circuit is going to represent logic states using voltage levels we must decide what level is to represent which logic state. We can choose either POSITIVE or NEGATIVE logic; these are defined as follows:

a. Positive Logic: The more positive level represents the TRUE state.

Note that we have some options within this definition eg we might choose that +5V represents the TRUE state and 0V the FALSE state. Alternatively 0V could represent the TRUE state and -5V the FALSE state. In either case the more positive level is representing the TRUE state.

b. Negative Logic: The more negative level represents the TRUE state.

Again we can have a range of choices similar to the case above.

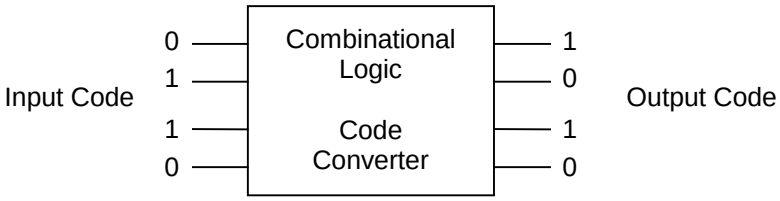
2. In practice we frequently use POSITIVE logic. You should, however, be familiar with the concept of negative logic since it has some particular advantages in certain conditions.

Combinational Logic and Sequential Logic.

1. In our study of logic circuits we will divide the topic into 2 distinct parts:

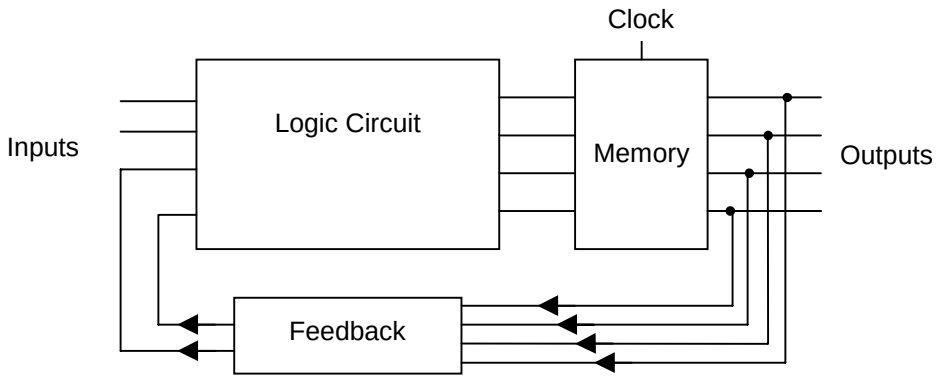
COMBINATIONAL LOGIC and SEQUENTIAL LOGIC

2. Combinational Logic. In combinational logic we will deal with circuits which, given certain input logic values, will automatically produce an output or outputs depending only on the inputs provided at that time. A simple example is a code converter. Here the input is one code of logic 1's and 0's and the output is some other code.



3. Sequential Logic. In a sequential logic circuit inputs arrive in sequence and the output now depends on the external inputs and the previous state of the outputs. This definition of sequential logic implies some form of memory and feedback so that the system input is aware of the previous output(s). When we consider sequential logic circuits we will find that, generally, the circuit will require an external clock to trigger a change of output.

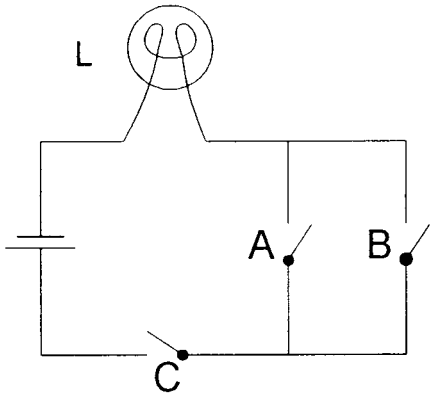
4. A good example of a sequential circuit is a counter where the count increments in response to an external clock pulse and the counter must be aware of the previous output to determine the next output after the clock pulse.



Sequential Logic Circuit with Memory, Feedback and Clock

Example of a Combinational Logic Circuit

1. We will start our consideration of logic circuits by looking at combinational logic; a simple combinational logic circuit is shown below:



In the circuit, if switch C is closed then closing either switch A or B will immediately operate the lamp, L . Clearly, it is the combination of inputs at A, B and C which produce the output at L.

The Truth Table.

1. A Truth Table is a powerful tool which is used in deciding the outputs from a given circuit or in stating the initial design requirements for a logic circuit. The Truth Table is drawn up to show all possible input states and the resulting output state, or states, produced.

2. Before we can draw up a Truth Table we must decide what input and output conditions represent the logic '0' and logic '1' states. For the circuit shown above we will choose the following definitions:

inputs:	switch open	: logic '0'
	switch closed	: logic '1'
outputs	lamp off	: logic '0'
	lamp on	: logic '1'

Having decided these definitions we can now draw-up a table to show the outputs resulting from all possible combinations of inputs:

Truth Table:

INPUTS			OUTPUT
A	**B**	**C**	**L**
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

To understand the table contents consider the first row, this tells us that if switches A, B and C are all OPEN then the lamp L, will be OFF. Alternatively, row 4 tells us that if switches C and B are closed, with A open, the lamp, L will be ON.

3. Notice that in completing the table to show all logic inputs we are effectively tabulating a binary count sequence - this is all combinations of 1's and 0's for the range of inputs given.

4. This Truth Table technique gives us an easily understood representation of the circuit operation. We will use truth tables in the next section to understand the functionality of the basic logic circuits which are available.

Conclusions.

In this introductory section we have considered some of the basic definitions which will be used throughout our study of logic circuits. The following summarises the important points:

a. In its pure mathematical form logic is the study of reasoning where at any time variables can have only one of 2 possible states: TRUE or FALSE.

b. When referring to the TRUE and FALSE states we will write logic '1' and logic '0', respectively.

c. We will generally use POSITIVE logic where the TRUE or logic '1' state is represented by the more positive voltage level.

d. Combinational logic is the study of circuits which automatically produce a change in output in response to a change in input conditions.

e. In Sequential logic we add memory capability and some feedback from output to input so that the output depends on the external inputs and on the previous state of the output(s). A clocking signal will normally be required.

f. Truth Tables will be used to determine output levels in response to input conditions. To draw up the Truth Table we tabulate each logic input condition and determine the resulting output required.

g. For a Truth Table having 'n' logic inputs we find that the range of input combinations conveniently follows the normal binary count sequence for an n-bit number.

COMBINATIONAL LOGIC - GATES AND CIRCUITS

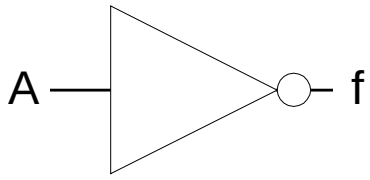
Introduction.

1. Surprisingly in logic there are only 3 basic functions! All other, more complex functions are modifications or combinations of these 3. The functions are: NOT, AND and OR. In this section we will consider these functions and the logic circuits or gates which are produced to replicate the functions in digital electronic circuits.

2. For each logic function we will consider the gate provided, its truth table and also the mathematical symbology used to represent the logic function. The mathematical representation was devised by an English mathematician called George Boole, hence the form of mathematics which uses this symbology is called BOOLEAN ALGEBRA.

The NOT Function.

Gate Symbol



Truth Table

A	f
0	1
1	0

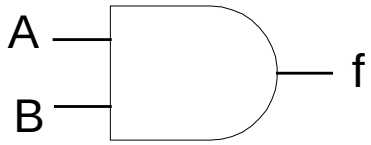
Boolean

$f = \overline{A}$

The NOT function carries out the process of logical inversion, ie for a logic '1' input we have a logic '0' output and vice-versa. The logic function can be seen from the truth table for input 'A' and output 'f'. The Boolean expression says that the output is the logical inverse of the input - the bar symbol, '¯', above the A indicates inversion. The NOT gate is also called an INVERTER.

The AND Function.

Gate Symbol



Truth Table

A	B	f
0	0	0
0	1	0
1	0	0
1	1	1

Boolean

$f = A \cdot B$

AND

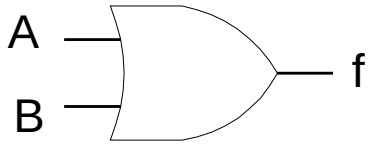
1. For an AND gate the logic function obeys the rule that we only get a TRUE or logic '1' output if all the inputs are TRUE, ie all the inputs are logic '1'. In the above example, therefore, we have a TRUE output if A AND B are both TRUE. This can be extended to any number of inputs, thus for 3 inputs: A, B and C we have a logic '1' output if A AND B AND C are all logic '1'.

2. Note that the Boolean expression uses the symbol '.' to mean AND. We are more familiar with this symbol representing multiplication. If you are new to digital logic it will help if when you read such expressions you make a conscious effort to mentally say AND so as not to confuse the two.

Problem:	draw the gate symbol and write out the truth table and Boolean expression for a 3-input AND gate with inputs: A, B, C and output f.

The OR Function.

Gate Symbol



Truth Table

A	B	f
0	0	0
0	1	1
1	0	1
1	1	1

Boolean

f = A + B

OR

1. The OR gate logic function obeys the rule that a TRUE or logic '1' output is generated if any of the inputs are TRUE. In the example shown above, the output will be TRUE if either A OR B OR both are TRUE. This can be extended to any number of inputs, thus for 3 inputs A, B and C, we have a logic '1' out if A OR B OR C or any combination of the 3 is logic '1'. In this case the Boolean expression would be:

$$f = A + B + C$$

2. Note that the Boolean expression uses the symbol '+' to mean OR. We are more familiar with this representing addition. Once again you may find it helpful to mentally read this as OR rather than PLUS so as not to confuse the two.

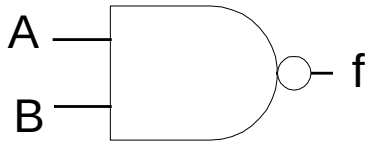
Other Logic Functions and Gates.

1. As stated previously, the 3 basic logic functions are NOT, AND and OR. However, these can appear in various combinations so other gates are available which provide some of the simple extended logic; in addition, we will find that these extended gates have other advantages.

2. We will consider 3 additional gates: The NAND, NOR and Exclusive OR (EOR).

The NAND Function.

Gate Symbol



Truth Table

A	B	f
0	0	0
0	1	1
1	0	1
1	1	1

Boolean

NOT

$$f = \overline{A . B}$$

AND

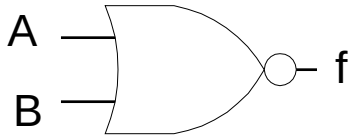
1. The NAND gate may be considered to be a composite gate consisting of an AND gate followed by a NOT gate. In the symbology used the small circle on the output indicates the NOT function. If you compare the Truth Table with that given for the AND gate earlier you will see that the outputs are the inverse of those given for the AND. The Boolean expression shows that the normal AND function has been INVERTED - hence the NOT, or inversion line spanning the whole function. Finally, note that NAND gates with more than 2 inputs are also available.

2. The NAND gate has greater use than simply giving us a conveniently inverted AND. We will see later that any logic function, no matter how complex, can be realised using all NAND gates in the circuit. This has great advantages. If, for example, you wish to produce a single integrated circuit device to perform your desired logic function then it is much easier to design a chip using all the same gates.

Problem:	write down the Boolean expression for a 5-input NAND gate having output f and inputs A, B, C, D and E.

The NOR Function.

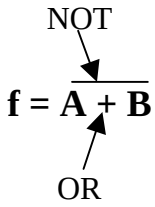
Gate Symbol



Truth Table

A	B	f
0	0	1
0	1	0
1	0	0
1	1	0

Boolean

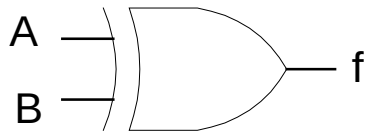


1. Like the NAND gate the NOR gate may be considered to be a composite gate consisting of an OR gate followed by a NOT gate. In the symbology used the small circle on the output indicates the NOT function. If you compare the Truth Table with that given for the OR gate earlier you will see that the outputs from the NOR gate are the inverse of those given for the OR. The Boolean expression shows that the normal OR function has been INVERTED, hence the NOT, or inversion line spanning the whole function. Finally, note that NOR gates with more than 2 inputs are also available.

2. We will see later that any logic function, no matter how complex, can be realised using all NOR gates in the circuit. This is a similar property to that of the NAND gate seen earlier.

The EXCLUSIVE OR (EOR) Function.

Gate Symbol



Truth Table

A	B	f
0	0	0
0	1	1
1	0	1
1	1	0

Boolean

$$f = A \oplus B$$

EOR

1. The exclusive OR gate is a combination of other basic logic gates as we shall see shortly. The Truth Table shows us that the function gives a TRUE result exclusively for the OR condition, ie the output is TRUE or logic '1' when A is TRUE, OR when B is TRUE but not when both are TRUE; it is a '*one OR the other*' situation.

2. Whilst the EOR function is produced by combining AND/OR/NOT gates in some way, the function is so important that it has been given its own Boolean notation - we see that the exclusivity of the OR function is emphasised by a ring around the normal OR symbol.

3. The EOR function is found frequently in problems concerned with signal processing; it is a function worth remembering since it will occur in subjects other than basic logic.

4. The EOR function can be produced in a variety of ways and we can see one realisation of the function from the Truth Table. Note that the Truth Table tells us (at lines 2 and 3) that the output (f) of the EOR gate will be logic '1' if:

A is logic '0' AND B is logic '1'

OR

A is logic '1' AND B is logic '0'

In Boolean arithmetic we indicate that a variable is in the LOW or logic '0' state by placing a bar above the symbol. If the variable is in the HIGH or logic '1' there is no bar above the symbol.

5. We can thus put the expression at paragraph 4 into Boolean logic by writing:

$$f = \overline{A}.B + A.\overline{B}$$

where: $\overline{A}.B$ says A is '0' and B is '1'

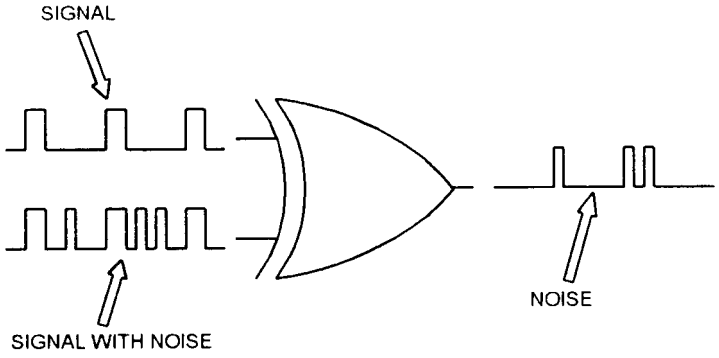
$A.\overline{B}$ says A is '1' and B is '0'

This is a relationship that is well worth remembering since it will occur many times in your studies.

6. Uses for the Exclusive OR Gate.

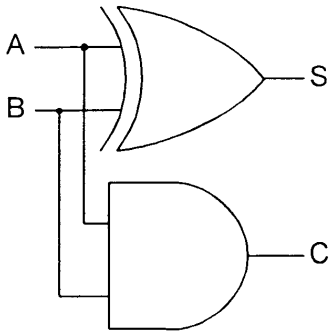
Some common uses are as follows:

a. Non-Equivalence Gate. The EOR gate will tell us if 2 logic inputs are not equal. From the Truth Table we can see that if the 2 inputs are not equal then the output from the gate is a logic '1'. If, for example, we were to pass similar digital signals through the gate, one including additional noise pulses, then the output would indicate the noise pulses since this would be the times when the 2 signals differed.



b. Binary Adder. The EOR gate is a fundamental component of a binary addition circuit. To be more precise, with the inclusion of an AND gate it allows us to produce a HALF ADDER circuit. This is a circuit which will correctly add together 2 binary digits and generate SUM and CARRY bits.

Half Adder Circuit



Truth Table

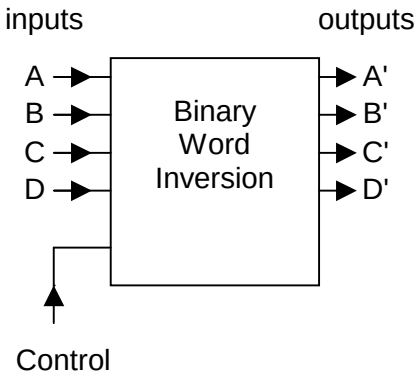
A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

S = SUM
C = CARRY

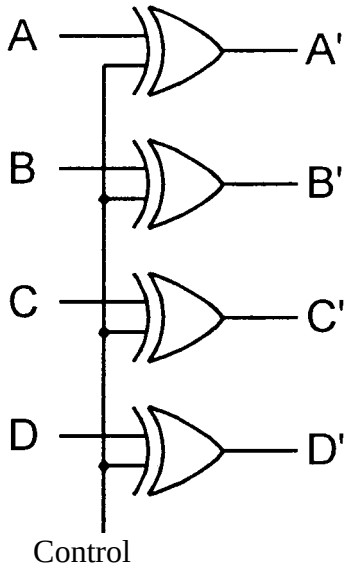
The EOR gate produces the correct output for the SUM, S, of 2 binary digits. In this application it is called a MODULO 2 adder since it produces the sum of 2 digits to the base 2; the AND gate produces the correct output for the carry, C, digit.

c. Binary Word Inversion under Logic Control. The input to this circuit is a data word together with a control signal. If the control signal is logic '0' then the output is the same as the data word input, if the control signal is a logic '1' then the output word is the 1's complement of the input.

SCHEMATIC

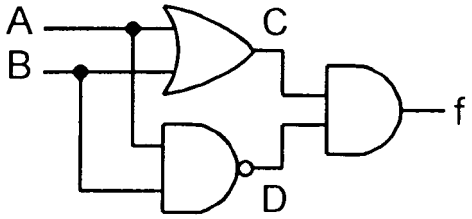


CIRCUIT

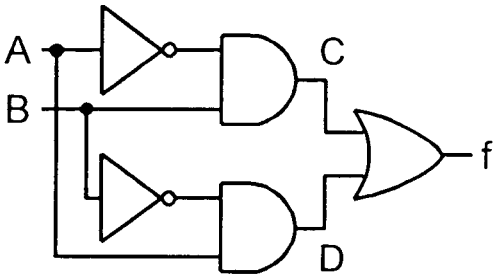


in the above circuit, if the input word, DCBA is 1010, and the control line is logic '0', then the output word, D'C'B'A', will also be 1010. Alternatively, if the control line is logic '1' then the output word will be 0101, the inverse of the input word.

7. EOR Configuration. As mentioned previously, the EOR gate is a combination of basic logic gates and it can be realised in a variety of ways. For the following gate combinations, complete the truth tables for the logic levels at C and D, produced by inputs A and B, hence show that the logic levels produced at output, f, indicate the EOR function.



A	B	C	D	f
0	0			
0	1			
1	0			
1	1			



A	B	C	D	f
0	0			
0	1			
1	0			
1	1			

8. Conclusions.

a. In pure logic there are only 3 fundamental functions for which we provide logic gates: the NOT, AND and OR.

b. The NOT function produces logical inversion which is indicated by the circle on the symbol output. In Boolean algebra inversion is indicated by a bar ('－') above the symbol.

c. The AND function only produces a TRUE or logic '1' output when all inputs are TRUE. In Boolean Algebra the AND function is represented by a '.' between symbols.

d. The OR function gives a TRUE or logic '1' output when any of the inputs, is TRUE. In Boolean Algebra the OR function is represented by a '+' between symbols.

e. Other useful, basic combinations of the fundamental gates are also provided. These are the NAND, NOR and EOR.

f. The NAND function is effectively an AND followed by a NOT function. A 2-input NAND gate with inputs A and B produces the Boolean function: $\overline{A.B}$

g. The NOR function is effectively an OR followed by a NOT function. A 2-input NOR gate with inputs A and B produces the Boolean function: $\overline{A+B}$

h. The EOR gate gives a TRUE output exclusively for the OR function, the output is TRUE only if one input OR the other is TRUE, not both. An EOR gate with inputs A and B produces the Boolean function: $\overline{A}.B + A.\overline{B}$

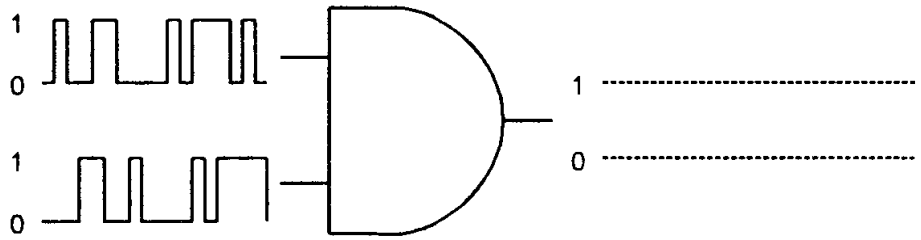
i. EOR gates occur frequently in signal processing. Simple examples of their use are: a NON-EQUIVALENCE gate, a MODULO-2 adder and a WORD INVERTER.

Student Examples.

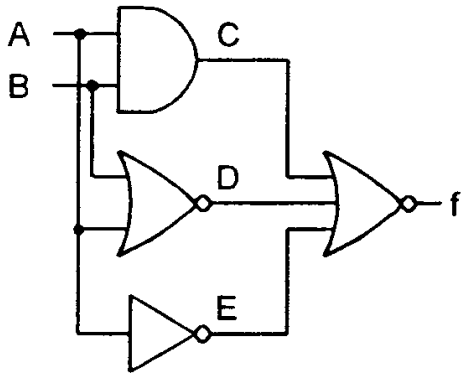
1. A particular, non-standard logic circuitry has voltage outputs which are either -3V or -9V. If positive logic is being used which level would represent logic '0'?

2. Explain the difference between combinational and sequential logic.

3. Draw the signal diagram at the output of the following AND gate given the input signals shown:



4. Complete the Truth Table for the following circuit:

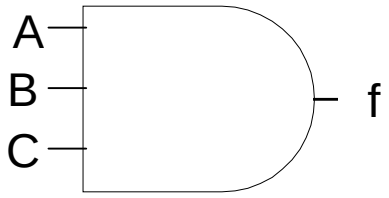


A	B	C	D	E	f
0	0				
0	1				
1	0				
1	1				

Answers to Problems and Student Examples.

Page 2-11, problem: The 3-input AND gate.

Gate Symbol



Truth Table

A	B	C	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

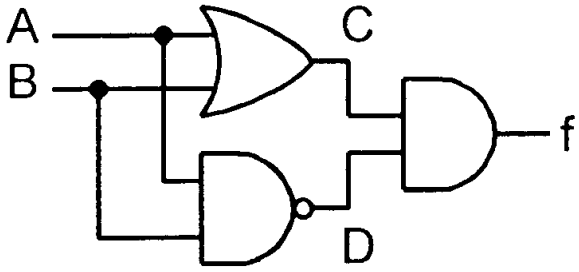
Boolean

$f = A \cdot B \cdot C$

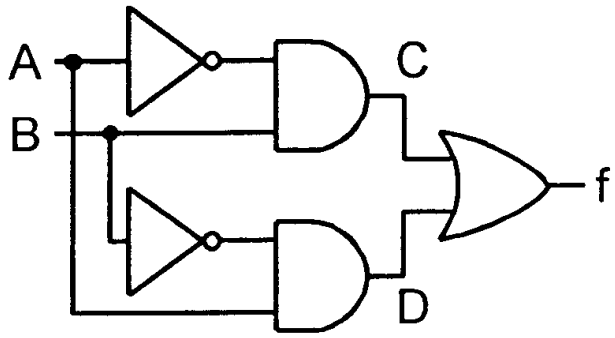
AND

Page 2-13, problem: Boolean expression for 5-input NAND gate $f = \overline{A.B.C.D.E}$

Page 2-18, EOR configurations:



A	B	C	D	f
0	0	0	1	0
0	1	1	1	1
1	0	1	1	1
1	1	1	0	0



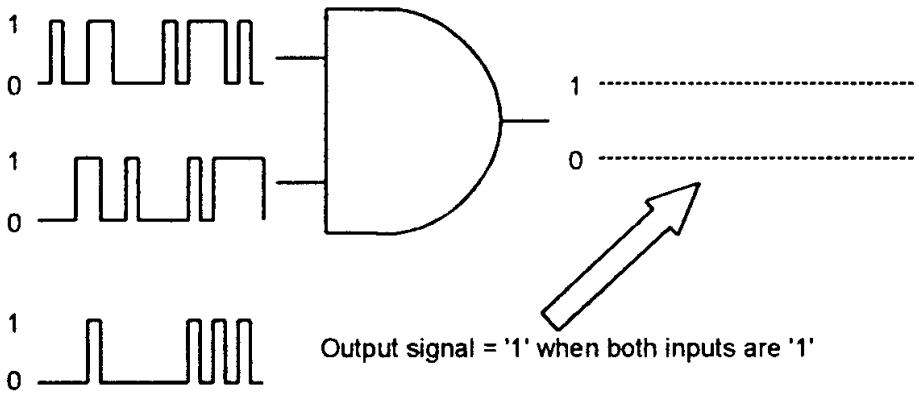
A	B	C	D	f
0	0	0	0	0
0	1	1	0	1
1	0	0	1	1
1	1	0	0	0

Page 2-20, Student Examples:

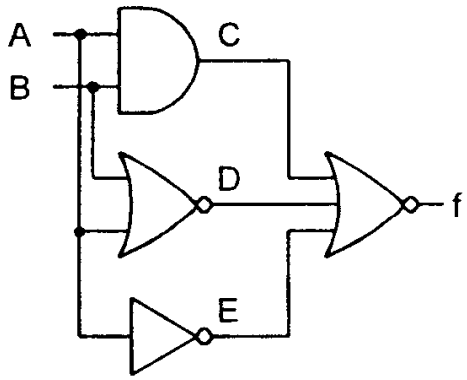
1. -9V, the lower voltage level.

2. See page 2-4.

3.



4.



A	B	C	D	E	f
0	0	0	1	1	0
0	1	0	0	1	0
1	0	0	0	0	1
1	1	1	0	0	0

Section 3

Sequential Logic
Bistables, Counters and Registers

CONTENTS

	Page
OBJECTIVES	3-2
Sequential Logic Circuits	3-3
Introduction	3-3
The JK Bistable	3-4
Operation of the JK	3-4
Toggle Mode	3-5
Set and Clear Terminals	3-5
The Binary Counter	3-7
Binary Count and Waveform Interpretation	3-7
Waveform Generation	3-7
A 4-bit Counter	3-8
Registers	3-9
Introduction	3-9
Serial and Parallel Registers	3-9
The Serial Register	3-9
Characteristics of the Serial Register	3-12
The Recirculating Shift Register	3-13
The Parallel Register	3-13
Characteristics of the Parallel Register	3-14
Hybrid Registers	3-14
Examples	3-17
Solutions to Student Problems and Examples	3-18

SEQUENTIAL LOGIC, BISTABLES, COUNTERS AND REGISTERS

OBJECTIVES

On completion of the material in this section the student should be able to:

a. Differentiate between Sequential and Combinational logic circuits.

b. Identify the need for some form of memory element in a sequential logic circuit.

c. Draw the circuit symbol and complete a state table for the JK Bistable.

d. Identify JK operating modes of: data store, quiescent state, toggle mode.

e. Identify the symbology used to represent clock inputs for devices using edge triggering.

f. Explain the Toggle Mode of Bistable operation and how this produces a frequency divide-by-two circuit.

g. Explain the functionality of set and clear terminals.

h. Explain how frequency divide-by-two circuits can be cascaded to produce a binary counter.

i. Draw the circuit diagram of a Binary Counter and label input clock and output count terminals.

j. Explain how a JK bistable can be configured to store the data bit from a single data input by placing an inverter between J and K inputs.

k. Draw the circuit of a serial shift register using JK bistables and explain how data is clocked in using a common clock line.

l. Draw the schematic diagram of a re-circulating shift register and explain its operation.

m. Draw the circuit diagram of a parallel register using JK bistables and identify inputs, outputs, write signal and read signal.

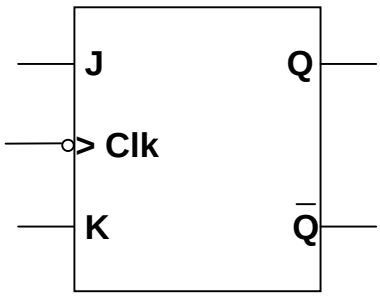
n. Draw the schematic circuit diagram of a hybrid register and identify the functionality of each pin.

o. Explain how the hybrid register can be used to produce serial-to-parallel and parallel-to-serial conversion.

SEQUENTIAL LOGIC CIRCUITS

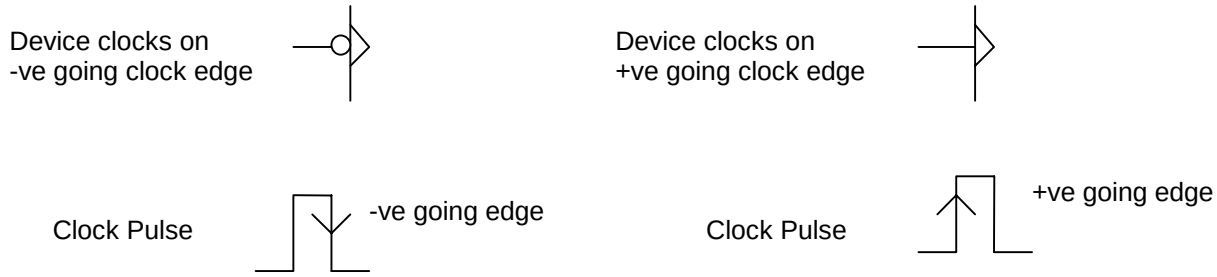
1. Introduction. As mentioned in Section 2, page 2-6, in the sequential logic circuit there are 2 essential changes from the pure combinational logic circuit. Firstly, there is more than likely a **clock** signal involved to trigger the output sequence; and secondly there is the need for a **memory** element. The memory element is otherwise called a **bistable** or **flip-flop** device and it is a fundamental component of many common circuits that you will encounter in your studies. In computing it is the basic storage element in a **RAM memory device,** it is also the fundamental component in 2 important circuits: the **counter** and the **register**. It is necessary to understand these circuits for aspects of computer architecture and also for understanding concepts in data communications and networking. I will therefore start by describing the functionality of a flip-flop and continue to cover features of counters and registers.

2. The JK Bistable. The diagram below is the standard symbol for a JK bistable or flip-flop (the letters JK simply come from the name of its designer - James Knapp).



The logic gates contained within this package do not concern us, but what is important is the device functionality - ie how it responds to different input signals. Pins labelled J and K are signal input pins, pins labelled Q and $\overline{Q}$ are outputs where output $\overline{Q}$ is the logical inverse of Q (ie if Q = '1' then $\overline{Q}$ = '0'). The name **BISTABLE** comes from the fact that this device has 2 stable output states: Q = 1, $\overline{Q}$ = 0 and Q = 0, $\overline{Q}$ = 1. Finally, the pin labelled Clk is a clock input.

3. Operation of the JK. Firstly data input on J and K only affects the output when the device is clocked. The clock symbology is important and is to be interpreted as follows:



Next, the way the output changes is best shown in a **state** table for the device. The state table is like a truth table but shows, for any inputs and present output, what the next outputs will be after the clock pulse:

Inputs		**Present Outputs before clock**		**Next Outputs after clock**		**Comments**
J	K	Q	$\overline{Q}$	Q	$\overline{Q}$	
0	0	0	1	0	1	
						Outputs unchanged
0	0	1	0	1	0	
0	1	0	1	0	1	
						Output Q cleared to 0
0	1	1	0	0	1	
1	0	0	1	1	0	
						Output Q set to 1
1	0	1	0	1	0	
1	1	0	1	1	0	
						Outputs toggle ie change state
1	1	1	0	0	1	

We see from the above, 3 effects on the output:

If J = 0 and K = 1 then after the clock is applied output Q is 0 ie we are storing a logic '0' at output Q.

If J = 1 and K = 0 then after the clock is applied output Q is 1 ie we are storing logic '1' at Q.

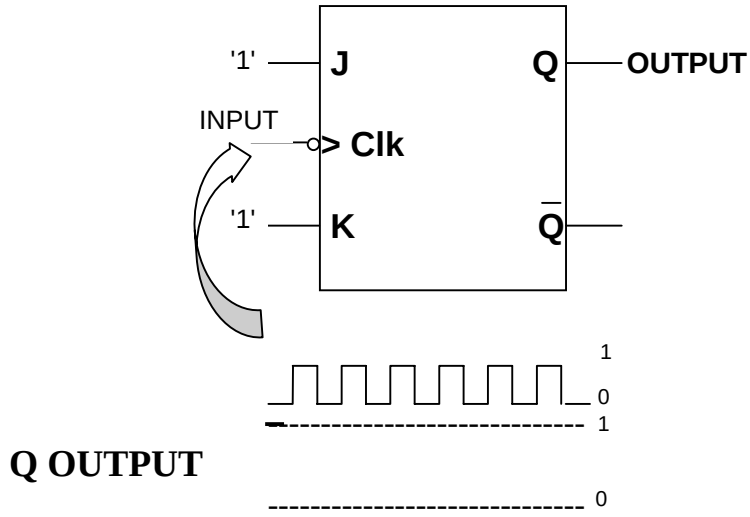
Both these states show the device acting as a **memory store** and they are the basis of operation of a computer memory device storing 1 bit of information. For 8 bits we need 8 bistables. In addition, this action is the basis of operation of a **register** which is a simple storage device for small amounts of data, say a single byte. We will see this later.

Alternatively, if J = K = 0 then after the clock pulse there is no change in the data stored - this is the **quiescent or rest** state.

Finally, if J = K = 1 then after the clock is applied the output changes to the opposite state, we say that it **toggles.** This first appears a little odd, however, it is most useful and is the basis of operation of a **counter** as we will see shortly.

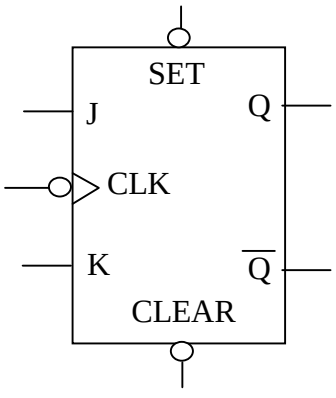
4. TOGGLE Mode.

A circuit is said to TOGGLE when its output alternates between the logic '0' and '1' states as it is clocked. We can arrange this effect by placing logic '1' on both J and K and allowing the circuit to clock freely:



Given the clock input signal shown in the diagram above, draw the output signal seen at Q, assuming that Q starts in the '1' state shown. What is the effect of this circuit? See solutions section for the answer. This is a characteristic of the TOGGLE mode of a JK device.

5. Set and Clear Terminals Some JK devices, and other circuits such as counters and registers, are available with SET and CLEAR terminals. A JK of this type is shown below:



First a note on symbology:

A terminal with a circle means that the terminal is active when a logic low (‘0’) is applied. Sometimes this condition is shown by the terminal name having a line above it with no circle, eg:

$\overline{\text{Clear}}$

This symbology is often used on counters or registers and means that the terminal is *active low*.

A terminal with no circle means that the terminal is active when a logic high (‘1’) is applied. It is said to be *active high*.

The operation of SET and CLEAR terminals is simple:

a. Set: When the set terminal is activated (in the case above, a logic '0' is applied) the Q output is automatically set to the '1' state irrespective of the clock. The $\overline{Q}$ output will then go to logic '0'. The outputs then stay in this state until some other signal, clock or clear, changes them.

b. Clear: When the clear terminal is activated (in the case above, a logic '0' is applied) the Q output is automatically cleared to the '0' state irrespective of the clock. The $\overline{Q}$ output will then go to logic '1'. The outputs then stay in this state until some other signal, clock or set, changes them.

Since these terminals function *irrespective of the clock*, they are said to be **asynchronous**. That is to say, their effects are not synchronised to the clock – they take effect immediately they are applied and, furthermore, they over-ride any clock signal. If a clock signal is applied whilst the clear terminal is active then the output would remain cleared.

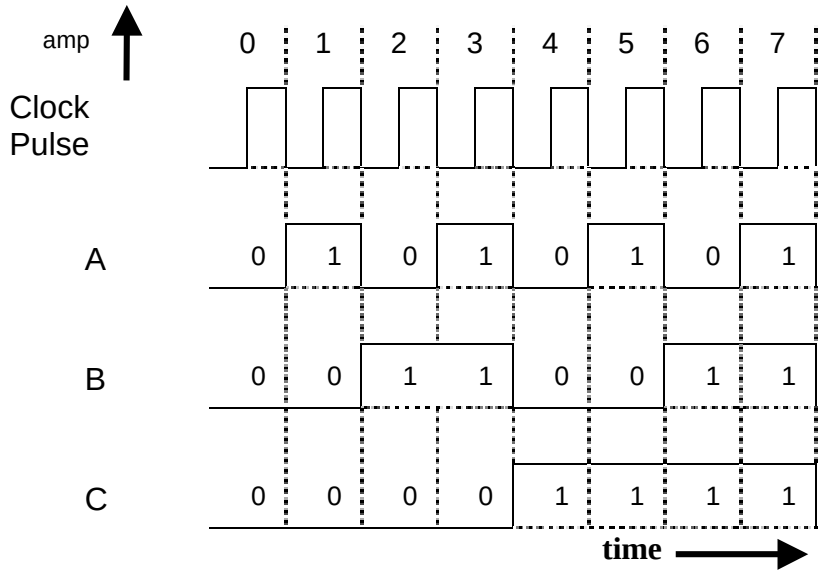
THE BINARY COUNTER

6. Binary Count and Waveform Interpretation. In the diagram below I show on the left a 3-bit binary count from 0 to 7. The 3 bits of the count are labelled C, B, A, where A is the LSB. On the right of the diagram I show each bit C, B, and A displayed as a waveform of amplitude, vertically, against time, horizontally. In addition, I show the clock pulse which we can imagine causing each change of count value. If at any instant along the horizontal axis we look vertically we see the voltage amplitudes at C, B and A. In each case they correspond to the binary count value at that time.

BINARY COUNT SEQUENCE

Count	C	B	A
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1

WAVEFORM INTERPRETATION



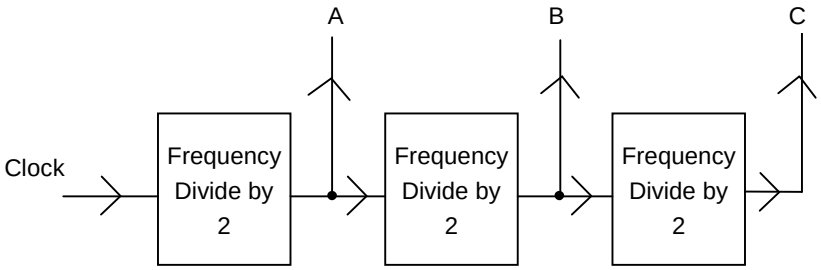
7. Waveform Generation. If we look at each waveform individually, starting with the Clock waveform, then A, B and C, we see the following relationships:

waveform A varies at HALF THE FREQUENCY of the clock

waveform B varies at HALF THE FREQUENCY of waveform A

waveform C varies at HALF THE FREQUENCY of waveform B

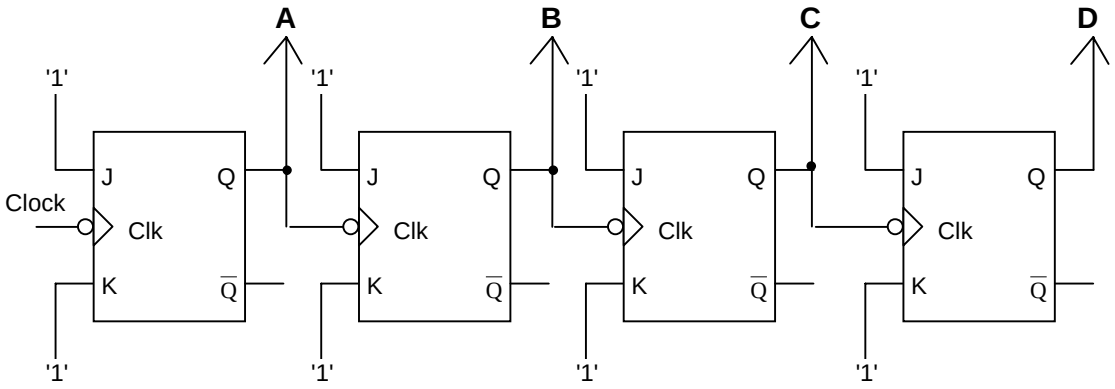
The result is clear - we can build a simple binary counter by simply cascading "FREQUENCY DIVIDE-BY-2" circuits as follows:



Also, as we have seen in para 4 above, the frequency divide by 2 circuit is easily made by configuring a JK in toggle mode.

8. A 4-bit Counter. A 4-bit counter can thus be constructed using JK devices as shown below:

4-bit Counter



Points to note in the counter circuit:

a. The input signal to the circuit is applied to the CLOCK (Clk) terminal on the first bistable. Any pulse inputs which are to be counted are applied to this input terminal.

b. Each bistable is configured in toggle mode, ie as frequency ÷ 2.

c. The Q output from each bistable is connected to the clock input of the following bistable in the chain. Hence each stage clocks the next stage, and so on down the chain.

d. The 4-bit count can be viewed by monitoring the 4 outputs. For convenience we might connect each output to a light emitting diode to get a visual display of the count.

e. The output 4-bit number is DCBA, where D is the MSB and A is the LSB.

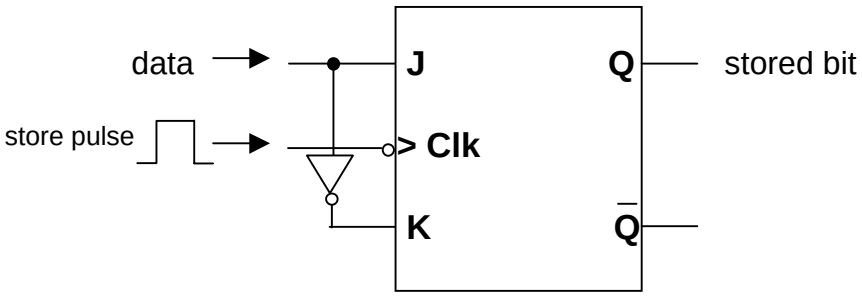
f. We can extend the count range by simply adding more bistables to the chain.

Student Problem: how many JK bistable devices will be required to build a counter to count from 0 to 63?

REGISTERS

Introduction.

1. Earlier we developed the concept of a Flip-Flop or Bistable circuit as a device which can hold or store one bit of data. Using a JK device we can arrange that data is stored on the falling (or rising) edge of a controlling clock pulse. Shown below is one arrangement of the JK which can be used to store 1-bit of data which is presented to the device on a single data line:



The data bit must be present and stable on the data line before the clock input is activated and the data bit will then be stored at Q. Note that an inverter is required to manufacture the logical inverse of the input data bit for the K input (ie if J = 1 then K must = 0 and if J = 0 then K must = 1).

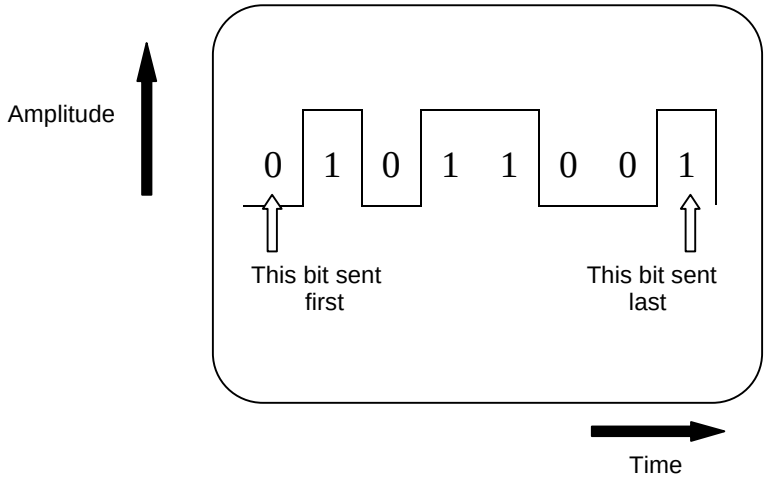
2. There are many occasions when we require to store a number of bits. In this case we will require a number of bistables all linked in some way to store data as it arrives at the data input(s). A device which will hold a number of data bits is called a **REGISTER**. We build a register using a number of bistables. There are 2 principle forms of register, each is named after the manner in which data is presented to the device. These are the **SERIAL** register and the **PARALLEL** register.

Serial and Parallel Registers.

3. The Serial Register. Serial data is delivered with each bit of the data word following in sequence along a single data line, or pair of lines since there must always be an earth, or common, line. In our diagrams we will assume that the common line is connected and always show the data line on its own. If we view serial data on an oscilloscope then it might appear as shown on the next page. Note that the oscilloscope displays signal amplitude against time so the first bit sent out is the one which appears earlier on the display. It is normal to transmit data "LSB first", so the diagram on the following page shows the byte

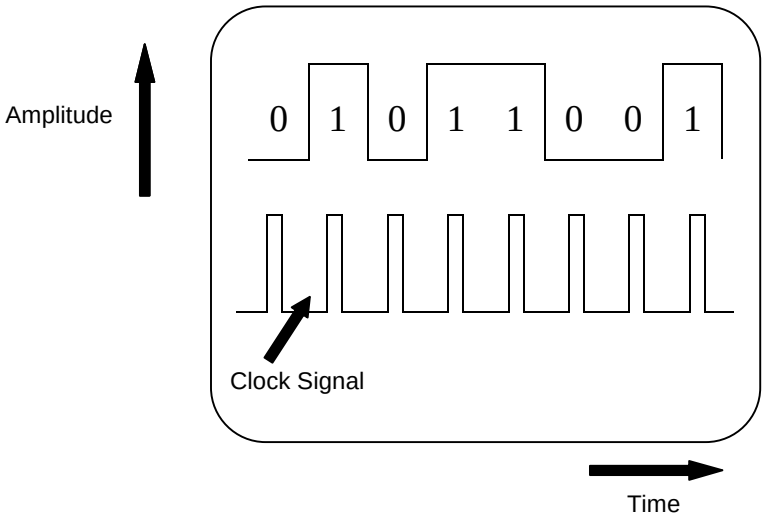
1 0 0 1 1 0 1 0

being transmitted serially.



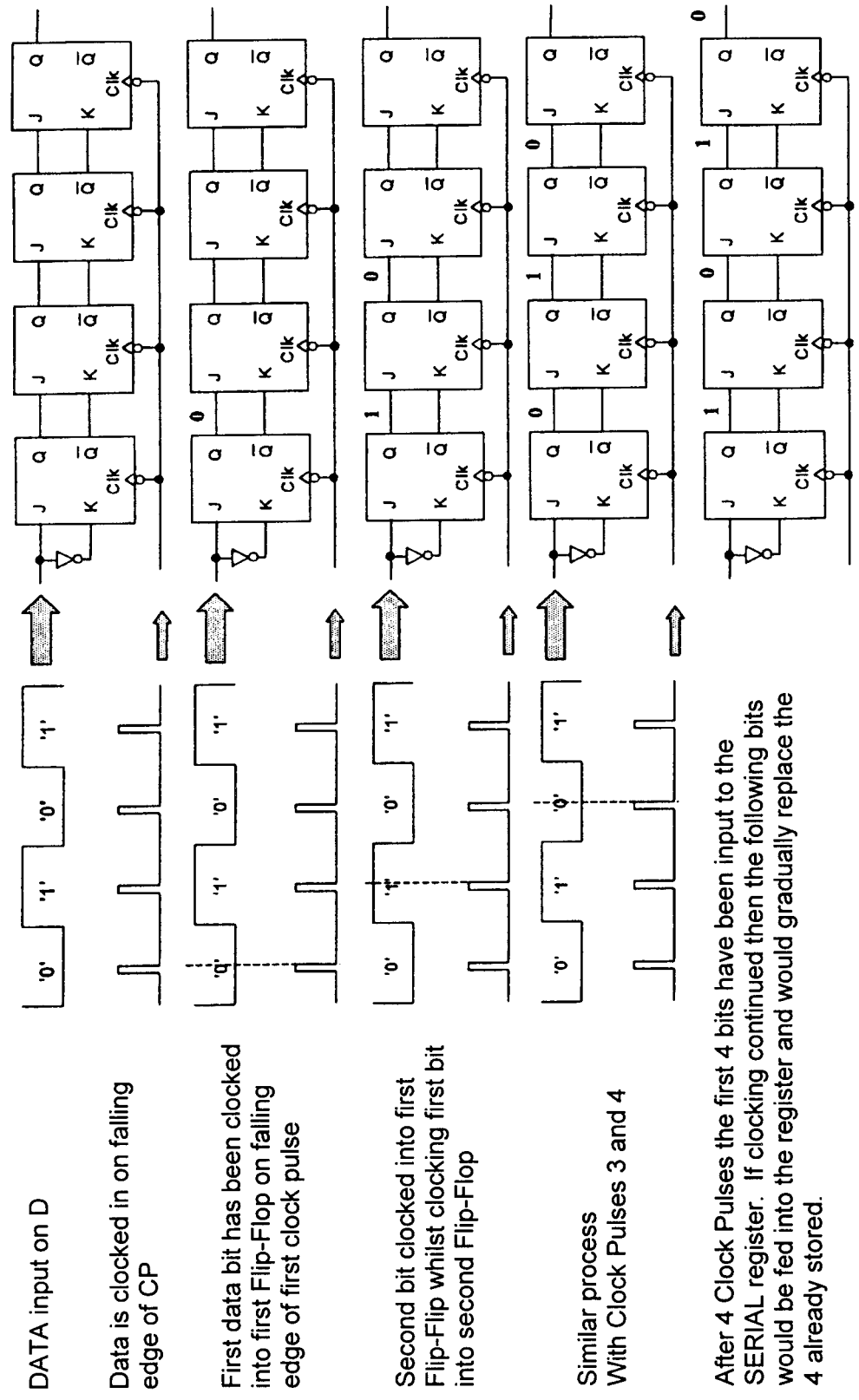
Serial Data Viewed on Oscilloscope

If we wish to store each bit of the data in a bistable then we need to apply the data to the data input and clock the bistable when each bit is stable. This is best done at the 'centre' of each bit. For this purpose we must supply clocking pulses at the correct time. A suitable clock is shown below on the second oscilloscope trace. Notice that the falling edge of each clock pulse is at the centre of each data bit:



Data with Bistable Clock Signal

To store the whole byte we must provide 8 bistables in cascade so that, as a new bit arrives to be stored in the first bistable the bit already there is passed on to be stored in the second, and so on down the chain of devices. The sequence of activity for receiving and storing 4 data bits in 4 JK bistables is shown below (note that the JK symbol has been slightly reconfigured to simplify the diagram):



You should note the following points in the serial diagram:

a. The first JK has been configured with an inverter to ensure that K is the inverse of J at the input.

b. Clocking data through any device is not instantaneous. There is thus a short time delay between the active clock edge and data changing at each bistable output - this is called the propagation delay.

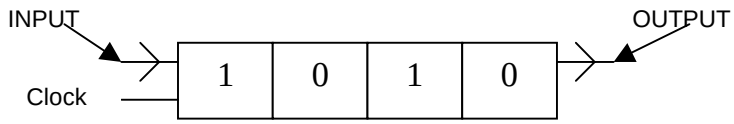
c. All the bistable clock inputs are connected in common so that, on each clock pulse falling edge, each bistable clocks through the data bit **presently applied** to its data input.

d. The data and clock signals are shown as though displayed on an oscilloscope, hence the earlier events are on the left hand side of the diagram.

e. Data is clocked into the input bistable and along the chain on the falling edge of the clock pulse. In the diagram I show this using a dotted line which progresses to the right as time advances.

4. Characteristics of the Serial Register. The serial register is also called a **SHIFT** register since when data in input to the register it shifts along the bistable chain. For simplicity the serial shift register can be viewed as shown below where each box represents one bistable holding one bit of data:

Simple representation of a Serial Shift Register



From the operation of the serial shift register considered above you should note the following characteristics:

a. Memory elements are INTERDEPENDENT. That is to say, data passes through different elements on its way into the register.

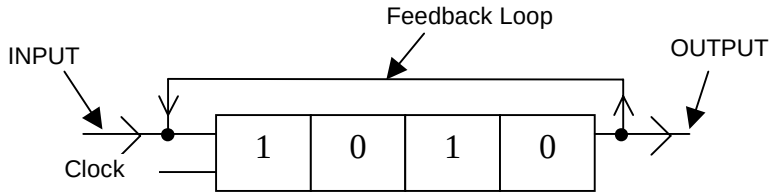
b. There is a single input line and a single output line.

c. For a data word of n bits we require n clock pulses to load the data into the register.

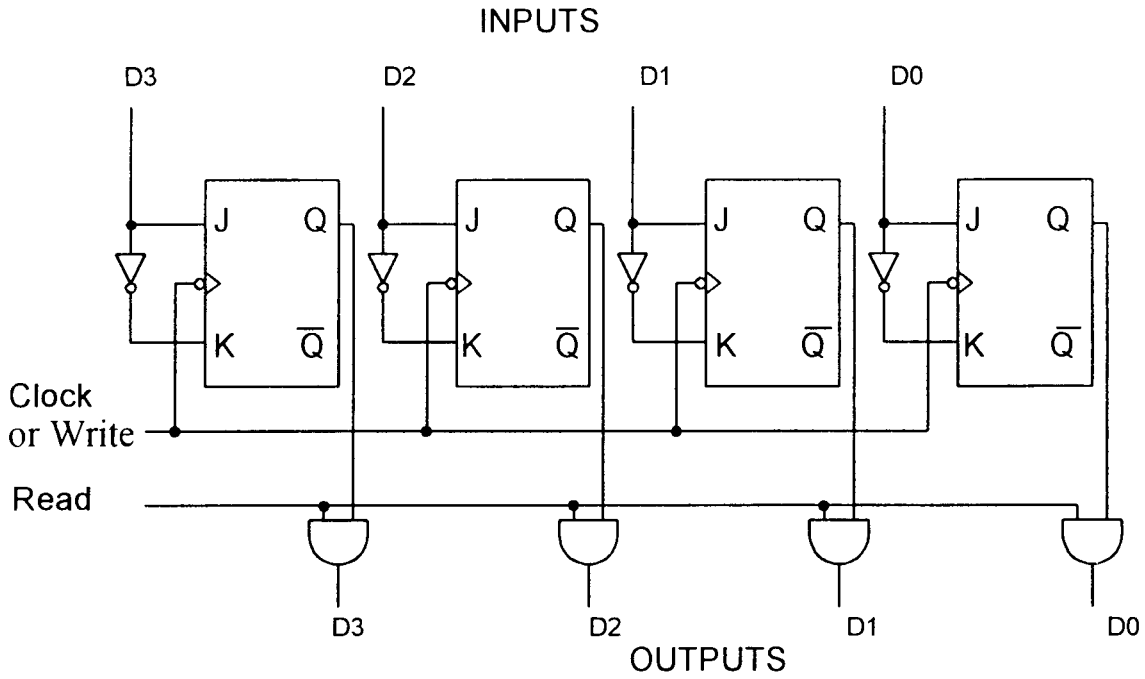
d. Normally the read-out process is destructive since data bits over-write one another as they pass down the register.

5. The Re-circulating Shift Register. We can overcome the problem in sub para 4d above by arranging for data to re-circulate in the register during read-out. We do this by feeding the output back to the input during read-out:

The Re-circulating Shift Register



6. The Parallel Register. In the parallel register data is presented in parallel form, that is to say, each bit of the data word has a separate line so that all bits are presented at the same instant to a set of bistables. A simple form of parallel register is shown below:



In the circuit shown above you should note the following points:

a. Each JK bistable has been configured to store the bit of data presented on the single input line.

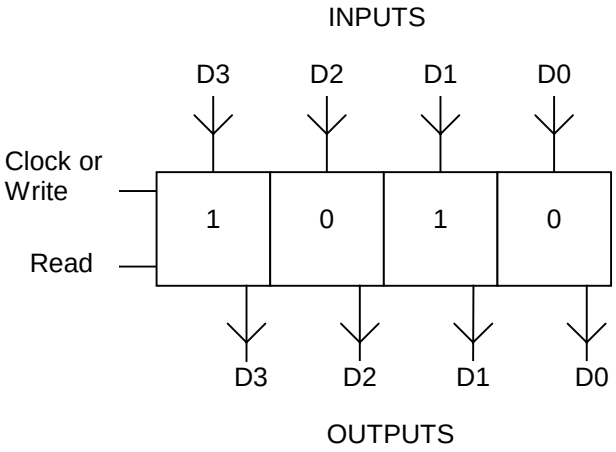
b. Data is stored in the register by clocking all the bistables at the same instant. This clock signal may be considered to be a **write** signal since it will write data into the register.

c. Each data bit is stored at the Q output.

d. Each Q terminal is fed to the parallel output.

e. In this circuit I have added the 'extra' **READ** control line so that all output lines stay at logic '0' until a READ pulse is applied. The data stored in the register is presented on the output lines for the duration of the READ pulse, ie whilst READ is high.

7. Characteristics of the Parallel Register. As with the Serial Shift Register we can represent the parallel register as a simple block diagram as shown below:



From the operation of the parallel register considered above you should note the following characteristics:

a. Memory elements are INDEPENDENT. That is to say that for a true parallel register there is no data path between neighbouring memory elements.

b. For an n-bit data word the parallel register requires n I/O lines.

c. The register is fast loading. All bits of the data word are stored (or written into the register) with one clock, or write, pulse.

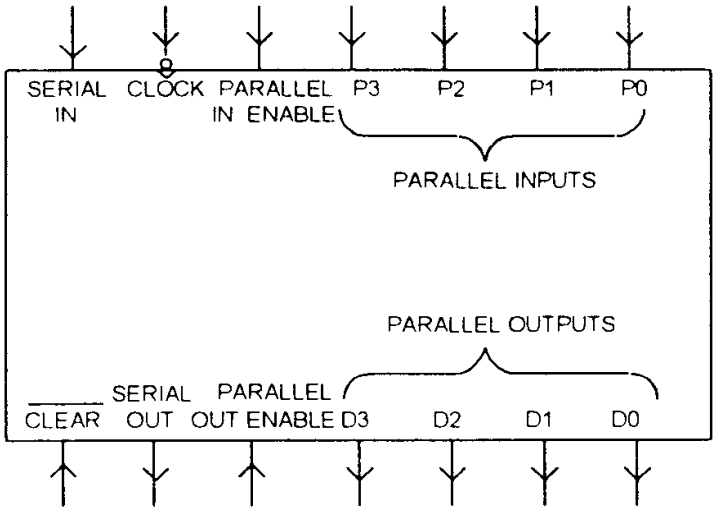
d. The read-out is in parallel and is non-destructive. That is to say, when data is read out it is **copied** to the new location rather than physically transferred.

8. Hybrid Registers. Hybrid registers can also be built. These are registers which can have Serial and/or Parallel inputs and outputs. Some common terms which you will meet with these registers are:

SIPO - Serial In, Parallel Out

PISO - Parallel In, Serial Out

Shown below is a simplified diagram of a register which has both Serial and Parallel inputs and outputs:



Note the following operating points for this circuit:

a. To load parallel data:

(1) The 4-bit parallel data is applied to inputs P3-P0.

(2) A logic '1' level is applied to PARALLEL IN ENABLE.

(3) A single clock pulse is applied to the CLOCK input to clock the parallel data into the register.

b. To load 4-bit serial data: a '0' is applied to PARALLEL IN ENABLE (to disable parallel input), the serial data is input at SERIAL IN and 4 clock pulses are applied to the CLOCK input.

c. To read data out in serial: a '0' is applied to PARALLEL IN ENABLE, the clock is applied to the CLOCK input and the serial data is output from SERIAL OUT.

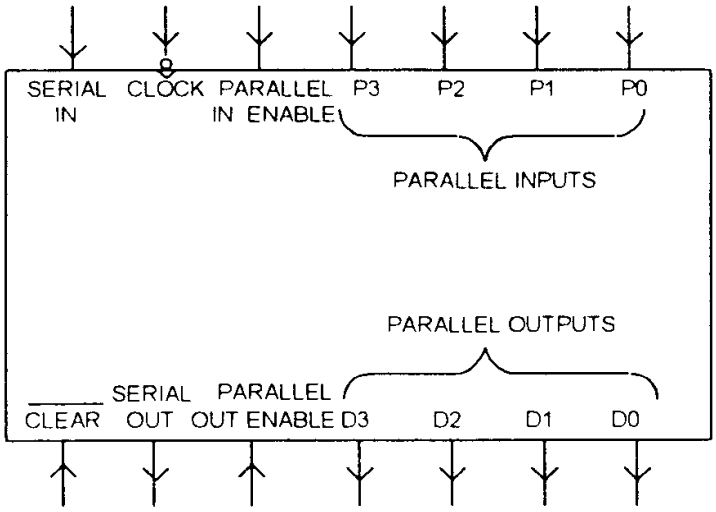
d. To read data out in parallel a logic '1' is applied to PARALLEL OUT and the 4-bit parallel data can be sensed at D3-D0 for as long as the '1' is applied.

e. The data may be clocked-in in serial form and read out in parallel form by activating pins in correct sequence. This is **serial-to-parallel** conversion.

f. The data may be written-in in parallel form and clocked out in serial form by activating pins in correct sequence. This is **parallel-to-serial** conversion.

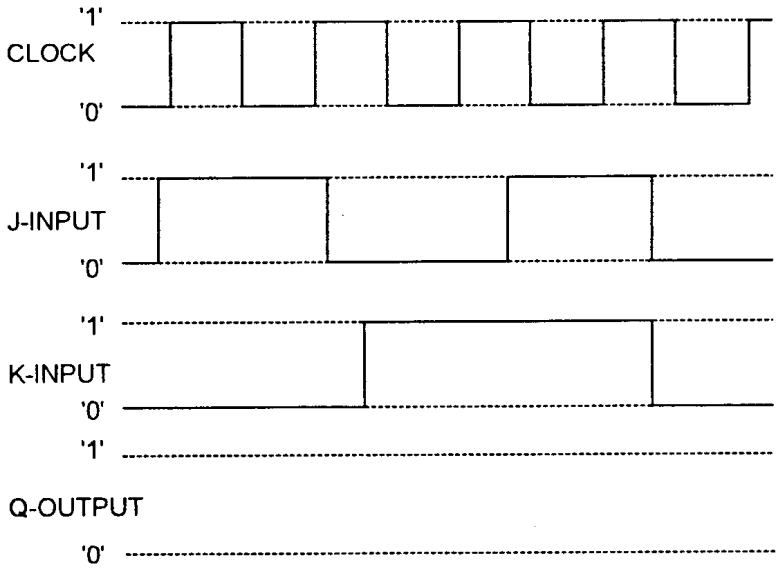
One particular use for a device such as this is to convert the data form from parallel to serial, or vice versa. To convert from parallel to serial format the data is presented to the device on lines P3 to P0 and stored in parallel form. It is then clocked out of the Serial Out line in serial form. This sort of facility must be provided for data which is to be transmitted in serial form, eg down a telephone line. You can deduce the necessary procedure for conversion from serial to parallel data format.

9. The block diagram below shows the parallel/serial register considered above. Itemise, in sequence, the steps you would take to load the nibble 0110 in parallel and then read-out the nibble in serial form.



EXAMPLES

1. The diagram below shows the signals at the clock and J and K inputs of a negative edge triggered, JK bistable. Sketch the waveform appearing at the Q output assuming that Q is initially '0'.



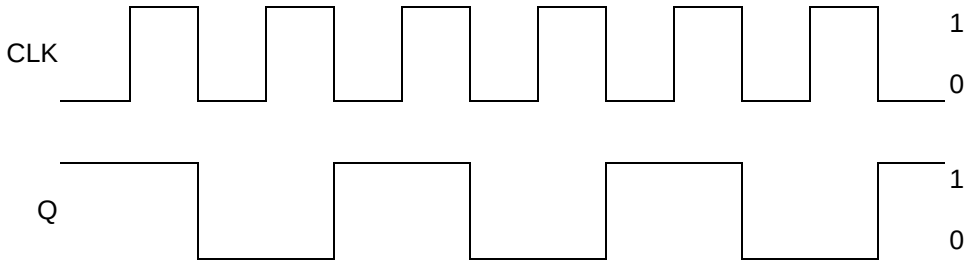
2. A circuit is required to take an input signal and divide its frequency by 16. Draw a suitable circuit diagram showing input and output signals. What other purpose might this circuit serve?

3. Draw the circuit diagram of a 4-bit serial shift register using negative edge triggered JK devices. Label input, output and clock lines.

4. A standard, 8-bit serial shift register, holds the data byte 6A hex. Draw the data waveform diagram (Amplitude vs Time) which will appear at the output as the data is clocked out using a regular clock waveform. Assume that logic '0' is applied to the shift register input for the duration of the read-out.

Solutions to Student Problems and Examples.

Page 3-5, para 4:



The output at Q changes level (toggles) on the negative going edge of the clock input. The effect is to produce an output at half the clock input frequency. This is a frequency divide-by-2 circuit.

Page 8, Student Problem.

The maximum count is 63 (decimal) which is 111111 in binary. The counter must therefore have 6 bits to hold the whole count. Hence 6 bistables will be required.

Pages 15/16 parallel/serial register operation.

Sequence of events as follows:

a. Data nibble 0110 applied to P3 P3 P1 P0 inputs.

b. PARALLEL IN ENABLE input raised to logic '1' to enable parallel input.

c. A single clock pulse applied to CLOCK to clock-in the parallel data.

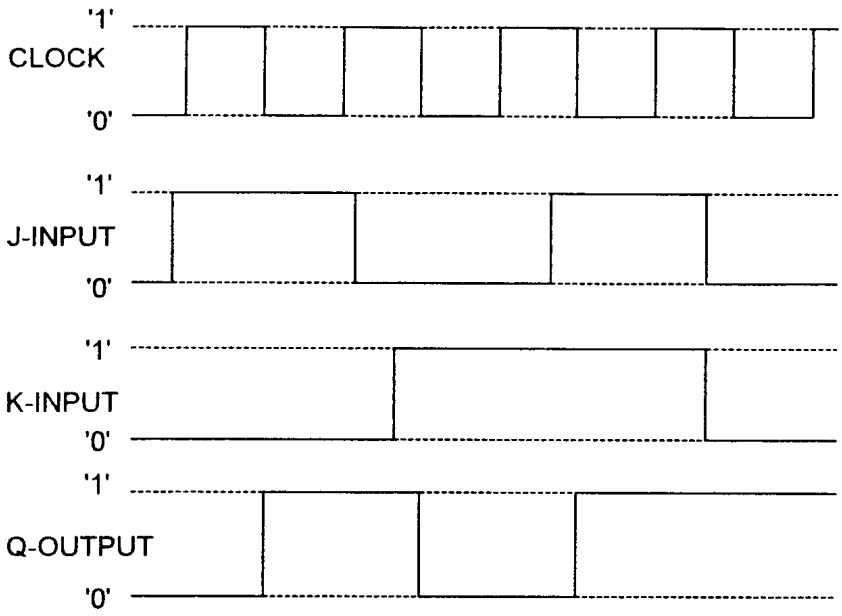
d. PARALLEL IN ENABLE returned to logic '0' to disable any further parallel input.

e. Clock pulses applied to CLOCK line to clock data out from SERIAL OUT.

Note: if, at the same time the data is not to be seen at the parallel output then PARALLEL OUT ENABLE would be held at logic '0'.

Examples page 17:

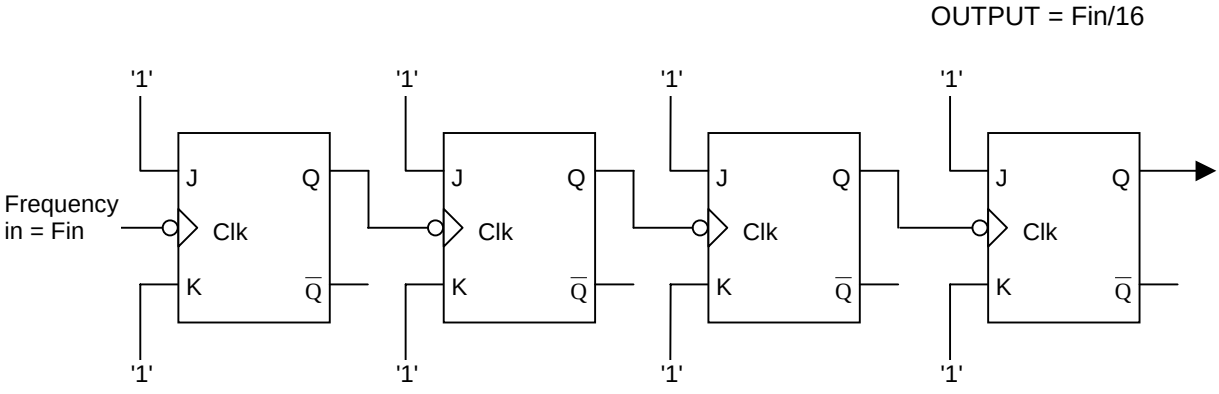
1.



The output is initially '0'.

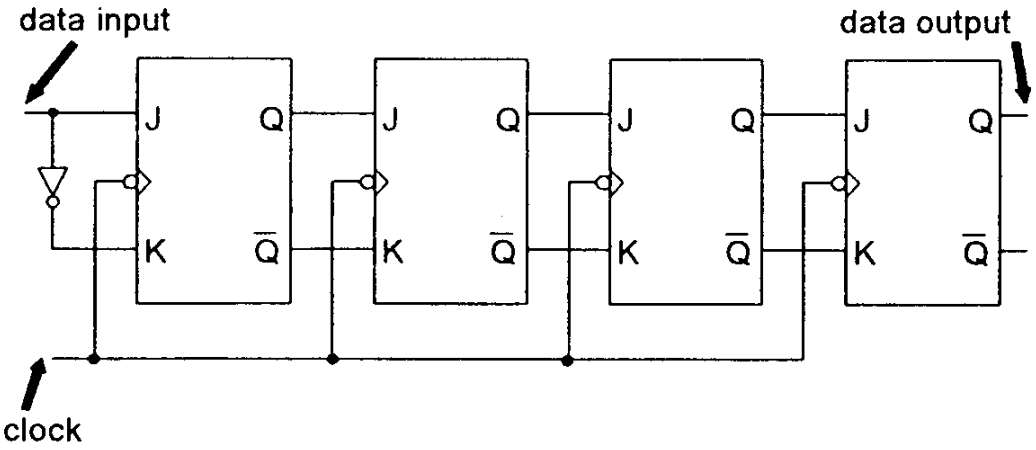
On the first -ve clock edge J = '1', K = '0' so Q sets to '1'
On the second -ve clock edge J = '0', k = '1' so Q resets to '0'
On the third -ve clock edge J = '1', K = '1' so Q toggles to '1'
On the fourth -ve clock edge J = '0', K = '0' so Q stays unchanged at '1'

2. To produce divide-by-16 we require 4 JKs each in toggle mode:



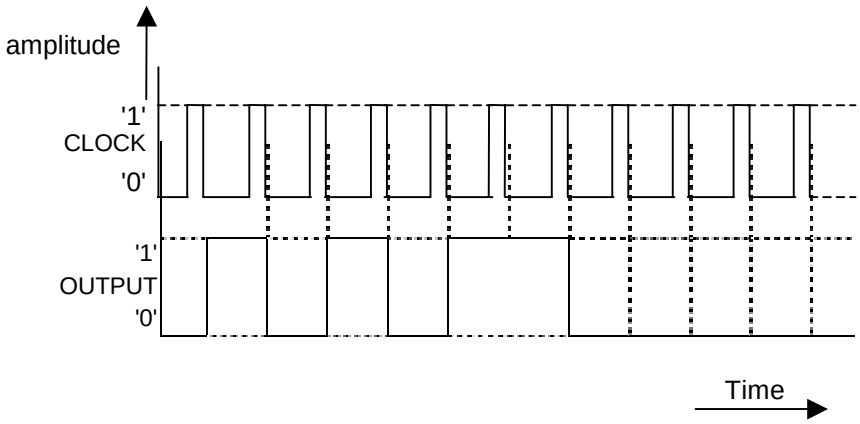
The circuit will also function as a 4-bit counter taking 4 output bits from each Qout.

3. 4-bit serial shift register:



note: inverter required on first JK but not on others since $\overline{Q}$ provides logical inverse of Q.

4.



Note: -ve edge triggered devices assumed.

Hexadecimal 6A is binary 01101010. The first bit to leave the register will be the zero on the RHS (data goes out LSB first). Hence the pattern shown emerges from the shift register.

Section 4

Microprocessors

CONTENTS

	Page
Introduction to Microprocessors	4-2
Computer System Components	4-4
CPU	4-4
Clock	4-4
Bus and Control Lines	4-5
Memory	4-6
Main Memory Devices	4-7
System Boot-Up Program	4-7
Input/Output Interfaces	4-8
The Microprocessor	4-9
Introduction	4-9
The Internal Architecture of the Microprocessor	4-9
Addresses and Data Buses	4-10
Multiplexers	4-10
Program Counter	4-10
Instruction Register	4-10
Instruction Decode	4-11
Control and Timing	4-11
Arithmetic and Logic Unit	4-11
Accumulator	4-11
Buffer Register	4-11
Flags or Status Register	4-12
Working Registers	4-12
Memory Address Register	4-12
The Format of Microprocessor Instructions	4-13
Introduction	4-13
Instruction Format	4-13
An Instruction Example	4-13
The Fetch-Execute Cycle	4-14
Summary	4-20
Programming Terminology and Instruction Types	4-21
The Computer Program	4-21
Instruction Groups	4-21
Methods of Representing Instructions	4-22
Assembler Code	4-23
Operands in Assembler Code	4-23
The Instruction Set	4-24
Solutions to Student Problems	4-28

INTRODUCTION TO MICROPROCESSORS

1. The microprocessor is one of the major developments of the last 30 years. It affects our lives in many ways that we are probably not even aware of, from everyday devices such as televisions, washing machines, car engines, alarm systems and personal computers, to more esoteric applications as satellite control, aircraft systems control, data communications and the powerful Cray computer.

2. In a similar way the microprocessor (or micro for short), plays a key role in many features of Royal Signals Systems and equipment. From data communications switches, used in Ptarmigan, to local area networks, radio sets and, of course, the much used PC. Indeed, to understand how computers work, their capabilities and limitations, how operating systems are designed and what features make them reliable and safe, an understanding of the microprocessor is required. To be more topical and up-to-date, it is undoubtedly the capabilities of the microprocessor which have led to the whole concept of 'Digitisation of the Battlefield' - without micro technology underpinning all the digital equipment, the concept would never have been dreamt of.

3. In this section we will consider some basic features of microprocessors to provide you with some familiarity before attending the course. On the course we will pick up from where this section leaves off and continue to develop principles of **Computer Architecture** and how they affect **Operating Systems** design. In the course of this study we will look at features of operating systems like MSDOS, UNIX and Windows to enable you to understand the advantages and limitations of each.

WHERE IT ALL BEGAN

4. Prior to 1971 the word microprocessor had not even been thought of. In this year, however, a (then) small company called Intel brought out the first device.

5. Intel had been given a contract to design the logic required for a small, hand-held calculator; they duly did the work and found that the device required a large number of basic integrated circuit components (ICs). This was a problem since at that time the demand for ICs had become so great that their supply was unreliable. In addition, there was a general idea floating around the academic world that IC technology was now sufficiently advanced that a simple computer, or at least the processing part, could conceivably be put on a single chip.

6. Having a computer on a single IC would have many advantages - only the supply of the single IC would be required, rather than many different ICs from many suppliers. Also, this new device would be very versatile - the same device could be programmed in different ways to perform different functions, and again this would reduce the supply problem - manufacturers only need one type of device rather than a store of many different components.

7. Needless to say, the fledgling company decided to take on the challenge of placing a computer processing section on a single chip - and they succeeded. The microprocessor was born and it was named the Intel 4004. The 4 in the name indicates that it could only deal with data words of size 4 bits - somewhat limited by today's standards, but it was only the beginning.

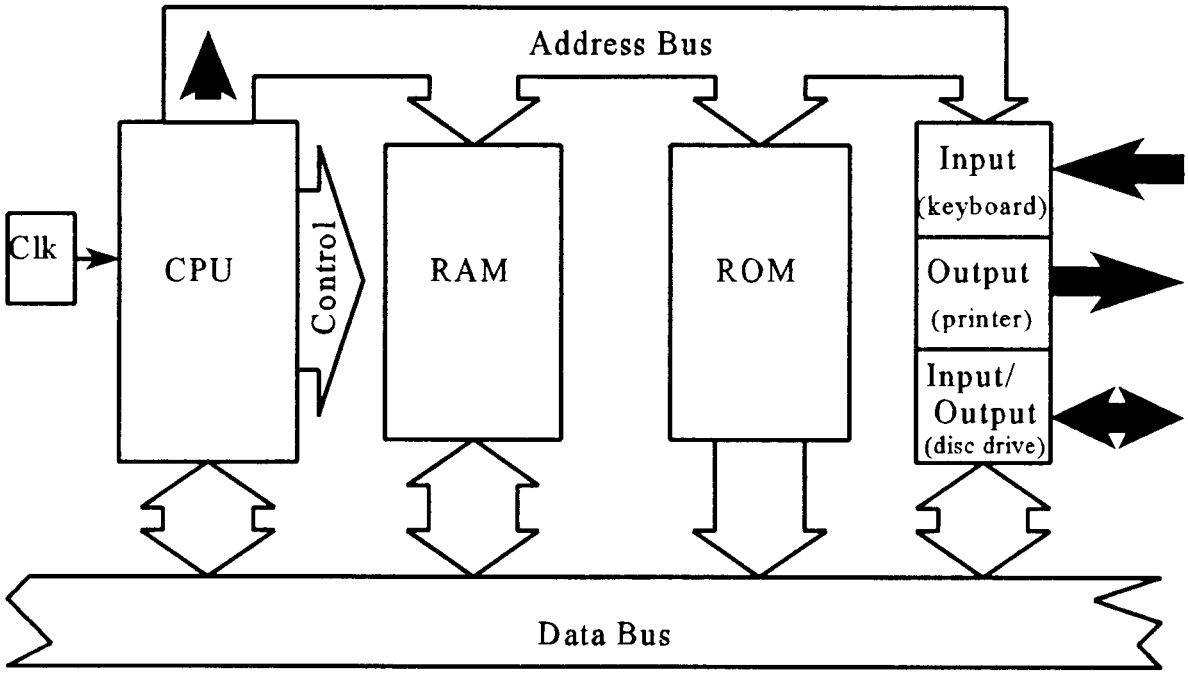
8. From these early beginnings the technology blossomed and competition between companies has produced a whole variety of micros with far greater capability. To give you some idea of how fast developments moved I list below some of the significant devices which have been produced and the companies involved. Note, in the microprocessor world some important characteristics are: word size (ie how big a number the processor can work on), memory size (measured in K, approximately 1000 locations, or M, approximately 1 million locations, or G, approximately 1 billion locations) and clock speed in MHz. We will cover these points in detail later but I use them below to indicate device capability.

Data	Manufacturer	Device	Word Size (bits)	Memory Size	Clock MHz	No of Transistors	
1972	Intel	4004	4	256	0.74	2300	
1974	Intel	8080	8	64K	2-4	4800	Became industry standard
1975	MOS Tech	6502	8	64K	2-4	4800	Used in BBC Micro
1976	Zilog	Z80	8	64K	2-4	8500	Used in Sinclair ZX 80
1978	Intel	8086/88	16	1M	4.77	29000	Used in the original IBM PC
1978	Motorola	68000	16	16M	6-8	68000	Used in Apple/Mac
1982	Intel	80286	16	16M	6-8	134000	Used in PC AT
1985	Intel	80386	32	4G	12.5-33	275000	
1989	Intel	80486	32	4G	33-66	1.18M	
1993	Intel	Pentium	64	4G	60-75	3.1M	
1998	Intel	Pentium	64	4G	3000		

9. We will now continue to consider some of the basic features of the microprocessor and microprocessor systems to give a basis from which we can develop computer architecture and operating systems design.

COMPUTER SYSTEM COMPONENTS

1. Any computer has a number of fundamental components which, when connected together in the correct way, enable the computer to function correctly. The basic parts of any computer system are shown below, and each is described in the following paragraphs.



2. **CPU**. The CPU is the Central Processing Unit; it is the brains of the computer system. In a microcomputer all functions of the CPU are carried out by a microprocessor - in the most modern IBM PC, for example, this would be a Pentium microprocessor. The CPU is responsible for a number of functions:

a. Controls the sequence in which program instructions are executed.

b. Decodes and interprets each instruction to understand what action is required.

c. Provides both internal and external control signals to execute any instruction.

d. Controls the flow of data and instructions on the buses.

e. Performs all arithmetic and logic operations.

We will consider in more detail how the micro achieves these functions in the next section.

10. **Clock (Clk)**. All timing and control in the CPU or micro is governed by the clock. This is normally a crystal oscillator which produces an accurate and stable timing waveform. The clock circuit is normally external to the CPU.

11. **Bus and Control Lines**. Communication with different elements in the computer system is carried out using buses and control lines. Note that the term 'bus' comes from the word omnibus indicating that it can be used by, or channel information to, or from, several users. The buses in a computer system are:

a. **Address Bus**. This consists of a number of parallel lines carrying address codes from the CPU to other system devices. There can be 16, 20, 24 or 32 depending on the processor. The number of address lines determines the number of memory locations that can be accessed. For example, if there is only one address line then this can only distinguish between 2 address locations: one location is accessed if the line carries a logic '0' and the other if the line carries a logic '1'. Two address lines can distinguish between 4 address locations for combinations of 00, 01, 10, 11 and 3 can distinguish 8 (000, 001, 010 etc to 111). The rule is clear:

number of memory locations is = 2^n where n is number of address lines.

Thus the Z80 micro has 16 address lines and so can access 2^{16} memory locations.

$$2^{16} = 65536$$

however, a more convenient way of expressing memory size is in K.

1K is the memory size for 10 address lines = 2^{10} = 1024. This is just greater than 1000 and so is given the name K - note capital K, not k.

$$\text{so } 2^{16} = 2^6 * 2^{10} = 64 * 2^{10} = 64K$$

The 8088 processor, which was used for the original IBM PC has 20 address lines, hence the address range for the PC was:

$$2^{20} = 2^{10} * 2^{10} = 1K * 1K = 1M$$

pronounced 1 Meg which is not 1 million but 1,048,576.

Student Problem: what is the address space for: (1) the 68000 microprocessor which has 24 address lines? (2) the Pentium which has 32 address lines?

b. **Data Bus**. This consists of a number of parallel lines along which the data bits pass to or from memory and I/O (input/output) devices. There can be 8, 16 or 32 data lines depending on the microprocessor. In a Z80 there are 8 data lines indicating that this is an 8-bit processor - it can only deal with 8-bits of data at any one time.

c. **Control Lines**. A number of control lines are provided to carry signals which control and synchronise the flow of data. There is no standard format for these control signals and they will vary between microprocessors.

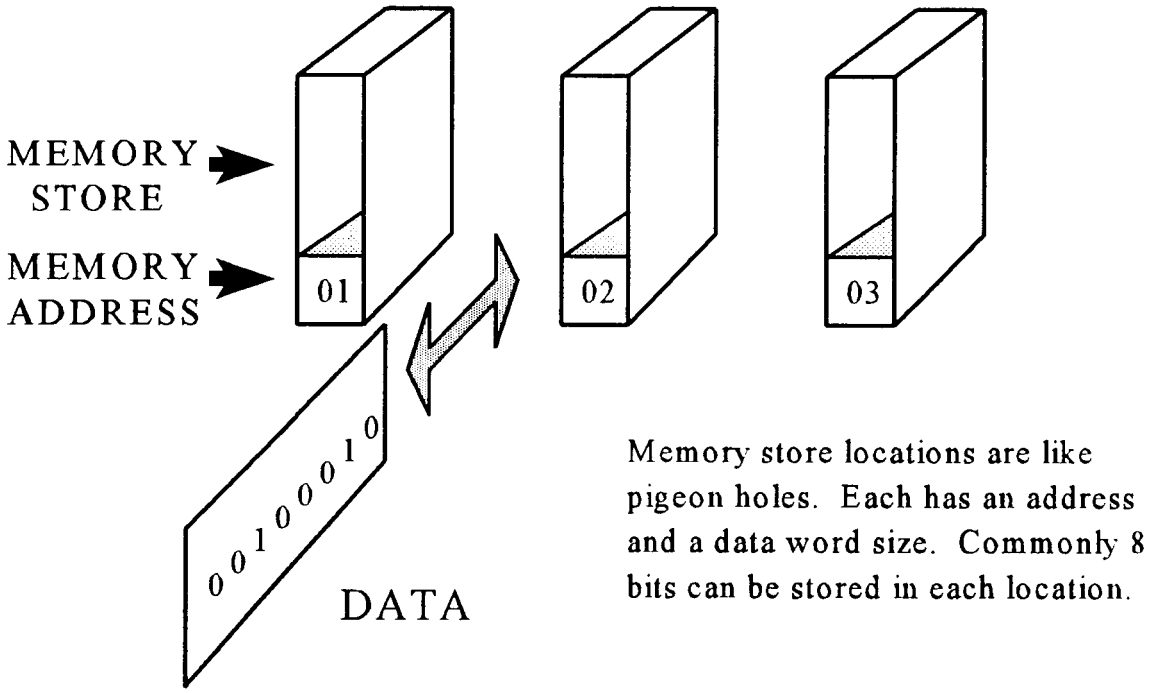
12. **Memory**. Generally there are 2 major categories of memory on a computer system: **primary memory (or main memory),** which is normally **semiconductor memory,** and **secondary memory** which today is most often a **floppy disc or hard disc drive**. In the diagram above I am showing primary memory in the form of semiconductor integrated circuits. In this case the memory can be considered simply as a series of pigeon holes or store locations, with each store location characterised by:

a. A unique ADDRESS which is the binary number output from the CPU on the address bus. To find memory device address range we use the same rule as for the micro:

number of memory locations on memory chip = 2^n n = no of address lines on chip

b. The length of the binary word (number of bits) which the memory device can hold in each address location. This is normally 8, but can be 16 or 32.

Simple View of Semiconductor Memory:



Another characteristic of memory devices is the **ORGANISATION**, this is the number of address locations the memory device holds and the word size at each location:

MEMORY ORGANISATION = no memory locations * word size

for example, the organisation of a memory chip having 10 address lines and 8 data lines will be:

organisation = 2^{10} * 8 bits
= 1K * 8 bits

This is pronounced one K by 8 bits - it can hold 1 K of 8-bit words.

13. **Main Memory Devices (ROM and RAM)**. Main memory is made up of ROM and RAM, semiconductor memory devices. The differences between these devices are as follows:

a. **Random Access Memory (RAM)**. In this type of memory device any location can be accessed at random (and immediately) by specifying its unique address. Data is written into RAM by a **WRITE** operation and is read from RAM in a **READ** operation. The main characteristic of RAM is its ability to read or write data, hence it is often referred to as **read/write memory**. Another major characteristic of RAM is that when power is removed from the chip, the data held in each location is destroyed. RAM is said to be **volatile** memory.

b. **Read Only Memory (ROM)**. Like RAM any location in ROM can be accessed at random and immediately by specifying its unique address. The term RAM above is therefore a little misleading, since ROM also has this characteristic. The main difference is that data can only be read from ROM, ie normally it is not possible to write new data. Also when power is removed the data in ROM is not destroyed. ROM is said to be **non-volatile memory**. ROM devices come in different forms:

(1) **Mask ROM**. Here the ROM is programmed at the time of manufacture. the manufacturer produces a mask to indicate permanent links on the chip and reproduces ROM devices with this mask. This is only economical when many devices (~10,000) are required.

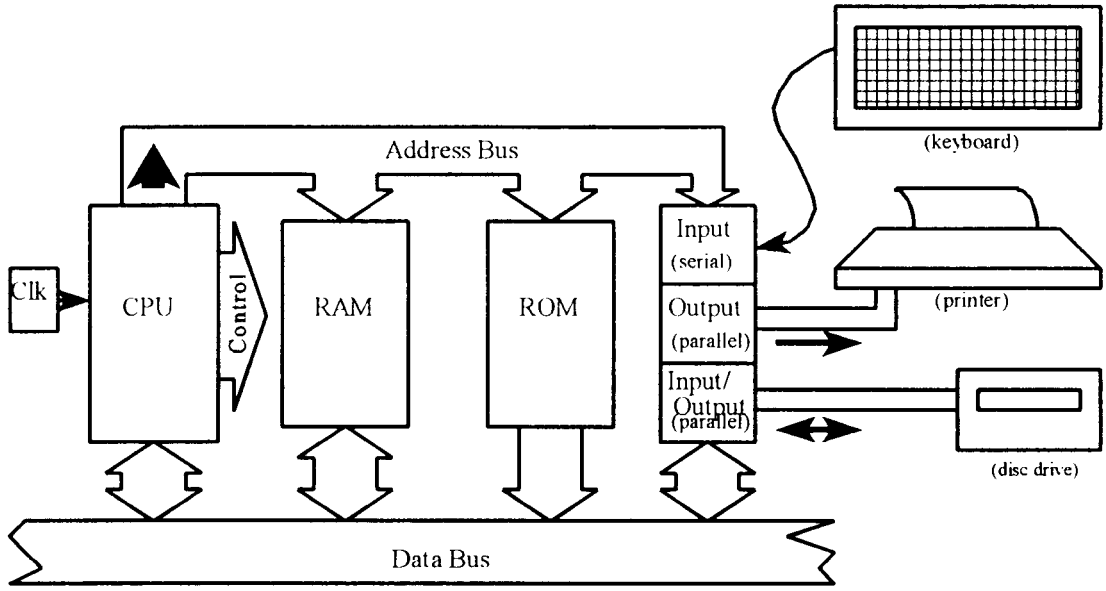
(2) **Programmable ROM (PROM)**. This form of ROM can be programmed by the user using a PROM programming device. Internally the PROM has fusible links which can be left intact or broken by the selective application of high voltages. The process is run completely on the programming device; all the user does is input the data required by hand or by download from a PC. Once programmed this device cannot be re-programmed.

(3) **Erasable Programmable ROM (EPROM)**. This type of memory behaves exactly the same as PROM but can be erased by exposure to UV light and then re-programmed in an EPROM PROGRAMMER. You can normally recognise this type of device since it has a 'window' in the chip through which the integrated circuit can be seen - this is normally covered with silver foil to keep out UV light! EPROM is useful for system development.

14. **System Boot-Up Program**. The ROM is normally used to hold the system boot-up program. This is a program which is run at switch-on time to initialise and prepare the system for use. Obviously it must remain in memory when power is removed, hence it is held in ROM. On the PC you may have heard of the BIOS (Basic Input Output System) program which is held in ROM; part of its purpose is to provide the boot-up program. Without the BIOS chips in your PC it cannot start up.

15. **Input/Output Interfaces**. These are frequently special input/output (I/O) integrated circuits which **organise** and **control** the flow of data between the CPU (or micro) and external devices (often called **peripheral devices**).

The microcomputer System with interfaces to serial and parallel peripheral devices.



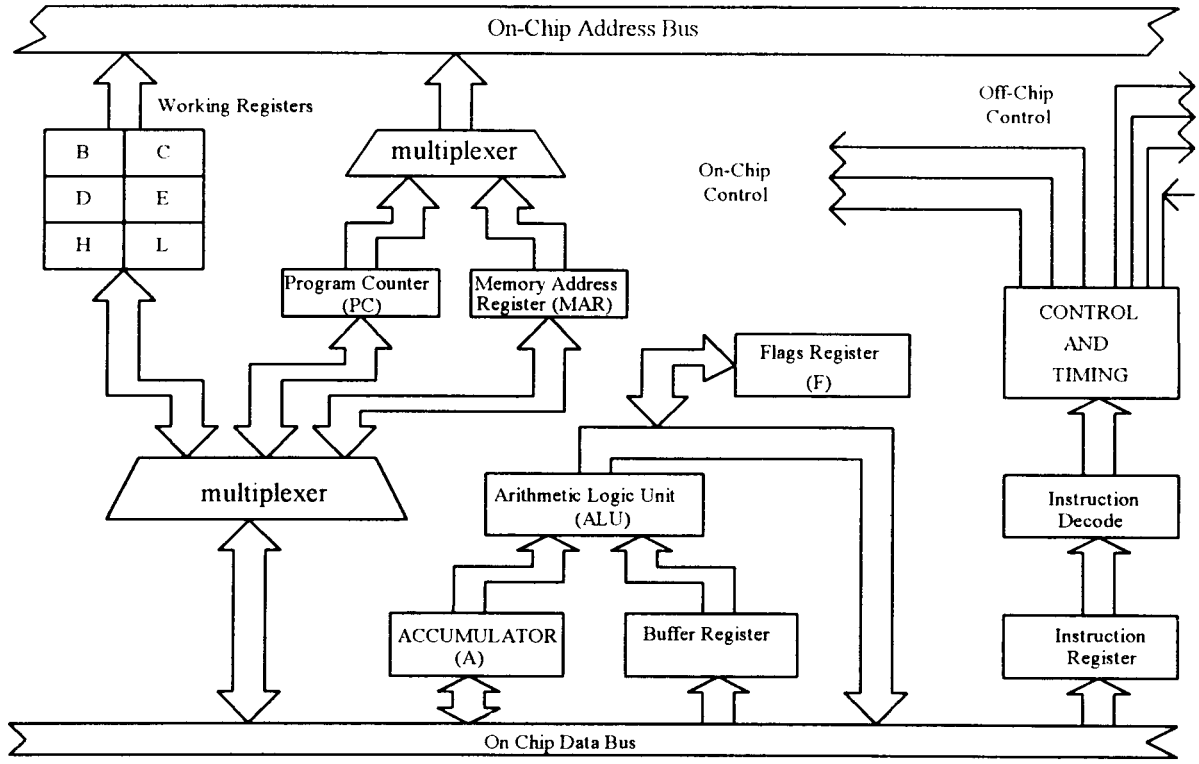
Such peripheral devices are printers, disc drives or keyboards. The interface circuits, together with relevant connectors, are called **PORTS**. Ports can be **bi-directional**, in which case they allow 2-way flow of data (the disc interface uses a port of this type), or they can be **unidirectional** (printer output or keyboard input). Ports can also be classified as **parallel** ports in which case they can input or output a full word at one instant on, say, 8 parallel lines (eg the disc drive port), or they can be **serial,** where a data word is transferred bit by bit along a single wire (for example the keyboard or a port to a modem for telephone signals).

Student Problem:	If, on the computer system diagram above, the RAM and ROM chips had not been labelled, how could you have distinguished between the two?

THE MICROPROCESSOR

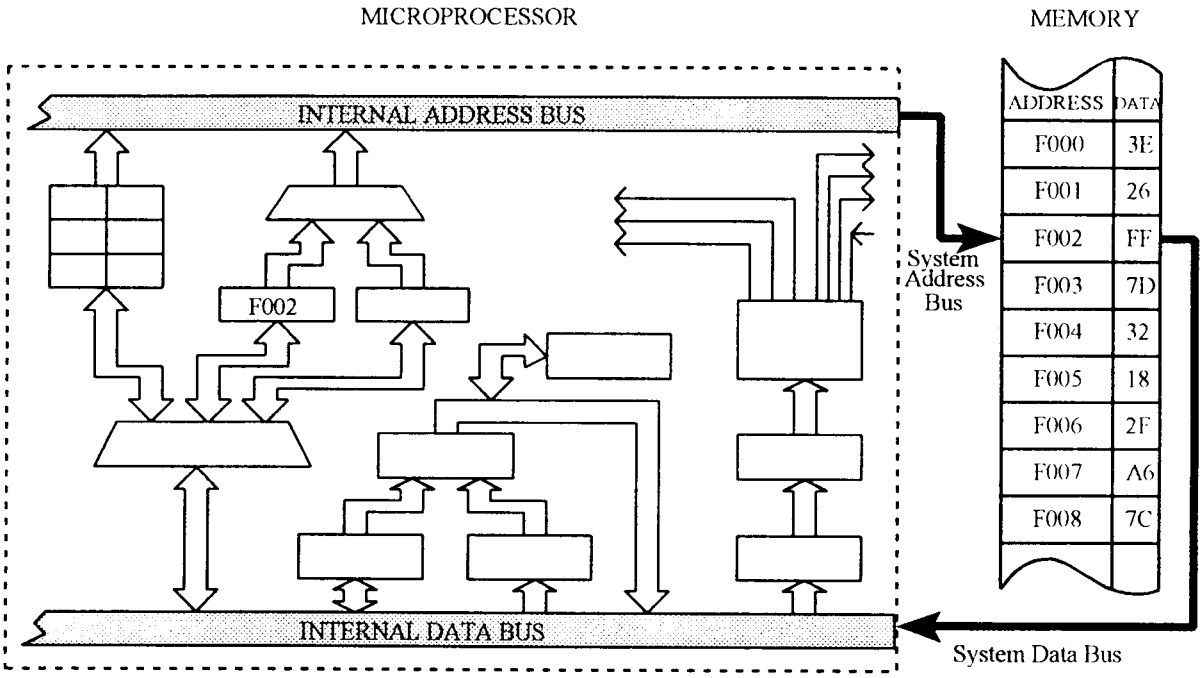
1. **Introduction**. In the previous section we have considered the components of a computer system which is using a microprocessor as its CPU. Now we will look inside the microprocessor to see its basic component architecture and how these components work together to enable the micro to execute programs. The good news is that you have already met all the components inside the micro in the logic sections. The microprocessor is basically a **sequential logic circuit** (ie it is a clocked circuit) and to perform its task it uses various registers, counters, flip-flops and general logic arrays. For simplicity we will consider the architecture of a general purpose 8-bit microprocessor, so all internal registers and counters will be 8-bit devices, and it will input and output data on an 8-bit data bus. In addition, and in common with normal 8-bit micros, there will be a 16 bit address bus.

2. **The Internal Architecture of the Microprocessor**. Shown below is the basic organisation of a general purpose microprocessor. In the following pages I will describe all the components and then we will consider how they work together to run programs which are held in primary memory (ROM or RAM).



This diagram will form the basis of our discussion in the next few pages so it might be worthwhile extracting this page and having it beside you as you read on.

3. **Address and Data Buses.** The internal Address and Data buses are simply internal extensions of the external buses which we saw in the computer system diagram. They enable internal devices to output addresses and to output or input instruction codes and data. It is important to bear in mind that these are connected to ROM and RAM devices in the external computer system. I show this in the diagram below where memory is shown with the address (16 bit address hence 4 hexadecimal digits) and data (8-bit data so 2 hexadecimal digits) and the way that address bus and data bus would be connected to any location. An address is output from the micro on the address bus, a memory location (F002 in this case) is accessed and the data held at that address (FF) is input on the data bus.



4. **Multiplexers**. These are **digital switching circuits** which select which of several inputs are connected to the address or data buses.

5. **Program Counter (PC)**. This is a **counter/register**. It is used when executing a program and it always holds the **address of the next instruction** byte to be accessed. In the diagram above, for example, if a program was in process of execution the address F002 would be held in the PC and would be output, via the multiplexer, onto the address bus for the code FF to be input to the micro. An important feature of the program counter is that, after each code has been fetched-in, the PC will be **incremented automatically** so that it points to the next memory location to be accessed (in this case F003).

6. **Instruction Register**. This register holds the instruction code for the current instruction which the micro is executing. In the diagram above, if the code FF which is being input is an instruction code it would automatically be placed into the instruction register.

7. **Instruction Decode**. Once the instruction code has been input to the instruction register the microprocessor must decode the instruction to prepare all the control and timing signals for correct execution of this instruction. This is where the instruction decoding is carried out. In effect it is a complex logic circuit that will produce different outputs depending on the instruction code received. Whilst you may never wish to design such a circuit it is essential to know of its existence for discussions of processor features later in the course.

8. **Control and Timing**. This is the logic which responds to the decoded signals. It carries out many different functions, for example:

a. Incrementing the PC after each code input.

b. Providing READ and WRITE signals to control reading and writing memory or Input/Output.

c. Setting internal control lines to activate multiplexers correctly or activate internal registers for reading or writing - recall section 3 page 3-15: registers require clock signals and read/write signals. These will be generated here.

Student Problem:	In my diagram of the general micro architecture on page 4-9 I show one line entering the Control and Timing Unit from an external pin. What essential signal do you think this line might carry?

9. **Arithmetic and Logic Unit (ALU)**. This is a complex combinational logic circuit which performs all arithmetic and logic operations on data which is held in the accumulator and/or buffer register. The ALU is the heart of the microprocessor; this circuit is responsible for all the calculations and data manipulations which we program the computer to perform. In 8-bit microprocessors the ALU can perform all types of logic operations but only the arithmetic operations of addition and subtraction. On larger 16-bit and 32-bit micros, however, the ALU can also carry out multiply and divide operations.

10. **Accumulator (A)**. The accumulator is a special register. It is used in all arithmetic and logic operations. Later, when we take a look at the style of instructions which we can use with the micro, we will see that the accumulator is involved in most of them - these operations are performed on data held in the accumulator. In addition, the result of any arithmetic or logic operation performed in the ALU is invariably placed in the accumulator, hence its name - it **accumulates** the results of any operations. Note on the diagram the data pathway from the output of the ALU back to the data bus and hence to the Accumulator.

11. **Buffer Register**. In arithmetic or logic instructions there are often 2 bytes of data to be added or subtracted or ANDed etc. One of these bytes is always placed in the accumulator, the other byte, meanwhile is channelled by the micro into the buffer register. When both are in position the ALU can then be triggered to carry out the required operation. The buffer register is thus used to hold the second of 2 bytes involved in an arithmetic or logic operation.

12. **Flags Register (F) or Status Register**. This is a collection of individual flip-flops (or bistables). When the ALU carries out any operation, the nature of the result is stored in these bistables. Each flip-flop is given a name to indicate the information it records. Common flags are:

Zero Flag (Z): sets to '1' if the result of an operation is zero.

Carry Flag (C): sets to '1' if the operation causes a carry out from the MSB.

Sign Flag (S): sets to '1' if the result appears to be negative - ie the MSB is '1'. Section 1, Number Systems and Codes, page 1-11 para 6 refers.

13. **Working Registers (B, C, D, E, H, L)**. Many microprocessors have a number of 'working registers' on-chip. These are very useful either to store data temporarily or to store data which will be used in operations. The great advantage of on-chip registers is that they are very fast to access, particularly when compared to memory. Loading data from an on-chip register or adding data from an on-chip register is much faster than loading or adding data which is held in memory. This is a matter which we shall discuss in greater detail later on the course. Generally for a micro to have a good set of on-chip working registers is a point in its favour.

14. **Memory Address Register (MAR)**. Finally there is the memory address register. This is used to hold the memory address of a data location which is to be read from or written to. You may think that this register is not required as the PC can do this job.

Student Problem:	why can the PC not be used to hold general memory addresses?

THE FORMAT OF MICROPROCESSOR INSTRUCTIONS

1. **Introduction**. Any microprocessor **program** is made up of a **set of instructions** which the processor executes in a particular order. Each instruction carries out a particular operation, and at the processor level these are very simple in nature. Some examples are:

Load a number from memory into the A register.

Add to the A register a number held in memory.

Subtract from the A register a number held in memory.

Load a number from the B register into the A register.

Any instruction is given a particular 8-bit binary code which the microprocessor can read into its Instruction Register, decode and then execute. All such instructions are recorded in a document called the **Instruction Set** - we will consider an instruction set later, but for now I would like to consider one instruction and the form which this instruction might take. In this section, to give realism to the work I shall consider instructions for a common and popular 8-bit processor the Z80. This processor was designed in 1976 but is still popular today.

2. **Instruction Format**. In the paragraph above I have referred to various instructions by describing them in words. These instructions are translated into instruction codes which the microprocessor can understand. In this form we call the instructions **machine code**. Any machine code instruction has 2 parts:

a. **The Operator**. In an 8-bit microprocessor this will be a single 8-bit code which defines the nature of the operation: add, subtract, load etc.

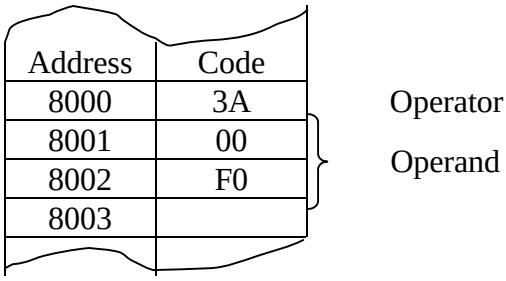
b. **The Operand**. This can be a single byte or a pair of bytes. It defines the data to be worked on or where that data can be found.

3. **An Instruction Example**. A common instruction is to load data from memory into the A register. So, the instruction we will consider is:

'Load the A register with the data byte held in memory at address hex F000'

In the Z80 the Machine Code for this instruction is: **3A 00 F0**

If this instruction was held in memory in the program at 8000, then it might look as below:



The Operator Code (or **OpCode**) 3A is the byte which must be fetched into the micro and decoded. It then sets all the control ready for what comes next.

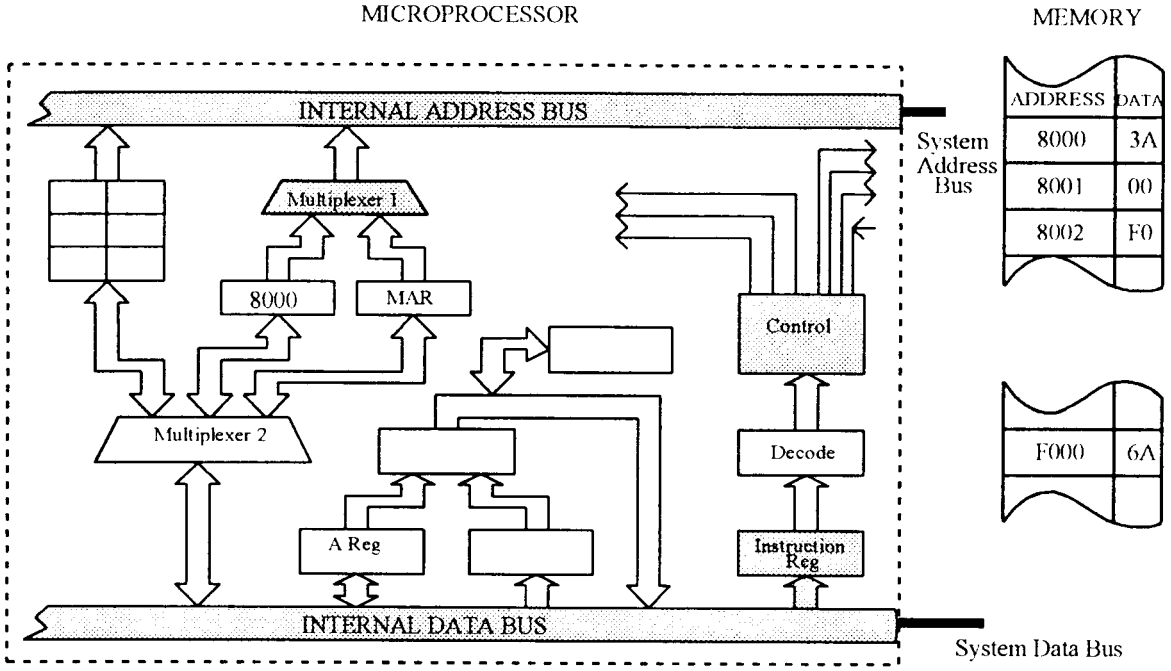
The **Operand**, in this case the memory address from which data is to be fetched into A, is next. It looks a little odd since it looks to be the wrong way around. This is because this address must be loaded into the micro and the micro always loads multi-byte data (or addresses) least significant (or low) byte first.

We will shortly look to see how the micro deals with this instruction but first a word about the basic microprocessor instruction processing procedure.

4. **The Fetch-Execute Cycle**. In carrying out each instruction in a program the microprocessor goes through a sequence of events known as the **fetch-execute cycle**:

> **The Fetch Sequence:** This involves getting the operator part of the instruction from memory and passing it into the micro, then getting any operand/s required. Also involved in this sequence will be the **decoding** of the instruction.
>
> **The Execute Sequence:** Involves actually doing what the instruction says.

To illustrate this activity we will take the instruction considered above and go through the sequence of events for '**load the A register with the data held in memory at address F000**'.



In the diagram above I show all parts of the microprocessor which will be involved in the fetch-execute cycle for this instruction. In addition, I show the main memory holding the instruction itself, which is stored from 8000, and the actual data which has to be loaded into the A register, ie the contents of memory location F000. The diagram also shows the PC holding 8000 - recall that, when the program is running, the PC holds the address of the next memory location to be accessed. At the start of this instruction, therefore, it will hold the instruction start address.

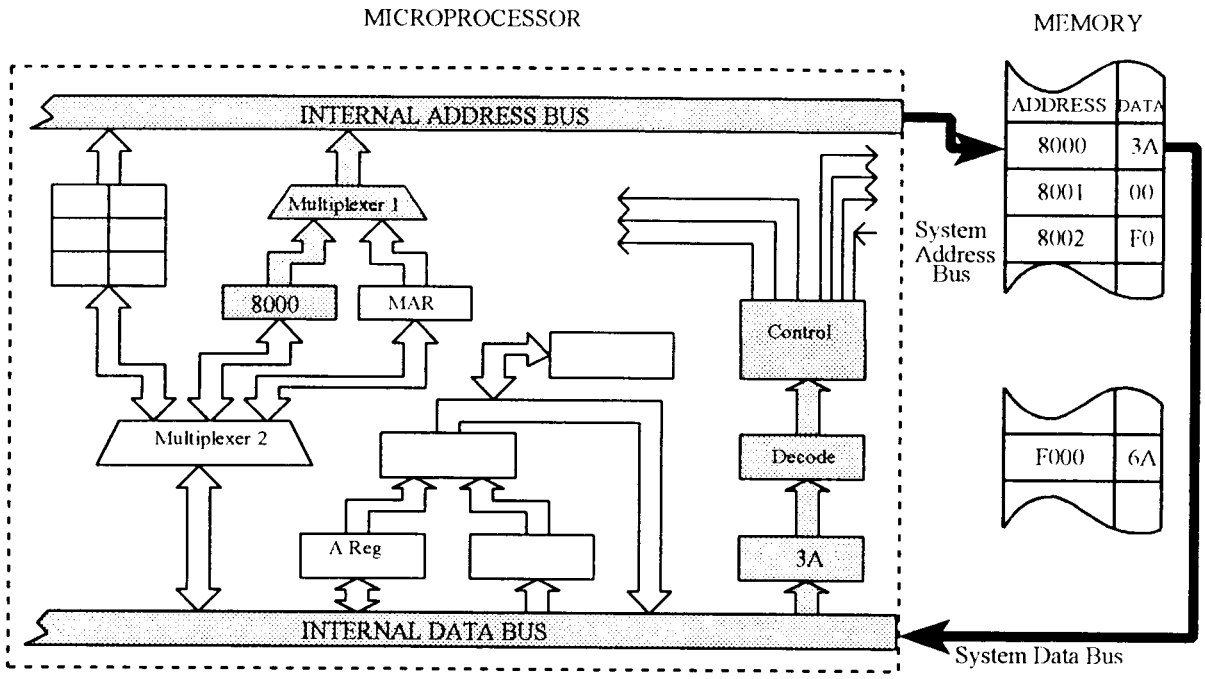
As this is the start of an instruction fetch, the micro will be aware of this and the control section will operate to do 2 things:

a. It will activate multiplexer 1 to channel; the PC onto the Address bus.

b. It will activate the Instruction Register to receive the byte which arrives on the data bus - it must be an instruction code since it is the start of the fetch.

We can see this activity on the next page . . . .

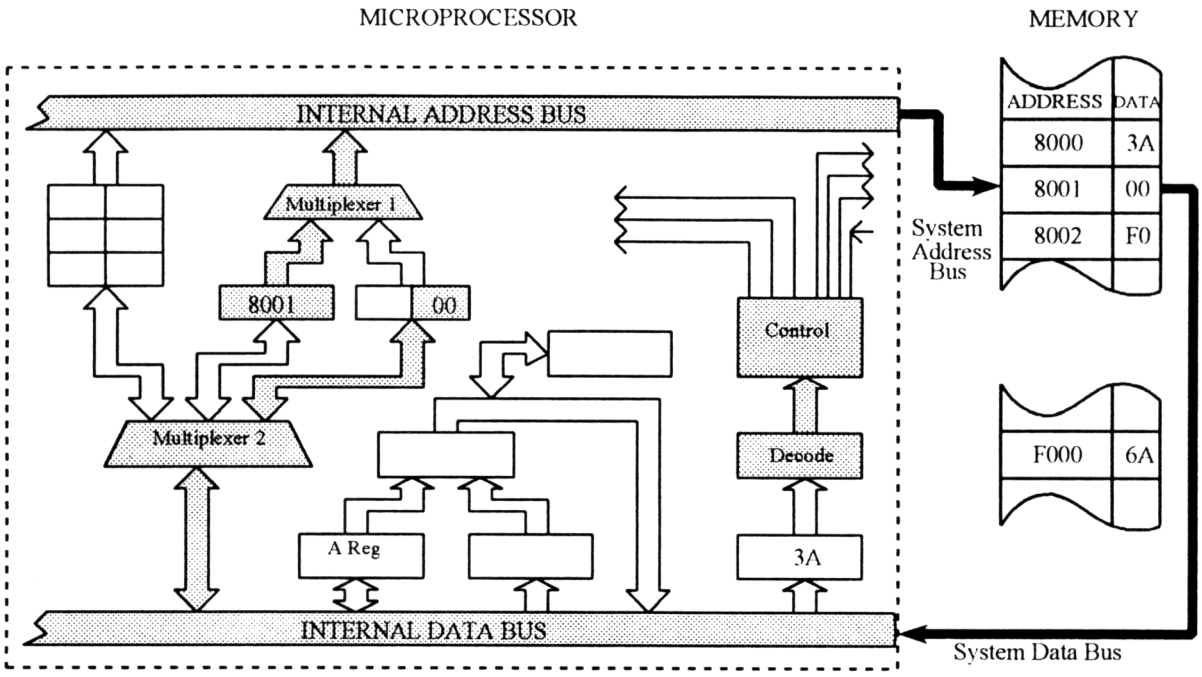
Fetching the Op code



In the diagram above we can see the PC contents (8000) being passed through the multiplexer to the address bus and access being made to main memory location 8000. The byte held there is brought into the micro and placed in the Instruction Register. Thereafter the decode section will decode this instruction and activate the control section to prepare for the next activity which is to fetch the memory address.

We can see this in the next diagram . . . .

Fetching the Address Low Byte



To prepare for fetching the memory address the control section will:

a. Increment the **PC** to 8001.

b. Activate **multiplexer 1** to output the PC.

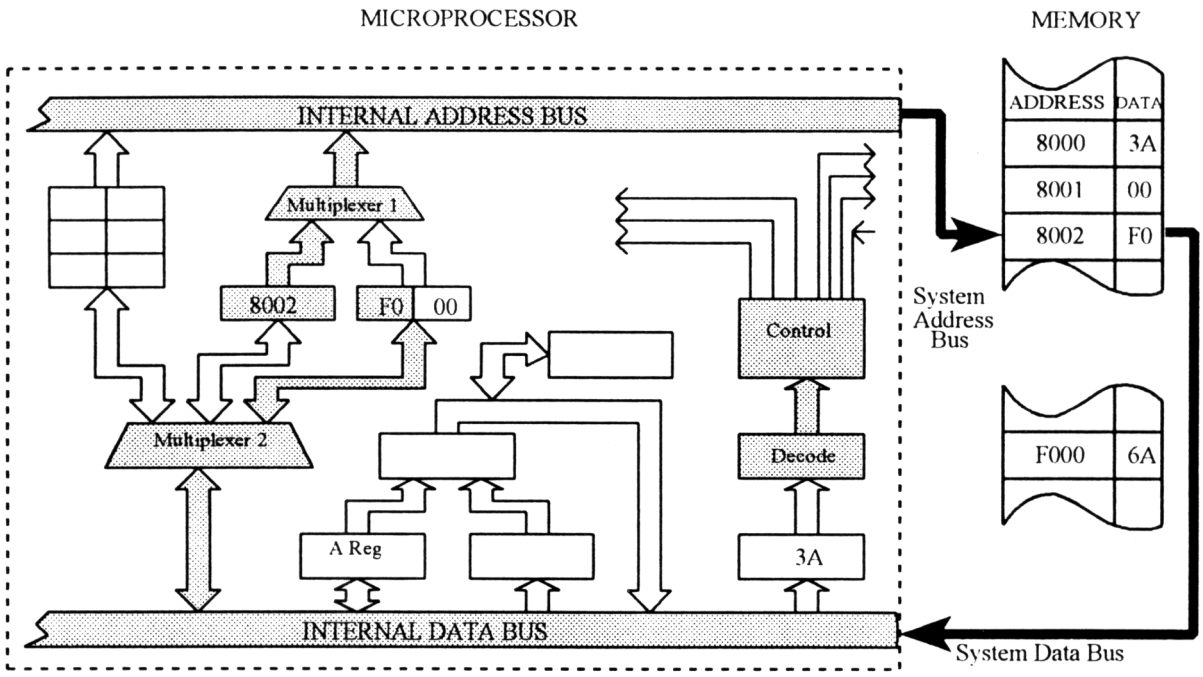
c. Activate **multiplexer 2** to channel the next byte to the low byte of the **MAR** - recall that the address is held in memory LOW BYTE first so the low byte will be accessed first.

d. Close-off access to the Instruction Register.

In the diagram we can see the PC value, 8001, is now output onto the address bus and the contents of this address are input to the micro and channelled into the MAR low byte. The control section will now prepare to fetch the high byte of the address.

We can see this in the next diagram . . . .

Fetching the Address High Byte



The control continues to work and will now:

a. Increment the **PC** to 8002.

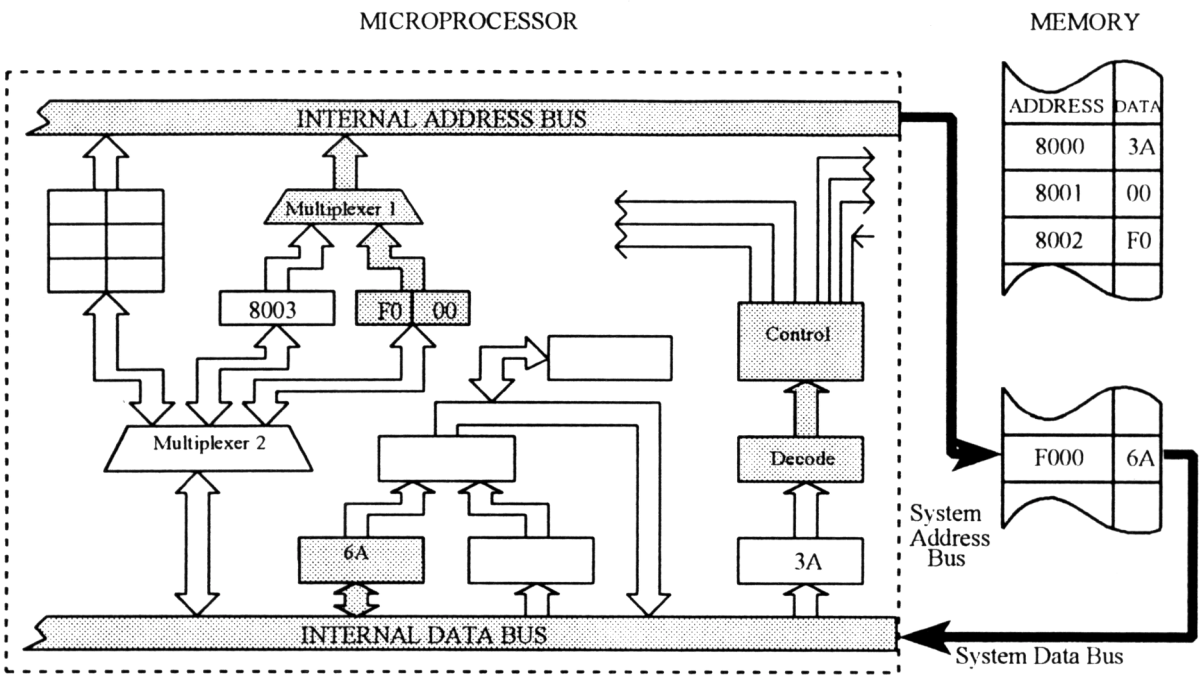
b. Continue to activate the **multiplexers.**

c. Set the **MAR** to receive the next byte into its high byte location.

In the diagram we can see the PC value, 8002, is now output on the address bus and the contents of this address are input to the micro and channelled into the MAR HIGH byte. The control section now prepares to access the data.

We can see this on the next page . . . .

The Execute Phase



Note that this is now the **EXECUTE** phase. All relevant information has been fetched into the micro and it can now carry out the instruction by loading the data at address F000 into the A register.

After the last action the control section will increment the PC and continue to prepare for loading the data byte into the A register by:

a. Activating **multiplexer 1** to channel the MAR contents to the address bus (not the PC).

b. Deactivating multiplexer 2.

c. Activating the A register to receive the byte which will now be input.

We can see this activity in the diagram above. The result is that the actions required by the instruction have now been achieved. The data byte from address F000 is in the A register and the PC is ready to start on the next instruction fetch.

5. **Summary**. In this part we have seen some important points regarding the operation of a microprocessor. To prepare for the final part of this section it will be useful to summarise what we have just learned.

a. Any instruction has 2 parts:

The operator or opcode - this is the codeword which the microprocessor decodes to understand what it must do.

The operand - this the additional information which the micro must have in the instruction to perform its task, for example the address from which data is obtained.

b. When the processor starts to execute an instruction the PC contains the instruction's address.

c. The first byte of any instruction is always an instruction byte and it is fetched into the micro and placed in the Instruction Register.

d. The instruction code is decoded in the decode section and this activates the control section to set-up all relevant control and timing.

e. All relevant operands are fetched into the micro under the control of the control unit. The PC increments after each fetch.

f. Finally, when all relevant information is in the micro the execution phase is carried out and the instruction is executed.

PROGRAMMING TERMINOLOGY AND INSTRUCTION TYPES

1. **The Computer Program**. In order for a microprocessor to accomplish a specific task we must write a program. The program can be regarded as a step-by-step sequence of instructions which enable the microprocessor to do whatever is required.

2. **Instruction Groups**. All microprocessors provide different categories of instructions which are conveniently listed in their different groups. Depending on the microprocessor being considered the instruction names will differ. Common instruction groups are:

Data Transfer: ie moving data from one place in the system to another. From memory into a register, from a register to memory or from one register to another:

Z80	-	LOAD (LD)
68000	-	MOVE (MOV)
8086	-	MOVE (MOV)

Arithmetic: these are common to most micros and they carry out the basic arithmetic operations, for example: ADD, SUBTRACT (SUB), INCREMENT (INC), DECREMENT (DEC) etc.

Logic: again these are common to most micros: AND, OR, XOR etc. Note, though, these are **bit-wise** operations. If a micro were to AND the two 8-bit words A6, 3E, the AND operation would be carried out on each bit pair as below, first the 2 LS bits (0 AND 0 = 0), then the next 2 bits (1 AND 1 = 1) all the way up to the 2 MS bits (1 AND 0 = 0):

A6	=	1010 0110
3E	=	0011 1110
A6 AND 3E	=	0010 0110

Logic operations are useful for manipulating data. If, for example, a word is received and only the lower 4 bits are relevant, we can **MASK-OFF** the upper 4 bits by ANDing the word with 0F:

Word received	=	1011 1010
Mask word	=	0000 1111
Result	=	0000 1010

ANDing anything with 0 results in 0, ANDing anything with 1 leaves the bit unchanged.

Student Problem:	An 8-bit data word is to be prepared for transmission and must have each of the 2 MS bits (ie bits 7 and 6) set to '1', irrespective of what they might currently hold. What bit-wise logic operation would achieve this?

Input/Output: These instructions are used to input and output data to **ports**. Some micros, like the Z80, can treat ports in a special way and have instructions particularly for port access. Others just treat ports like memory locations and so use the same instructions as for memory:

Z80	:	IN, OUT
68000	:	MOVE (MOV)
8086	:	IN, OUT

Transfer Control: These instructions cause the program execution to change to a different location; we talk of **jumping** or **branching** to a new location. We use these types of instructions when we want a program to loop, ie to repeat the same operation a number of times:

Z80	:	JMP (ie jump)
68000	:	JMP, BRA (ie branch)
8086	:	JMP

3. **Methods of Representing Instructions**. A microprocessor instruction can be represented in a number of ways:

a. **Description**. As a written description of the actions that will be carried out, for example:

'Load the A register with data byte held at memory address F000'.

b. **Binary Code**. The actual binary codes that will be input by the processor to understand what it must do. For the instruction above this is:

0011 1010 0000 0000 1111 0000

c. **Hexadecimal Code**. This is an improvement over binary code since it reduces the number of digits which we must remember. However, like binary code it gives no indication of the instruction's purpose. For the instruction above this would be:

3A 00 F0

You should note that the binary or hexadecimal forms of the instruction are also called **machine code**.

d. **Mnemonic Code**. The word mnemonic means 'memory aid'. This form is a shorthand method of representing the instruction. It is convenient to use this form since the instruction indicates what operation is performed. However, the instruction must still be transformed into the hexadecimal or binary (ie machine code) form before the micro can decode it and understand what is to be done. The instruction above might appear as below in mnemonic form:

LD A,($F000)

This form of instruction is also called **Assembler** form and the process of converting it from this form into the binary code form is carried out by a program called an **assembler**.

4. **Assembler Code**. In para 3d above I have mentioned assembler code which is the mnemonic form of the instruction. This is easier for the programmer to use since it gives an indication of the operation to be carried out. I now want to concentrate on this instruction format since an understanding of it will help us in our future studies of processor types.

5. **Operands in Assembler Code**. We have met the term operand before in our consideration of a machine code instruction, in that context it is the data or address code following the Opcode. The operand tells us where the data will be obtained from or sent to. In a similar way we have operands in assembler code and they, once again, tell us where the data will be obtained from or sent to. The general form of an assembler code instruction is thus as follows:

Operation Destination Operand Source Operand

where **Destination Operand** is a way of expressing where the result of the operation will be placed, and the **Source Operand** is a way of expressing from where the data to be operated on will be obtained. For example:

LD A,B

Here the operation is **Load data**, in other words move data.

The **destination** for the move is the A register.

The **source,** where the data will be moved from, is the B register.

(Note that in processor terms load or move means copy. The data byte will still remain in the B register but a copy of it will be placed in The A register.)

Other examples:

	Instruction	**Operation**	**Destination**	**Source**
LD	A,($7000)	load	A reg	Memory Address hex 7000
LD	($F000),A	load	Memory Address hex F000	A reg
LD	A,$80	load	A reg	Next data byte, hex 80
LD	E,A	load	E reg	A reg

Notice in the above examples that brackets around the operand number denote that it is a memory address, whilst an operand number with no brackets denotes that this is actual data to be used.

6. **The Instruction Set**. As we have seen, an instruction is an **opcode** which defines a specific operation, together with the **operand data** required. The **instruction set** is a listing of all instruction codes in their mnemonic (or assembler) form together with the same instructions in their binary (or hexadecimal) ie machine code forms.

Given on the next page are some common Z80 instructions in both assembler and machine code forms. They are just a part of the instruction set for this processor. The full instruction set holds more than 150 different instructions.

Some notes on the instructions given:

a. Any instruction is shown in its general form followed by a specific example.

b. Any general address is shown as **(hl)** meaning **h**igh byte, **l**ow byte. Remember that in machine code the low byte is entered first. Also note that an address is shown in brackets, this is to distinguish it from a number value which has no brackets.

c. Any single byte data (ie number value) in the instruction is shown as n.

d. Input/Output instructions are not shown. We will consider these on the course.

e. Jump instructions are shown in 2 forms:

Unconditional (JP hl): this means that this jump to a new program address will be made irrespective of what may have happened previously.

Conditional (JP Z,hl): this means that the jump to a new program address will only be made if the condition designated occurred in the last arithmetic or logic operation and so this flag is set. If not then the program continues with the next instruction in sequence.

f. A **$** shown before a number means that number is in hexadecimal form. A number shown without a $ means the number is in decimal form.

Extract from Z80 Instruction Set

Instruction Group	Assembler Form		Machine Code Form			Effect
Data Transfer	LD	A,(hl)	3A	1	h	Copy data byte from memory address to A reg.
	LA	A,($4000)	3A	00	40	
	LD	(hl),A	32	1	h	Copy data byte from A register to memory address ie store data in memory.
	LD	($FC00),A	32	00	FC	
	LD	A,n	3E	n		Copy data byte designated in the instruction to the A reg.
	LD	A,$60	3E	60		
	LD	B,n	06	n		Copy data byte designated in the instruction to the B reg.
	LD	B,$6E	06	6E		
	LD	A,B	40			Copy data byte held in the B reg into the A reg.
Arithmetic	ADD	A,n	C6	n		Add to the contents of the A reg the data byte designated in the instruction, the result left in A reg.
	ADD	A,$77	C6	77		
	ADD	A,B	80			Add to the contents of the A reg the data byte held in the B reg, the result left in A reg.
	SUB	n	D6	n		Subtract from the contents of the A reg the data byte designated in the instruction, result left in A.
	SUB	$3D	D6	3D		
	DEC	B	05			Reduce the contents of the B reg by 1.
Logic	AND	n	E6	n		AND the contents of the A reg with data byte designated in the instruction, the result left in A
	AND	$28	E6	28		
	AND	D	A2			AND the contents of the A reg with the contents of the D reg, result left in A.
	OR	n	F6	n		OR the contents of the A reg with the data byte designated in the instruction, result left in A.
	OR	$F7	F6	F7		
Transfer Control	JP	hl	C3	1	h	Jump to address given in the instruction.
	JP	$9000	C3	00	90	
	JP	Z,hl	CA	1	h	Jump to the address given in the instruction IF the result of the last operation is Zero, ie the Z flag is set.
	JP	Z,$E000	CA	00	EO	
	JP	NZ,hl	C2	1	h	Jump to the address given in the instruction IF the result of the last operation was not Zero, ie the Z flag is not set.
	JP	NZ,$3F00	C2	00	3F	

Student Problem 1:

The following shows an assembler code program. And the address in memory at which each instruction will be held.

a. For each assembler code instruction write out the machine code form. Note that this activity is called assembling the program.

b. Complete the instruction addresses assuming that they follow one another in sequence.

c. Explain what the program achieves.

Address	**Assembler Code**		**Machine Code**
F000	LD	A,$00	
F002	LD	B,$04	
	ADD	A,$05	
	DEC	B	
	JP	NZ,$F004	

Student Problem 2:

The following is a Z80 program in machine code form.

a. Convert the program to assembler form. Note this activity is called disassembling the program.

b. Explain what the program does.

Address	**Assembler Code**	**Machine Code**		
6000		3A	00	FC
6003		E6	80	
6005		CA	0D	60
6008		06	00	
600A		C3	0F	60
600D		06	FF	

Solutions to Student Problems.

Page 4-5, para 11a.

(1) 68000: number of address lines = n = 24

range of addresses $= 2^n$

$= 2^{24}$

$= 2^4 * 2^{10} * 2^{10}$

$= 16 * 1K * 1K$

$= 16 * 1M$

$= 16M$

(2) Pentium: number of address lines = n = 32

range of addresses $= 2^n$

$= 2^{32}$

$= 2^2 * 2^{10} * 2^{10} * 2^{10}$

$= 16 * 1K * 1K * 1K$

$= 4 * 1G$

$= 4G$

G stands for gigabyte, ie 1024 megabytes.

Page 4-8, para 15.

Data from ROM is shown coming OUT of the memory device only - ie READ ONLY. Data from the RAM is shown both coming out (READ) and going in (WRITE).

Page 4-11, para 8.

This control line will carry the **clock signal.** This provides all the essential timing in the Control and Timing Unit, which is used for sequencing the events in the microprocessor.

Page 4-12, para 14.

The function of the Program Counter (PC) is to hold the current program memory addresses. It is essential that it always '**points**' to the **next** program memory location during the execution of any program code.

If the PC were loaded with the data memory address it would lose track of where program execution has reached. It cannot therefore be used simultaneously for the 2 jobs. For this reason the MAR is provided, and any operand (ie data) address is held here so that the PC can be left to hold only program addresses.

Page 4-21.

Refer back to Section 2 - (Basic Logic Definitions, Symbols and Circuits), Page 2-12. The OR function is such that if either input is logic '1' then the output will be logic '1'. Thus OR-ing any input with '1' will give a '1' result.

Alternatively, OR-ing any input with logic '0' does not alter the bit at the output.

The 'bitwise' logic operation is therefore to OR the input value with a number which has logic '1' in the 2 MS Bits and logic '0' in the other bits.

Example:

If the input value is	01001010
We can OR with	11000000
Producing result	11001010

We see that under any circumstances the 2 MS Bits are each set to '1' and the other bits are all unchanged.

Page 4-26, Students Problem 1.

Complete machine code and instruction addresses are as follows:

Address	**Assembler Code**		**Machine Code**		
F000	LD	A,$00	3E	00	
F002	LD	B,$04	06	04	
F004	ADD	A,$05	C6	05	
F006	DEC	B	05		
F007	JP	NZ,$F004	C2	04	F0

Having loaded the A register with 00 and the B register with 04 the program repetitively adds 05 to the A register and decrements the B register until B is zero. The end result is the number decimal 20 (hex 14) is held in the A register.

What the program achieves is that the number 5 is multiplied by 4.

The way the program achieves this is to carry out multiple addition. That is, we add the number to itself the number of times we want to multiply it. You will recall that 8-bit micros have no multiply instruction. Any multiplication required is, therefore, carried out in software using this multiple addition.

Page 4-27, Student Problem 2.

Completed disassembly is as follows:

Address	**Assembler Code**		**Machine Code**		
6000	LD	A,($FC00)	3A	00	FC
6003	AND	$80	E6	80	
6005	JP	Z,$600D	CA	0D	60
6008	LD	B,$00	06	00	
600A	JP	$600F	C3	0F	60
600D	LD	B,$FF	06	FF	

Note that the next program memory location after the last instruction is $600F.

This program does the following:

Loads a byte from address FC00 into the A register.
ANDs the byte with 1000000 ($80) so all bits other than the MSB are **masked** out.
The next instruction checks if the result is zero, ie since only the MSB is left it really checks if the MSB of the start word was 0.
IF the result was zero it then jumps to $600D where it loads the B register with $FF, and then continues with whatever is next.
IF the result was not zero, ie the MSB is '1' it loads the B register with 00.
It then jumps **unconditionally**, ie it has no option, to the next instruction after the last one shown.

The net effect is: **a byte is input from address FC00**
the MSB of the byte is checked
IF the MSB is '0' then the B register is loaded with $FF
Else, if the MSB is '1' then the B register is loaded with $00
In either case the program continues after the last instruction

Note that this example shows how we can control program execution (we can transfer control to a different place in the program) using the **conditional** jump instruction.