Unit 1 – Data Representation

Data Representation	**5 Hrs.**
Data Representation	**1.5 Hrs.**
Data Types	
Number Systems	
Alphanumeric Representation	
Complements (r's and r-1's)	
Fixed point Representation	**1 Hr.**
Integer Representation	
Arithmetic Addition, Subtraction, and Overflows	
Decimal Fixed point Representation	
Floating point Representation	**1 Hr.**
Binary and Decimal Codes	**1 Hr.**
Gray, BCD, ASCII, Excess-3 Codes	
Error Detection Code	**0.5 Hr.**
Parity bit, Parity checker and Parity generator	

Data Types

In computer science, a data type or simply type is a classification identifying one of various types of data, such as real, integer or Boolean, that determines the possible values for that type; the operations that can be done on values of that type; the meaning of the data; and the way values of that type can be stored.

Data types are used within type systems, which offer various ways of defining, implementing and using them. Common data types may include:

- Integers
- Booleans
- Characters
- Floating-point numbers
- Alphanumeric strings

The computer registers contain either data or control information. Data are numbers and other binary-coded information that are operated on. Control information is a bit or group of bits used to specify the sequence of command signals needed for data manipulation.

Possible data types in registers:

- Numbers used in computations
- Letters of the alphabet used in data processing
- Other discrete symbols used for specific purposes

b i i z a y @ g m a i l . c o m

All types of data, except binary numbers, are represented in binary-coded form. Numbers are represented by a string of digit symbols.

Based on number systems two basic data types are implemented in the computer system: fixed point numbers and floating point numbers. Representing numbers in such data types is commonly known as fixed point representation and floating point representation.

Number System

Number systems are used to describe the quantity of something or represent certain information. Number of digits used in a number system is called its ***base*** or ***radix*** (r). We can categorize number system as below:

Binary number system (r = 2)

Binary number system was introduced by an Indian- scholar ***Pingala*** in around 5th -2nd centuries in BC. Long and short syllables were used by him to illustrate the two types of numbers, it is more like Morse code. Gottfried Leibniz in 1679 introduced the modern type of binary number system which we still use. The number system with only two digit 0 and 1 is known as binary number system. Here, zero is represented by a symbol '0' and one is represented as '1'. Binary is another way of saying ***base-2***.

Binary number system has the base (or radix) 2 and the numbers in this system are formed with two digits 1 & 0.

Binary	0	1	10	11	100	101	110	111	1000	1001	1010
Decimal	0	1	2	3	4	5	6	7	8	9	10

2^6	2^5	2^4	2^3	2^2	2^1	2^0
1	0	0	1	1	0	1
64+0+0+8+4+0+1						
87						

Figure: Representing binary number 1001101

EXAMPLE: Binary Number: $(10101)_2$

Calculating Decimal Equivalent:

Step	Binary Number	Decimal Number
Step 1	10101_2	$((1 \times 2^4) + (0 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0))_{10}$
Step 2	10101_2	$(16 + 0 + 4 + 0 + 1)_{10}$
Step 3	10101_2	21_{10}

b i i z a y @ g m a i l . c o m

$2^3 = 8$	$2^2 = 4$	$2^1 = 2$	$2^0 = 1$	Decimal Equivalent
0	0	0	0	0
0	0	0	1	1
0	0	1	0	2
0	0	1	1	3
0	1	0	0	4
0	1	0	1	5
0	1	1	0	6
0	1	1	1	7
1	0	0	0	8
1	0	0	1	9
1	0	1	0	10
1	0	1	1	11
1	1	0	0	12
1	1	0	1	13
1	1	1	0	14
1	1	1	1	15

Table: The Binary counting sequence

Octal Number System (r = 8)

The octal numeral system, or "***oct***" for short, is the ***base-8*** number system, and uses the digits 0 to 7. When we count up one from the 7, we need a new placement to represent what we call 8 since an 8 doesn't exist in Octal. So, after 7 it is 10.

Octal	0	1	2	3	4	5	6	7	10	11	12...	17	20...	30...	77	100
Decimal	0	1	2	3	4	5	6	7	8	9	10...	15	16...	24...	63	64

8^3	8^2	8^1	8^0
3	6	2	3
1536+384+16+3			
1939			

Figure: Representing octal number 3623

EXAMPLE: Octal Number: $(12570)_8$

Calculating Decimal Equivalent:

Step	Octal Number	Decimal Number
Step 1	12570_8	$((1 \times 8^4) + (2 \times 8^3) + (5 \times 8^2) + (7 \times 8^1) + (0 \times 8^0))_{10}$
Step 2	12570_8	$(4096 + 1024 + 320 + 56 + 0)_{10}$
Step 3	12570_8	5496_{10}

b i i z a y @ g m a i l . c o m

Hexadecimal Number system (r = 16)

The hexadecimal system is ***base-16***. As its base implies, this number system uses sixteen symbols to represent numbers. So, in hexadecimal, the total list of symbols to use is 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F.

Hexadecimal	9	A	B	C	D	E	F	10	11...	19	1A	1B	1C...	9F	A0
Decimal	9	10	11	12	13	14	15	16	17	25	26	27	28	159	160

16^3	16^2	16^1	16^0
2	D	B	7
8192+3328+176+7			
11703			

Figure: Representing hexadecimal number 2DB7

EXAMPLE: Hexadecimal Number: $(19FDE)_{16}$

Calculating Decimal Equivalent:

Step	Binary Number	Decimal Number
Step 1	$19FDE_{16}$	$((1 \times 16^4) + (9 \times 16^3) + (F \times 16^2) + (D \times 16^1) + (E \times 16^0))_{10}$
Step 2	$19FDE_{16}$	$((1 \times 16^4) + (9 \times 16^3) + (15 \times 16^2) + (13 \times 16^1) + (14 \times 16^0))_{10}$
Step 3	$19FDE_{16}$	$(65536 + 36864 + 3840 + 208 + 14)_{10}$
Step 4	$19FDE_{16}$	106462_{10}

Decimal Number System (r = 10)

The number system that we use in our day-to-day life is the decimal number system. The word decimal comes from "***decem***", the Latin word for ten. The decimal number system (also called ***base-10*** or occasionally ***denary***) has 10 as its base. Any number, from huge quantities to tiny fractions, can be written in the decimal system using only the ten basic symbols 1, 2, 3, 4, 5, 6, 7, 8, 9, and 0.

Values are represented by the digits and their positions in the number and the type of number system is called ***Positional Number System***. In decimal number system, the successive positions to the left of the decimal point represents: units, tens, hundreds, thousands and so on.

Each position represents a specific power of the base (10). For example, the decimal number 1234 consists of the digit 4 in the units position, 3 in the tens position, 2 in the hundreds position, and 1 in the thousands position, and its value can be written as:

b i i z a y @ g m a i l . c o m

= (1x1000) + (2x100) + (3x10) + (4x1)
= (1x103) + (2x102) + (3x101) + (4x100)
= 1000 + 200 + 30 + 1
= $(1234)_{10}$

Number System Conversion

Decimal to Other Base System

Steps:

Step 1 - Divide the decimal number to be converted by the value of the new base.
Step 2 - Get the remainder from Step 1 as the rightmost digit (least significant digit) of new base number.
Step 3 - Divide the quotient of the previous divide by the new base.
Step 4 - Record the remainder from Step 3 as the next digit (to the left) of the new base number.

Repeat Steps 3 and 4, getting remainders from right to left, until the quotient becomes zero in Step 3.

The last remainder thus obtained will be the most significant digit (MSD) of the new base number.

Example: Decimal Number: $(29)_{10}$

Calculating Binary Equivalent:

Step	Operation	Result	Remainder
Step 1	29 / 2	14	1
Step 2	14 / 2	7	0
Step 3	7 / 2	3	1
Step 4	3 / 2	1	1
Step 5	1 / 2	0	1

As mentioned in Steps 2 and 4, the remainders have to be arranged in the reverse order so that the first remainder becomes the least significant digit (LSD) and the last remainder becomes the most significant digit (MSD).

Decimal Number: $(29)_{10}$ = Binary Number: $(11101)_2$

b i i z a y @ g m a i l . c o m

Example: Conversion of Decimal $(41.6875)_{10}$ to Binary

Integer = 41

41	
20	1
10	0
5	0
2	1
1	0
0	1

Fraction = 0.6875

0.6875
2
1.3750
x 2
0.7500
x 2
1.5000
x 2
1.0000

$(41)_{10} = (101001)_2$ $(0.6875)_{10} = (0.1011)_2$

$$(41.6875)_{10} = (101001.1011)_2$$

Other base system to Decimal System

Steps:

Step 1 - Determine the column (positional) value of each digit (this depends on the position of the digit and the base of the number system).
Step 2 - Multiply the obtained column values (in Step 1) by the digits in the corresponding columns.
Step 3 - Sum the products calculated in Step 2. The total is the equivalent value in decimal.

Example: Binary Number: $(11101)_2$

Calculating Decimal Equivalent:

Step	Binary Number	Decimal Number
Step 1	11101_2	$((1 \times 2^4) + (1 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0))_{10}$
Step 2	11101_2	$(16 + 8 + 4 + 0 + 1)_{10}$
Step 3	11101_2	29_{10}

Binary Number: $(11101)_2$ = Decimal Number: $(29)_{10}$

Other Base System to Non-Decimal System

Steps:

Step 1 - Convert the original number to a decimal number (base 10).
Step 2 - Convert the decimal number so obtained to the new base number.

Example: Octal Number: $(25)_8$

b i i z a y @ g m a i l . c o m

Calculating Binary Equivalent:

STEP 1: CONVERT TO DECIMAL

Step	Octal Number	Decimal Number
Step 1	25_8	$((2 \times 8^1) + (5 \times 8^0))_{10}$
Step 2	25_8	$(16 + 5)_{10}$
Step 3	25_8	21_{10}

Octal Number: $(25)_8$ = Decimal Number: $(21)_{10}$

STEP 2: CONVERT DECIMAL TO BINARY

Step	Operation	Result	Remainder
Step 1	21 / 2	10	1
Step 2	10 / 2	5	0
Step 3	5 / 2	2	1
Step 4	2 / 2	1	0
Step 5	1 / 2	0	1

Decimal Number: $(21)_{10}$ = Binary Number: $(10101)_2$

Octal Number: $(25)_8$ = Binary Number: $(10101)_2$

Shortcut method - Binary to Octal

Steps:

Step 1 - Divide the binary digits into groups of three (starting from the right).
Step 2 - Convert each group of three binary digits to one octal digit.

Example: Binary Number: $(10101)_2$

Calculating Octal Equivalent:

Step	Binary Number	Octal Number
Step 1	10101_2	010 101
Step 2	10101_2	$2_8\ 5_8$
Step 3	10101_2	25_8

Binary Number: $(10101)_2$ = Octal Number: $(25)_8$

b i i z a y @ g m a i l . c o m

Shortcut method - Octal to Binary

Steps:

Step 1 - Convert each octal digit to a 3 digit binary number (the octal digits may be treated as decimal for this conversion).
Step 2 - Combine all the resulting binary groups (of 3 digits each) into a single binary number.

Example: Octal Number: $(25)_8$

Calculating Binary Equivalent:

Step	Octal Number	Binary Number
Step 1	25_8	$2_{10}\ 5_{10}$
Step 2	25_8	$010_2\ 101_2$
Step 3	25_8	010101_2

Octal Number: $(25)_8$ = Binary Number: $(10101)_2$

Shortcut method - Binary to Hexadecimal

Steps:

Step 1 - Divide the binary digits into groups of four (starting from the right).
Step 2 - Convert each group of four binary digits to one hexadecimal symbol.

Example: Binary Number: $(10101)_2$

Calculating hexadecimal Equivalent:

Step	Binary Number	Hexadecimal Number
Step 1	10101_2	0001 0101
Step 2	10101_2	$1_{10}\ 5_{10}$
Step 3	10101_2	15_{16}

Binary Number: $(10101)_2$ = Hexadecimal Number: $(15)_{16}$

Shortcut method - Hexadecimal to Binary

Steps:

Step 1 - Convert each hexadecimal digit to a 4 digit binary number (the hexadecimal digits may be treated as decimal for this conversion).
Step 2 - Combine all the resulting binary groups (of 4 digits each) into a single binary number.

Example: Hexadecimal Number: $(15)_{16}$

b i i z a y @ g m a i l . c o m

Calculating Binary Equivalent:

Step	Hexadecimal Number	Binary Number
Step 1	15_{16}	$1_{10}\ 5_{10}$
Step 2	15_{16}	$0001_2\ 0101_2$
Step 3	15_{16}	00010101_2

Hexadecimal Number: $(15)_{16}$ = Binary Number: $(10101)_2$

Complements

Complements are used in the digital computers in order to simplify the subtraction operation and for the logical manipulations. For each radix-r system (radix r represent base of number system) there are two types of complements:

R's Complement

The radix complement is referred to as the r's complement. R's complement of a number N is defined as $\mathbf{r^n - N}$

Where, N is the given number
r is the base of number system
n is the number of digits in the given number

To get the R's complement fast, add 1 to the low-order digit of its (R-1)'s complement

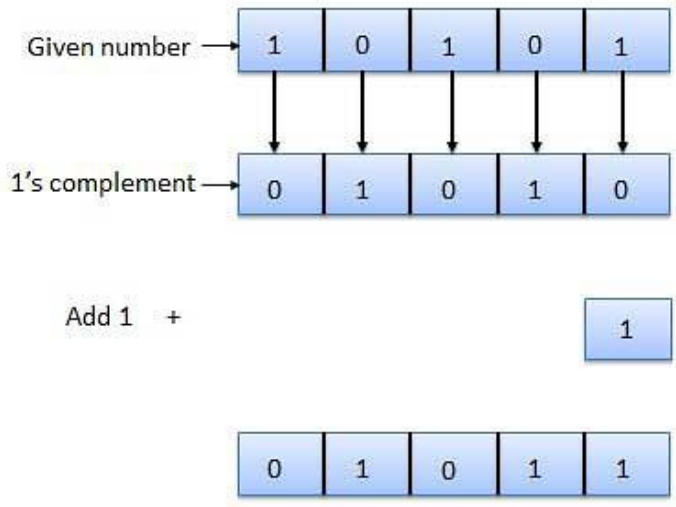
Example:

- 10's complement
- 2's complement

2's complement

The 2's complement of binary number is obtained by adding 1 to the Least Significant Bit (LSB) of 1's complement of the number. 2's complement = 1's complement + 1.

Example of 2's Complement is as follows:



b i i z a y @ g m a i l . c o m

Example 1:
Using 2's complement, subtract 1010100 – 1000011

X – Y

X	=	1010100
2's complement of Y	=	+ 0111101
Sum	=	10010001
Discard end carry 2^7	=	- 10000000
Answer: X - Y	=	0010001

Example 2:
Using 2's complement, subtract 1000011 – 1010100

Y – X

Y	=	1000011
2's complement of X	=	+0101100
Sum	=	1101111
No end carry		
Answer: Y – X - (2's complement of 1101111)	=	-0010001

10's complement
We have to add 1 with the 9's complement of any number to obtain the desired 10's complement of that number. Or, if we want to find out the 10's complement directly, we can do it by following the following formula, (10^n - number), where n = number of digits in the number. An example is given below to illustrate the concept of obtaining 10's complement.

Example:
The 10's complement of 546700 is 1000000 – 546700 = 453300

1	0	0	0	0	0	0
–	5	4	6	7	0	0
	4	5	3	3	0	0

The 10's complement of 12389 is 100000 - 12389 = 87611

1	0	0	0	0	0	
–		1	2	3	8	9
		8	7	6	1	1

b i i z a y @ g m a i l . c o m

Other Examples:
The 10's complement of decimal 2389 = $(10^4 - 1) - 2389 + 1 = 7611$.
The 10's complement of decimal 012389 = $(10^6 - 1) - 012389 + 1 = 987602$
The 10's complement of decimal 246700 = $(10^6 - 1) - 246700 + 1 = 753300$

Example 1:
Using 10's complement, subtract 72532 – 3250

M – N

M	= 72532
10's complement of N	= +96750 (99999 – 03250) + 1
Sum	= 169282
Discard end carry 10^5	= -100000
Answer:	= 69282

Example 2:
Using 10's complement, subtract 3250 – 72532

M – N

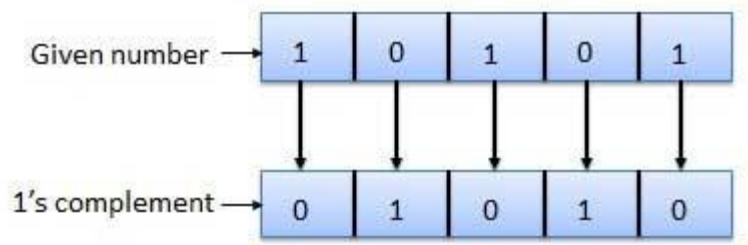
M	= 03250
10's complement of N	= +27468 (99999 – 72532) + 1
Sum	= 30718
No end carry	
Answer: - (10's complement of 30718)	= - 69282

(R-1)'s Complement
The diminished radix complement is referred to as the (r-1)'s complement. (R-1)'s complement of a number N is defined as **$(r^n - 1) - N$**.
Where, N is the given number
r is the base of number system
n is the number of digits in the given number

To get the (R-1)'s complement fast, subtract each digit of a number from (R-1)

Example:
- 9's complement
- 1's complement

1's complement
The 1's complement of a number is found by changing all 1's to 0's and all 0's to 1's. This is called as taking complement or 1's complement. Example of 1's Complement is as follows:



Other Examples:
The 1's complement of 1011000 is 0100111
The 1's complement of 0101101 is 1010010

Example 1:
Using 1's complement, subtract X – Y = 1010100 – 1000011

	X	=	1010100
	1's complement of Y	= +	0111100 (+1 End-around carry)
	Sum	=	10010000
			+ 1
Answer:	X - Y	=	0010001

Example 2:
Using 1's complement, subtract Y – X 1000011 – 1010100

Y	=	1000011
1's complement of X	= +	0101011
Sum	=	1101110
No end carry		
Answer: Y – X - (1's complement of 1101110)	=	-0010001

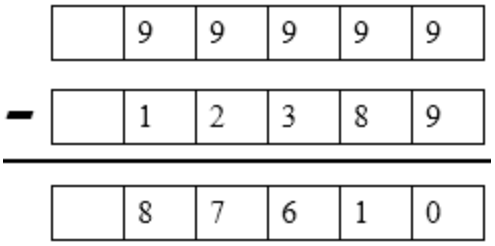
9's complement
To obtain the 9's complement of any number we have to subtract the number with $(10^n - 1)$ where n = number of digits in the number, or in a simpler manner we have to divide each digit of the given decimal number with 9.

Example: The 9's complement of 546700 is 999999 - 546700= 453299

	9	9	9	9	9	9
–	5	4	6	7	0	0
	4	5	3	2	9	9

b i i z a y @ g m a i l . c o m

Example: The 9's complement of 12389 is 99999 - 12389 = 87610



Other Examples:

- The 9's complement of 546700 is 999999 – 546700 = 453299
- The 9's complement of 012398 is 999999 – 012398 = 987601

9's complement subtraction:

Example 1:
A = 215
B = 155
We want to find out A-B by 9's complement subtraction method. First we have to find out 9's complement of B.

```
 999
-155
 844
```

Now we have to add 9's complement of B to A

```
 215
+844
1059
```

The left most bit of the result is called carry and is added back to the part of the result without it

```
 059
  +1
 060
```

This is the final answer.

Example 2:
A = 4567
B = 1234
We need to find out A – B. 9's complement of B is

b i i z a y @ g m a i l . c o m

```
  9999
-1234
  8765
```

Adding 9's complement of B with A, we get

```
  4567
+8765
13332
```

Adding the carry with the result we get

```
  3332
   +1
  3333
```

Now the answer is – (3333), because if there is no carry the answer will be – (9's complement of the answer).

Fixed point representations

Fixed point representation is a method of storing numbers in binary format. Fixed point refers to a method of representing numbers with a fractional part on an ALU that only handles integer operations.

Integer Representation

Integers are whole numbers or fixed-point numbers with the radix point fixed after the least-significant bit. They are contrast to real numbers or floating-point numbers, where the position of the radix point varies. It is important to take note that integers and floating-point numbers are treated differently in computers.

Computers use a fixed number of bits to represent an integer. The commonly-used bit-lengths for integers are 8-bit, 16-bit, 32-bit or 64-bit. Besides bit-lengths, there are two representation schemes for integers:

1. ***Unsigned Integers***: can represent zero and positive integers.
2. ***Signed Integers:*** can represent zero, positive and negative integers. Three representation schemes had been proposed for signed integers:
    a. Sign-Magnitude representation
    b. 1's Complement representation
    c. 2's Complement representation

Advantages and Disadvantages of unsigned notation
Advantages:

- One representation of zero
- Simple addition

b i i z a y @ g m a i l . c o m

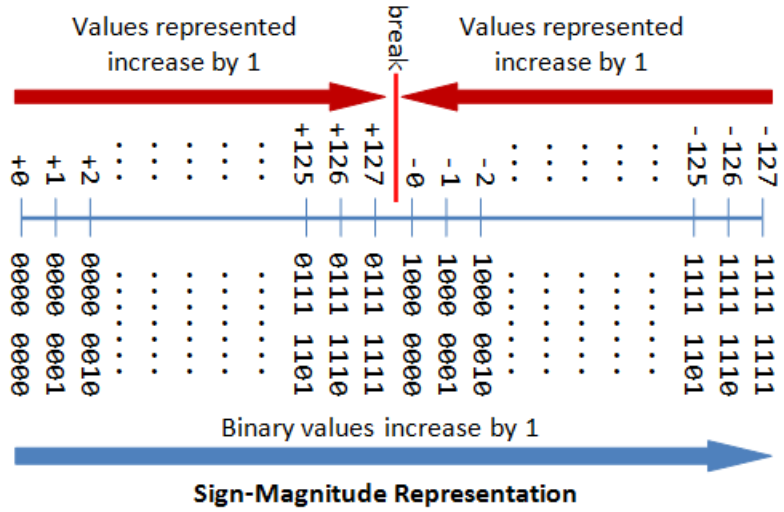
Disadvantages

- Negative numbers cannot be represented.
- The need of different notation to represent negative numbers.

In all the above three schemes, the most-significant bit (MSB) is called the sign bit. The sign bit is used to represent the sign of the integer - with 0 for positive integers and 1 for negative integers. The magnitude of the integer, however, is interpreted differently in different schemes.

Sign-Magnitude Representation

- sign bit is 0 for positive, 1 for negative
- magnitude part = absolute value of number
- 2 representations for 0. (0000 and 1000)



Sign-Magnitude Representation

Example 1: Suppose that n=8 and the binary representation is $(0\ 100\ 0001)_2$.

Sign bit is 0 ⇒ positive
Absolute value is $(100\ 0001)_2 = (65)_{10}$
Hence, the integer is $+(65)_{10}$

Example 2: Suppose that n=8 and the binary representation is $(1\ 000\ 0001)_2$.

Sign bit is 1 ⇒ negative
Absolute value is $(000\ 0001)_2 = (1)_{10}$
Hence, the integer is $-(1)_{10}$

Example 3: Suppose that n=8 and the binary representation is $(0\ 000\ 0000)_2$.

Sign bit is 0 ⇒ positive
Absolute value is $(000\ 0000)_2 = (0)_{10}$
Hence, the integer is $+(0)_{10}$

b i i z a y @ g m a i l . c o m

Example 4: Suppose that n=8 and the binary representation is $(1\ 000\ 0000)_2$.

Sign bit is 1 ⇒ negative

Absolute value is $(000\ 0000)_2 = (0)_{10}$

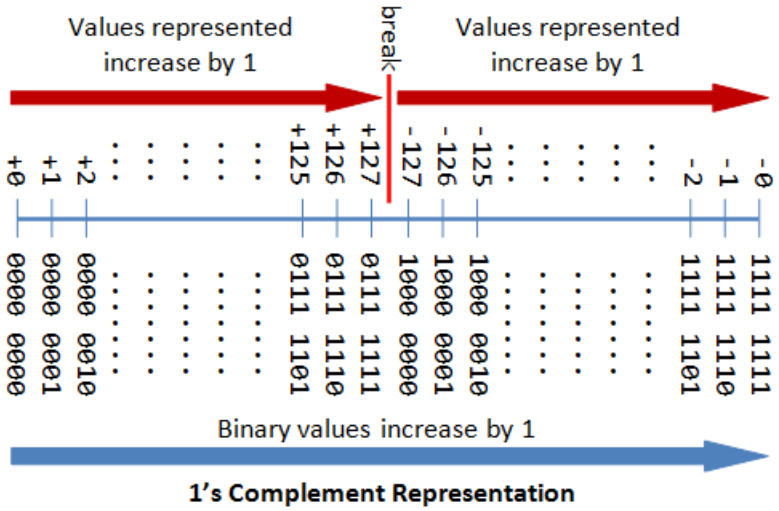
Hence, the integer is $-(0)_{10}$

The drawbacks of sign-magnitude representation are:

- Addition and subtractions are difficult.
- Signs and magnitude, both have to carry out the required operation.
- There are two representations and for the number zero, which could lead to inefficiency and confusion.
  $(0000\ 0000)_2 = +0_{10}$
  $(1000\ 0000)_2 = -0_{10}$
- Positive and negative integers need to be processed separately.

1's complement Representation

- sign bit (MSB) is 1 for negative, 0 for positive
- negation is complement, e.g. 5 = 0101, -5 = 1010
- two representations for 0: 1111 and 0000



1's Complement Representation

Example 1: Suppose that n=8 and the binary representation $(0\ 100\ 0001)_2$.

Sign bit is 0 ⇒ positive

Absolute value is $(100\ 0001)_2 = (65)_{10}$

Hence, the integer is $+(65)_{10}$

Example 2: Suppose that n=8 and the binary representation $(1\ 000\ 0001)_2$.

Sign bit is 1 ⇒ negative

Absolute value is the complement of $(000\ 0001)_2$, i.e., $(111\ 1110)_2 = (126)_{10}$

Hence, the integer is $-(126)_{10}$

b i i z a y @ g m a i l . c o m

Example 3: Suppose that n=8 and the binary representation $(0\ 000\ 0000)_2$.

Sign bit is 0 ⇒ positive

Absolute value is $(000\ 0000)_2 = (0)_{10}$

Hence, the integer is $+(0)_{10}$

Example 4: Suppose that n=8 and the binary representation $(1\ 111\ 1111)_2$.

Sign bit is 1 ⇒ negative

Absolute value is the complement of $(111\ 1111)_2$, i.e., $(000\ 0000)_2 = (0)_{10}$

Hence, the integer is $-(0)_{10}$

Again, the drawbacks are:

- There are two representations for zero.
  $(0000\ 0000)_2 = +0_{10}$
  $(1111\ 1111)_2 = -0_{10}$
- The positive integers and negative integers need to be processed separately.

2's complement Representation

- sign bit (MSB) is 1 for negative, 0 for positive
- add 1 to 2's complement negative numbers
- only one representation for 0
- negation is complement + 1, e.g. 5 = 0101, -5 = 1010 + 1 = 1011

Example 1: Suppose that n=8 and the binary representation $(0\ 100\ 0001)_2$.

Sign bit is 0 ⇒ positive

Absolute value is $(100\ 0001)_2 = (65)_{10}$

Hence, the integer is $+(65)_{10}$

Example 2: Suppose that n=8 and the binary representation $(1\ 000\ 0001)_2$.

Sign bit is 1 ⇒ negative

Absolute value is the complement of $(000\ 0001)_2$ plus 1, i.e., $(111\ 1110)_2 + (1)_2 = (127)_{10}$

Hence, the integer is $-(127)_{10}$

Example 3: Suppose that n=8 and the binary representation $(0\ 000\ 0000)_2$.

Sign bit is 0 ⇒ positive

Absolute value is $(000\ 0000)_2 = (0)_{10}$

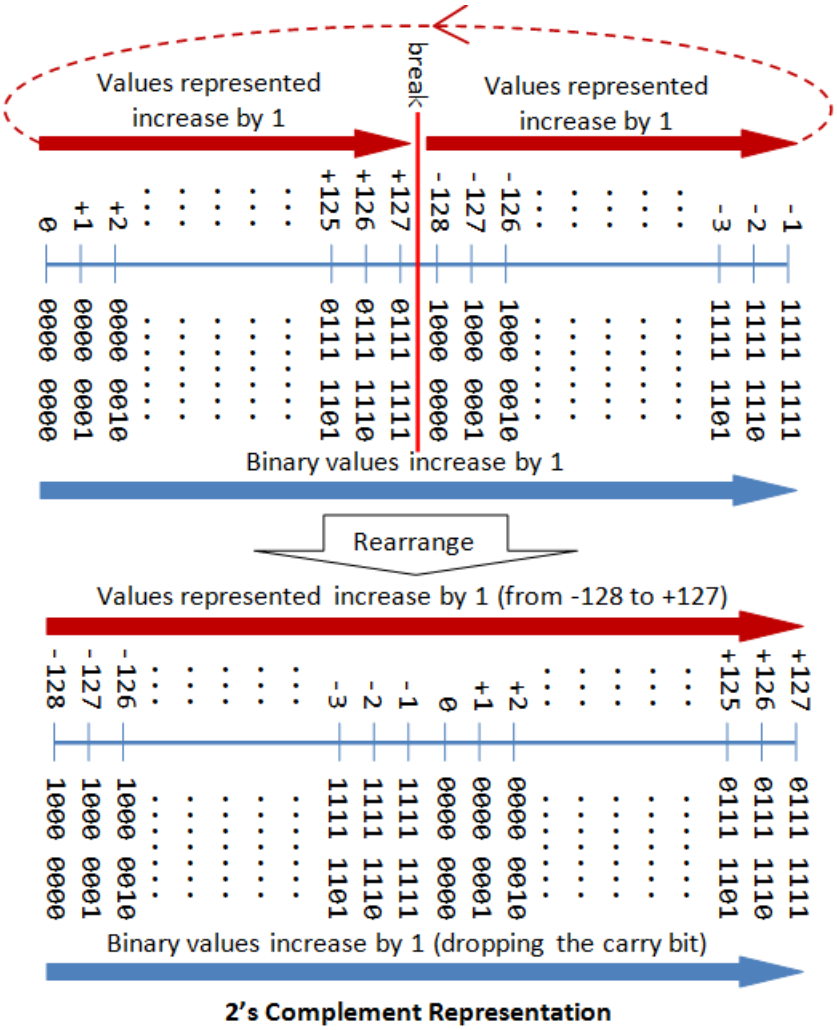
Hence, the integer is $+(0)_{10}$

Example 4: Suppose that n=8 and the binary representation $(1\ 111\ 1111)_2$.

Sign bit is 1 ⇒ negative

Absolute value is the complement of $(111\ 1111)_2$ plus 1, i.e., $(000\ 0000)_2 + (1)_2 = (1)_{10}$

Hence, the integer is $-(1)_{10}$



2's Complement Representation

Arithmetic Addition and Subtraction of Signed Numbers

Arithmetic Addition

- The addition of 2 numbers in the signed-magnitude system follows the rules of ordinary arithmetic.
- Compare their signs, if the signs are the same, we add the two magnitudes and give the sum the common sign. Look for Overflow.
- If the signs are different, we subtract the smaller magnitudes from the larger and give the result the sign of the larger magnitude.
- Negative numbers must be in 2's complement and that the sum obtained after the addition if negative is in 2's-complement form.

b i i z a y @ g m a i l . c o m

+6	00000110	-6	11111010
+13	00001101	+13	00001101
+19	00010011	+7	00000111

+6	00000110	-6	11111010
-13	11110011	-13	11110011
-7	11111001	-19	11101101

6 + 9

6	0110
+ 9	1001
15	1111 = 01111

-6 + 9

9	1001
- 6	0110
3	0011 = 00011

6 + (- 9)

9	1001
- 6	0110
- 3	0011 = 10011

-6 + (-9)

6	0110
+ 9	1001
-15	1111 = 11111

Overflow: 9 + 9 or (-9) + (-9)

9	1001
+ 9	1001
overflow (1)	0010

In each of the 4 cases, the operation performed is always addition, including the sign-bits. Any carry out of the sign bit is discarded and negative results are automatically in 2's complement form.

Arithmetic Subtraction

Take the 2's complement of the subtrahend (including the sign bit) and add it to the minuend (including the sign bit). A carry out of the sign-bit position is discarded.

$(\pm A) - (+B) = (\pm A) + (-B)$
$(\pm A) - (-B) = (\pm A) + (+B)$

Example:

(-6) - (-13) = +7, in binary with 8-bits this is written as:

-6 → 11111010

b i i z a y @ g m a i l . c o m

-13 → 11110011 (2's complement form)

Subtraction is changed to addition by taking 2's complement of the subtrahend (-13) to give (+13).

-6 → 11111010
+13 → 00001101
+7 → 100000111 (discarding end carry).

Overflow

When two numbers of n digits are added and the sum occupies n+1 digits, we say that an overflow has occurred. A result that contains n+1 bits can't be accommodated in a resister with a standard length of n-bits. For this reason many computers detect the occurrence of an overflow setting corresponding flip-flop. An overflow may occur if two numbers added are both positive or both negative.

Example: Two signed binary numbers +70 and +80 are stored in two 8-bit resisters.

+70	0	1000110	-70	1	0111010
+80	0	1010000	-80	1	0110000
+150	1	0010110	-150	0	1101010

Since the sum of numbers 150 exceeds the capacity of the resister (since 8-bit resister can store values ranging from +127 to -128), hence the overflow.

Overflow Detection

An overflow condition can be detected by observing two carries: carry into the sign bit position and carry out of the sign bit position. Consider example of above 8-bit resister, if we take the carry out of the sign bit position as a sign bit of the result, 9-bit answer so obtained will be correct. Since answer cannot be accommodated within 8-bits, we say that an overflow occurred.

If these two carries are equal ==> no overflow
If these two carries are not same ==> overflow condition is produced.

If two carries are applied to an exclusive-OR gate, an overflow will be detected when output of the gate is equal to 1.

Decimal Fixed-Point Representation

Decimal number representation = f (binary code used to represent each decimal digit). Output of this function is called the Binary coded Decimal (BCD). A 4-bit decimal code requires 4 flip-flops for each decimal digit.

Example: 4385 = $(0100\ 0011\ 1000\ 0101)_{BCD}$

Disadvantages of BCD representation:

b i i z a y @ g m a i l . c o m

- wastage of memory
- Circuits for decimal arithmetic are quite complex.

Advantages of BCD representation:

- Eliminate the need for conversion to binary and back to decimal. (since applications like Business data processing requires less computation than I/O of decimal data, hence electronic calculators perform arithmetic operations directly with the decimal data (in binary code))

For the representation of signed decimal numbers in BCD, sign is also represented with 4-bits, plus with 4 0's and minus with 1001 (BCD equivalent of 9). Negative numbers are in 10's complement form.

Consider the Addition: (+375) + (-240) = +135 [0→positive, 9→negative in case of radix 10]

$$\begin{array}{rll} 0 & 375 & (0000\ 0011\ 0111\ 1010)_{BCD} \\ +9 & 760 & (1001\ 0111\ 0110\ 0000)_{BCD} \\ \hline 0 & 135 & (0000\ 0001\ 0011\ 0101)_{BCD} \end{array}$$

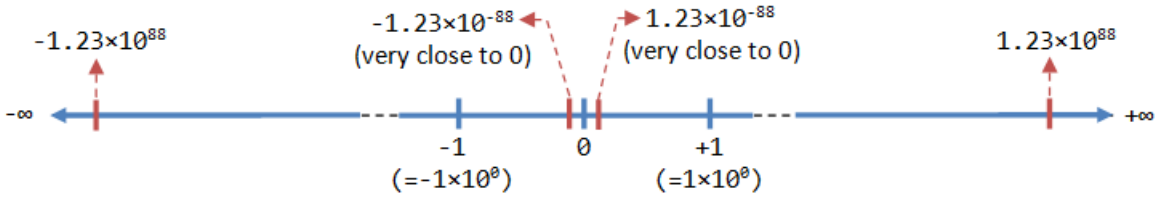
To obtain the 10's complement of a BCD number, first take the 9's complement and then add one to the least significant digit.

Limitation of Fixed-Point Representation

To represent large numbers or very small numbers we need a very long sequences of bits. This is because we have to give bits to both the integer part and the fraction part.

Floating Point Representation

A floating-point number (or real number) can represent a very large (1.23×10^{88}) or a very small (1.23×10^{-88}) value. It could also represent very large negative number (-1.23×10^{88}) and very small negative number (-1.23×10^{88}), as well as zero, as illustrated:



Floating-point Numbers (Decimal)

In the decimal system there are 2 ways of floating point representation: Scientific Notation and Floating point notation.

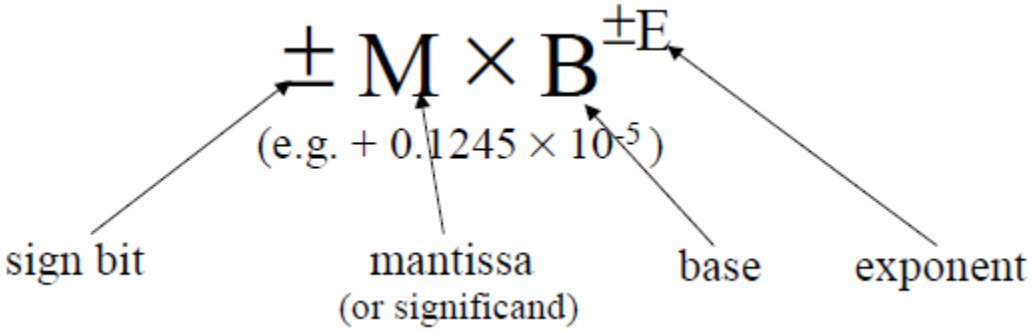
A floating-point number is typically expressed in the scientific notation, with a fraction (F), and an exponent (E) of a certain radix (r), in the form of $F \times r^E$. Decimal numbers use radix of 10 ($F \times 10^E$); while binary numbers use radix of 2 ($F \times 2^E$).

b i i z a y @ g m a i l . c o m

Representation of floating point number is not unique. For example, the number 55.66 can be represented as 5.566×10^{1}, 0.5566×10^{2}, 0.05566×10^{3}, and so on. The fractional part can be normalized. In the normalized form, there is only a single non-zero digit before the radix point. For example, decimal number 123.4567 can be normalized as 1.234567×10^{2}; binary number $(1010.1011)_2$ can be normalized as $(1.011011)_2\times2^{3}$.

Example:

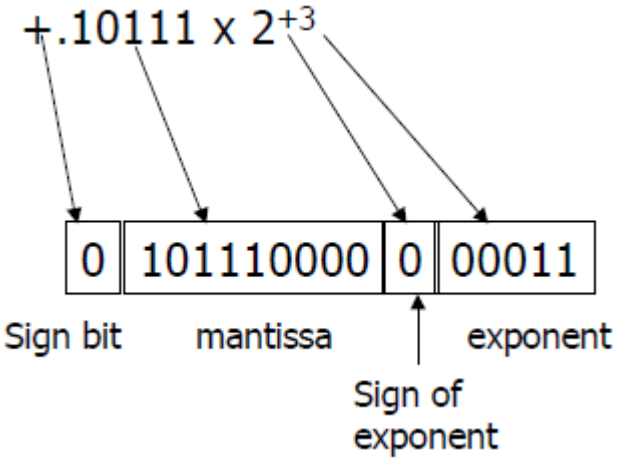
	Scientific notation	***Floating point notation***
1,245,000,000,000 =	1.245×10^{12}	0.1245×10^{13}
0.00001245 =	1.245×10^{-5}	0.1245×10^{-4}
-0.00001245 =	-1.245×10^{-5}	-0.1245×10^{-4}



Thus, there are three parts in the floating-point representation:

a. The **sign bit** (S) is self-explanatory (0 for positive numbers and 1 for negative numbers).
b. For the **exponent** (E), a so-called **bias** (or excess) is applied so as to represent both positive and negative exponent. The bias is set at half of the range. For single precision with an 8-bit exponent, the bias is 127 (or excess-127). For double precision with a 11-bit exponent, the bias is 1023 (or excess-1023).
c. The **fraction** (F) (also called the **mantissa** or **significand**) is composed of an implicit leading bit (before the radix point) and the fractional bits (after the radix point). The leading bit for normalized numbers is 1; while the leading bit for denormalized numbers is 0.

Suppose we use 16 bit words



b i i z a y @ g m a i l . c o m

It is important to note that floating-point numbers suffer from loss of precision when represented with a fixed number of bits (e.g., 32-bit or 64-bit). This is because there are infinite number of real numbers (even within a small range of says 0.0 to 0.1).

It is also important to note that floating number arithmetic is very much less efficient than integer arithmetic. It could be speed up with a so-called dedicated floating-point co-processor. Hence, use integers if your application does not require floating-point numbers.

In computers, floating-point numbers are represented in scientific notation of fraction (F) and exponent (E) with a radix of 2, in the form of $F \times 2^E$. Both E and F can be positive as well as negative.

Note:
In Floating Point Number representation, only Mantissa (M) and Exponent (E) are explicitly represented. The Radix (R) and the position of the Radix Point are implied.

Example
A binary number +1001.11 in 16-bit floating point number representation (6-bit exponent and 10-bit fractional mantissa):

	0	0 00100	100111000
	Sign	Exponent	Mantissa
or	0	0 00101	010011100

Normalization
A floating-point number is said to be normalized if the most significant digit of the mantissa is nonzero. For example, decimal number 350 is normalized but 00035 is not. The 8-bit number 00011010 is not normalized. Normalize it by fraction = 11010000 and exponent = -3

Representation of Zero
- Zero
  Mantissa = 0

- Real Zero
  Mantissa = 0
  Exponent = smallest representable number which is represented as 00 ... 0

Binary and Decimal Codes

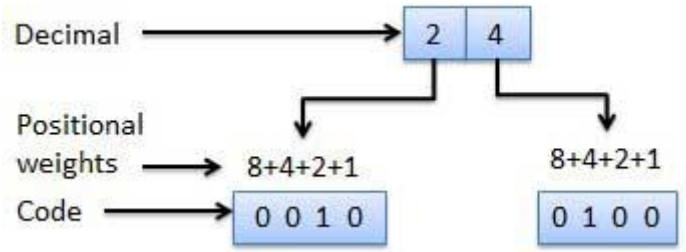
Internally, digital computers operate on binary numbers. When interfacing to humans, digital processors, e.g. pocket calculators, communication is decimal-based. Input is done in decimal then converted to binary for internal processing. For output, the result has to be converted from its internal binary representation to a decimal form.

To be handled by digital processors, the decimal input (output) must be coded in binary in a digit by digit manner. For example, to input the decimal number 957, each digit of the number is individually coded and the number is stored as 100101010111. Thus, we need a specific code for each of the 10 decimal digits. There is a variety of such codes, such as:

1. Weighted codes
2. Non-Weighted codes
3. Self-Complementing codes
4. Reflective codes
5. Alphanumeric codes
6. Error detecting and correcting codes

1. Weighted Codes

Weighted binary codes are those binary codes which obey the positional weight principle. Each position of the number represents a specific weight. Several systems of the codes are used to express the decimal digits 0 through 9. In these codes each decimal digit is represented by a group of four bits.



Each bit has a positional value of 8, 4, 2 or 1 in binary codes. Examples are: 8421, 2421, 3321, 4221, 5211, 5311, 5421, 6311, 7421, 742'1', 842'1'. All the above codes are used to represent a given decimal digit into four bit binary word.

Decimal Number	8421	2421	3321	4221	5311	5421	6311	7421	742'1'	842'1'
0	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
1	0001	0001	0001	0001	0001	0001	0001	0001	0111	0111
2	0010	0010	0010	0010	0011	0010	0011	0010	0110	0110
3	0011	0011	0011	0011	0100	0011	0100	0011	0101	0101
4	0100	0100	0101	1000	0101	0100	0101	0100	0100	0100
5	0101	1011	0110	0111	1000	0101	0111	0101	1010	1011
6	0110	1100	0111	1100	1001	0110	1000	0110	1001	1010
7	0111	1101	1101	1101	1010	0111	1001	0111	1000	1001
8	1000	1110	1110	1110	1100	1011	1011	1001	1111	1000
9	1001	1111	1111	1111	1101	1100	1100	1010	1110	1111

b i i z a y @ g m a i l . c o m

2. Non-Weighted Codes:

Each bit has no positional value

1. Excess-3 code
2. Gray code
3. Five bit BCD

3. Self-Complementing codes or Reflective codes

Code for one digit will be the complement of other

1. 2421
2. 5211
3. Excess-3

4. Sequential Codes

Succeeding number is one more than the previous one

1. 8421
2. Excess-3

5. Alphanumeric codes

1. American Standard Code for Information Interchange (ASCII)
2. Extended Binary Coded Decimal Interchange Code (EBCDIC)
3. Hollerith Code
4. Five bit Baudot Code
5. Morse Code

6. Error Detection and Correction Codes

For reliable transmission and storage of digital data, error detection and correction is required. Examples of codes which permit error detection and error correction are: Parity Codes, Hamming Code, etc.

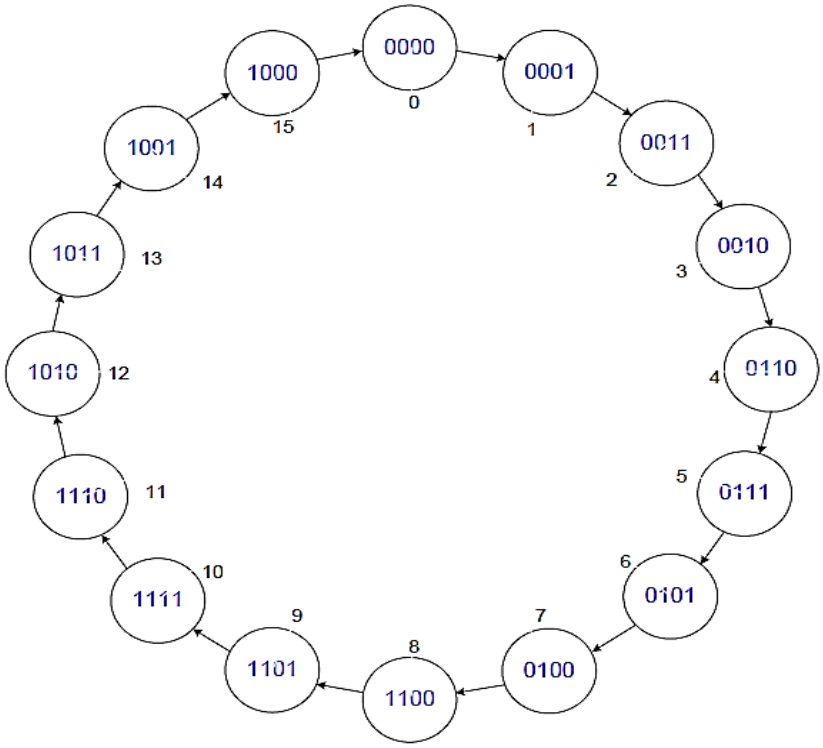
Gray Code:

It is the non-weighted code and it is not arithmetic codes. That means there are no specific weights assigned to the bit position. It has a very special feature that has only one bit will change, each time the decimal number is incremented as shown in fig. As only one bit changes at a time, the gray code is called as a ***unit distance code***. The gray code is a ***cyclic code***. Gray code cannot be used for arithmetic operation.

Decimal	BCD	Gray
0	0 0 0 0	0 0 0 0
1	0 0 0 1	0 0 0 1
2	0 0 1 0	0 0 1 1
3	0 0 1 1	0 0 1 0
4	0 1 0 0	0 1 1 0
5	0 1 0 1	0 1 1 1
6	0 1 1 0	0 1 0 1
7	0 1 1 1	0 1 0 0
8	1 0 0 0	1 1 0 0
9	1 0 0 1	1 1 0 1

b i i z a y @ g m a i l . c o m

The Gray code consists of sixteen 4-bit code words to represent the decimal Numbers 0 to 15. For Gray code, successive code words differ by only one bit from one to the next as shown in the table above and further illustrated in the Figure below.



Application of Gray Code

- Gray code is popularly used in the shaft position encoders.
- A shaft position encoder produces a code word which represents the angular position of the shaft.

BCD Code:

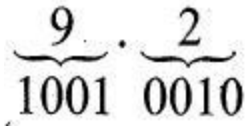
One commonly used code is the **Binary Coded Decimal (BCD)** also known as **packet decimal** code which corresponds to the first 10 binary representations of the decimal digits 0-9. The BCD code requires 4 bits to represent the 10 decimal digits. In the BCD, with four bits we can represent sixteen numbers (0000 to 1111). But in BCD code only first ten of these are used (0000 to 1001). A total of 6 combinations will be unused. The position weights of the BCD code are 8, 4, 2, 1.

Group of 4 binary bits is a **nibble**. A nibble representing a number greater than 9 is invalid BCD. The remaining six code combinations i.e. 1010 to 1111 are invalid in BCD. Below is a list of the decimal numbers 0 through 9 and the binary conversion.

Decimal	0	1	2	3	4	5	6	7	8	9
BCD	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001

b i i z a y @ g m a i l . c o m

The 8421 BCD code for 9.2 is 1001.0010.



BCD for 9.2

The 4221 BCD code for 9.2 is 1111.0010.
The 5421 BCD code for 9.2 is 1100.0010.

Note: The numbers 4, 2, 2, 1 in 4221 BCD and 5, 4, 2 and 1 in 5421 BCD represent weights of the relevant bits.

Advantages of BCD Codes

- It is very similar to decimal system.
- We need to remember binary equivalent of decimal numbers 0 to 9 only, such as those found in digital clocks or digital voltmeters.

Disadvantages of BCD Codes

- The addition and subtraction of BCD have different rules.
- The BCD arithmetic is little more complicated.
- BCD needs more number of bits than binary to represent the decimal number. So BCD is less efficient than binary.

ASCII Code:

The American Standard-Code for Information Interchange (ASCII) pronounced "as-kee" is a 7-bit code based on the ordering of the English alphabets. The ASCII codes are used to represent alphanumeric data in computer input/output. It represents a total of 128 characters.

These include 95 printable characters including 26 upper-case letters (A to Z), 26 lowercase letters (a to z), 10 numerals (0 to 9) and 33 special characters such as mathematical symbols, space character etc. It also defines codes for 33 non-printing obsolete characters except for carriage return and/or line feed. The below table lists the 7 bit ASCII code containing the 95 printable characters.

The format of ASCII code for each character is $X_6X_5X_4X_3X_2X_1X_0$ where each X is 0 or 1. For instance, letter D is coded as 1000100.

More examples are:
The ASCll-7 code for ‘d’ is 1100100 as seen from the table.
The ASCll-7 code for '+' is 0101011 as seen from the table.

An eight-bit version of the ASCII code, known as US ASCII-8 or ASCII-8, has also been developed. Since it uses 8-bits, so this version of ASCII can represent a maximum of 256 characters.

(Digits) $X_3X_2X_1X_0$	$X_6X_5X_4$ (Zoned bits) 010	011	100	101	110	111
0000	SP	0	@	P		p
0001	!	1	A	Q	a	q
0010	"	2	B	R	b	r
0011	#	3	C	S	c	s
0100	$	4	D	T	d	t
0101	%	5	E	U	e	u
0110	&	6	F	V	f	v
0111	(	7	G	W	g	w
1000	)	8	H	X	h	x
1001	*	9	I	Y	i	y
1010	*	:	J	Z	j	z
1011	+	;	K	[	k	{
1100	,	<	L	\	l	;
1101	-	=	M	]	m	y
1110	.	>	N	^	n	~
1111	/	?	O	-	o	DEL

ASCII - 7 Code table containing 95 printable characters

Digits $X_3X_2X_1X_0$	$X_7X_6X_5X_4$ (Zoned bits) 0101	1010	1011
0000	0		P
0001	I	A	Q
0010	2	B	R
0011	3	C	S
0100	4	D	T
0101	S	E	U
0110	6	F	V
0111	7	G	X
1000	8	H	Y
1001	9	I	Z
1010		J	
1011		K	
1100		L	
1101		M	
1110		N	
1111		O	

ASCII-8 Code table containing codes for some characters

Example 1: With an ASCII-7 keyboard, each keystroke produces the ASCII equivalent of the designated character. Suppose that you type PRINT X. What is the output of an ASCII-7 keyboard?

b i i z a y @ g m a i l . c o m

Solution: The sequence is as follows:
The ASCII-7 equivalent of P = 101 0000
The ASCII-7 equivalent of R = 101 0010
The ASCII-7 equivalent of I = 1001010
The ASCII-7 equivalent of N = 100 1110
The ASCII-7 equivalent of T = 1-010100
The ASCII-7 equivalent of space = 010 0000
The ASCII-7 equivalent of X = 101 1000

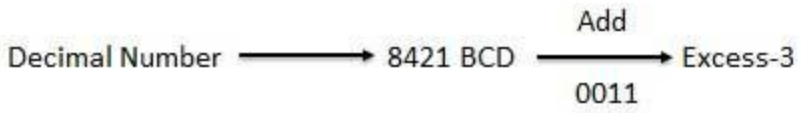
So the output produced is 10100001010010100101010011101010100010000001011000. The output in hexadecimal equivalent is 50 52 49 4E 54 30 58

Example2: A computer sends a message to another computer using an odd-parity bit. Here is the message in ASCII-8 Code.
1011 0001
1011 0101
1010 0101
1010 0101
1010 1110
What do these numbers mean?

Solution
On translating, the 8-bit numbers into their equivalent ASCII-8 code we get the word 1011 0001 (Q), 10110101 (U), 10100101 (E), 1010 0101 (E), 1010 1110 (N)

So, on translation we get QUEEN as the output.

Excess-3 Code:
The Excess-3 code is also called as XS-3 code. It is non-weighted code used to express decimal numbers. The Excess-3 code words are derived from the 8421 BCD code words adding $(0011)_2$ or $(3)_{10}$ to each code word in 8421. The excess-3 codes are obtained as follows:



An Example: The code of a decimal 0 is 0011, that of 6 is 1001, etc. Some other are shown in table below:

b i i z a y @ g m a i l . c o m

Decimal	BCD 8 4 2 1	Excess-3 BCD + 0011
0	0 0 0 0	0 0 1 1
1	0 0 0 1	0 1 0 0
2	0 0 1 0	0 1 0 1
3	0 0 1 1	0 1 1 0
4	0 1 0 0	0 1 1 1
5	0 1 0 1	1 0 0 0
6	0 1 1 0	1 0 0 1
7	0 1 1 1	1 0 1 0
8	1 0 0 0	1 0 1 1
9	1 0 0 1	1 1 0 0

Advantages of Binary Code
Following is the list of advantages that binary code offers.

- Binary codes are suitable for the computer applications.
- Binary codes are suitable for the digital communications.
- Binary codes make the analysis and designing of digital circuits if we use the binary codes.
- Since only 0 & 1 are being used, implementation becomes easy.

Binary to BCD Conversion

Steps:

Step 1 -- Convert the binary number to decimal.
Step 2 -- Convert decimal number to BCD.

Example: convert $(11101)_2$ to BCD.

Step 1 - Convert to Decimal
Binary Number: $(11101)_2$

Calculating Decimal Equivalent:

Step	Binary Number	Decimal Number
Step 1	11101_2	$((1 \times 2^4) + (1 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0))_{10}$
Step 2	11101_2	$(16 + 8 + 4 + 0 + 1)_{10}$
Step 3	11101_2	29_{10}

Binary Number: $(11101)_2$ = Decimal Number: $(29)_{10}$

b i i z a y @ g m a i l . c o m

Step 2 - Convert to BCD

Decimal Number: $(29)_{10}$

Calculating BCD Equivalent. Convert each digit into groups of four binary digits equivalent.

Step	Decimal Number	Conversion
Step 1	29_{10}	$0010_2\ 1001_2$
Step 2	29_{10}	00101001_{BCD}

Result

$(11101)_2 = (00101001)_{BCD}$

BCD to Binary Conversion

Steps

Step 1 -- Convert the BCD number to decimal.
Step 2 -- Convert decimal to binary.

Example: convert $(00101001)_{BCD}$ to Binary.

STEP 1 - CONVERT TO BCD
BCD Number: $(00101001)_{BCD}$

Calculating Decimal Equivalent. Convert each four digit into a group and get decimal equivalent or each group.

Step	BCD Number	Conversion
Step 1	$(00101001)_{BCD}$	$0010_2\ 1001_2$
Step 2	$(00101001)_{BCD}$	$2_{10}\ 9_{10}$
Step 3	$(00101001)_{BCD}$	29_{10}

BCD Number: $(00101001)_{BCD}$ = Decimal Number: $(29)_{10}$

STEP 2 - CONVERT TO BINARY
Used long division method for decimal to binary conversion.

Decimal Number: $(29)_{10}$

Calculating Binary Equivalent:

b i i z a y @ g m a i l . c o m

Step	Operation	Result	Remainder
Step 1	29 / 2	14	1
Step 2	14 / 2	7	0
Step 3	7 / 2	3	1
Step 4	3 / 2	1	1
Step 5	1 / 2	0	1

As mentioned in Steps 2 and 4, the remainders have to be arranged in the reverse order so that the first remainder becomes the least significant digit (LSD) and the last remainder becomes the most significant digit (MSD).

Decimal Number: $(29)_{10}$ = Binary Number: $(11101)_2$

Result

$(00101001)_{BCD} = (11101)_2$

BCD to Excess-3

Steps

Step 1 -- Convert BCD to decimal.
Step 2 -- Add (3)10 to this decimal number.
Step 3 -- Convert into binary to get excess-3 code.

Example: convert $(1001)_{BCD}$ to Excess-3.

STEP 1 - CONVERT TO DECIMAL
$(1001)_{BCD} = (9)_3$

STEP 2 - ADD 3 TO DECIMAL
$(9)_{10} + (3)_{10} = (12)_{10}$

STEP 3 - CONVERT TO EXCESS-3
$(12)_{10} = (1100)_2$

Result

$(1001)_{BCD} = (1100)_{XS-3}$

Excess-3 to BCD Conversion

Steps

b i i z a y @ g m a i l . c o m

Step 1 -- Subtract $(0011)_2$ from each 4 bit of excess-3 digit to obtain the corresponding BCD code.

Example: convert $(10011010)_{XS-3}$ to BCD.

Given XS-3 number = 1 0 0 1 1 0 1 0
Subtract $(0011)_2$ = 0 0 1 1 0 0 1 1

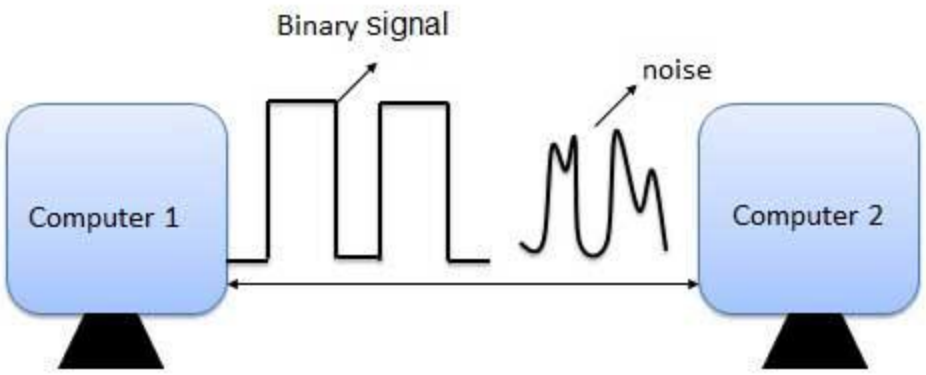
BCD = 0 1 1 0 0 1 1 1

Result

$(10011010)_{XS-3} = (01100111)_{BCD}$

Error Detection and Correction Code

Error means a condition when output information is not same as input information. When transmission of digital signals takes place between two systems such as a computer, the transmitted signal is combined with the "Noise". The noise can introduce an error in the binary bits travelling from one system to other. That means 0 may change to 1 or a 1 may change to 0.



Error Detecting Codes

Whenever a message is transmitted then there are changes that it get scrambled by noise or data gets corrupted. When we add some additional data to a given digital message which can help us to detect if an error occurred during transmission of the message then adding such data is called an error-detecting code. A simple example of error-detecting code is ***parity check***.

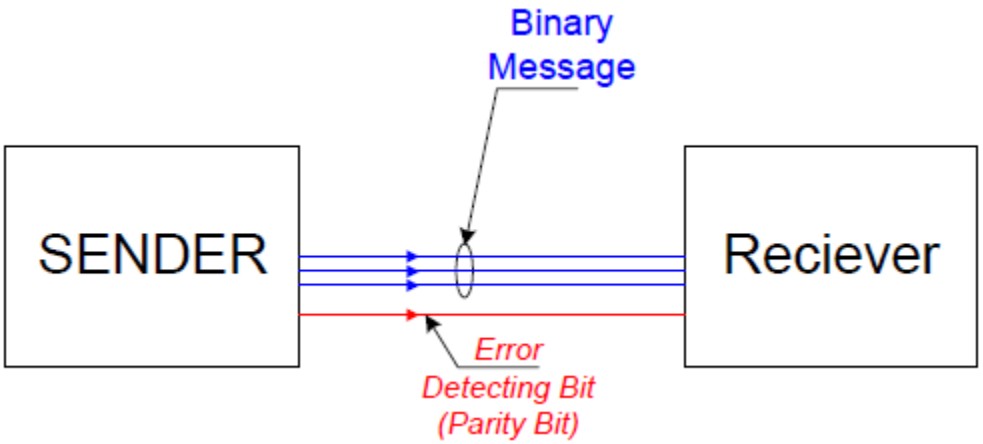
Error Correcting Codes

Along with Error detecting code, we can also pass some data which help to figure out the original message from the corrupt message that we received. This type of code is called an error-correcting code. Error correcting codes also deploys the same strategy as error detecting codes but additionally, such codes also detects the exact location of corrupt bit. In error correcting code, parity check has simple way to detect error along with a sophisticated mechanism to determine the corrupt bit location. Once corrupt bit is located, its value is reverted (from 0 to 1 or 1 to 0) to get the original message.

How to detect and correct errors?

For the detection and correction of these errors, one or more than one extra bits are added to the data bits at the time transmitting. These extra bits are called as ***parity bits***. They allow detection or correction of the errors. The data bits along with the parity bits form a ***code word***.

Binary information may be transmitted through some communication medium, e.g. using wires or wireless media. A corrupted bit will have its value changed from 0 to 1 or vice versa. To be able to detect errors at the receiver end, the sender sends an extra bit (parity bit) with the original binary message.



A parity bit is an extra bit included with the n-bit binary message to make the total number of 1's in this message (including the parity bit) either odd or even. If the parity bit makes the total number of 1's an odd (even) number, it is called odd (even) parity. The table shows the required odd (even) parity for a 3-bit message.

Three-Bit Message			Odd Parity Bit	Even Parity Bit
X	Y	Z	P	P
0	0	0	1	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	0	1

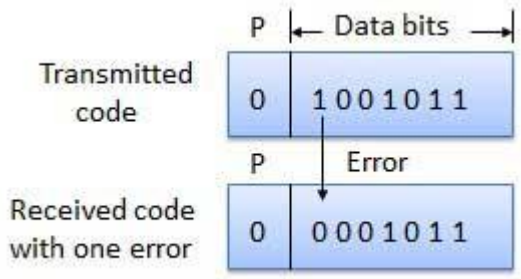
b i i z a y @ g m a i l . c o m

At the receiver end, an error is detected if the message does not match have the proper parity (odd/even). Parity bits can detect the occurrence 1, 3, 5 or any odd number of errors in the transmitted message. No error is detectable if the transmitted message has 2 bits in error since the total number of 1's will remain even (or odd) as in the original message. In general, a transmitted message with even number of errors cannot be detected by the parity bit.

Binary information may be transmitted through some communication medium, e.g. using wires or wireless media. Noise in the transmission medium may cause the transmitted binary message to be corrupted by changing a bit from 0 to 1 or vice versa. To be able to detect errors at the receiver end, the sender sends an extra bit (parity bit).

How does error detection take place?

The parity checking at the receiver can detect the presence of an error if the parity of the receiver signal is different from the expect parity. That means if is known that the parity of the transmitted signal is always going to be "even" and if the received signal has an odd parity then the receiver can conclude that the received signal is not correct. If presence of error is detected then the receiver will ignore the received byte and request for retransmission of the same byte to the transmitter.



Parity Generator

Parity generator and checker networks are logic circuits constructed with exclusive-OR functions. Consider a 3-bit message to be transmitted with an odd parity bit. At the sending end, the odd parity is generated by a parity generator circuit. The output of the parity checker would be 1 when an error occurs i.e. no. of 1's in the four inputs is even.

$P = \overline{x \oplus y \oplus z}$

Message (xyz)	Parity bit (odd)
000	1
001	0
010	0
011	1
100	0
101	1
110	1
111	0

b i i z a y @ g m a i l . c o m

Parity Checker
Considers original message as well as parity bit

$e = \overline{p \oplus x \oplus y \oplus z}$

e= 1 => No. of 1's in pxyz is even => Error in data
e= 0 => No. of 1's in pxyz is odd => Data is error free

Circuit diagram for parity generator and parity checker is shown below. At the sending end, the message is applied to a parity generator. The message, including the parity bit, is transmitted. At the receiving end, all the incoming bits are applied to a parity checker. Any odd number of errors are detected.

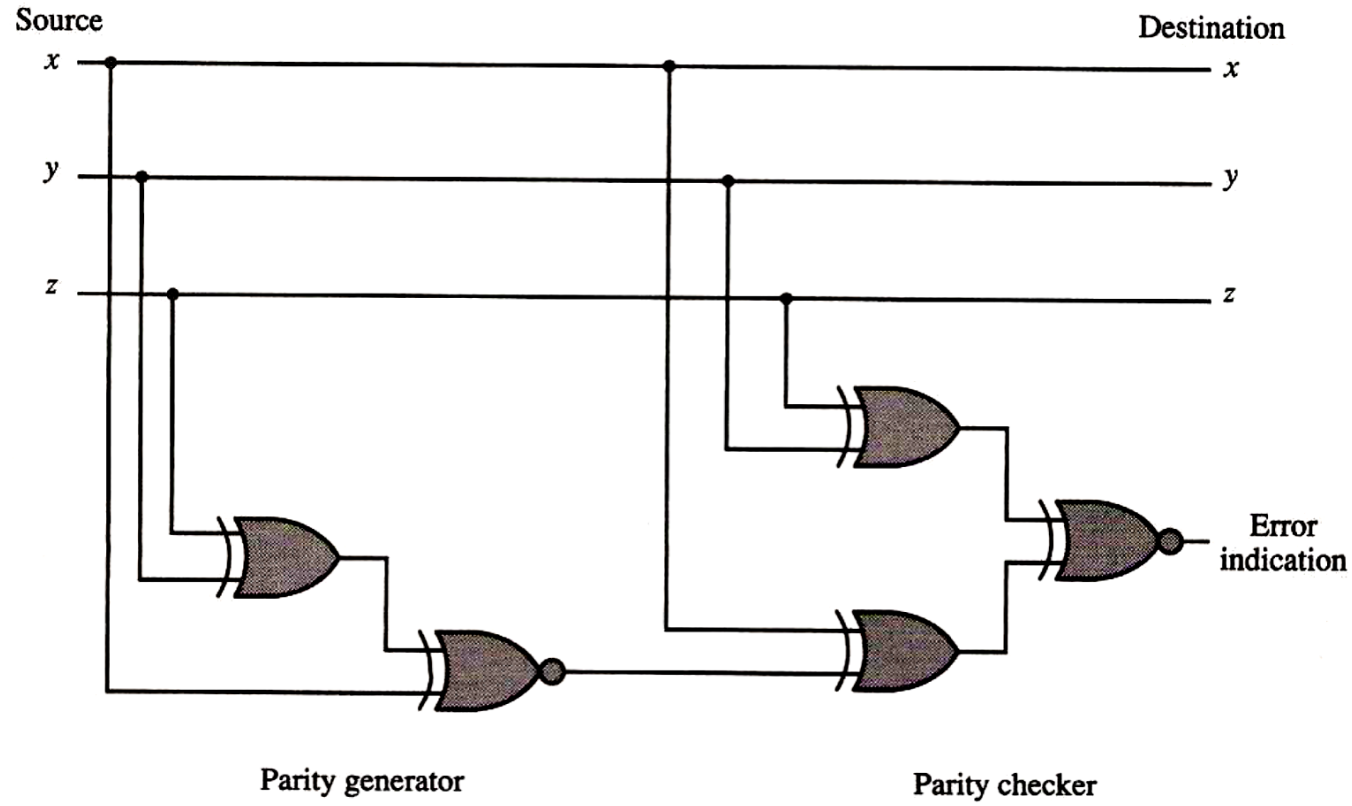


Figure: Error detection with odd parity bit

Parity generators and checkers are constructed with XOR gates (odd function). An odd function generates 1 if an odd number of input variables are 1.

Book References:

(1) Andrew S. Tanenbaum, "Structured Computer Organization", Fourth Edition.
(2) M. Morris Mano, "Computer System Architecture", Pearson, 3rd Ed, 2004.
(3) M. Morris Mano, "Logic and Computer Design Fundamentals", Pearson Education, 2nd Edition
(4) John P. Hayes, "Computer Architecture & Organization".
(5) William Stalling, "Computer Organization & Architecture".

Web References:

(6) http://en.wikipedia.org
(7) http://www.eda.kent.ac.uk/
(8) http://ecomputernotes.com/
(9) http://www.tutorialspoint.com/

Assignments:

(1) Why do 0 through 9 have ASCII values?
(2) $(37)_{10}$ has 0010 0101 in signed magnitude notation. Find the signed magnitude of $-(37)_{10}$?
(3) Using the signed magnitude notation find the 8-bit binary representation of the decimal value $(24)_{10}$ and $-(24)_{10}$.
(4) Find the signed magnitude of $-(63)_{10}$ using 8-bit binary sequence?
(5) Determine the decimal value represented by 10001011 in each of the following three systems.
   - Unsigned notation?
   - Signed magnitude notation?
   - Two's complements?
(6) Use the ASCII table to write the ASCII code for the following:
   - CIS110
   - 6 = 2*3
(7) Derive the circuit for a 3-bit parity generator and a 4-bit parity checker using even parity bit.
(8) Differentiate between parity checker and parity generator. (T.U. 2066)
(9) Explain the error detection code with example. (T.U. 2068)
(10) Explain the subtraction algorithm with signed 2's compliment. (T.U. 2067)
(11) What is an error detection code? Explain with example. (T.U. 2070)
(12) Differentiate between fixed point representation and floating point representation. (T.U. 2069)
(13) Write short notes on:
   - Alphanumeric Representation
   - Parity Generator (T.U. 2069)

A Gentle Advice:

Please go through your text books and reference books for detail study!!! Thank you all.

Notes Compiled By:

Bijay Mishra
biizay.blogspot.com
9813911076 or 9841695609

Unit 2 – Micro-operations

Micro-operations	**7 Hrs.**
Arithmetic Micro-operations	**3 Hrs.**
Add Micro-operation	
Subtract Micro-operation	
Binary Adder	
Binary Subtractor	
Binary Adder-Subtractor	
Binary Incrementer	
Arithmetic Circuit	
Logic Micro-operations	**1.5 Hrs.**
Logic Micro-operations	
Implementations and Applications	
Shift Micro-operations	**1.5 Hrs.**
Logical Shift	
Circular Shift	
Arithmetic Shift	
Combinational Circuit Shifter	
Arithmetic Logic Shift Unit	**1 Hr.**

Micro-operations

Micro-operations (also known as a micro-ops or µops) are the operations which are used to create assembly language instruction. The operations performed on the data stored in registers are called micro-operations. Some common example of micro-operation are:

1) Register Transfer Micro-operation: The main purpose of Register Transfer Micro operation is to transfer binary information from register to another register.

2) Arithmetic Micro-operation: The main purpose of Arithmetic Micro-operation is to perform arithmetic operation on numeric data.

3) Logical Operation: The main purpose of Logical Operation is to perform bit manipulation on numeric data.

4) Shift Micro-operation: The main purpose of Shift Micro-operation is to shift the temporary data which are present in register.

Register Transfer Micro-operation

Register transfer language is a symbolic language and a convenient tool for describing the internal organization of digital computers. It can also be used to facilitate the design process of digital systems.

Copying the contents of one register to another is a register transfer. Registers are designated by capital letters, sometimes followed by numbers (E.g. R1, IR, MAR, etc.).

A register transfer is indicated as: R2 ← R1

The statement "R2 ← R1" denotes a transfer of the content of the R1 into resister R2.

A register transfer such as "R3 ← R5" implies that the digital system has:

- the data lines from the source register (R5) to the destination register (R3)
- Parallel load in the destination register (R3)
- Control lines to perform the action

Control Function

Often actions need to only occur if a certain condition is true. In digital systems, this is often done via a control signal, called a control function.

Example: P: R2 ← R1 i.e. if (P = 1) then (R2 ← R1)
Which means "if P = 1, then load the contents of register R1 into register R2".

If two or more operations are to occur simultaneously, they are separated with commas.

Example: P: R3 ← R5, MAR ← IR

Here, if the control function P = 1, load the contents of R5 into R3, and at the same time (clock), load the contents of register IR into register MAR

Symbols	Description	Examples
Capital letters & numerals	Denotes a register	MAR, R2
Parentheses ()	Denotes a part of a register	R2(0-7), R2(L)
Arrow ←	Denotes transfer of information	R2 ← R1
Colon :	Denotes termination of control function	P:
Comma ,	Separates two micro-operations	A ← B, B ← A

Table: Basic symbols for register-transfer

Arithmetic Micro-operation

The basic arithmetic micro-operations are

- Addition
- Subtraction
- Increment
- Decrement

The additional arithmetic micro-operations are

- Add with carry
- Subtract with borrow
- Transfer/Load

Summary of Typical Arithmetic Micro-Operations:

R3 ← R1 + R2	Contents of R1 plus R2 transferred to R3
R3 ← R1 - R2	Contents of R1 minus R2 transferred to R3
R2 ← R2’	Complement the contents of R2 (1’s Complement)
R2 ← R2’+ 1	2's complement the contents of R2 (Negate)
R3 ← R1 + R2’+ 1	R1 plus 2's complement the contents of R2 (Subtraction)
R1 ← R1 + 1	Increment the content of R1 by 1
R1 ← R1 - 1	Decrement the content of R1 by 1

Binary Adder

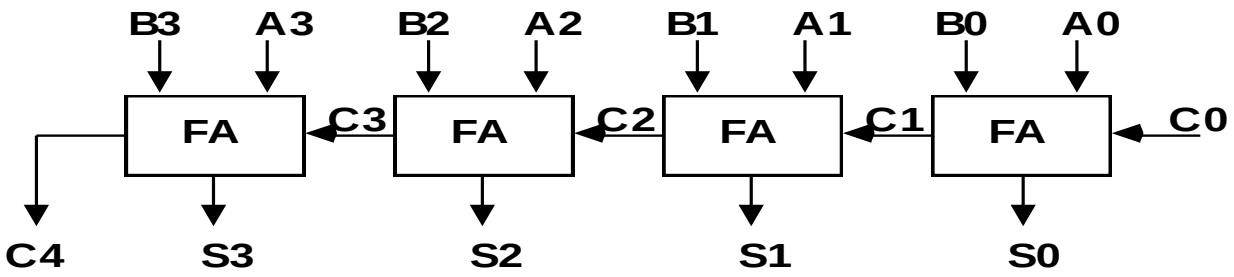


Figure: 4-bit binary adder

To implement the add micro-operation with hardware, we need the resisters that hold the data and the digital component that performs the arithmetic addition. The digital circuit that generates the arithmetic sum of two binary numbers of any lengths is called ***Binary Adder***.

The binary adder is constructed with the full-adder circuit connected in cascade, with the output carry from one full-adder connected to the input carry of the next full-adder.

An n-bit binary adder requires n full-adders. The output carry from each full-adder is connected to the input carry of the next-high-order-full-adder. Inputs A and B come from two registers R1 and R2.

The subtraction A-B can be carried out by the following steps

- Take the 1’s complement of B (invert each bit)
- Get the 2’s complement by adding 1
- Add the result to A

Binary Adder-Subtractor

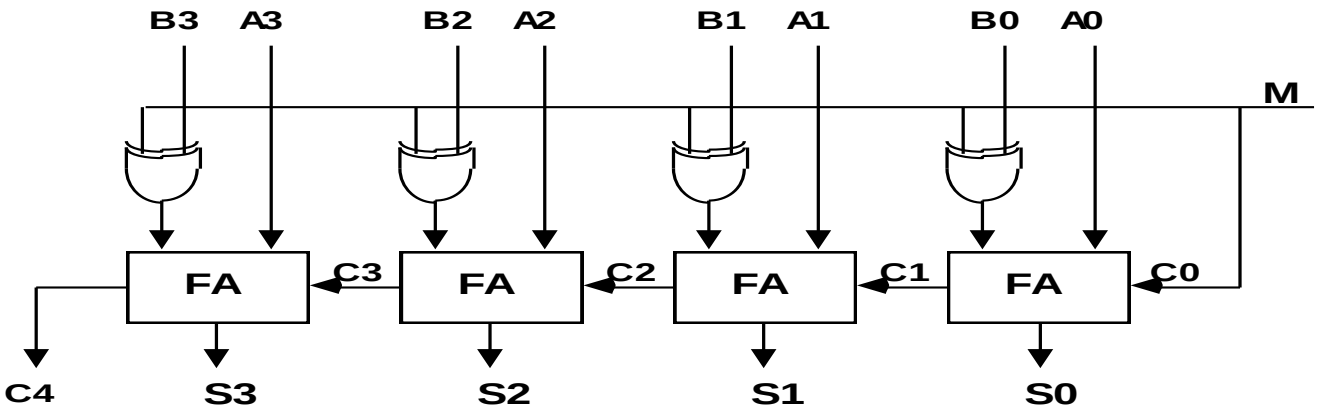


Figure: 4-bit adder-subtractor

The addition and subtraction operations can be combined into one common circuit by including an exclusive-OR gate with each full-adder.

The mode input M controls the operation the operation. When M=0, the circuit is an adder and when M=1 the circuit becomes a subtractor. Each exclusive-OR gate receives input M and one of the inputs of B.

1. When M=0: B $\oplus$ M = B $\oplus$ 0 = B, i.e. full-adders receive the values of B, input carry is B and circuit performs A+B.
2. When M=1: B $\oplus$ M = B $\oplus$ 1 = B' and C_0= 1, i.e. B inputs are all complemented and 1 is added through the input carry. The circuit performs A + (2's complement of B).

Binary Incrementer

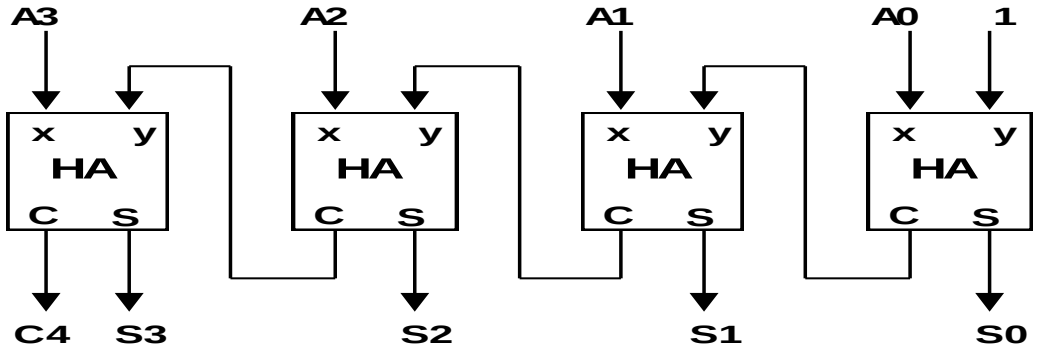


Figure: 4-bit binary Incrementer

The increment micro-operation adds one to a number in a register. For example, if a 4-bit register has a binary value 0110, it will go to 0111 after it is incremented. Increment micro-operation can be done with a combinational circuit (half-adders connected in cascade) independent of a particular register.

Arithmetic Circuit

The arithmetic micro-operations can be implemented in one composite arithmetic circuit. By controlling the data inputs to the adder (basic component of an arithmetic circuit), it is possible to obtain different types of arithmetic operations. In the circuit below contains:

- 4 full-adders
- 4 multiplexers (controlled by selection inputs S0 and S1)
- Two 4-bit inputs A and B and a 4-bit output D
- Input carry c_{in} goes to the carry input of the full-adder.

Output of the binary adder is calculated from the arithmetic sum: $D = A + Y + c_{in}$

By controlling the value of Y with the two selection inputs S1 & S0 and making c_{in}= 0 or 1, it is possible to generate the 8 arithmetic micro-operations listed in the table below:

Select			Input	Output	
S_1	S_0	C_{in}	Y	$D = A + Y + C_{in}$	Microoperation
0	0	0	B	$D = A + B$	Add
0	0	1	B	$D = A + B + 1$	Add with carry
0	1	0	$\overline{B}$	$D = A + \overline{B}$	Subtract with borrow
0	1	1	$\overline{B}$	$D = A + \overline{B} + 1$	Subtract
1	0	0	0	$D = A$	Transfer A
1	0	1	0	$D = A + 1$	Increment A
1	1	0	1	$D = A - 1$	Decrement A
1	1	1	1	$D = A$	Transfer A

Table: Arithmetic Circuit Function Table

When S_1S_0 = 00, the value of B is applied to the Y inputs of the adder. If C_{in} = 0, the output D = A + B. If C_{in} = 1, output D = A + B + 1. Both cases perform the add micro-operation with or without adding the input carry.

When S_1S_0 = 01, the complement of B is applied to the Y inputs of the adder. If C_{in} = 1, then D =A +B'+ 1. This produces A plus the 2's complement of B, which is equivalent to a subtract with borrow, that is, A – B – 1.

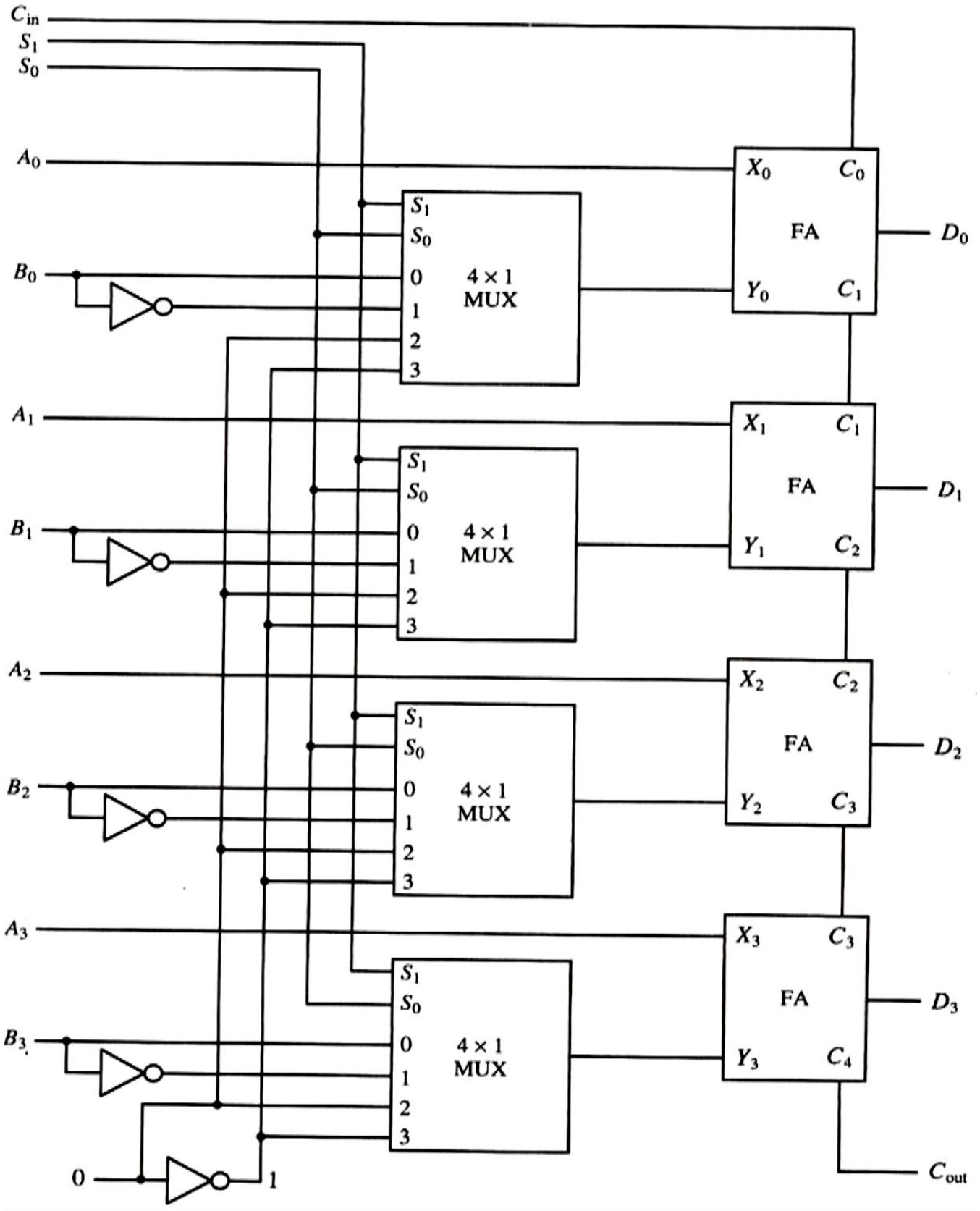


Figure: 4-bit Arithmetic Circuit

b i i z a y @ g m a i l . c o m

When S_1S_0 = 10, the input from B are neglected, and instead, all 0's are inserted into the Y inputs. The output becomes D = A + 0 + C_{in}. This gives D = A when C_{in} = 0 and D = A +1 when C_{in} = 1. In the first case we have a direct transfer from input A to output D. In the second case, the value of A is incremented by 1.

When S_1S_0 = 11, all 1's are inserted into the Y inputs of the adder to produce the decrement operation D = A –1 when C_{in}. This is because a number with all 1's is equal to the 2's complement of 1 (the 2's complement of binary 0001 is 1111). Adding a number A to the 2's complement of 1 produces F = A +2's complement of 1 = A – 1. When C_{in} = 1, then D = A – 1 + 1 = A, which causes a direct transfer from input A to output D. Note that the micro-operation D = A is generated twice, so there are only seven distinct micro-operations in the arithmetic circuit.

Logic Micro-operation

Logic operations specify binary operations for strings of bits stored in registers and treat each bit separately. Logic micro-operations are bit-wise operations, i.e., they work on the individual bits of data. They are useful for bit manipulations on binary data and for making logical decisions based on the bit value.

Example: The XOR of R1 and R2 is symbolized by P: R1 ← R1 ⊕ R2

Example: R1 = 1010 and R2 = 1100
1010 Content of R1
1100 Content of R2
0110 Content of R1 after P = 1

Symbols used for logical micro-operations:

1. OR: ∨
2. AND: ∧
3. XOR: ⊕

The + sign has two different meanings: logical OR and Summation

- When + is in a micro-operation, then summation
- When + is in a control function, then OR

Example: P + Q: R1 ← R2 + R3, R4 ← R5 ∨ R6

There are 16 different logic operations that can be performed with two binary variables

TABLE: Truth Tables for 16 Functions of Two Variables

x	y	F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}	F_{11}	F_{12}	F_{13}	F_{14}	F_{15}
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

b i i z a y @ g m a i l . c o m

Boolean function	Microoperation	Name
$F_0 = 0$	$F \leftarrow 0$	Clear
$F_1 = xy$	$F \leftarrow A \wedge B$	AND
$F_2 = xy'$	$F \leftarrow A \wedge \overline{B}$	
$F_3 = x$	$F \leftarrow A$	Transfer A
$F_4 = x'y$	$F \leftarrow \overline{A} \wedge B$	
$F_5 = y$	$F \leftarrow B$	Transfer B
$F_6 = x \oplus y$	$F \leftarrow A \oplus B$	Exclusive-OR
$F_7 = x + y$	$F \leftarrow A \vee B$	OR
$F_8 = (x + y)'$	$F \leftarrow \overline{A \vee B}$	NOR
$F_9 = (x \oplus y)'$	$F \leftarrow \overline{A \oplus B}$	Exclusive-NOR
$F_{10} = y'$	$F \leftarrow \overline{B}$	Complement B
$F_{11} = x + y'$	$F \leftarrow A \vee \overline{B}$	
$F_{12} = x'$	$F \leftarrow \overline{A}$	Complement A
$F_{13} = x' + y$	$F \leftarrow \overline{A} \vee B$	
$F_{14} = (xy)'$	$F \leftarrow \overline{A \wedge B}$	NAND
$F_{15} = 1$	$F \leftarrow$ all 1's	Set to all 1's

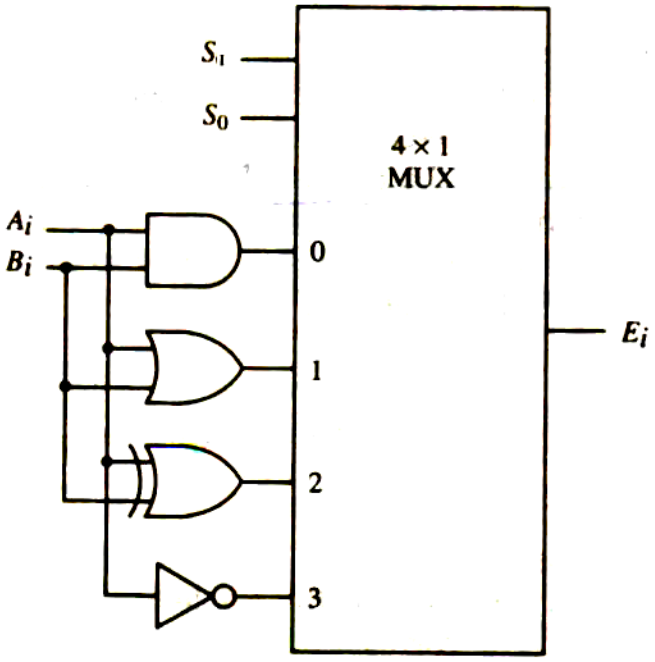
Table: Sixteen Logic Micro-operations

Hardware Implementations and Applications

The hardware implementation of logic micro-operations requires that logic gates be inserted for each bit or pair of bits in the registers to perform the required logic function. Although there are 16 logic micro-operations, most computers use only four --- AND, OR, XOR (exclusive-OR), and complement by which all others can be derived.

Figure below shows one stage of a circuit that generates the four basic logic micro-operations. It consists of four gates and a multiplexer. Each of the four logic operations is generated through a gate that performs the required logic. The outputs of the gates are applied to the data inputs of the multiplexer. The two selection inputs S_1 and S_0 choose one of the data inputs of the multiplexer and direct its value to the output.

The diagram shows one typical stage with subscript i. For a logic circuit with n bits, the diagram must be repeated n times for i = 0, 1, 2, …, N–1. The selection variables are applied to all stages. The function table lists the logic micro-operations obtained for each combination of the selection variables.



(a) Logic diagram

S_1	S_0	Output	Operation
0	0	$E = A \wedge B$	AND
0	1	$E = A \vee B$	OR
1	0	$E = A \oplus B$	XOR
1	1	$E = \bar{A}$	Complement

(b) Function table

Figure: One stage of Logic Circuit

Logic micro operations are very useful for manipulating individual bits or a portion of a word stored in a register. They can be used to change bit values, delete a group of bits, or insert new bit values into a register. The following examples show how the bits of one register (designated by A) are manipulated by logic micro-operations as a function of the bits of another register (designated by B). In a typical application, register A is a processor register and the bits of register B constitute a logic operand extracted from memory and placed in register B.

The **selective-set operation** sets to 1 the bits in A where there are corresponding 1's in B

1010	A before
1100	B (logic operand)
1110	A after

$A \leftarrow A \vee B$

The **selective-complement operation** complements bits in A where there are corresponding 1's in B

1010	A before
1100	B (logic operand)
0110	A after

$A \leftarrow A \oplus B$

The **selective-clear operation** clears to 0 the bits in A only where there are corresponding 1's in B

1010	A before
1100	B (logic operand)
0010	A after

$A \leftarrow A \wedge \bar{B}$

b i i z a y @ g m a i l . c o m

The **mask operation** is similar to the selective-clear operation, except that the bits of A are cleared only where there are corresponding 0's in B

1010	A before
1100	B (logic operand)
1000	A after

$A \leftarrow A \wedge B$

The **insert operation** inserts a new value into a group of bits. This is done by first masking the bits to be replaced and then Oring them with the bits to be inserted.

0110	1010	A before
0000	1111	B (mask)
0000	1010	A after masking

0000	1010	A before
1001	0000	B (insert)
1001	1010	A after insertion

The **clear operation** compares the bits in A and B and produces an all 0's result if the two number are equal

1010	A
1010	B
0000	$A \leftarrow A \oplus B$

Shift Micro-operation

Shift micro-operations are used for serial transfer of data. They are also used in conjunction with arithmetic, logic, and other data-processing operations.

There are three types of shifts

1. Logical shift
2. Circular shift
3. Arithmetic shift

Symbolic designation	Description
$R \leftarrow \text{shl } R$	Shift-left register R
$R \leftarrow \text{shr } R$	Shift-right register R
$R \leftarrow \text{cil } R$	Circular shift-left register R
$R \leftarrow \text{cir } R$	Circular shift-right register R
$R \leftarrow \text{ashl } R$	Arithmetic shift-left R
$R \leftarrow \text{ashr } R$	Arithmetic shift-right R

Figure: Shift Micro-operations

b i i z a y @ g m a i l . c o m

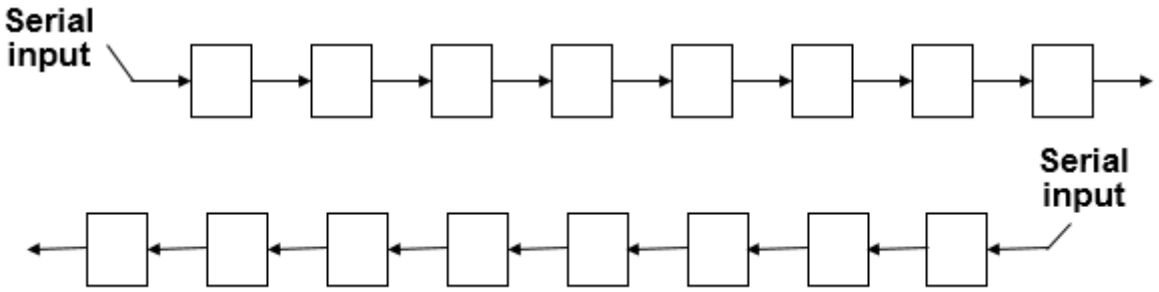
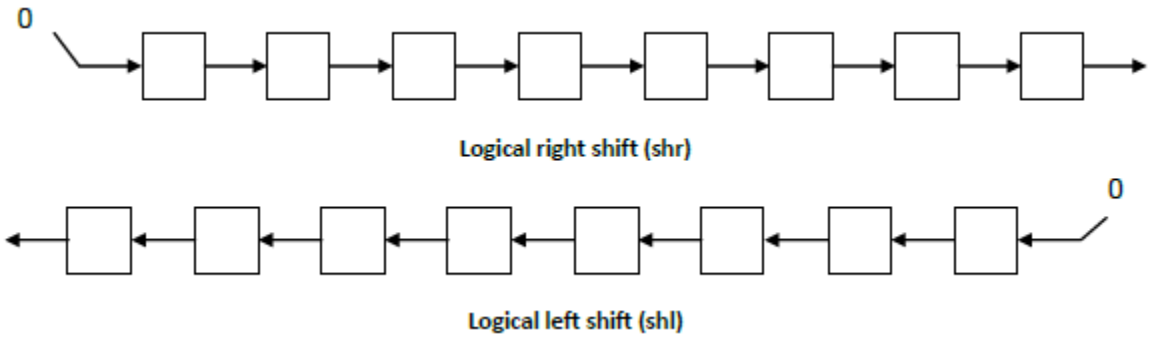


Figure: Right Shift Operation and Left Shift Operation

Logical Shift

A logical shift is one that transfers 0 through the serial input. The symbols ***shl*** and ***shr*** are for logical shift-left and shift-right by one position respectively.

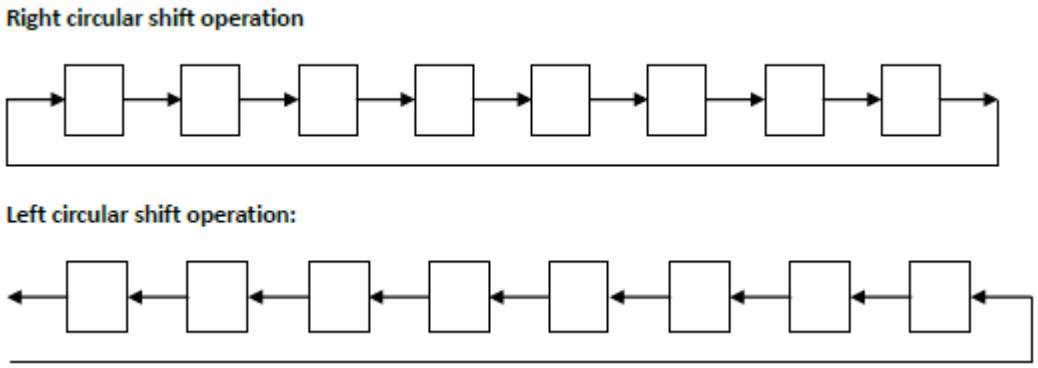
Example:
R1 ← shl R1
R2 ← shr R2



Circular Shift

The circular shift (aka rotate) circulates the bits of the register around the two ends without loss of information. The symbols ***cil*** and ***cir*** are for circular shift left and right respectively.

Examples:
R2 ← cir R2
R3 ← cil R3

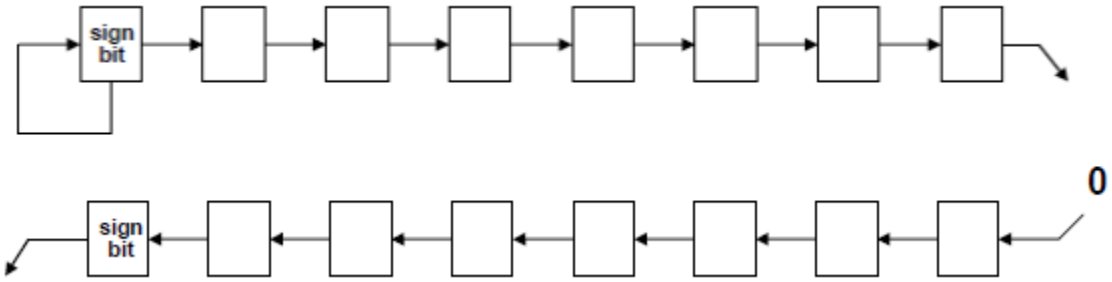


b i i z a y @ g m a i l . c o m

Arithmetic Shift

The arithmetic shift shifts a signed binary number to the left or right. To the left is multiplying by 2, to the right is dividing by 2 (i.e. an arithmetic left shift multiplies a signed number by 2 and an arithmetic right shift divides a signed number by 2).

Arithmetic shifts must leave the sign bit unchanged because the sign of the number remains the same when it is multiplied or divided by 2. The left most bit in a resister holds a sign bit and remaining hold the number. Negative numbers are in 2's complement form. The figure below shows the arithmetic shift right and left respectively.



Examples:
R2 ← ashr R2
R3 ← ashl R3

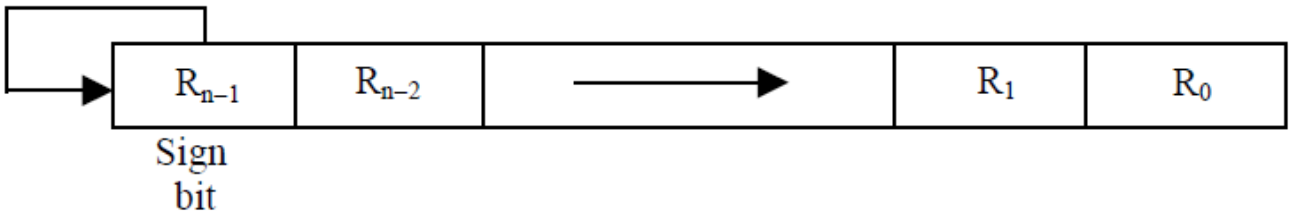


Figure: Arithmetic Shift Right

Arithmetic Shift-right

Arithmetic shift-right leaves the sign bit unchanged and shifts the number (including a sign bit) to the right. Thus R_{n-1} remains same; R_{n-2} receives input from R_{n-1} and so on.

Arithmetic Shift-left

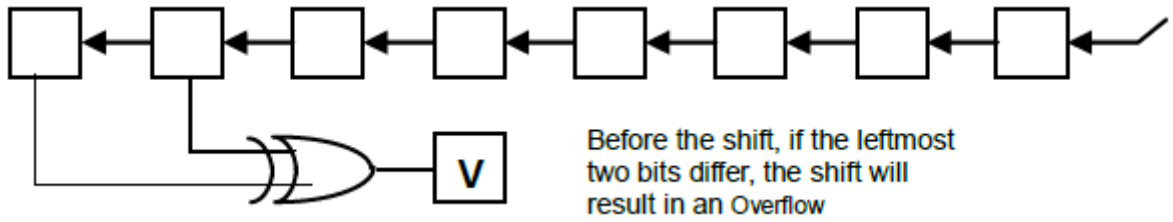
Arithmetic shift-left inserts a 0 into R_0 and shifts all other bits to left. Initial bit of R_{n-1} is lost and replaced by the bit from R_{n-2}.

Overflow case during arithmetic shift-left

A sign reversal occurs if the bit in R_{n-1} changes in value after the shift. This happens if the multiplication causes an overflow. Thus, left arithmetic shift operation must be checked for the overflow.

An overflow occurs after an arithmetic shift-left if before shift $R_{n-1} \neq R_{n-2}$. An overflow flip-flop V_s can be used to detect the overflow: $V_s = R_{n-1} \oplus R_{n-2}$. If V = 0, there is no overflow but if V = 1, overflow is detected.

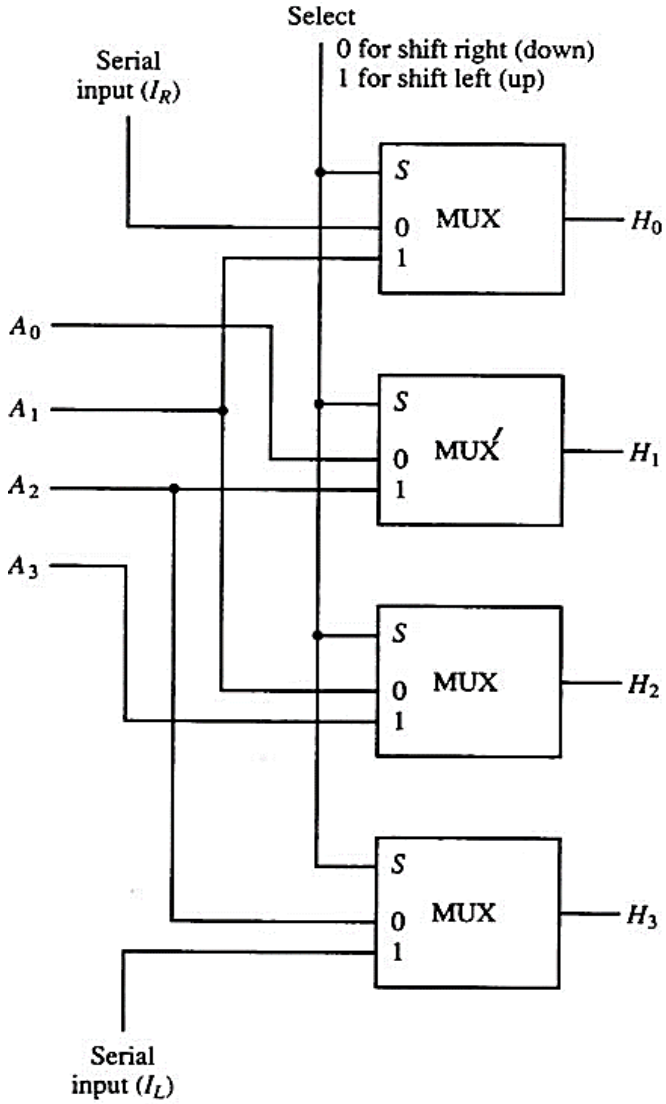
b i i z a y @ g m a i l . c o m



Hardware implementation of shift micro-operations

A bi-directional shift unit with parallel load could be used to implement this. Two clock pulses are necessary with this configuration: one to load the value and another to shift. In a processor unit with many registers it is more efficient to implement the shift operation with a combinational circuit. The content of a register to be shifted is first placed onto a common bus and the output is connected to the combinational shifter, the shifted number is then loaded back into the register.

This can be constructed with multiplexers.



Function table

Select	Output			
S	H_0	H_1	H_2	H_3
0	I_R	A_0	A_1	A_2
1	A_1	A_2	A_3	I_L

Figure 4-bit combinational circuit shifter.

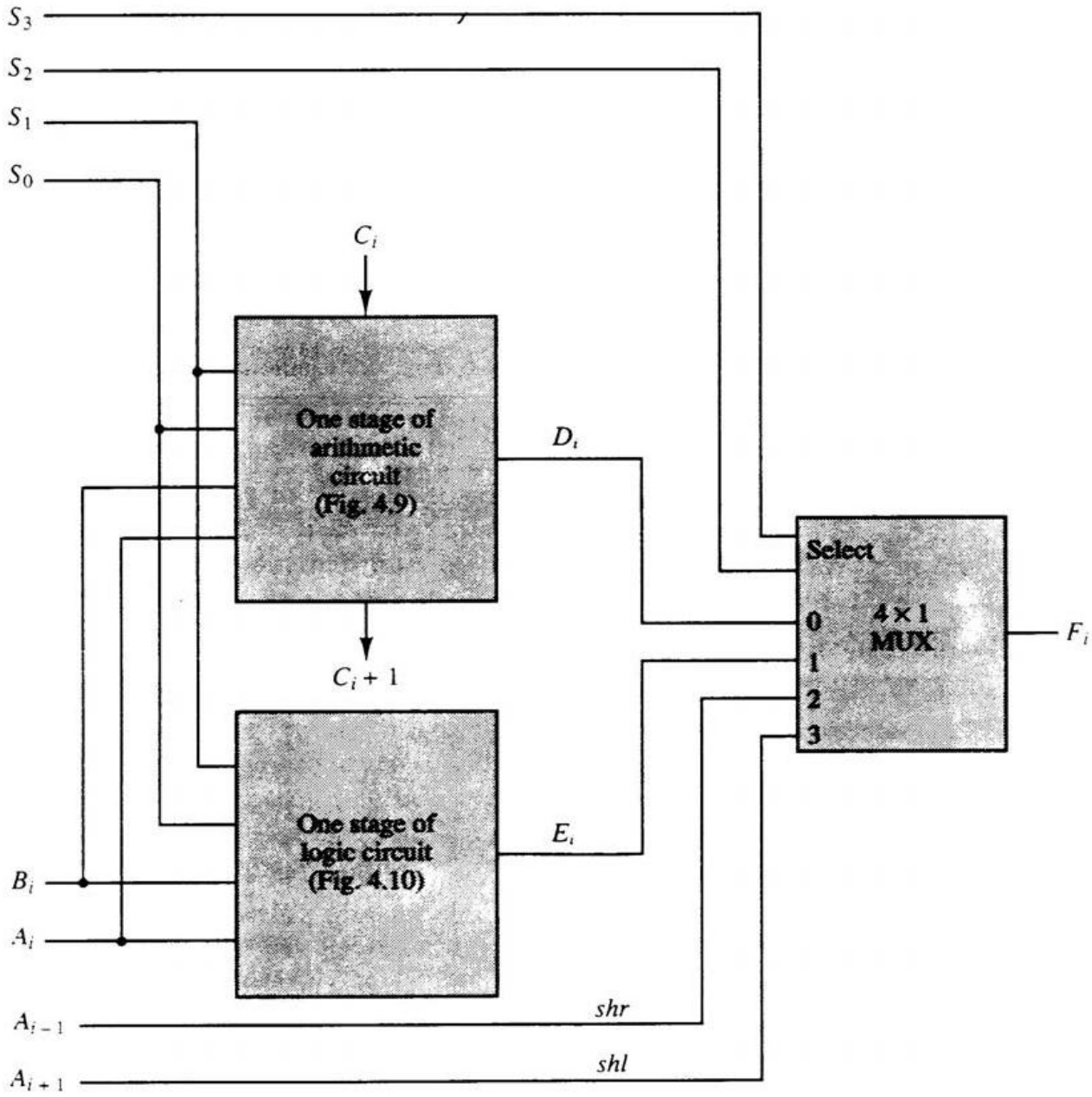
b i i z a y @ g m a i l . c o m

It has 4 data inputs A0 through A3 and 4 data outputs H0 through H3. There are two serial inputs, one for shift-left (IL) and other for shift-right (IR). When S = 0: input data are shifted right (down in fig). When S = 1: input data are shifted left (up in fig).

Arithmetic Logic Shift Unit

The arithmetic logic unit (ALU) is a common operational unit connected to a number of storage registers. To perform a micro-operation, the contents of specified registers are placed in the inputs of the ALU. The ALU performs an operation and the result is then transferred to a destination register. The ALU is a combinational circuit so that the entire register transfer operation from the source registers through the ALU and into the destination register can be performed during one clock pulse period.

Figure One stage of arithmetic logic shift unit.



b i i z a y @ g m a i l . c o m

The diagram above shows just one typical stage. The circuit must be repeated n times for an n-bit ALU.

i. Input A_i and B_i are applied to both the arithmetic and logic units. A particular micro-operation is selected with inputs S_1 and S_0.
ii. A 4x1 MUX at the output chooses between an arithmetic output in D_i and a logic output in E_i.
iii. The data inputs to the multiplexer are selected with inputs S_3 and S_2.
iv. The other two data inputs to the MUX receive inputs A_{i-1} for the shift right operation and A_{i+1} for the shift left operation.
v. C_{in} is the selection variable for the arithmetic operation.
vi. The circuit provides eight arithmetic operation, four logic operations and two shift operations. Each operation is selected with the five variables S_3, S_2, S_1, S_0 and C_{in}. The input carry C_{in} is used for arithmetic operations only.
vii. The table lists the 14 operations of the ALU. The first eight are arithmetic operation and are selected with S_3 S_2 = 00. The next four are logic operation and are selected with S_3 S_2 = 01 and last two operation are shift operation and are selected with S_3 S_2 = 10 and 11.

TABLE Function Table for Arithmetic Logic Shift Unit

Operation select						
S_3	S_2	S_1	S_0	C_{in}	Operation	Function
0	0	0	0	0	$F = A$	Transfer A
0	0	0	0	1	$F = A + 1$	Increment A
0	0	0	1	0	$F = A + B$	Addition
0	0	0	1	1	$F = A + B + 1$	Add with carry
0	0	1	0	0	$F = A + \overline{B}$	Subtract with borrow
0	0	1	0	1	$F = A + \overline{B} + 1$	Subtraction
0	0	1	1	0	$F = A - 1$	Decrement A
0	0	1	1	1	$F = A$	Transfer A
0	1	0	0	×	$F = A \wedge B$	AND
0	1	0	1	×	$F = A \vee B$	OR
0	1	1	0	×	$F = A \oplus B$	XOR
0	1	1	1	×	$F = \overline{A}$	Complement A
1	0	×	×	×	$F = \text{shr } A$	Shift right A into F
1	1	×	×	×	$F = \text{shl } A$	Shift left A into F

Book References:

(1) Andrew S. Tanenbaum, "Structured Computer Organization", Fourth Edition.
(2) M. Morris Mano, "Computer System Architecture", Pearson, 3rd Ed, 2004.
(3) John P. Hayes, "Computer Architecture & Organization".

Web References:

(4) http://en.wikipedia.org
(5) http://www.cs.uwm.edu/
(6) http://www.transtutors.com/

Assignments:

(1) What do you mean by shift micro-operations? Explain. (T.U. 2066)
(2) What do you mean by logic micro-operations? (T.U. 2067)
(3) Differentiate between logic micro operations and shift micro operations. (T.U. 2068)
(4) Explain the arithmetic logic shift unit. (T.U. 2069)
(5) Design the binary adder-subtractor with example. (T.U. 2070)
(6) Design a 2-bit adder and logic circuit capable of performing AND, ADD, complement and shift left operation.
(7) Give the hardware realization of 4-bit arithmetic circuit capable of doing addition, subtraction, increment, decrement etc. Give the function table and explain its operation.
(8) The 8-bit registers A, B, C & D are loaded with the value $(F2)_H$, $(FF)_H$, $(B9)_H$ and $(EA)_H$ respectively. Determine the register content after the execution of the following sequence of micro-operations sequentially.
    (i) $A \leftarrow A + B$, $C \leftarrow C + Shl(D)$
    (ii) $C \leftarrow C \wedge D$, $B \leftarrow B + 1$.
    (iii) $A \leftarrow A - C$.
    (iv) $A \leftarrow Shr(B) \oplus Cir(D)$
(9) A half-adder is a combinational logic circuit that has two inputs, X and Y and two outputs, S and C that are the sum and carry-out, respectively, resulting from binary addition of X and Y.
    (i) Design a half-adder as a two-level AND-OR circuit.
    (ii) Show how to implement a full adder, by using half adders.
(10) Show the implementation of a full adder using half adders
(11) Differentiate between arithmetic shift and logical shift.
(12) Register A is having S-bit number 11011001. Determine the operand and logic micro-operation to be performed in order to change the value in A to:
    (i) 01101101
    (ii) 11111101
    (iii) Starting from an initial value R = 11011101, determine the sequence of binary values in R after a logical shift left, followed by circular shift right, followed by a logical shift right and a circular shift left.

A Gentle Advice:

Please go through your text books and reference books for detail study!!! Thank you all.

Notes Compiled By:

Bijay Mishra
biizay.blogspot.com
9813911076 or 9841695609

Unit 3 - Fundamental of Computer Organization and Design

Fundamental of Computer Organization and Design	**7 Hrs.**
Computer Register	**1 Hr.**
Registers for the Basic Computer and Common Bus	
Computer Instructions	**1.5 Hrs.**
Instruction Format Basic Instructions Instruction Set Completeness Types of Instruction (Memory Reference, Register Reference, I/O)	
Instruction Cycle	**1 Hr.**
Phases of Instruction Cycle Fetch and Decode Flowchart for Instruction Cycle	
Input and Output and Interrupt	**1.5 Hrs.**
I/O Configuration Input-Output Instruction Types of Interrupts Program Interrupt Interrupt Cycle	
Basic Computer Design and Accumulator Logic	**2 Hrs.**
Basic Hardware Components Flowchart for Computer Operation Control Logic Gates Control of Register and Memory Control of Common Bus Control of Flip-flop Design of Accumulator Logic (Control of AC Register, Adder and Logic Circuit)	

Computer Registers

Register are used to quickly accept, store, and transfer data and instructions that are being used immediately by the CPU, there are various types of registers those are used for various purpose. Among of the some mostly used registers named as Accumulator, Data Register, Address Register, Program Counter, Memory Data Register, Index register, Memory Buffer Register.

Types of Registers:

A processor has many registers to hold instructions, addresses, data, etc. Some of them are:

Instruction Register (IR)
Instruction Register (16 bits) holds the instruction code of the instruction currently executing. Outputs of this register are hardwired to specific logic in the control unit, which interprets the bits to generate control signals.

Address Register (AR)
Address Register (12 bits) is used to interface with the memory unit. All memory-references are initiated by loading the memory address into AR.

Temporary Register (TR)
Temporary Register (16 bits) is an extra register for storing data or addresses.

Input and Output Registers
The Basic Computer has one input device and one output device. The Input Register (INPR) holds an 8 bit character gotten from an input device. The Output Register (OUTR) holds an 8 bit character to be send to an output device.

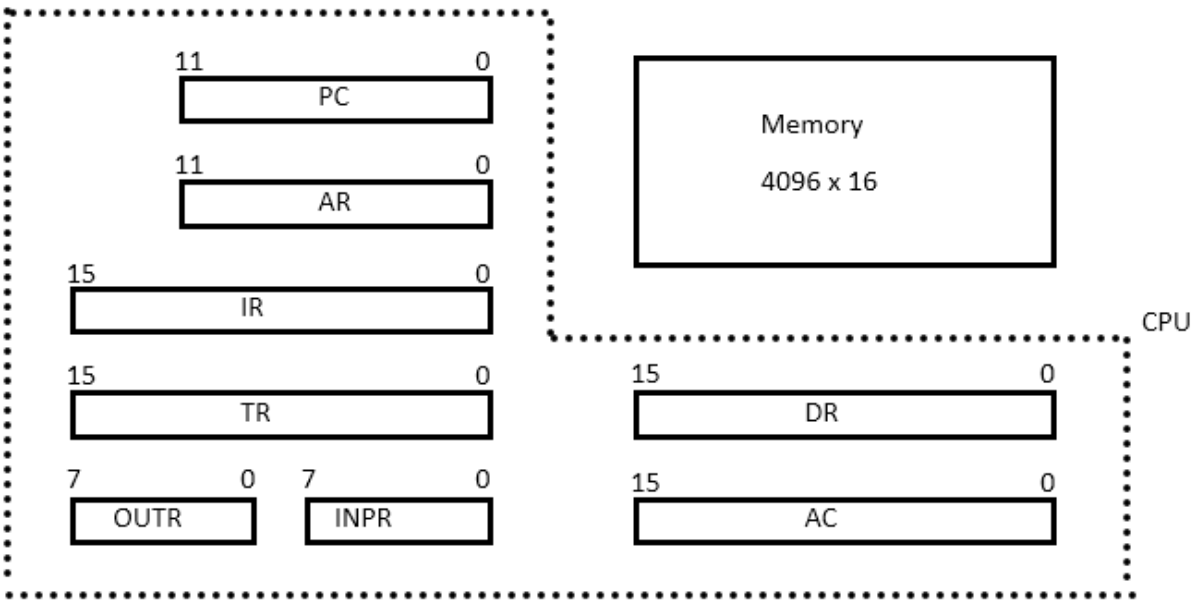


Figure: Registers in the Basic Computer

Program Counter (PC)
The program counter (PC) is commonly called the instruction pointer (IP) in Intel x86 microprocessors, and sometimes called the Instruction Address Register (IAR), or just part of the instruction sequencer in some computers, is a processor register.

Program Counter (12 bits) holds memory address of current/next instruction to be executed. Updated as part of the instruction cycle. Usually incremented, but may be parallel loaded by jump/branch instructions. It keeps track of the next memory address of the instruction that is to be executed once the execution of the current instruction is completed. In other words, it holds the address of the memory location of the next instruction when the current instruction is executed by the microprocessor.

Accumulator Register (AC)
This Register is used for storing the Results those are produced by the System. When the CPU will generate Some Results after the Processing then all the Results will be Stored into the AC Register.

Accumulator (16 bits) is used for all mathematical, logic, and shift operations except incrementing and clearing other registers (most have built-in increment and clear capability). It is the destination for all ALU operations, and a source for all dyadic (two-operand) operations.

Data Register (DR)
A register used in microcomputers to temporarily store data being transmitted to or from a peripheral device. Data Register (16 bits) is used to contain a second operand for dyadic operations such as Add, Sub, AND, OR.

DR	16	Data Register	Holds memory operand
AR	12	Address Register	Holds address for memory
AC	16	Accumulator	Processor register
IR	16	Instruction Register	Holds instruction code
PC	12	Program Counter	Holds address of instruction
TR	16	Temporary Register	Holds temporary data
INPR	8	Input Register	Holds input character
OUTR	8	Output Register	Holds output character

Table: List of Basic Computer Registers

Internal BUS Structure
The registers in the basic computer are connected using a bus. Most registers have load, increment, and clear capabilities built-in. This eliminates the need to use the ALU or the BUS for increment and clear, and hence we can perform these operations on any register in parallel with other micro-operations. AR outputs are the memory address bus. They are directly connected to the address input of the memory unit. AC and DR outputs hardwired into ALU. Hence, operands for dyadic operations such as add, sub, and, or must be in AC and DR. Inputs of INPR are hardwired from the input device. We cannot transfer anything into INPR from the bus. Outputs of INPR are hardwired to ALU. We can only transfer INPR to AC. Outputs of OUTR hardwired to the output device. We cannot transfer from OUTR to the bus. Memory data inputs and outputs are connected directly to the internal bus.

S_2	S_1	S_0	Register
0	0	0	x
0	0	1	AR
0	1	0	PC
0	1	1	DR
1	0	0	AC
1	0	1	IR
1	1	0	TR
1	1	1	Memory

b i i z a y @ g m a i l . c o m

Three control lines, S_2, S_1, and S_0 control which register the bus selects as its input. Either one of the registers will have its load signal activated, or the memory will have its write signal activated. The 12-bit registers, AR and PC, have 0's loaded onto the bus in the high order 4 bit positions. When the 8-bit register OUTR is loaded from the bus, the data comes from the low order 8 bits on the bus.

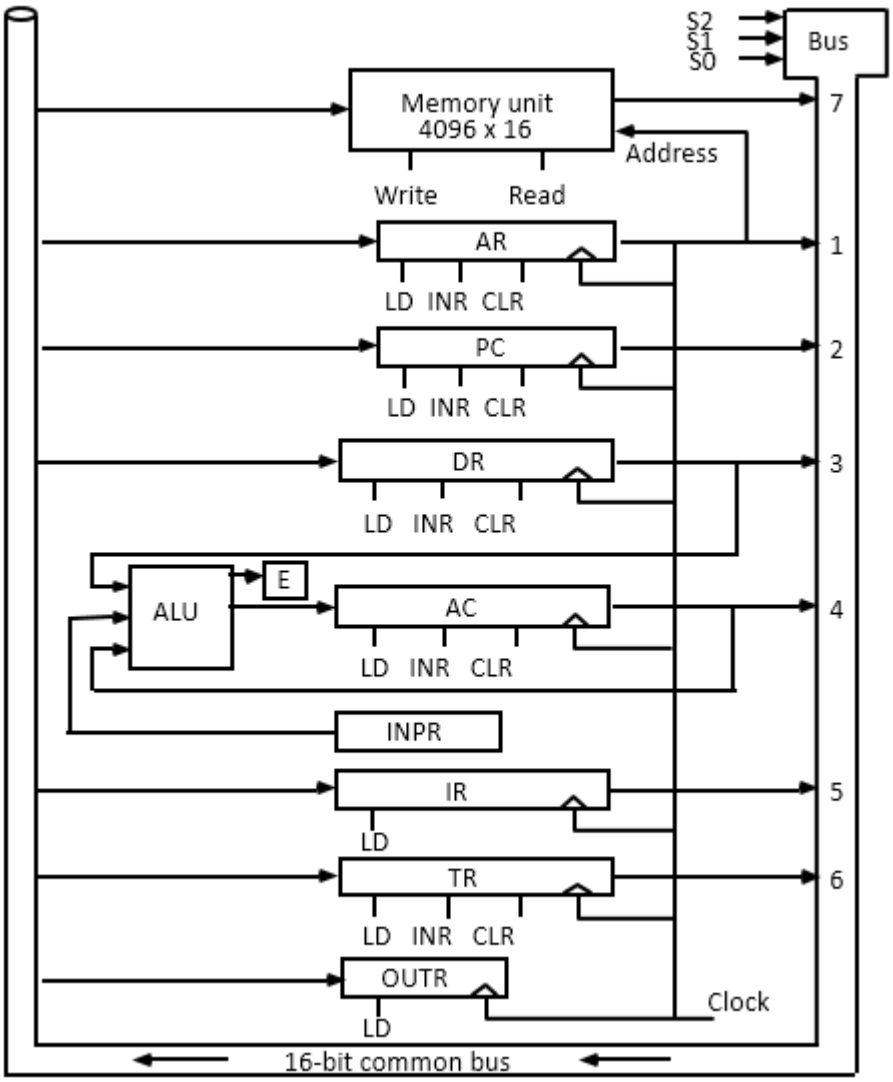


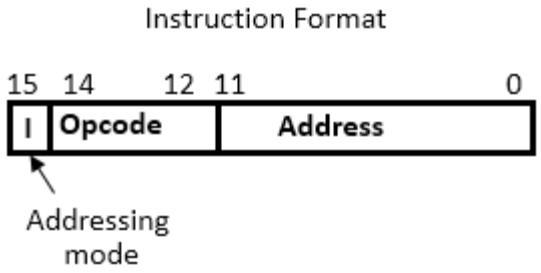
Figure: Common BUS System

Instruction Codes

Instructions are encoded as binary instruction codes. A computer instruction is often divided into two parts:

- An opcode (Operation Code) that specifies the operation for that instruction
- An address that specifies the registers and/or locations in memory to use for that operation

In the basic computer, since the memory contains 4096 (= 2^{12}) words, we needs 12 bit to specify which memory address this instruction will use. In the basic computer, bit 15 of the instruction specifies the addressing mode (0: direct addressing, 1: indirect addressing). Since the memory words, and hence the instructions, are 16 bits long, that leaves 3 bits for the instruction's opcode.



Some Common Addressing Modes

Direct: Instruction code contains address of operand; 1 memory-references after fetching instruction

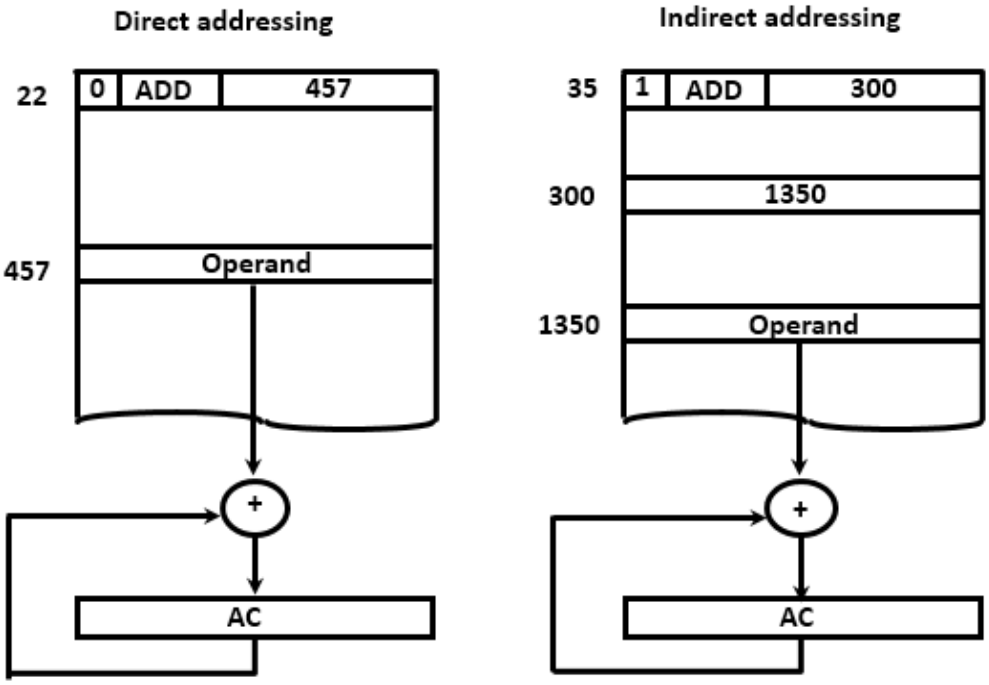
Immediate: Instruction code contains operand; No memory-reference after fetching instruction

Indirect: Instruction code contains address of address of operand; 2 memory-references after fetching instruction.

Effective Address (EA)

The address that can be directly used without modification to access an operand for a computation-type instruction, or as the target address for a branch-type instruction. It is the actual address of the data in memory.

Mode	Effective Address
Immediate	Address of the instruction itself
Direct	Address contained in the instruction code
Indirect	Address at the address in the instruction code

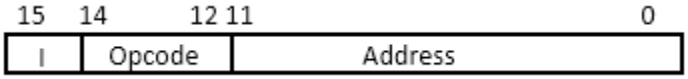


b i i z a y @ g m a i l . c o m

Computer Instructions

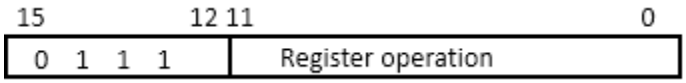
All Basic Computer instruction codes are 16 bits wide. There are 3 instruction code formats:

Memory-Reference Instructions take a single memory address as an operand, and have the format:



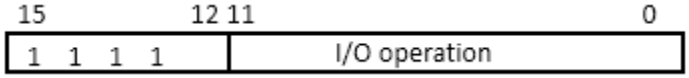
(OP-code = 000 - 110)
If I = 0, the instruction uses direct addressing.
If I = 1, addressing in indirect addressing.

Register-Reference Instructions operate solely on the AC register, and have the following format:



(OP-code = 111, I = 0)

Input/Output Instructions have the following format:



(OP-code =111, I = 1)

Instruction Set Completeness

A computer should have a set of instructions so that the user can construct machine language programs to evaluate any function that is known to be computable.

Instruction Types

- *Functional Instructions*
  - Arithmetic, logic, and shift instructions
  - ADD, CMA, INC, CIR, CIL, AND, CLA

- *Transfer Instructions*
  - Data transfers between the main memory and the processor registers
  - LDA, STA

- *Control Instructions*
  - Program sequencing and control
  - BUN, BSA, ISZ

- *Input/output Instructions*
  - Input and output
  - INP, OUT

Symbol	Hex Code I = 0	Hex Code I = 1	Description
AND	0xxx	8xxx	AND memory word to AC
ADD	1xxx	9xxx	Add memory word to AC
LDA	2xxx	Axxx	Load AC from memory
STA	3xxx	Bxxx	Store content of AC into memory
BUN	4xxx	Cxxx	Branch unconditionally
BSA	5xxx	Dxxx	Branch and save return address
ISZ	6xxx	Exxx	Increment and skip if zero
CLA	7800		Clear AC
CLE	7400		Clear E
CMA	7200		Complement AC
CME	7100		Complement E
CIR	7080		Circulate right AC and E
CIL	7040		Circulate left AC and E
INC	7020		Increment AC
SPA	7010		Skip next instr. if AC is positive
SNA	7008		Skip next instr. if AC is negative
SZA	7004		Skip next instr. if AC is zero
SZE	7002		Skip next instr. if E is zero
HLT	7001		Halt computer
INP	F800		Input character to AC
OUT	F400		Output character from AC
SKI	F200		Skip on input flag
SKO	F100		Skip on output flag
ION	F080		Interrupt on
IOF	F040		Interrupt off

Table: Basic Computer Instructions

Instruction Cycle

The CPU performs a sequence of micro-operations for each instruction. The sequence for each instruction of the Basic Computer can be refined into 4 abstract phases:

1. Fetch an instruction from memory
2. Decode the instruction
3. Read the effective address from memory if the instruction has an indirect address
4. Execute the instruction

After an instruction is executed, the cycle starts again at step 1, for the next instruction.

Instruction Fetch and Decode

Program execution begins with: PC ← address of first instruction, SC ← 0

After this, the SC is incremented at each clock cycle until an instruction is completed, and then it is cleared to begin the next instruction. This process repeats until a HLT instruction is executed, or until the power is shut off.

The instruction fetch and decode phases are the same for all instructions, so the control functions and micro-operations will be independent of the instruction code.

Everything that happens in this phase is driven entirely by timing variables T_0, T_1 and T_2. Hence, all control inputs in the CPU during fetch and decode are functions of these three variables alone.

T0: AR ← PC
T1: IR ← M [AR], PC ← PC + 1
T2: $D_0, \ldots, D_7$ ← Decode IR(12-14), AR ← IR(0-11), I ← IR(15)

For every timing cycle, we assume SC ← SC + 1 unless it is stated that SC ← 0.

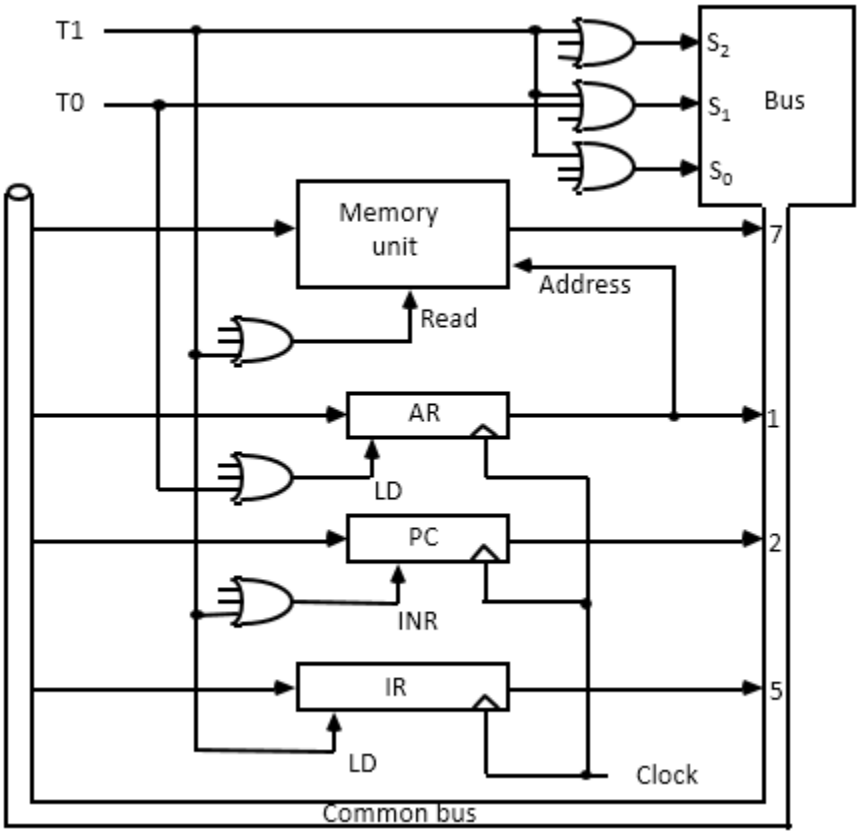


Figure: Register transfers for the fetch phase

The operation $D_{0\text{-}7}$ ← Decode IR(12-14) is not a register transfer like most of our micro-operations, but is actually an inevitable consequence of loading a value into the IR register. Since the IR outputs 12-14 are directly connected to a decoder, the outputs of that decoder will change as soon as the new values of IR(12-14) propagate through the decoder.

Note that incrementing the PC at time T_1 assumes that the next instruction is at the next address. This may not be the case if the current instruction is a branch instruction. However, performing the increment here will save time if the next instruction immediately follows, and will do no harm if it doesn't. The incremented PC value is simply overwritten by branch instructions.

Likewise, loading AR with the address field from IR at T_2 is only useful if the instruction is a memory-reference instruction. We won't know this until T_3, but there is no reason to wait since there is no harm in loading AR immediately.

Determining the Instruction Type

By time T_2, the opcode has been decoded by the decoder attached to IR(12-14), and the control signals D_{0-7} are available. At pulse T_2, IR(15) is loaded into the I flip-flop. Hence, all of these signals are available for use at pulse T_3.

D_7 indicates that the opcode field is 111, and this is either a register or I/O instruction. (i.e. it is not a memory-reference instruction.) The I bit allows us to distinguish between register and I/O instructions.

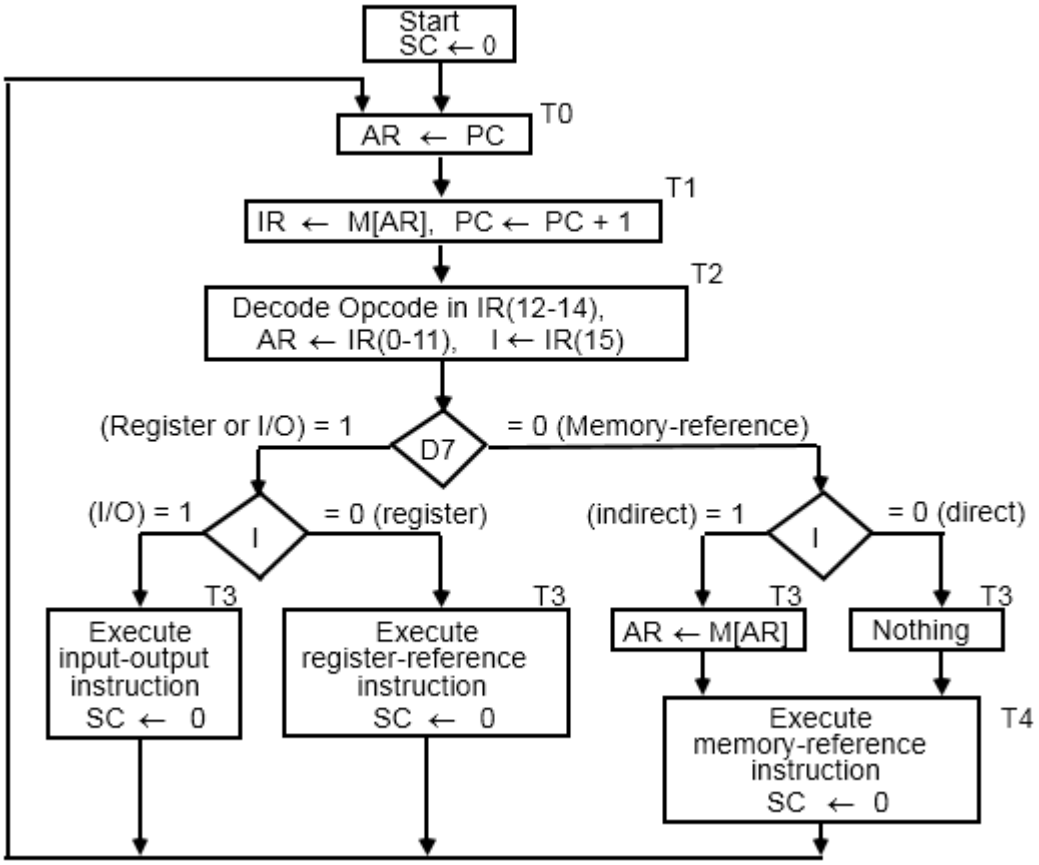


Figure: Flowchart for instruction cycle

D_7' indicates a memory-reference instruction. In this case, the I bit determines the addressing mode. What happens at time T_3 therefore depends on the two variables D_7 and I.

- Register-reference: $D_7I'T_3$: Execute register-reference instruction.
- I/O Reference: D_7IT_3: Execute I/O instruction.
- Memory-reference (Indirect addressing): $D_7'IT_3$: AR ← M[AR]

- Memory-reference (Direct addressing): $D_7'I'T_3$: Nothing. Effective address is already in AR. This wastes a clock cycle when direct addressing is used, but it simplifies the memory-reference execute phase by ensuring that the CPU is in a known state at time T_4.

Register-reference Execute Phase

Register Reference Instructions are identified when

- $D_7 = 1$, $I = 0$
- Register reference instruction is specified in b_0 - b_{11} of IR
- Execution starts with timing signal T_3

Hence:

- $r = D_7I'T_3$ => Register Reference Instruction
- $B_i = IR(i)$, i = 0,1,2,...,11

	r:	SC ← 0
CLA	rB_{11}:	AC ← 0
CLE	rB_{10}:	E ← 0
CMA	rB_9:	AC ← AC’
CME	rB_8:	E ← E’
CIR	rB_7:	AC ← shr AC, AC(15) ← E, E ← AC(0)
CIL	rB_6:	AC ← shl AC, AC(0) ← E, E ← AC(15)
INC	rB_5:	AC ← AC + 1
SPA	rB_4:	if (AC(15) = 0) then (PC ← PC+1)
SNA	rB_3:	if (AC(15) = 1) then (PC ← PC+1)
SZA	rB_2:	if (AC = 0) then (PC ← PC+1)
SZE	rB_1:	if (E = 0) then (PC ← PC+1)
HLT	rB_0:	S ← 0 (S is a start-stop flip-flop)

Memory-Reference Instructions

Each memory ref instruction is indicated by a unique D_i signal. For the memory-reference execute phase, all control inputs in the CPU are functions of timing signals T_4 or later, I, and one of the variables D_0 through D_6.

Symbol	Operation Decoder	Symbolic Description
AND	D_0	AC ← AC ∧ M[AR]
ADD	D_1	AC ← AC + M[AR], E ← C_{out}
LDA	D_2	AC ← M[AR]
STA	D_3	M[AR] ← AC
BUN	D_4	PC ← AR
BSA	D_5	M[AR] ← PC, PC ← AR + 1
ISZ	D_6	M[AR] ← M[AR] + 1, if M[AR] + 1 = 0 then PC ← PC+1

The effective address of the instruction is in AR and was placed there during timing signal T_2 when I = 0, or during timing signal T_3 when I = 1. Memory cycle is assumed to be short enough to complete in a CPU cycle. The execute phase for memory-reference instructions begins at time T_4. Several memory-reference instructions operate on AC and an operand from memory.

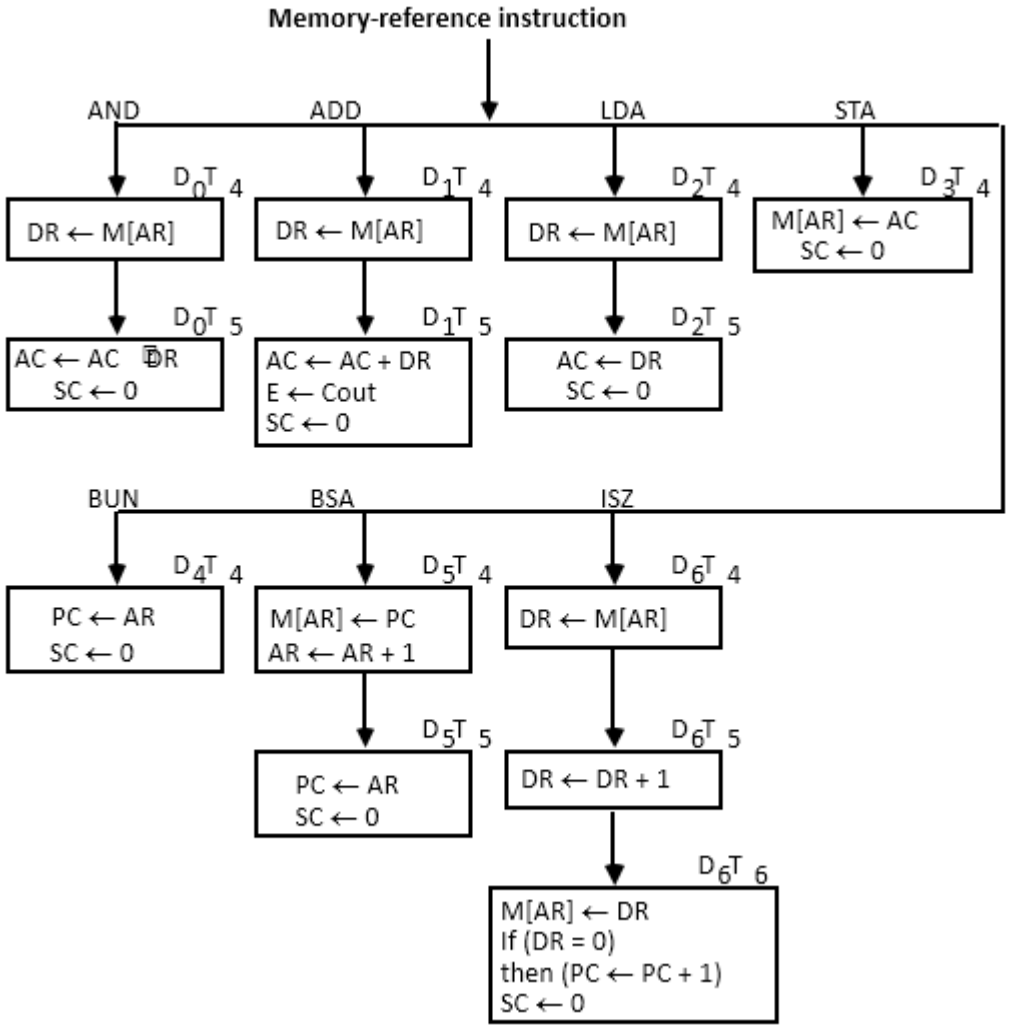


Figure: Flowchart for memory-reference instruction

AND to AC

D_0T_4: DR ← M[AR]

D_0T_5: AC ← AC ∧ DR, SC ← 0

ADD to AC

D_1T_4: DR ← M[AR]

D_1T_5: AC ← AC + DR, E ← C_{out}, SC ← 0

b i i z a y @ g m a i l . c o m

LDA: Load to AC

D_2T_4: DR ← M[AR]
D_2T_5: AC ← DR, SC ← 0

STA: Store AC

D_3T_4: M[AR] ← AC, SC ← 0

BUN: Branch Unconditionally

D_4T_4: PC ← AR, SC ← 0

BSA: Branch and Save Return Address

M[AR] ← PC, PC ← AR + 1

BSA:

D_5T_4: M[AR] ← PC, AR ← AR + 1
D_5T_5: PC ← AR, SC ← 0

ISZ: Increment and Skip-if-Zero

D_6T_4: DR ← M[AR]
D_6T_5: DR ← DR + 1
D_6T_6: M[AR] ← DR, if (DR = 0) then (PC ← PC + 1), SC ← 0

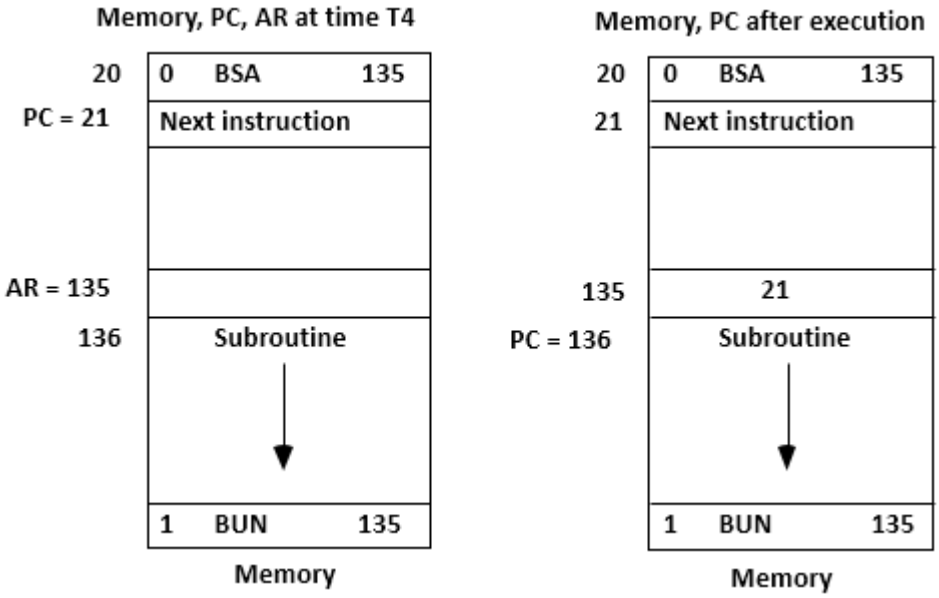
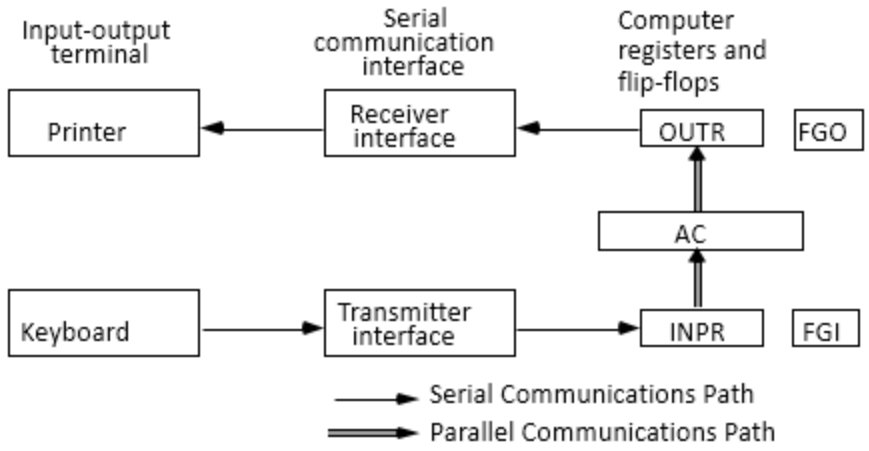


Figure: Example of BSA instruction execution

Input-Output and Interrupt

Hardware Summary

The Basic Computer I/O consists of a simple terminal with a keyboard and a printer/monitor.
- The terminal sends and receives serial information
- The serial information from the keyboard is shifted into INPR
- The serial information for the printer is stored in the OUTR
- INPR and OUTR communicate with the terminal serially and with the AC in parallel.
- The flags are needed to synchronize the timing difference between I/O device and the computer



INPR	Input register - 8 bits
OUTR	Output register - 8 bits
FGI	Input flag - 1 bit
FGO	Output flag - 1 bit
IEN	Interrupt enable - 1 bit

The register reference instructions are recognized by:

- $D_7IT_3 = p$
- $IR(i) = B_i$, i = 6, …, 11

	p:	SC ← 0	Clear SC
INP	pB_{11}:	AC(0-7) ← INPR, FGI ← 0	Input char. to AC
OUT	pB_{10}:	OUTR ← AC(0-7), FGO ← 0	Output char. from AC
SKI	pB_9:	if(FGI = 1) then (PC ← PC + 1)	Skip on input flag
SKO	pB_8:	if(FGO = 1) then (PC ← PC + 1)	Skip on output flag
ION	pB_7:	IEN ← 1	Interrupt enable on
IOF	pB_6:	IEN ← 0	Interrupt enable off

Table: Input-Output Instructions

Interrupts

With interrupts, the running program is not responsible for checking the status of I/O devices. Instead, it simply does its own work, and assumes that I/O will take care of itself!

When a device becomes ready, the CPU hardware initiates a branch to an I/O subprogram called an interrupt service routine (ISR), which handles the I/O transaction with the device.

An interrupt can occur during any instruction cycle as long as interrupts are enabled. When the current instruction completes, the CPU interrupts the flow of the program, executes the ISR, and then resumes the program. The program itself is not involved and is in fact unaware that it has been interrupted.

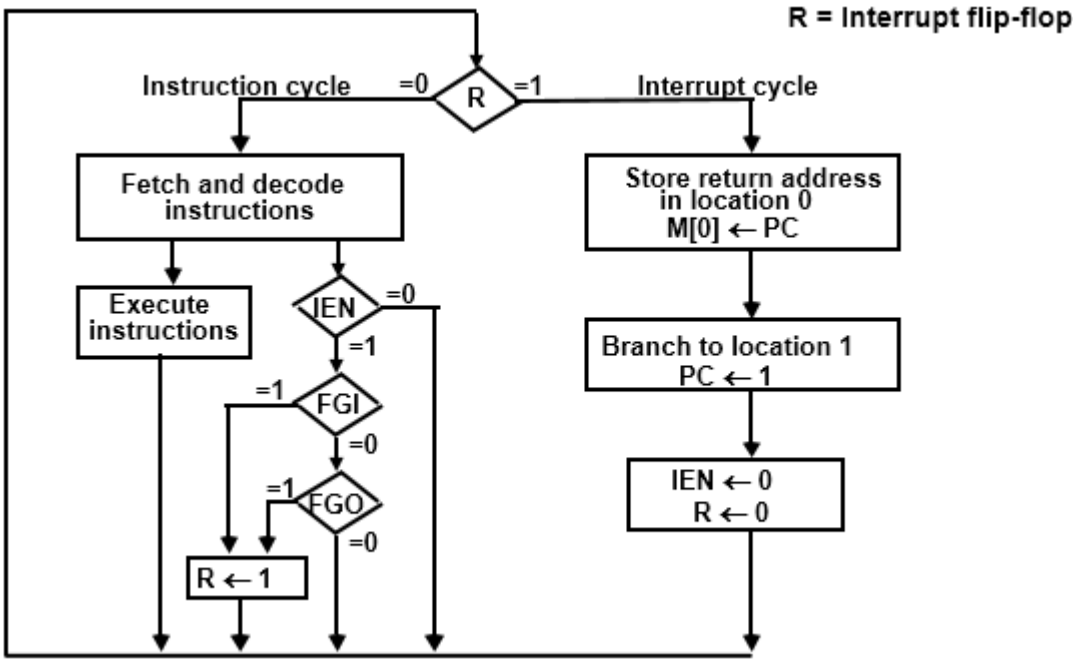


Figure: Flowchart for Interrupt Cycle

Interrupts can be globally enabled or disabled via the IEN flag (flip-flop). If interrupts are enabled, then when either FGI or FGO gets set, the R flag also gets set. (R = FGI V FGO) This allows the system to easily check whether any I/O device needs service. Determining which one needs service can be done by the ISR. If R = 0, the CPU goes through a normal instruction cycle. If R = 1, the CPU branches to the ISR to process an I/O transaction.

The interrupt cycle is a HW implementation of a branch and save return address operation. At the beginning of the next instruction cycle, the instruction that is read from memory is in address 1. At memory address 1, the programmer must store a branch instruction that sends the control to an interrupt service routine. The instruction that returns the control to the original program is "indirect BUN 0".

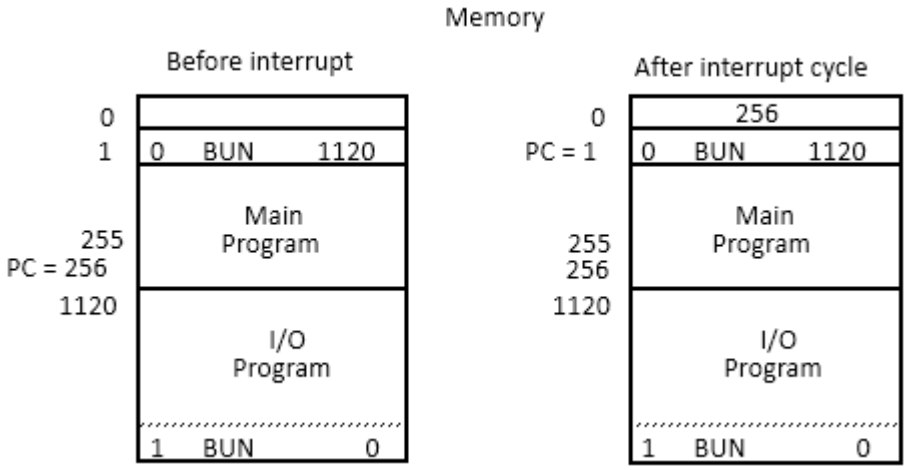


Figure: Interrupt Cycle demonstration

Register Transfer Statements for Interrupt Cycle:

R F/F ← 1 if IEN (FGI + FGO) $T_0'T_1'T_2' \Leftrightarrow T_0'T_1'T_2'$ (IEN) (FGI + FGO): R ← 1

The fetch and decode phases of the instruction cycle must be modified:

Replace T_0, T_1, T_2 with R'T0, R'T_1, R'T_2

The Interrupt Cycle:

RT_0: AR ← 0, TR ← PC

RT_1: M[AR] ← TR, PC ← 0

RT_2: PC ← PC + 1, IEN ← 0, R ← 0, SC ← 0

Book References:

(1) Andrew S. Tanenbaum, "Structured Computer Organization", Fourth Edition.
(2) M. Morris Mano, "Computer System Architecture", Pearson, 3rd Ed, 2004.
(3) M. Morris Mano, "Logic and Computer Design Fundamentals", Pearson Education, 2nd Edition
(4) John P. Hayes, "Computer Architecture & Organization".
(5) W. Stallings, "Computer Organization and Architecture – Designing for Performance", Prentice Hall of India, 7th Ed, 2007
(6) V.C. Hamacher, Z. G. Veranesic, and S. G. Zaky, "Computer Organization", Tata McGraw Hill, 5th Ed, 2002.
(7) D. A. Pattersen and J. L. Hennesy, "Computer Organization and Design: The Hardware Software Interface", Elsevier, 2nd Ed, 2006.

Web References:

(8) http://en.wikipedia.org
(9) http://www.cs.uwm.edu/
(10) http://www.transtutors.com/
(11) http://ecomputernotes.com/
(12) http://www.tutorialspoint.com/

Assignments:

(1) Explain interrupt cycle in detail with a flowchart.
(2) What is the difference between a direct and an indirect address instruction?
(3) A computer uses a memory unit with 256K words of 32 bits each. A binary instruction code is stored in one word of memory. The instruction has four parts: an indirect bit, an operation code, a register code part to specify one of 64 registers, and an address part.
    (a) How many bits are there in the operation code, the register code part, and the address part?
    (b) Draw the instruction word format and indicate the number of bits in each part.
(4) What do you understand by instruction set completeness? Explain.
(5) Explain the computer instruction with example. (T.U. 2066)
(6) What do you mean by instruction format? Explain. (T.U. 2067)
(7) Differentiate between direct and indirect addressing modes. (T.U. 2067)
(8) Explain the I/O instruction with example. (T.U. 2068)
(9) What do you mean by computer register and computer instructions? Explain. (T.U. 2069)
(10) Write short notes on the following:
    (a) Interrupt Cycle (T.U. 2068)

A Gentle Advice:

Please go through your text books and reference books for detail study!!! Thank you all.

Notes Compiled By:

Bijay Mishra
biizay.blogspot.com
9813911076 or 9841695609

Unit 4 - Control Unit

Control Unit **5 Hrs.**

Control Memory **1 Hr.**

Control Word, Control Memory, Stored Program Organization

Hardwired Control **1 Hr.**

Introduction, Timing and Control, Control Unit of Basic Computer, Timing Signal

Micro-programmed Control **2 Hrs.**

Micro-program Control Organization

Address Sequencing

- Introduction, Conditional Branching, Mapping of Instructions, Subroutines

Micro-programs

- Micro-instruction and micro-operation Format, Symbolic microinstructions, Symbolic micro-program, Binary micro-program

Design of control unit **1 Hr.**

F-field Decoding, Micro-program Sequencer

Control Memory

Control units that use dynamic microprogramming use a writable control memory. This type of memory can be used for writing (to change the microprogram) but is used mostly for reading. A memory that is part of a control unit is called a **control memory**.

The control unit in a digital computer initiates sequences of **micro-operations**. The control variables can be represented by a string of 1's and 0's called a **control word**. The control information is shared in a control memory, in a **micro-programmed** organization. A micro-programmed control unit is a control unit whose binary control variables are stored in memory. Each word in control memory contains within it is a **microinstruction.** The control memory is programmed to begin the desired sequence of micro-operation.

A sequence of microinstructions constitutes a **microprogram**. When the control signals are generated by hardware, it is **hardwired**. In a bus-oriented system, the control signals that specify micro-operations are groups of bits that select the paths in multiplexers, decoders, and ALUs.

The control memory is usually a ROM, which stores all control information permanently. The **control address register** (CAR) specifies the address of the microinstruction, and the control data register holds the microinstruction read from memory.

b i i z a y @ g m a i l . c o m

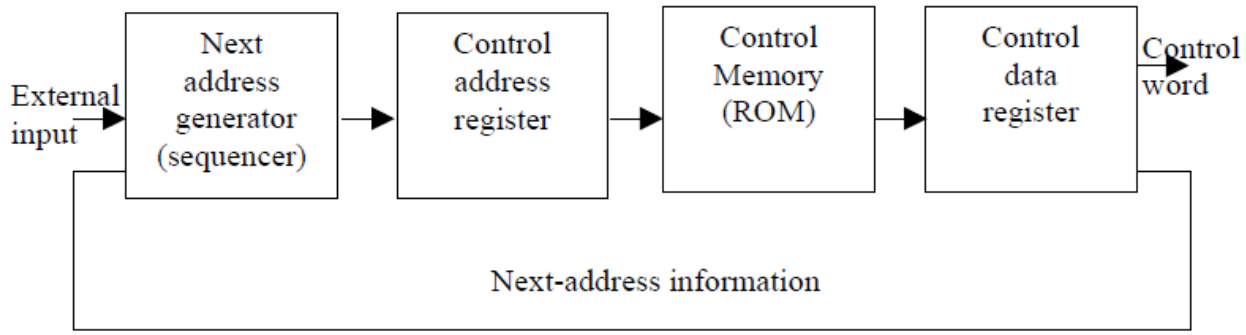


Figure: Micro-programmed Control Organization

The microinstruction contains a control word that specifies one or more micro-operations for the data processor. Once these operations are executed, the control must determine the next address. The location of the next microinstruction is generally the one next in sequence, otherwise, it may be located somewhere else in the control memory. For this reason it is necessary to use some bits of the present microinstruction to control the generation of the address of the next microinstruction. The next address may also be a function of external input conditions.

While the micro-operations are being executed, the next address is computed in the **next address generator** circuit and then transferred into the control address register to read the next micro-instruction. Hence a microinstruction contains bits for initiating micro-operations in the data processor part and bits that determine the address sequence for the control memory.

A microprogram **sequencer** is the next address generator, as it determines the address sequence that is read from control memory. The address of the next microinstruction can be specified in several ways depending on the sequencer inputs.

Typical functions of a microprogram sequencer are:

- incrementing the CAR by one
- loading into the CAR and address from control memory
- transferring an external address
- loading an initial address to start the control operations

The **control data register** (CDR) stores the present microinstruction while the next address is computed and read from memory. The data register is also called a **pipeline register**. It allows the execution of the micro-operations specified by the control word simultaneously with the generation of the next microinstruction. This configuration requires a two-phase clock, with one clock applied to the address register and the other to the data register.

The main advantage of the micro-programmed control is that once the hardware configuration is built, there should be no need for further hardware or wiring changes. If we want to make a different control sequence for the system, all we need to do is to specify a different set of microinstructions for control memory. The hardware configuration should not be changed for different operations. We have to change only the microprogram residing in control memory.

b i i z a y @ g m a i l . c o m

Timing and Control

Timing pulses are used in sequencing the micro-operations in an instruction. A master clock generator is used for controlling the timing for all register in a computer system. A state of a register cannot be changed by a clock pulse until it is enabled by the control signal, which are generated in the control unit and provide control inputs for multiplexers, processor register, and micro-operations. The control organization is of two types; hardwired control and micro-programmed control.

Hardwired Control

In a hardwired control, the control signals are generated by using the collection of combinational circuits. The main advantage of the hardwired control is that, it can be optimized to produce a fast mode of operation. Whenever a change or modification is to be done in the design, then the wiring among the various components needs to be done.

Micro-Programmed Control

In a micro-programmed control, a control memory is used for storing control information which is also programmed for initiating the sequence of micro-operations. Whenever any change or modification is required in the design, it can be done by updating the micro-program in the control memory.

Control Unit

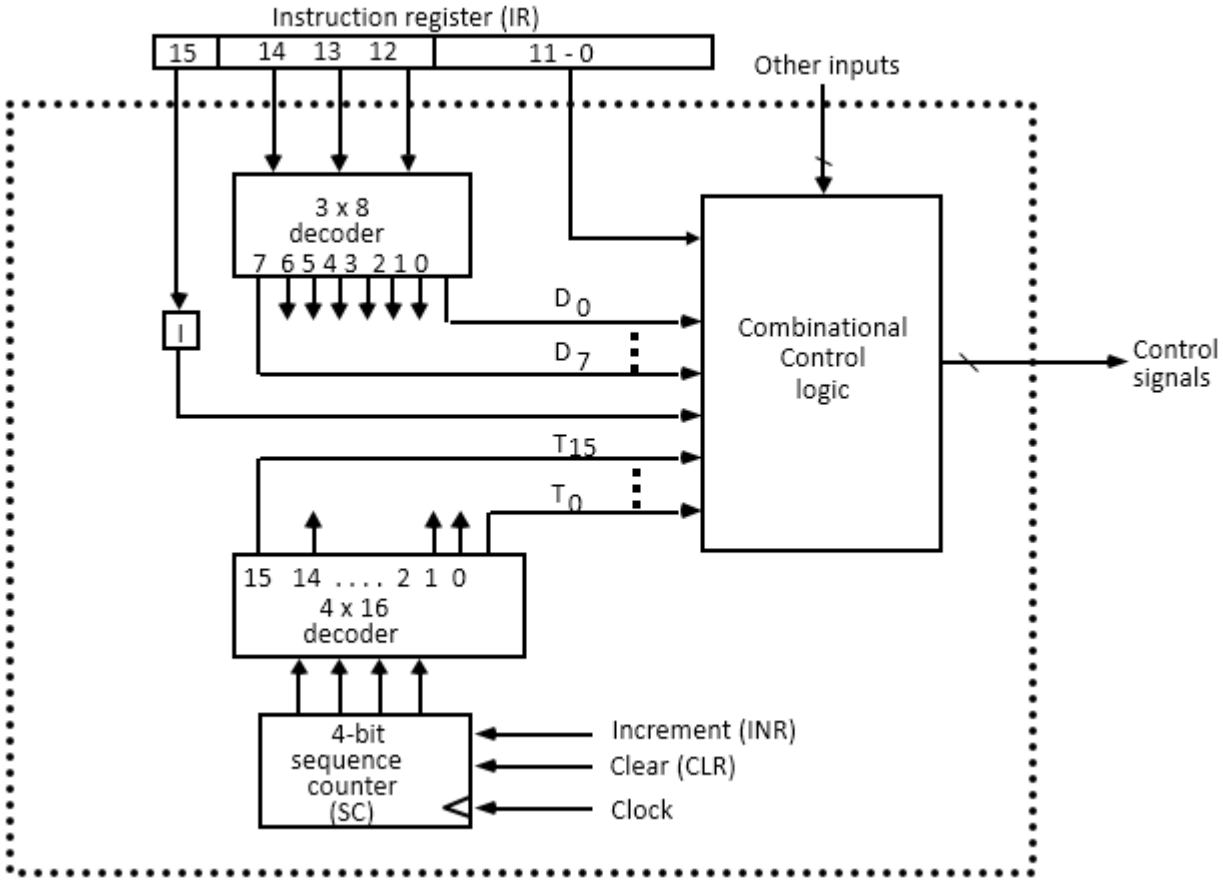


Figure: Control unit of Basic Computer

b i i z a y @ g m a i l . c o m

Control unit consists of two decoders, a sequence counter, and number of logic gates. When an instruction is read from the memory, it is placed in the instruction register (IR). The instruction register is divided into three parts; addressing mode, opcode, and address. Control unit is responsible for interpreting the instruction code and providing the necessary control needed for processing these instructions. Control unit uses the instruction format for interpreting the instruction.

Timing is generated by 4-bit sequence counter and 4x16 decoder. The SC can be incremented or cleared. Example: $T_0, T_1, T_2, T_3, T_4, T_0, T_1, \ldots$

Assume: At time T_4, SC is cleared to 0 if decoder output D_3 is active. This is expressed as:

$$D_3T_4: SC \leftarrow 0$$

Initially, the CLR input of SC is active. The last three waveform on figure below shows how SC is cleared when $D_3T_4 = 1$. Output D_3 from the operation decoder becomes active at the end of the timing signal T_2. When timing signal T_4 becomes active, the output of the AND gate that implements the control function D_3T_4 becomes active. This signal is applied to the CLR input of SC. On the next positive clock transition the counter is cleared to 0. This causes the timing signal T_0 to become active instead of T_5 that would have been active if SC were incremented instead of cleared.

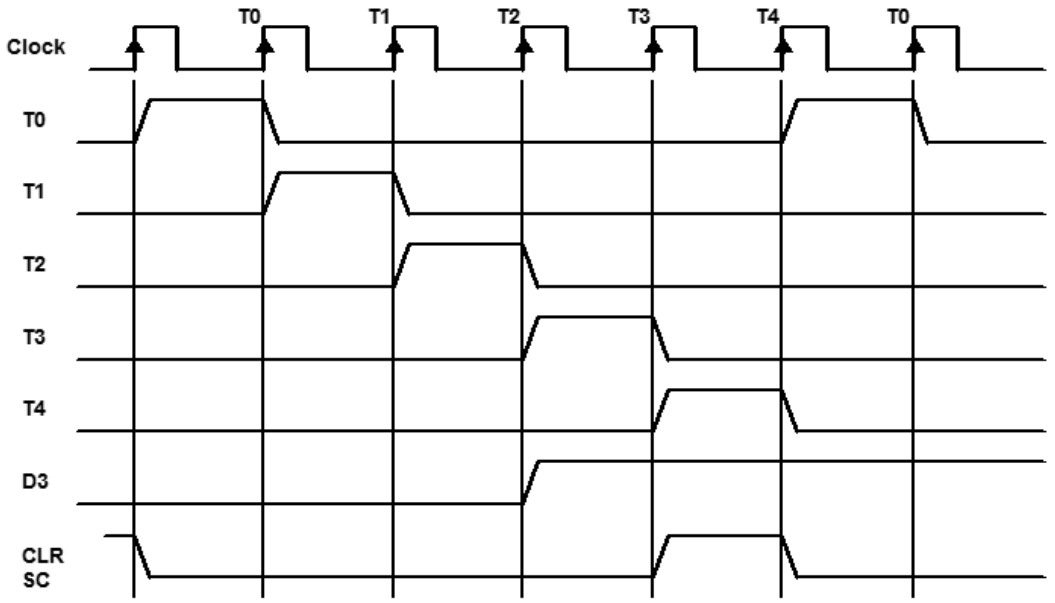


Figure: Example of Control Timing Signal

Address Sequencing

The microprogram consists of microinstructions that specify various internal control signals for execution of register micro-operations. Process of finding address of next microinstruction to be executed is called **address sequencing**. Address sequencer must have capabilities of finding address of next micro-instruction in following situations:

- In-line Sequencing
- Unconditional Branch

- Conditional Branch
- Subroutine Call and Return
- Looping
- Mapping from instruction opcode to address in control memory

Microinstructions are stored in control memory in groups, with each group specifying a routine. Each computer instruction has its own microprogram routine to generate the micro-operations. The hardware that controls the address sequencing of the control memory must be capable of sequencing the microinstructions within a routine and be able to branch from one routine to another.

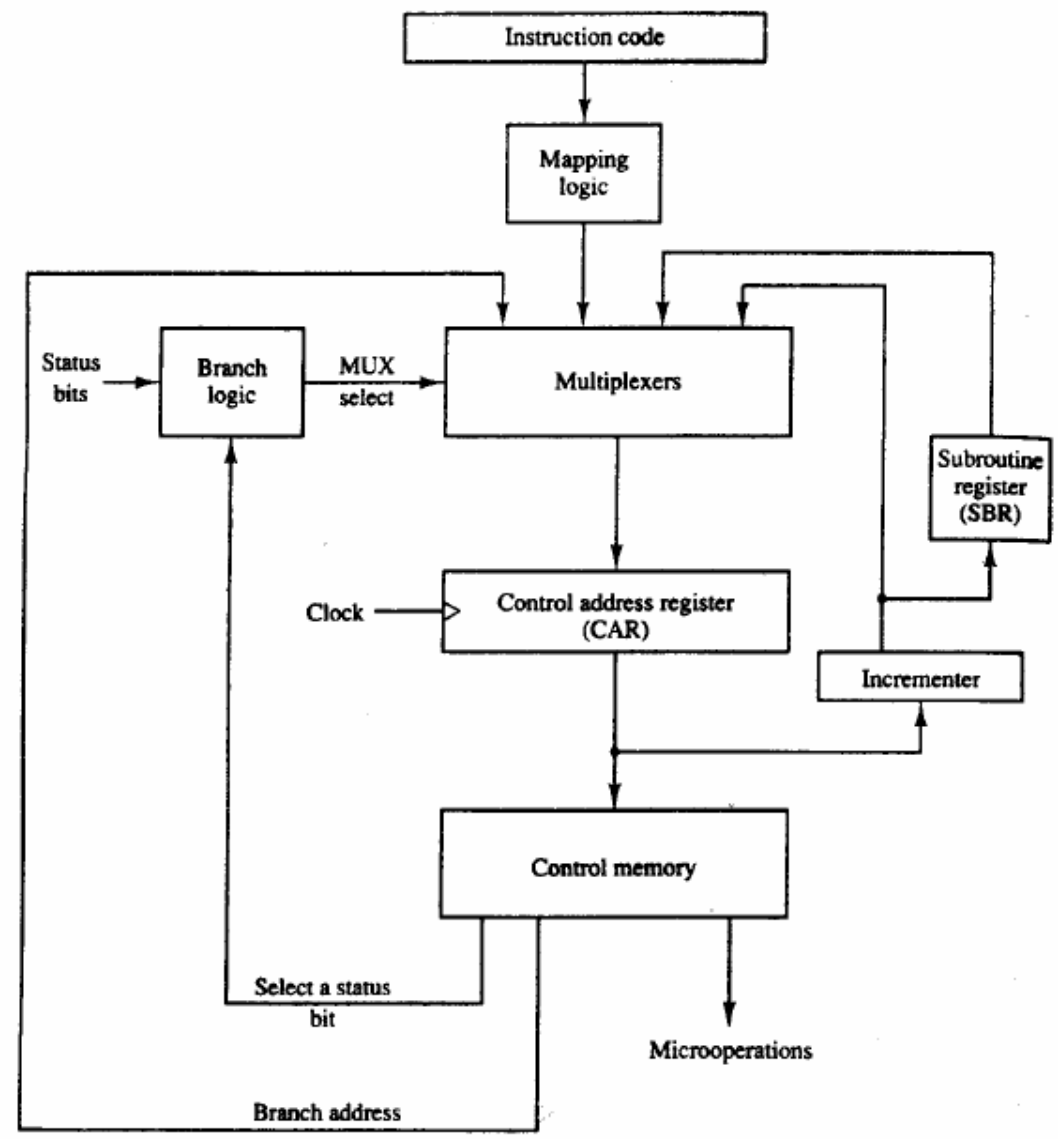


Figure: Selection of Address for Control Memory

The steps that the control must undergo during the execution of a single computer instruction are:

- Load an initial address into the CAR when power is turned on in the computer. This address is usually the address of the first microinstruction that activates the instruction fetch routine. IR holds instruction.

- The control memory then goes through the routine to determine the effective address of the operand. AR holds operand address.
- The next step is to generate the micro-operations that execute the instruction by considering the opcode and applying a mapping.
- After execution, control must return to the fetch routine by executing an unconditional branch.

The microinstruction in control memory contains a set of bits to initiate micro-operations in computer registers and other bits to specify the method by which the next address is obtained.

Conditional Branching

Conditional branching is obtained by using part of the microinstruction to select a specific status bit in order to determine its condition. The status conditions are special bits in the system that provide parameter information such as the carry-out of an adder, the sign bit of a number, the mode bits of an instruction, and i/o status conditions. The status bits, together with the field in the microinstruction that specifies a branch address, control the branch logic. The branch logic tests the condition, if met then branches, otherwise, increments the CAR. If there are 8 status bit conditions, then 3 bits in the microinstruction are used to specify the condition and provide the selection variables for the multiplexer.

If Condition is true, set the appropriate field of status register to 1. Conditions are tested for O (overflow), N (negative), Z (zero), C (carry), etc. Then test the value of that field if the value is 1 take branch address from the next address field of the current microinstruction). Otherwise simple increment the address.

Unconditional Branching

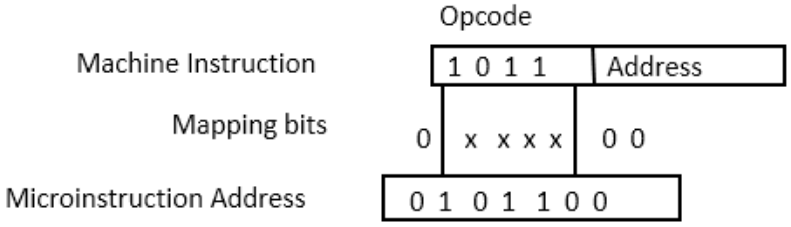
For unconditional branching, fix the value of one status bit to be one load the branch address from control memory into the CAR.

Mapping of Instruction

A special type of branch exists when a microinstruction specifies a branch to the first word in control memory where a microprogram routine is located. The status bits for this type of branch are the bits in the opcode. Assume an opcode of four bits and a control memory of 128 locations. The mapping process converts the 4-bit opcode to a 7-bit address for control memory. This provides for each computer instruction a microprogram routine with a capacity of four microinstructions.

Another Approach of Mapping

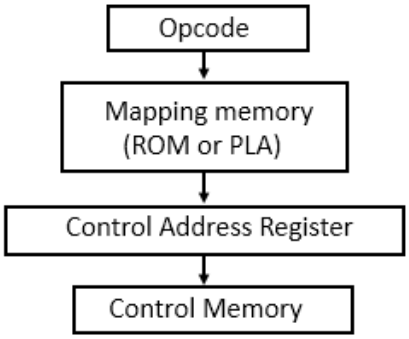
Modify opcode to use it as an address of control memory.



b i i z a y @ g m a i l . c o m

Mapping Function Implemented by ROM or PLA

Use opcode as address of ROM where address of control memory is stored and then use that address as an address of control memory.



Subroutines

Subroutines are programs that are used by other routines to accomplish a particular task and can be called from any point within the main body of the microprogram. Frequently many micro-programs contain identical section of code. Microinstructions can be saved by employing subroutines that use common sections of microcode.

Microprogram Example

Once we have a configuration of a computer and its micro-programmed control unit, the designer generates the microcode for the control memory. Code generation of this type is called microprogramming and is similar to conventional machine language programming. The block diagram of computer consists of:

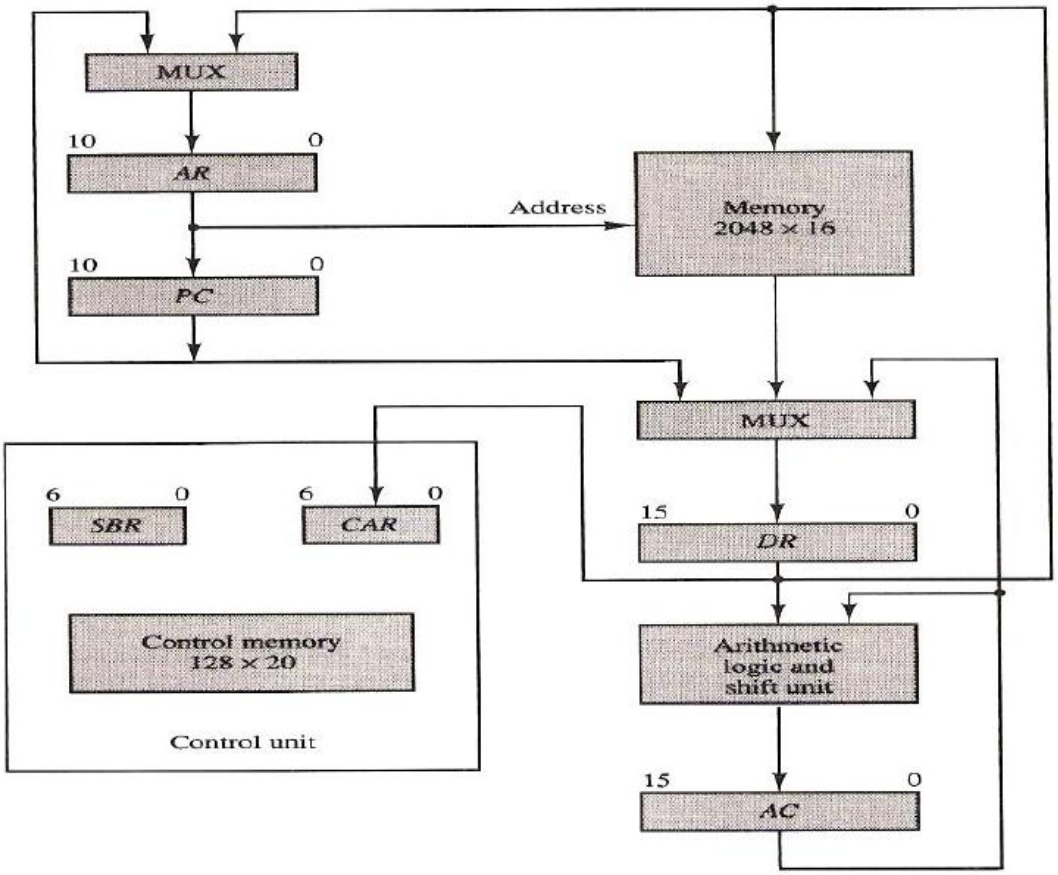


Figure: Computer hardware configuration

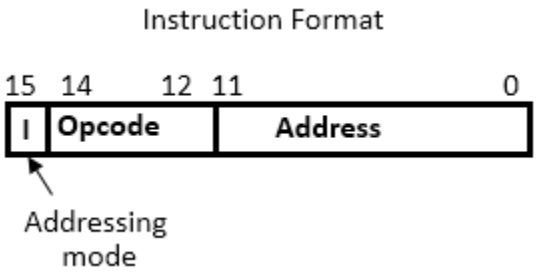
b i i z a y @ g m a i l . c o m

Transfer of information among registers in the processor is through Multiplexers rather than a bus.

Two memory units:
Main memory – stores instructions and data
Control memory – stores microprogram

Four processor registers:
Program counter – PC
Address register – AR
Data register – DR
Accumulator register - AC

Two control unit registers:
Control address register – CAR
Subroutine register – SBR



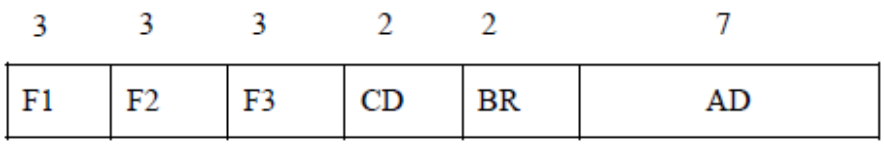
Three fields for an instruction:
I = 1-bit for indirect addressing
Opcode = 4-bit
Address Field = 11-bit

The example will only consider the following 4 of the possible 16 memory instructions

Symbol	Opcode	Description
ADD	0000	AC ← AC + M[EA]
BRANCH	0001	If (AC < 0) then (PC ← EA)
STORE	0010	M[EA] ← AC
EXCHANGE	0011	AC ← M[EA], M[EA] ← AC

Note: (EA is the effective address)

The microinstruction format is composed of 20 bits with four parts:



b i i z a y @ g m a i l . c o m

Three fields F1, F2, and F3 specify micro-operations for the computer [3 bits each].
The **CD** field selects status bit conditions [2 bits]
The **BR** field specifies the type of branch to be used [2 bits]
The **AD** field contains a branch address [7 bits]

Each of the three micro-operation fields can specify one of seven possibilities. Therefore only 21 micro-operations are used. No more than three micro-operations can be chosen for a microinstruction. If fewer than three are needed, the code 000 = NOP is used.

Five letters are used to specify a transfer-type micro-operation. First two designate the **source register**, Third is a '**T**', and Last two designate the **destination register.**

F1	Microoperation	Symbol
000	None	NOP
001	$AC \leftarrow AC + DR$	ADD
010	$AC \leftarrow 0$	CLRAC
011	$AC \leftarrow AC + 1$	INCAC
100	$AC \leftarrow DR$	DRTAC
101	$AR \leftarrow DR(0\text{-}10)$	DRTAR
110	$AR \leftarrow PC$	PCTAR
111	$M[AR] \leftarrow DR$	WRITE

F2	Microoperation	Symbol
000	None	NOP
001	$AC \leftarrow AC - DR$	SUB
010	$AC \leftarrow AC \vee DR$	OR
011	$AC \leftarrow AC \wedge DR$	AND
100	$DR \leftarrow M[AR]$	READ
101	$DR \leftarrow AC$	ACTDR
110	$DR \leftarrow DR + 1$	INCDR
111	$DR(0\text{-}10) \leftarrow PC$	PCTDR

F3	Microoperation	Symbol
000	None	NOP
001	$AC \leftarrow AC \oplus DR$	XOR
010	$AC \leftarrow \overline{AC}$	COM
011	$AC \leftarrow$ shl AC	SHL
100	$AC \leftarrow$ shr AC	SHR
101	$PC \leftarrow PC + 1$	INCPC
110	$PC \leftarrow AR$	ARTPC
111	Reserved	

b i i z a y @ g m a i l . c o m

CD	Condition	Symbol	Comments
00	Always = 1	U	Unconditional branch
01	$DR(15)$	I	Indirect address bit
10	$AC(15)$	S	Sign bit of AC
11	$AC = 0$	Z	Zero value in AC

BR	Symbol	Function
00	JMP	$CAR \leftarrow AD$ if condition = 1
		$CAR \leftarrow CAR + 1$ if condition = 0
01	CALL	$CAR \leftarrow AD$, $SBR \leftarrow CAR + 1$ if condition = 1
		$CAR \leftarrow CAR + 1$ if condition = 0
10	RET	$CAR \leftarrow SBR$ (Return from subroutine)
11	MAP	$CAR(2\text{–}5) \leftarrow DR(11\text{–}14)$, $CAR(0,1,6) \leftarrow 0$

Table: Symbols and Binary Code for Microinstruction Fields

Symbolic Microinstructions

A symbolic microprogram can be translated into its binary equivalent by means of an assembler. Each line of an assembly language microprogram defines a symbolic microinstruction and is divided into five fields: Label, micro-operations, CD, BR, and AD. The fields specify the following information:

1. The label field may be empty or it may specify a symbolic address. Terminate with a colon (:)
2. The micro-operations field consists of 1-3 symbols, separated by commas. Only one symbol from each F field. If NOP, then translated to 9 zeros
3. The condition field specifies one of the four conditions: U, I, S or Z
4. The branch field has one of the four branch symbols
5. The address field has three formats
   a. A symbolic address – must also be a label
   b. The symbol NEXT to designate the next address in sequence
   c. Empty if the branch field is RET or MAP and AD is converted to 7 zeros

Design of Control Unit

After getting the micro-operations we have to execute these micro-operations but before that we need to decode them. The 9-bits of the micro-operation field are divided into 3 subfields of 3 bits each. The control memory output of each subfield must be decoded to provide distinct micro-operations. The outputs of the decoders are connected to the appropriate inputs in the processor unit. The Figure below shows 3 decoders and connections that must be made from their outputs.

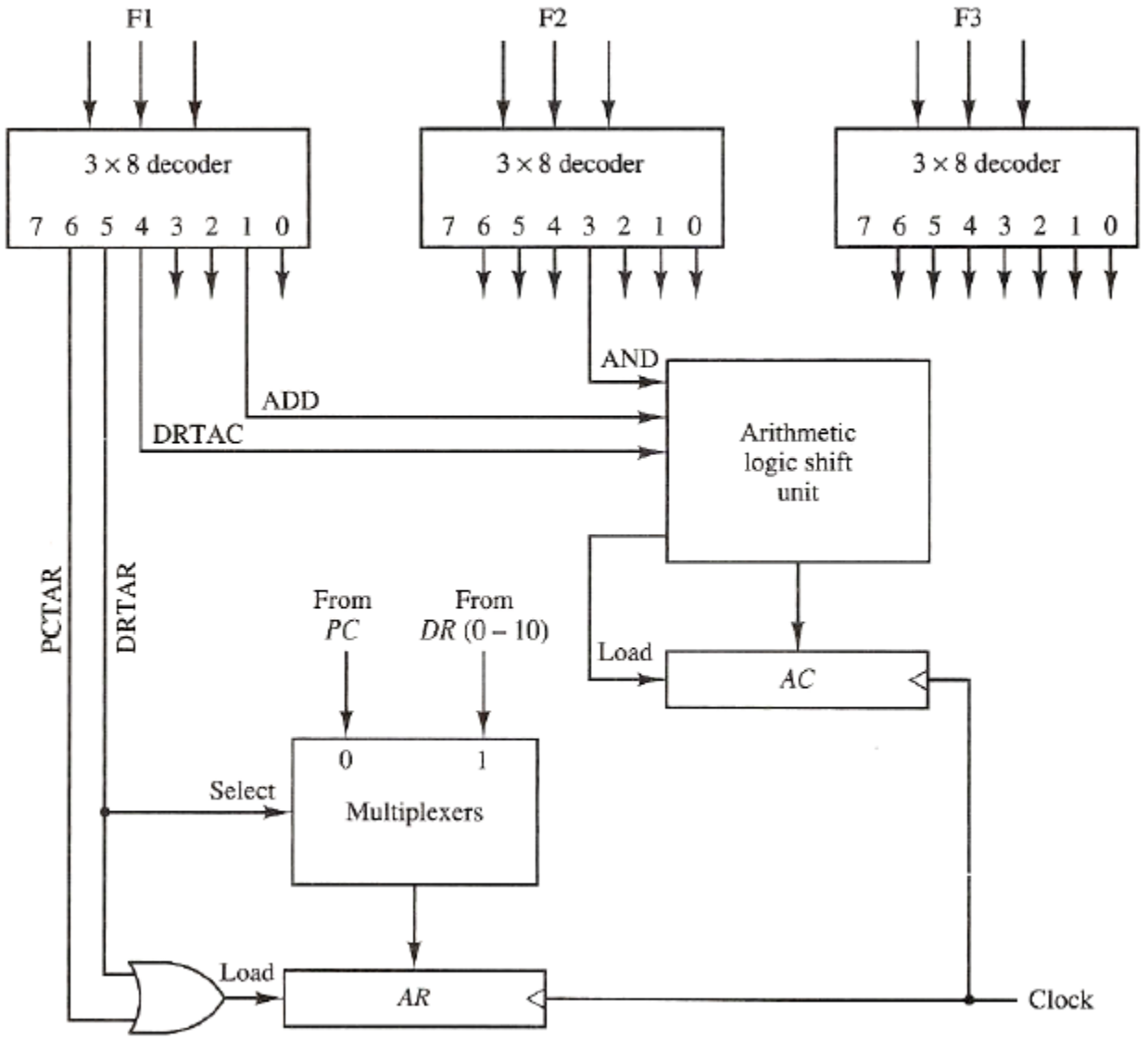


Fig. Decoding of Micro-operation Fields.

Three decoders and some of the connections that must be made from their outputs. Each of the three fields of the microinstruction presently available in the output of control memory are decoded with a 3x8 decoder to provide eight outputs.

- when F1 = 5, transfers the content of DR(0-10) to AR (DRTAR)
- when F1 = 6 there is a transfer from PC to AR (PCTAR)
- Outputs 5 and 6 of decoder F1 are connected to the load input of AR so that information is transferred to AR.

The multiplexers select the information from DR when output 5 is active and from PC when output 5 is inactive. Because we have 8 micro-operations represented with the help of 3 bits in every table and also we have 3 such tables possible we have decoded these micro-operations field bits with three 3 x 8 decoders.

b i i z a y @ g m a i l . c o m

After getting the micro-operations, we have to give it to particular circuits, the data manipulation type of micro-operations like AND, ADD, Sub and so on we give to ALU and the corresponding results moved to AC. The ALU has been provided data from AC and DR.

And for data transfer type of instructions like in the case of PCTAR or DRTAR we need to simply transfer the values. Because we have two options for data transfer in AR we are taking the help of MUX to choose one. We will take 2 x 1 MUX and one select line which is attached with DRTAR micro-operation signal. That means if DRTAR is high then MUX will choose DR to transfer the data to AR else PC's data will be moved to AR. And the corresponding data movement will be done with the help of load high or not. If any of the values is high the value will be loaded to AR.

The clock signal is provided for the synchronization of micro-operations.

Instead of using gates to generate the control signals marked by the symbols AND, ADD, and DR. These inputs will now come from the outputs of the decoders associated with the symbols AND, ADD, and DRTAC respectively. The other outputs of the decoders that are associated with an AC operation must also be connected to the arithmetic logic shift unit in a similar fashion.

Microprogram Sequencer:

The basic components of a micro-programmed control unit are the control memory and the circuits that select the next address. The address selection part is called a micro-program sequencer. It can be constructed with digital functions to suit a particular application. Main purpose is to present an address to the control memory so that a microinstruction may be read and executed.

Design of input logic:

The input logic circuit in the figure below has three inputs, I_0, I_1, and T, and three outputs S_0, S_1, and L. Variables S_0 and S_1 select one of the source addresses for CAR. Variable L enables the load input in SBR. The binary values of the two selection variables determine the path in the multiplexer.

For example, with $S_1S_0 = 10$, multiplexer input number 2 is selected and establishes a transfer path from SBR to CAR. Note that each of the four inputs as well as the output of MUX 1 contains a 7-bit address.

The truth table can be used to obtain the simplified Boolean functions for the input logic circuit:

$S_1 = I_1$

$S_0 = I_1I_0 + I'_1T$

$L = I'_1I_0T$

The circuit can be constructed with three AND gates, an OR gate and an inverter. The truth table for the input logic circuit is shown in table below:

b i i z a y @ g m a i l . c o m

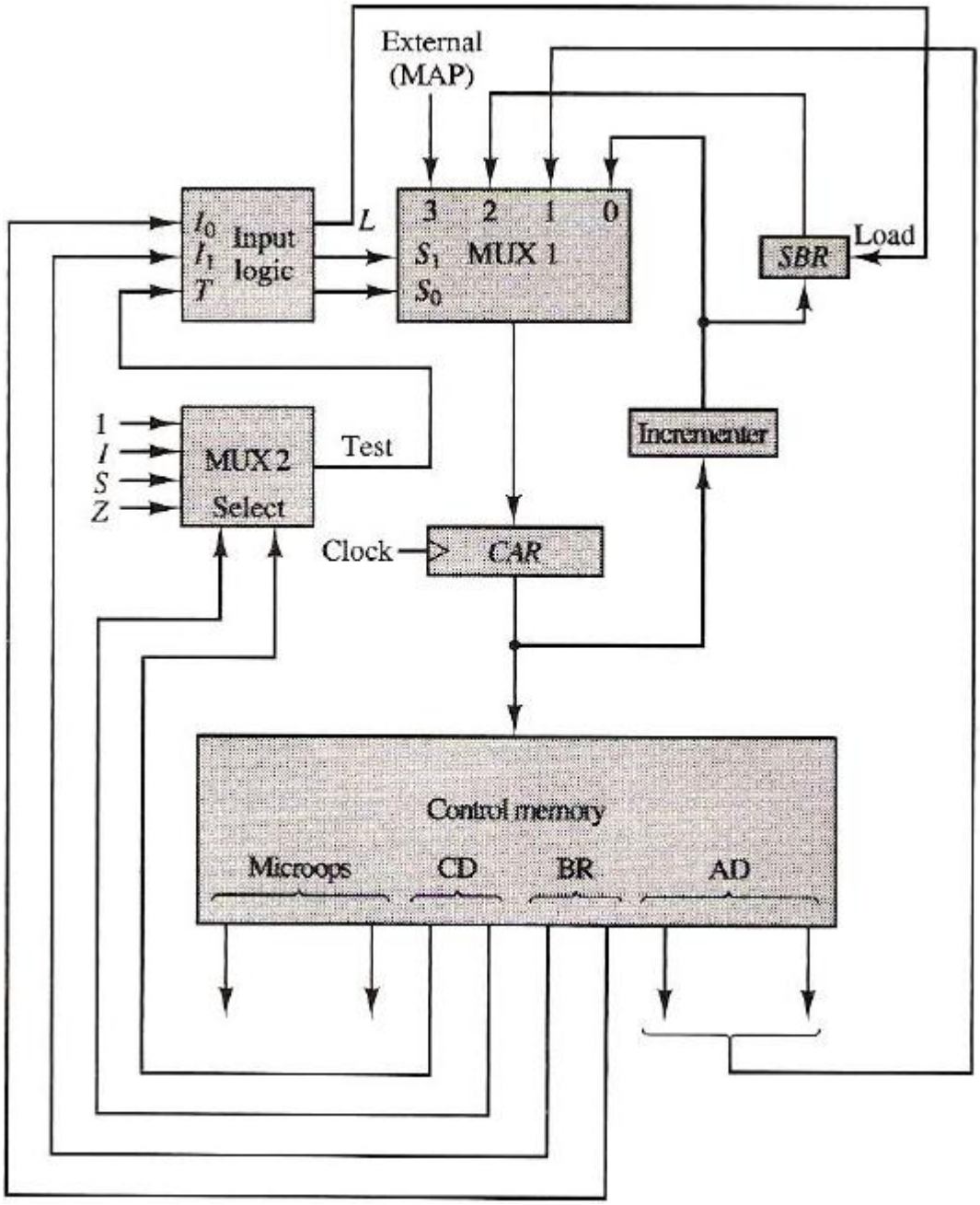


Figure: Microprogram Sequencer for a Control Memory

BR Field		Input I_1	I_0	T	MUX 1 S_1	S_0	Load *SBR* L
0	0	0	0	0	0	0	0
0	0	0	0	1	0	1	0
0	1	0	1	0	0	0	0
0	1	0	1	1	0	1	1
1	0	1	0	×	1	0	0
1	1	1	1	×	1	1	0

b i i z a y @ g m a i l . c o m

References:

1. Andrew S. Tanenbuam, "Structured Computer Organization", PHI
2. J. P. Hayes, "Computer Architecture and Organization", McGraw Hill, 3rd Ed, 1998.
3. M. Morris Mano, "Computer System Architecture", Pearson, 3rd Ed, 2004.
4. M. Morris Mano, "Digital Design", Pearson Education, Third Edition
5. M. Morris Mano, "Logic and Computer Design Fundamentals", Pearson Education, 2nd Edition
6. V.C. Hamacher, Z. G. Veranesic, and S. G. Zaky, "Computer Organization", Tata McGraw Hill, 5th Ed, 2002.
7. W. Stallings, "Computer Organization and Architecture – Designing for Performance", Prentice Hall of India, 7th Ed, 2007
8. D. A. Pattersen and J. L. Hennesy, "Computer Organization and Design: The Hardware Software Interface", Elsevier, 2nd Ed, 2006.

Assignments:

(1) What is the difference between a microprocessor and a microprogram? Is it possible to design a microprocessor without a microprogram? Are all micro-programmed computers also microprocessor?

(2) Explain hardwired control and micro-programmed control. Is it possible to have a hardwired control associated with a control memory?

(3) Define: micro-operation, microinstruction, microprogram, and microcode.

(4) Using table 7-1 from textbook (Morris Mano, 3rd Edition), give the 9-bit micro-operation field for the following micro-operations:

   a. AC ← AC + 1, DR ← DR + 1
   b. PC ← PC + 1, DR ← M[AR]
   c. DR ← AC, AC ← DR

(5) What do you mean by field decoding? Explain with block diagram. (T.U. 2066)

(6) Explain the microprogram sequence with an example along with a suitable diagram and a truth table. (T.U. 2067)

(7) Differentiate between hardwired and microprogram control unit. (T.U. 2067 and 2070)

(8) What do you mean by control memory? Explain the microinstructions and micro-operation format. (T.U. 2068)

(9) What is the general model of Microprogram Control Unit? Explain the major steps while designing the microprogram control unit. (T.U. 2070)

A Gentle Advice:

Please go through your text books and reference books for detail study!!! Thank you all.

Notes Compiled By:

Bijay Mishra
biizay.blogspot.com
9813911076 or 9841695609

Unit 5 - Central Processing Unit

Central Processing Unit **6 Hrs.**

Register Organization **1 Hr.**

Bus system of CPU, Control Word, ALU and Micro-operation for CPU

Register Stack and Memory Stack **1 Hr.**

LIFO and Stack Pointer, Register Stack, Memory Stack

One address and two address instruction **1 Hr.**

Instruction format, One address instruction, Two address instruction, Three address instruction, and Zero address instruction

Addressing modes **1 Hr.**

Introduction, Implied mode, Immediate mode, Register mode, Register indirect mode, Auto-increment/Auto-decrement mode, Relative address mode, Indexed addressing mode, Base register addressing mode

Data transfer and Manipulation **1 Hr.**

Basic operations, Data transfer instructions

Data manipulation instructions

- Types, Arithmetic instructions, Logical and bit manipulation instruction, Shift instruction

Introduction to RISC and CISC **1 Hr.**

Introduction to RISC and CISC, Characteristics of RIAC and CISC, Overlapped Register Window

Introduction

The part of the computer that performs the bulk of data-processing operations is called the central processing unit and is referred to as the CPU. The Arithmetic Logic Unit (ALU) and the Control Unit (CU) together are termed as the Central Processing Unit (CPU). The CPU is the most important component of a computer's hardware. The CPU is made up of three major parts, as shown in figure below:

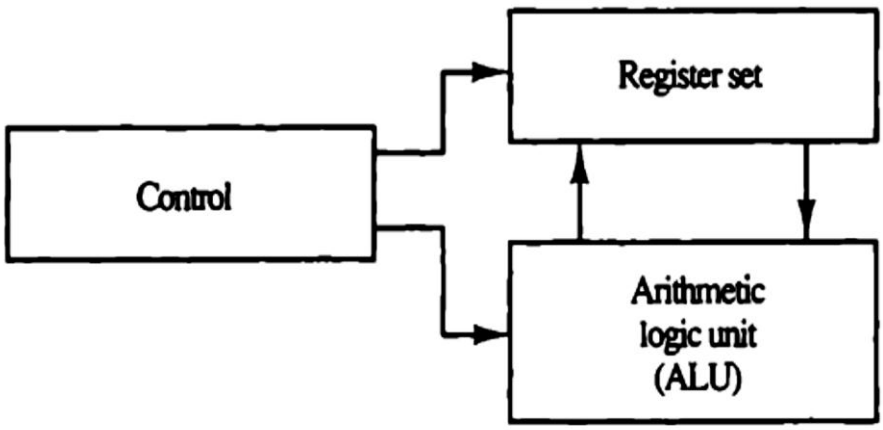


Figure: Components of CPU

b i i z a y @ g m a i l . c o m

1. A set of registers for holding binary information.
2. An arithmetic and logic unit (ALU) for performing data manipulation, and
3. A control unit that coordinates and controls the various operations and initiates the appropriate sequence of micro-operations for each task.

Computer instructions are normally stored in consecutive memory locations and are executed in sequence one by one. The control unit allows reading of an instruction from a specific address in memory and executes it with the help of ALU and Register.

All the arithmetic and logical Operations are performed in the CPU in special storage areas called registers. The size of the register is one of the important considerations in determining the processing capabilities of the CPU. Register size refers to the amount of information that can be held in a register at a time for processing. The larger the register size, the faster may be the speed of processing.

The register set stores intermediate data used during the execution of the instructions. The arithmetic logic unit (ALU) performs the required micro-operations for executing the instructions. The control unit supervises the transfer of information among the registers and instructs the ALU as to which operation to perform.

General Register Organization

The number and the nature of registers is a key factor that differentiates among computers. For example, Intel Pentium has about 32 registers. Some of these registers are special registers and others are general-purpose registers.

The general-purpose registers as the name suggests can be used for various functions. For example, they may contain operands or can be used for calculation of address of operand etc.

When a large number of registers are included in the CPU, it is most efficient to connect them through a common bus system. The registers communicate with each other not only for direct data transfers, but also while performing various micro-operations. Hence it is necessary to provide a common unit that can perform all the arithmetic, logic, and shift micro-operations in the processor.

Why we need CPU registers?

During instruction execution, we could store pointers, counters, return addresses, temporary results and partial products in some locations in RAM, but having to refer memory locations for such applications is time consuming compared to instruction cycle. So for convenient and more efficient processing, we need processor registers (connected through common bus system) to store intermediate results.

A bus organization for seven CPU registers is shown in figure below. The output of each register is connected to two multiplexers (MUX) to form the two buses A and B. The selection lines in each multiplexer select one register or the input data for the particular bus. The A and B buses form the inputs to a common arithmetic logic unit (ALU). The operation selected in the ALU determines the arithmetic or logic micro-operation that is to be performed.

The result of the micro-operation is available for output data and also goes into the inputs of all the registers. The register that receives the information from the output bus is selected by a decoder. The decoder activates one of the register load inputs, thus providing a transfer path between the data in the output bus and the inputs of the selected destination register.

For example, to perform the operation R1 ← R2 + R3 the control must provide binary selection variables to the following selector inputs:

1. MUX A selector (SELA): to place the content of R2 into bus A.
2. MUX B selector (SELB): to place the content of R3 into bus B.
3. ALU operation selector (OPR): to provide the arithmetic addition A + B.
4. Decoder destination selector (SELD): to transfer the content of the output bus into R1

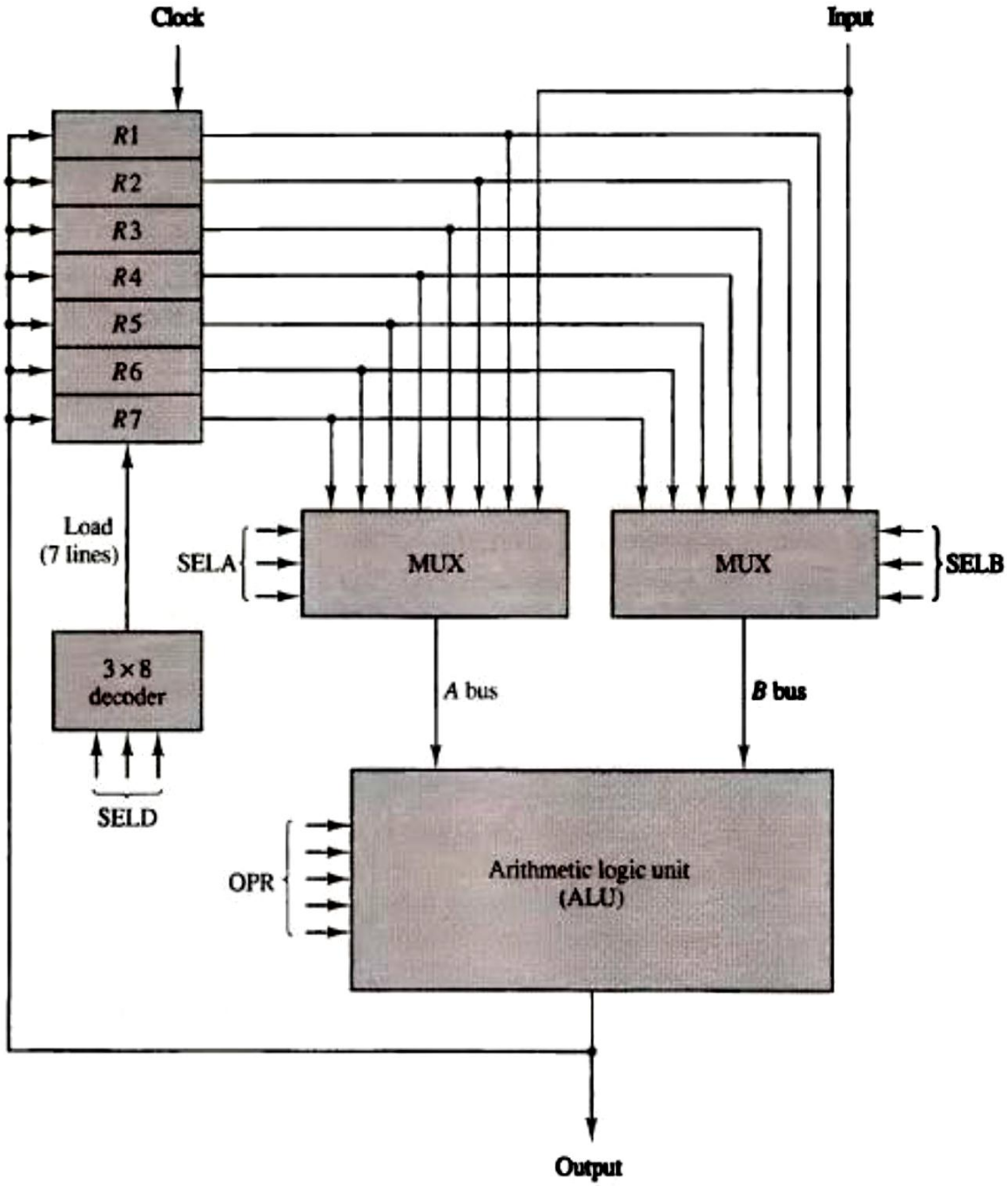


Figure: Register set with common ALU

b i i z a y @ g m a i l . c o m

Control Word

There are 14 binary selection inputs in the unit, and their combined value specifies a control word. The 14-bit control word is defined in figure below.

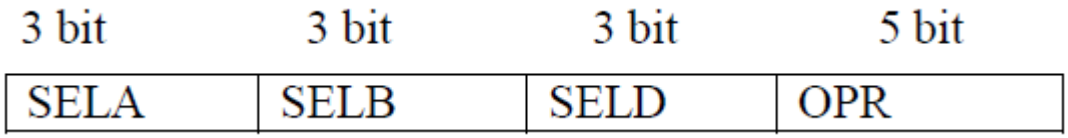


Figure: Control Word

It consists of four fields. Three fields contain three bits each, and one field has five bits. The three bits of SELA select a source register for the A input of the ALU. The three bits of SELB select a register for the B input of the ALU. The three bits of SELD select a destination register using the decoder and its seven load outputs. The five bits of OPR select one of the operations in the ALU. The 14-bit control word when applied to the selection inputs specify a particular micro-operation.

Binary Code	SELA	SELB	SELD
000	Input	Input	None
001	R1	R1	R1
010	R2	R2	R2
011	R3	R3	R3
100	R4	R4	R4
101	R5	R5	R5
110	R6	R6	R6
111	R7	R7	R7

Table: Encoding of register selection fields

OPR Select	Operation	Symbol
00000	Transfer A	TSFA
00001	Increment A	INCA
00010	Add A + B	ADD
00101	Subtract A – B	SUB
00110	Decrement A	DECA
01000	AND A and B	AND
01010	OR A and B	OR
01100	XOR A and B	XOR
01110	Complement A	COMA
10000	Shift right A	SHRA
11000	Shift left A	SHLA

Table: Encoding of ALU operations

b i i z a y @ g m a i l . c o m

Examples of Microoperations

Suppose, we have the subtract microperation as: R1 ← R2 – R3. A control word of 14-bits is needed to specify a micro-operation in the CPU. The control word for a given micro-operation can be derived from the selection variables. Here, R1 ← R2 – R3, specifies R2 for the A input of the ALU, R3 for the B input of the ALU, R1 for the destination register, and the ALU operation to subtract A-B. Thus the control word is specified by the four fields and can obtain as follows:

Field:	SELA	SELB	SELD	OPR
Symbol:	R2	R3	R1	SUB
Control Word:	010	011	001	00101

Hence the control word for this micro-operation was: 01001100100101

	Symbolic Designation				
Microoperation	SELA	SELB	SELD	OPR	Control Word
$R1 \leftarrow R2 - R3$	R2	R3	R1	SUB	010 011 001 00101
$R4 \leftarrow R4 \vee R5$	R4	R5	R4	OR	100 101 100 01010
$R6 \leftarrow R6 + 1$	R6	—	R6	INCA	110 000 110 00001
$R7 \leftarrow R1$	R1	—	R7	TSFA	001 000 111 00000
Output ← R2	R2	—	None	TSFA	010 000 000 00000
Output ← Input	Input	—	None	TSFA	000 000 000 00000
$R4 \leftarrow$ shl $R4$	R4	—	R4	SHLA	100 000 100 11000
$R5 \leftarrow 0$	R5	R5	R5	XOR	101 101 101 01100

Table: Examples of Microoperations for CPU

Stack Organization

Stack is a Last In First Out (LIFO) list. Stack is a storage that stores information in such a manner that the item stored last is the first item retrieved. Stack consists of a memory unit with address register that holds the address for the stack called a stack pointer (SP) which always points at the top item in the stack. The two operation of stack are the insertion and deletion of items. The operation of insertion is called ***PUSH*** and the deletion is called ***POP***.

Register Stack

It is the collection of finite number of registers. Stack pointer (SP) points to the register that is currently at the top of stack. Diagram shows 64-word register stack. 6-bit address SP points stack top. Currently 3 items are placed in the stack: A, B and C do that content of SP is now 3 (actually 000011). 1-bit registers FULL and EMTY are set to 1 when the stack is full and empty respectively. DR is data register that holds the binary data to be written into or read out of the stack.

/* Initially, SP = 0, EMPTY = 1 (true), FULL = 0 (false) */

PUSH Operation: The PUSH operation is implemented with the following sequence of micro-operations:

SP ← SP + 1	Increment stack pointer.
M[SP] ← DR	Write item on top of the stack.
If (SP = 0) then (FULL ← 1)	Check if stack is full
EMTY ← 0	Mark the stack not empty.

b i i z a y @ g m a i l . c o m

POP Operation: The POP operation is implemented with the following sequence of micro-operations:

DR ← M[SP]	Read item from the top of stack
SP ← SP - 1	Decrement stack pointer
If (SP = 0) then (EMTY ← 1)	Check if stack is empty
FULL ← 0	Mark the stack not full

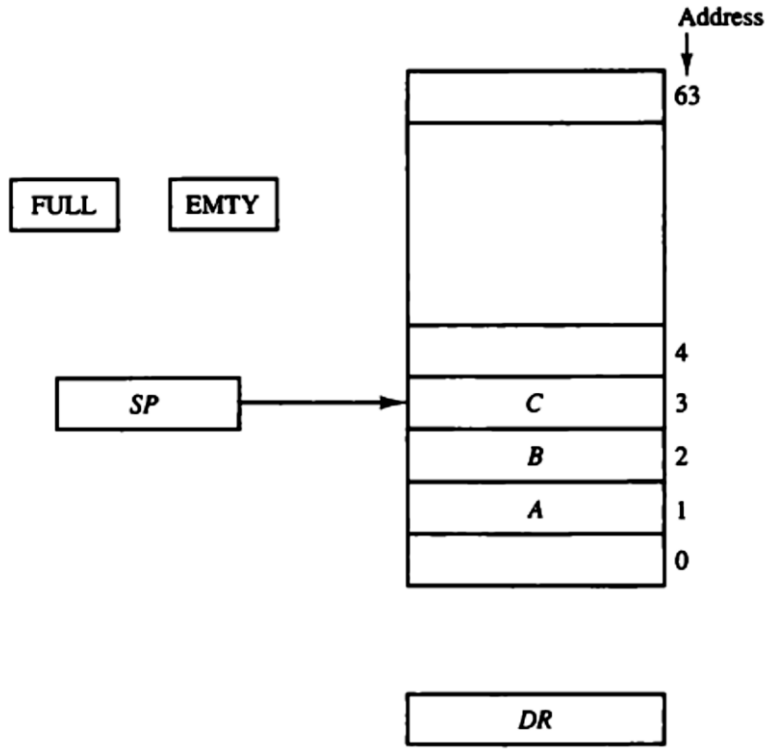


Figure: Block diagram of a 64-word stack

Memory Stack

A stack can exist as a stand-alone unit or can be implemented in a random-access memory attached to a CPU. The implementation of a stack in the CPU is done by assigning a portion of memory to a stack operation and using a processor register as a stack pointer. The program counter PC points at the address of the next instruction in the program. The address register AR points at an array of data. The stack pointer SP points at the top of the stack. The three registers are connected to a common address bus, and either one can provide an address for memory. PC is used during the fetch phase to read an instruction. AR is used during the execute phase to read an operand. SP is used to push or pop items into or from the stack. The initial value of SP is at maximum address location and the stack grows with decreasing addresses.

PC: used during fetch phase to read an instruction.
AR: used during execute phase to read an operand.
SP: used to push or pop items into or from the stack.

In the figure below, initial value of SP is 4001 and stack grows with decreasing addresses. First item is stored at 4000, second at 3999 and last address that can be used is 3000. No provisions are available for stack limit checks.

Push Operation: A new item is inserted with the push operation as follows:
SP ← SP - 1
M[SP] ← DR

POP Operation: A new item is deleted with a pop operation as follows:
DR ← M[SP]
SP ← SP + 1

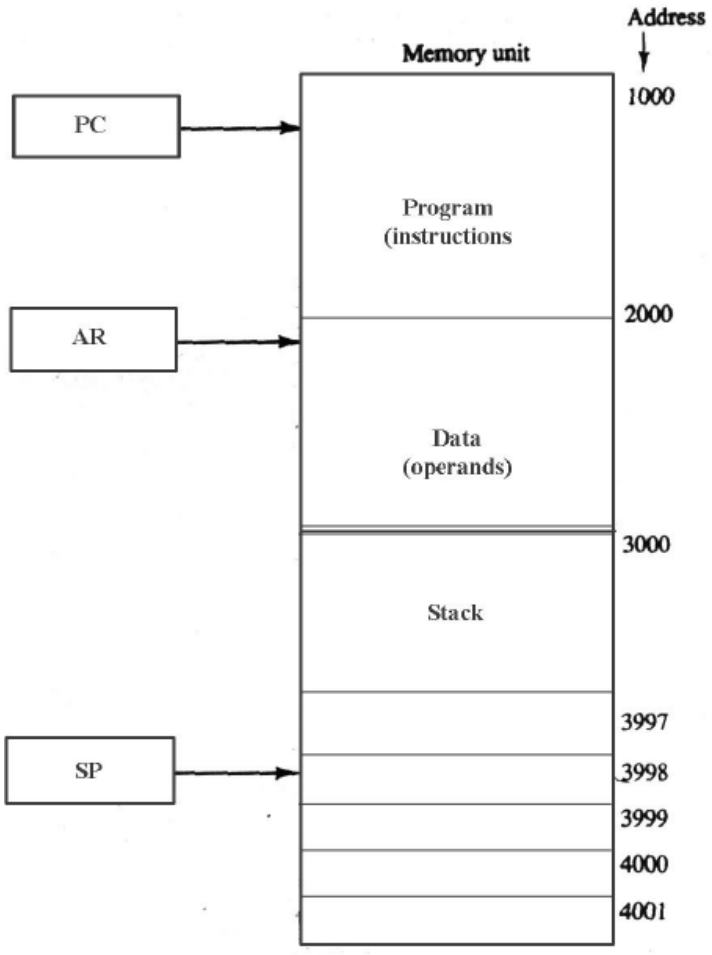


Figure: Computer memory with program, data, and stack segments

Instruction Formats

Instruction format is the function of the control unit within the CPU to interpret each instruction code. The bits of the instruction are divided into groups called **fields**. The most common fields are:

1. Operation code field – Specifies the operation to be performed.
2. Address field – Designates a memory address or a processor register
3. Mode field – specifies the way the operand or effective address is determined

A **register address** is a binary number of k bits that defines one of 2^k registers in the CPU. • The instructions may have several different lengths containing varying number of addresses. The number of address fields in the instruction format of a computer depends on the internal organization of its registers.

b i i z a y @ g m a i l . c o m

Processor Organization

Most computers fall into one of the three following processor organizations:

1. Single Register (Accumulator) organization
2. General register organization
3. Stack organization

The **Single Register (Accumulator) Organization** uses one address field. Example: ADD X, where X is the address of the operand. The ADD instruction results in operation AC ← AC + M[X], where AC is the accumulator register and M[X] denotes memory word located at address X.

```
Example:
ADD X          // AC ← AC + M[X]
LDA Y          // AC ← M[Y]
```

- Basic Computer is a good example
- Accumulator is the only general purpose register
- Uses implied accumulator register for all operations

The **General Register Organization** uses three address fields. Example: ADD R1, R2, R3, where R1, R2, and R3 are the registers. The above ADD instruction results in the operation R1 ← R2 + R3

```
Example:
ADD R1, R2, R3      // R1 ← R2 + R3
ADD R1, R2          // R1 ← R1 + R2
MOV R1, R2          // R1 ← R2
ADD R1, X           // R1 ← R1 + M[X]
```

- Used by most modern processors
- Any of the registers can be used as the source or destination for computer operations.

The **Stack Organization** would require one address field for PUSH/POP operations and none for operation-type instructions. Example: PUSH X, which pushes the word at address X on top of the stack. The instruction ADD in a stack computer consists of opcode only with no address field.

```
Example:
PUSH X          // TOS ← M[X]
ADD             // TOS = TOP(S) + TOP(S)
```

- All operations are done with the stack
- For example, an OR instruction will pop the two top elements from the stack, do a logical OR on them, and push the result on the stack.

b i i z a y @ g m a i l . c o m

Types of Instruction

Instruction format of a computer instruction usually contains 3 fields: operation code field (opcode), address field and mode field. The number of address fields in the instruction format depends on the internal organization of CPU.

On the basis of no. of address field we can categorize the instruction as below:

Three-Address Instructions

Computers with three-address instruction formats can use each address field to specify either a processor register or a memory operand.

Example: X = (A+B) * (C + D)

```
ADD R1, A, B        // R1 ← M [A] + M [B]
ADD R2, C, D        // R2 ← M[C] + M [D]
MUL X, R1, R2       // M[X] ← R1 * R2
```

It is assumed that the computer has two processor registers, R1 and R2. The symbol M[A] denotes the operand at memory address symbolized by A. The advantage of the three-address format is that it results in short programs when evaluating arithmetic expressions. The disadvantage is that the binary-coded instructions require too many bits to specify three addresses.

Two-Address Instructions

These instructions are most common in commercial computers. Here again each address field can specify either a processor register or a memory word.

Example: X = (A+B) * (C + D)

```
MOV R1, A           // R1 ← M [A]
ADD R1, B           // R1 ← R1 + M [A]
MOV R2, C           // R2 ← M[C]
ADD R2, D           // R2 ← R2 + M [D]
MUL R1, R2          // R1 ← R1 * R2
MOV X, R1           // M[X] ← R1
```

The MOV instruction moves or transfers the operands to and from memory and processor registers. The first symbol listed in an instruction is assumed to be both a source and the destination where the result of the operation is transferred.

One-Address Instructions

One-address instruction uses an implied accumulator (AC) register for all data manipulation. All operations are done between AC and memory operand.

Example: X = (A+B) * (C + D)

```
LOAD A              // AC ← M [A]
ADD B               // AC ← AC + M [B]
STORE T             // M [T] ← AC
```

```
LOAD C          // AC ← M[C]
ADD D           // AC ← AC + M [D]
MUL T           // AC ← AC * M [T]
STORE X         // M[X] ← AC
```

T is the address of a temporary memory location required for storing the intermediate result.

Zero-Address Instructions

A stack-organized computer uses this type of instructions. The name "zero-address" is given to this type of computer because of the absence of an address field in the computational instructions.

Example: X = (A+B) * (C + D)

```
PUSH A          // TOS ← A
PUSH B          // TOS ← B
ADD             // TOS ← (A + B)
PUSH C          // TOS ← C
PUSH D          // TOS ← D
ADD             // TOS ← (C + D)
MUL             // TOS ← (C + D) * (A + B)
POP X           // M[X] ← TOS
```

A stack-organized computer does not use an address held for the instructions ADD and MUL. The PUSH and POP instructions, however, need an address held to specify the operand that communicates with the stack.

Addressing Modes

The addressing mode specifies a rule for interpreting or modifying the address field of the instruction before the operand is actually referenced. The decoding step in the instruction cycle determines the operation to be performed, the addressing mode of the instruction, and the location of the operands.

Purpose of Addressing Modes:

1. To give programming versatility to the user by providing such facilities as pointers to memory, counters for loop control, indexing of data, and program relocation.
2. To reduce the number of bits in the addressing field of the instruction. The availability of the addressing modes gives the experienced assembly language programmer flexibility for writing programs that are more efficient with respect to the number of instructions and execution time.

Types of Addressing Modes:

Implied Addressing Mode (or, Stack Addressing Mode):

In this mode the operands are specified implicitly in the definition of the instruction. There is no need to specify address in the instruction. For example, the instruction "Complement Accumulator" is an implied mode instruction because the operand in the accumulator register is implied in the definition of the instruction. In fact, all register reference instructions that use an accumulator are implied-mode instructions. Example: ADD X; PUSH Y;

Immediate Addressing Mode:
In this mode the operand is specified in the instruction itself. The operand field contains the actual operand to be used in conjunction with the operation specified in the instruction. There is no need to specify address in the instruction. However, operand itself needs to be specified. Immediate-mode instructions are useful for initializing registers to a constant value. Example: ADD 5

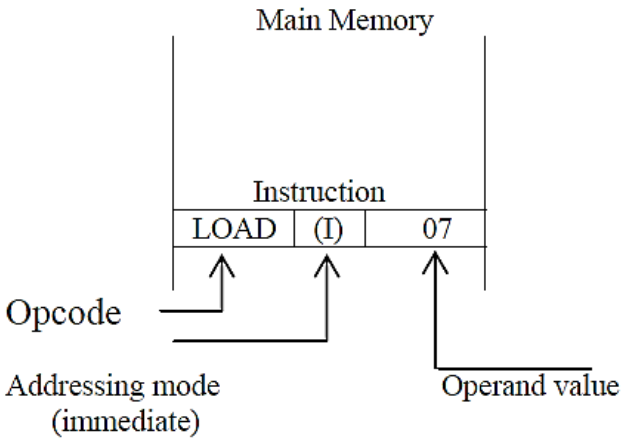


Figure: Immediate Addressing Mode

Register Addressing Mode:
In this mode the operands are in registers that reside within the CPU. Address specified in the instruction is the address of a register. Designated operand need to be in a register. The particular register is selected from a register field in the instruction. A k-bit field can specify any one of 2^k registers.

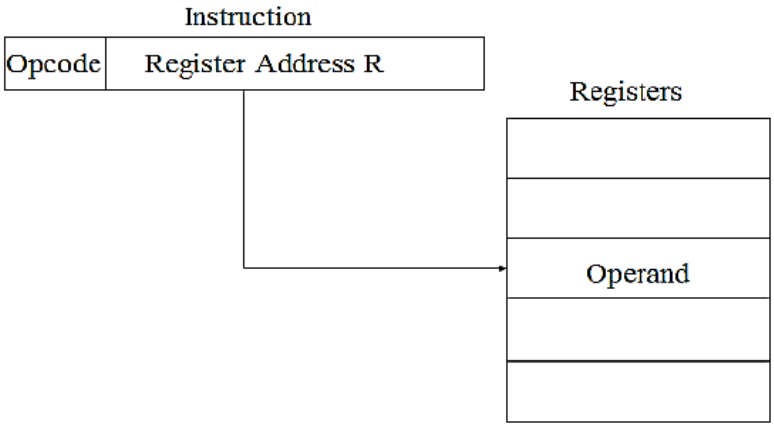


Figure: Register Addressing Mode

Register Indirect Addressing Mode:
In this mode the instruction specifies a register in the CPU whose contents give the address of the operand in memory. The advantage of a register indirect mode instruction is that the address field of the instruction uses fewer bits to select a register than would have been required to specify a memory address directly. EA (effective address) = content of R.

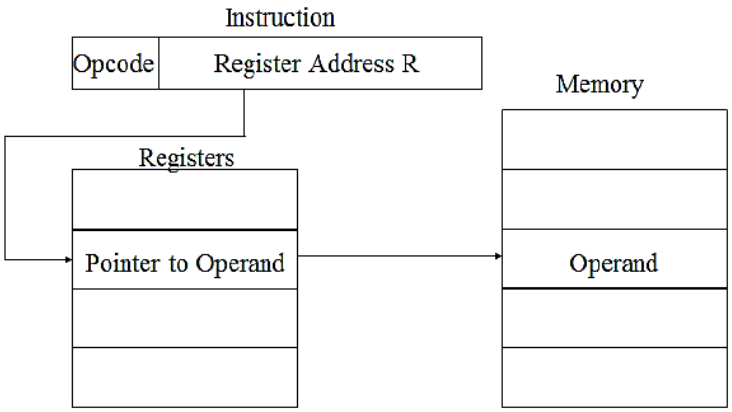


Figure: Register Indirect Addressing Mode

Autoincrement (or Autodecrement) Addressing Modes:
It is similar to register indirect mode except that the register is incremented or decremented after (or before) its value is used to access memory. When address stored in the register refers to a table of data in memory, it is necessary to increment or decrement the register after every access to the table.

Direct Addressing Mode:
In this mode the effective address is equal to the address part of the instruction. Instruction specifies the memory address which can be used directly to access the memory. The operand resides in memory and its address is given directly by the address field of the instruction. EA= IR (address).

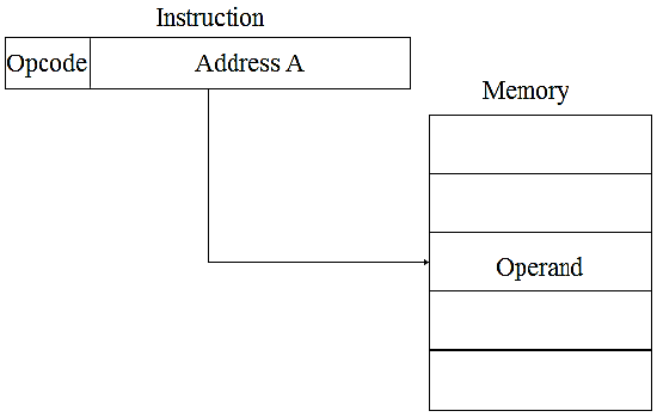


Figure: Direct Addressing Mode

In a branch-type instruction the address field specifies the actual branch address. Example: ADD A
— Adds contents of cell A to accumulator
— Look in memory at address A for operand

Indirect Addressing Mode:
In this mode the address field of the instruction gives the address where the effective address is stored in memory i.e. the address field of an instruction specifies the address of a memory location that contains the address of the operand. Control fetches the instruction from memory and uses its address part to access memory again to read the effective address. EA= M[IR (address)].

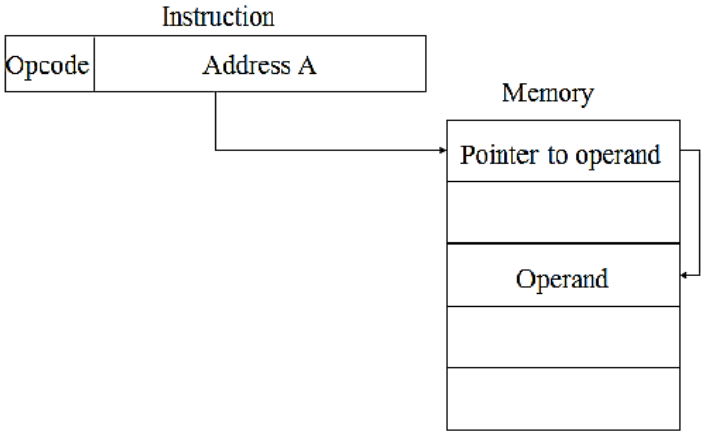


Figure: Indirect Addressing Mode

Displacement Addressing Mode:

A very powerful mode of addressing combines the capabilities of direct addressing and register indirect addressing. The effective address is EA = A + (R)

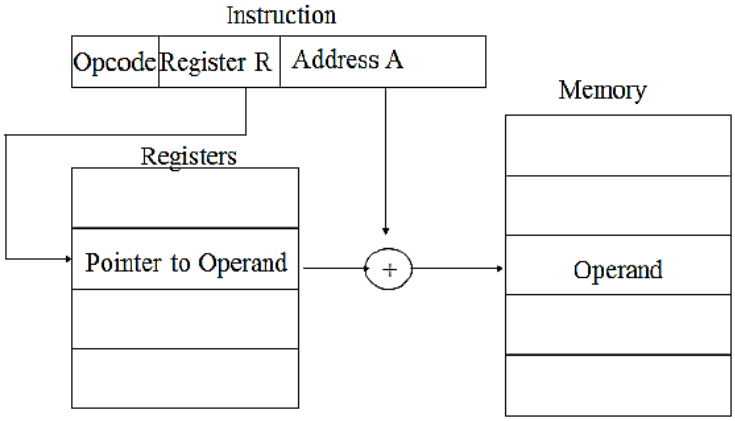


Figure: Displacement Addressing Mode

Relative Addressing Modes:

The Address field of an instruction specifies the part of the address which can be used along with a designated register (e.g. PC) to calculate the address of the operand.

- Address field of the instruction is short
- Large physical memory can be accessed with a small number of address bits

Three different relative addressing modes exists:

1. ***PC Relative Addressing Mode:*** EA = PC + IR(address)
2. ***Indexed Addressing Mode:*** EA = IX + IR(address) { IX is index register }
3. ***Base Register Addressing Mode:*** EA = BAR + IR(address)

Numerical Example of Addressing Modes

We have 2-word instruction “load to AC” occupying addresses 200 and 201. First word specifies an operation code and mode and second part specifies an address part (500). Mode field specify any one of a number of modes. For each possible mode we calculate effective address (EA) and operand that must be loaded into AC.

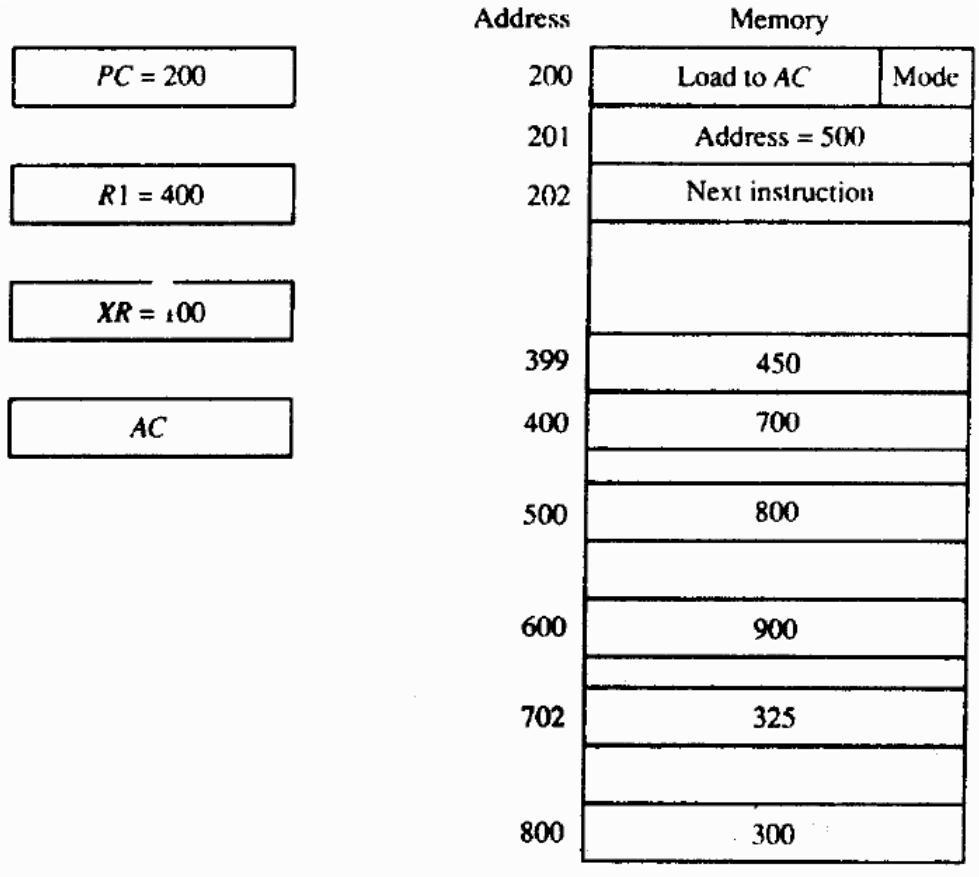


Figure: Numerical Example for Addressing Modes

Direct addressing mode: EA = address field 500 and AC contains 800 at that time.

Immediate mode: Address part is taken as the operand itself. So AC = 500. (Obviously EA = 201 in this case)

Indirect mode: EA is stored at memory address 500. So EA=800. And operand in AC is 300.

Relative mode:

- PC relative: EA = PC + 500=702 and operand is 325. (since after fetch phase PC is incremented)
- Indexed addressing: EA=XR+500=600 and operand is 900.

Register mode: Operand is in R1, AC = 400

Register indirect mode: EA = 400, so AC=700

Autoincrement mode: Same as register indirect except R1 is incremented to 401 after execution of the instruction.

Autodecrement mode: Decrements R1 to 399, so AC is now 450.

b i i z a y @ g m a i l . c o m

Following table shows the value of effective address and operand loaded into AC for 9 addressing modes.

Addressing Mode	Effective Address	Content of *AC*
Direct address	500	800
Immediate operand	201	500
Indirect address	800	300
Relative address	702	325
Indexed address	600	900
Register	—	400
Register indirect	400	700
Autoincrement	400	700
Autodecrement	399	450

Data Transfer and Manipulation

Computers give extensive set of instructions to give the user the flexibility to carryout various computational tasks. The actual operations in the instruction set are not very different from one computer to another although binary encodings and symbol name (operation) may vary. So, most computer instructions can be classified into 3 categories:

1. Data transfer instructions
2. Data manipulation instructions
3. Program control instructions

Data Transfer Instructions:

Data transfer instructions causes transfer of data from one location to another without modifying the binary information content. The most common transfers are:

- between memory and processor registers
- between processor registers and I/O
- between processor register themselves

Table below lists 8 data transfer instructions used in many computers.

Name	Mnemonic
Load	LD
Store	ST
Move	MOV
Exchange	XCH
Input	IN
Output	OUT
Push	PUSH
Pop	POP

Load: Denotes transfer from memory to registers (usually AC)
Store: Denotes transfer from a processor registers into memory
Move: Denotes transfer between registers, between memory words or memory & registers.
Exchange: Swaps information between two registers or register and a memory word.
Input & Output: Transfer data among registers and I/O terminals.
Push & Pop: Transfer data among registers and memory stack.

Instructions described above are often associated with the variety of addressing modes. Assembly language uses special character to designate the addressing mode. E.g. # sign placed before the operand to recognize the immediate mode. (Some other assembly languages modify the mnemonics symbol to denote various addressing modes, e.g. for load immediate: LDI).

Example: Consider load to accumulator instruction when used with 8 different addressing modes:

Mode	Assembly Convention	Register Transfer
Direct address	LD ADR	$AC \leftarrow M[ADR]$
Indirect address	LD @ADR	$AC \leftarrow M[M[ADR]]$
Relative address	LD $ADR	$AC \leftarrow M[PC + ADR]$
Immediate operand	LD #NBR	$AC \leftarrow NBR$
Index addressing	LD ADR(X)	$AC \leftarrow M[ADR + XR]$
Register	LD R1	$AC \leftarrow R1$
Register indirect	LD (R1)	$AC \leftarrow M[R1]$
Autoincrement	LD (R1)+	$AC \leftarrow M[R1], R1 \leftarrow R1 + 1$

Data Manipulation Instructions:

Data manipulation instructions provide computational capabilities for the computer. These are divided into 3 parts:

1. Arithmetic instructions
2. Logical and bit manipulation instructions
3. Shift instructions

Arithmetic Instructions:

The four basic arithmetic operations are addition, subtraction, multiplication and division. "Increment" operation increase the value by one and "decrement" decrease the value by one. The instruction "Add with carry" performs the addition on two operands plus the value of the carry from the previous computation. Similarly, "Subtract with borrow" instruction subtracts two words and borrow which may have resulted from a previous subtract operation. The "negate" instruction reverse the sign of an integer when represented in the signed 2's complement form.

Typical arithmetic instructions are listed below:

Name	Mnemonic
Increment	INC
Decrement	DEC
Add	ADD
Subtract	SUB
Multiply	MUL
Divide	DIV
Add with carry	ADDC
Subtract with borrow	SUBB
Negate (2's complement)	NEG

b i i z a y @ g m a i l . c o m

Logical and Bit Manipulation Instructions:

Logical instructions perform binary operations on strings of bits stored in registers and are useful for manipulating individual or group of bits representing binary coded information. Logical instructions each bit of the operand separately and treat it as a Boolean variable.

Name	Mnemonic
Clear	CLR
Complement	COM
AND	AND
OR	OR
Exclusive-OR	XOR
Clear carry	CLRC
Set carry	SETC
Complement carry	COMC
Enable interrupt	EI
Disable interrupt	DI

The AND, OR, and XOR instructions produce the corresponding logical operations on individual bits of the operands. Clear is implemented by using AND with one input as 0. XOR is used to complement selected bits by using one input as 1. Other operation can be implemented similarly.

Shift Instructions.

Instructions to shift the content of an operand are quite useful and are often provided in several variations (bit shifted at the end of word determine the variation of shift). Shift instructions may specify 3 different shifts:

1. Logical shifts
2. Arithmetic shifts
3. Rotate-type operations

The table below lists 4 types of shift instructions.

Name	Mnemonic
Logical shift right	SHR
Logical shift left	SHL
Arithmetic shift right	SHRA
Arithmetic shift left	SHLA
Rotate right	ROR
Rotate left	ROL
Rotate right through carry	RORC
Rotate left through carry	ROLC

Logical shift inserts 0 at the end position. Arithmetic shift left inserts 0 at the end (identical to logical left shift) and arithmetic shift right leave the sign bit unchanged (should preserve the sign). Rotate instructions produce a circular shift. Rotate left through carry instruction transfers carry bit to right and so is for rotate shift right.

Program Control Instructions

Instructions are always stored in successive memory locations and are executed accordingly. Program control instructions provide decision-making capabilities and change the program path. Typically, the program counter is incremented during the fetch phase to the location of the next instruction.

A program control type of instruction may change the address value in the program counter and cause the flow of control to be altered. This provides control over the flow of program execution and a capability for branching to different program segments. Some of the program control instructions are:

Name	Mnemonic
Branch	BR
Jump	JMP
Skip	SKP
Call	CALL
Return	RET
Compare (by subtraction)	CMP
Test (by ANDing)	TST

Branch (usually one address instruction) and jump instructions can be changed interchangeably. Skip is zero address instruction and may be conditional & unconditional. Call and return instructions are used in conjunction with subroutine calls.

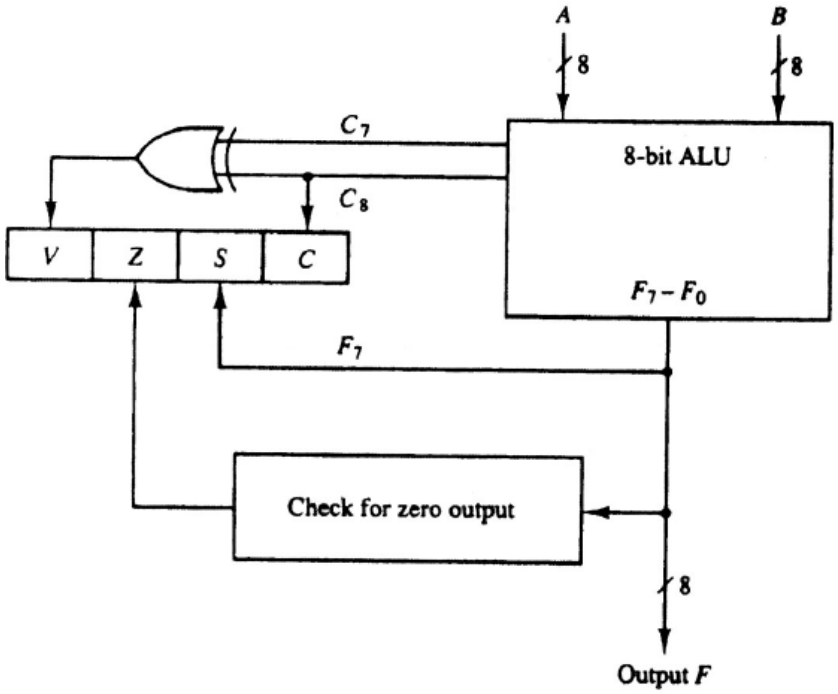


Figure: Status Register bits

Status Bit Conditions:

The four status bits are symbolized by C, S, Z, and V. The bits are set or cleared as a result of an operation performed in the ALU.

b i i z a y @ g m a i l . c o m

1. Bit C (carry) is set to 1 if the end carry C8 is 1 .It is cleared to 0 if the carry is 0.

2. Bit S (sign) is set to 1 if the highest-order bit F7 is 1. It is set to 0 if the bit is 0.

3. Bit Z (zero) is set to 1 if the output of the ALU contains all 0's. It is cleared to 0 otherwise. In other words, Z = 1 if the output is zero and Z = 0 if the output is not zero.

4. Bit V (overflow) is set to 1 if the exclusive-OR of the last two carries is equal to 1, and cleared to 0 otherwise. This is the condition for an overflow when negative numbers are in 2's complement. For the 8-bit ALU, V = 1 if the output is greater than +127 or less than -128.

Conditional Branch Instructions

The commonly used branch instructions are listed below in table.

Mnemonic	Branch condition	Tested condition
BZ	Branch if zero	$Z = 1$
BNZ	Branch if not zero	$Z = 0$
BC	Branch if carry	$C = 1$
BNC	Branch if no carry	$C = 0$
BP	Branch if plus	$S = 0$
BM	Branch if minus	$S = 1$
BV	Branch if overflow	$V = 1$
BNV	Branch if no overflow	$V = 0$
	Unsigned compare conditions ($A - B$)	
BHI	Branch if higher	$A > B$
BHE	Branch if higher or equal	$A \geq B$
BLO	Branch if lower	$A < B$
BLOE	Branch if lower or equal	$A \leq B$
BE	Branch if equal	$A = B$
BNE	Branch if not equal	$A \neq B$
	Signed compare conditions ($A - B$)	
BGT	Branch if greater than	$A > B$
BGE	Branch if greater or equal	$A \geq B$
BLT	Branch if less than	$A < B$
BLE	Branch if less or equal	$A \leq B$
BE	Branch if equal	$A = B$
BNE	Branch if not equal	$A \neq B$

b i i z a y @ g m a i l . c o m

Each mnemonic is constructed with the letter B (for branch) and an abbreviation of the condition name. When the opposite condition state is used, the letter N (for no) is inserted to define the 0 state. Thus BC is Branch on Carry, and BNC is Branch on No Carry. If the stated condition is met, the address specified by the instruction receives program control. If not, control continues with the instruction that follows.

The conditional instructions can be associated also with the jump, skip, call, or return type of program control instructions. The zero status bit is employed for testing if the result of an ALU operation is equal to zero or not. The carry bit is employed to check if there is a carry out of the most significant bit position of the ALU. It is also used in conjunction with the rotate instructions to check the bit shifted from the end position of a register into the carry position.

The sign bit reflects the state of the most significant bit of the output from the ALU. S = 0 denotes a positive sign and S = 1, a negative sign. Therefore, a branch on plus checks for a sign bit of 0 and a branch on minus checks for a sign bit of 1. It is worth noticeable that these two conditional branch instructions can be used to check the value of the most significant bit whether it represents a sign or not. The overflow bit is used in conjunction with arithmetic operations done on signed numbers in 2's complement representation.

Subroutine Call and Return

A subroutine is a self-contained sequence of instructions that does away the computational tasks. A subroutine is employed a number of times during the execution of a program. Wherever a subroutine is called to perform its function, a branch is executed to the beginning of the subroutine to start executing its set of instructions. After the subroutine has been executed, a branch is reverted to the main program. Various names are assigned to the instruction that transfers program control to a subroutine. For example, call subroutine, jump to subroutine, branch to subroutine, etc.

A call subroutine instruction comprises of an operation code with an address that specifies the beginning of the subroutine. As such two operations are included for execution of instruction (1) storage of the address of next instruction available in the program counter (the return address) in a temporary location so that the subroutine knows where to return, and (2) transfer of control to the beginning of the subroutine. The last instruction of every subroutine, referred as return from subroutine, causes transfer of returns address from the temporary location into the program counter. Consequently, program control is transferred to the instruction whose address was originally stored in the temporary location.

Program Interrupt:

Program interrupt can be described as a transfer of program control from a currently running program to another service program on a request generated externally or internally. After the service program is executed, the control returns to the original program.

The interrupt procedure is identical to a subroutine call except for three variations: (1) The interrupt is usually generated by an internal or external signal rather than from the execution of an instruction (except for software interrupt); (2) the address of the interrupt service program is determined by the hardware rather than from the address field of an instruction; and (3) an interrupt procedure usually stores all the information necessary to define the state of the CPU rather than storing only the program counter.

Types of Program Interrupt:
Three types of interrupts are:

1. External interrupts
2. Internal interrupts
3. Software interrupts

External interrupts come from input-output (I/O) devices, from a timing device, from a circuit monitoring the power supply, or from any other external source. Various examples that cause external interrupts are I/O device requesting transfer of data, I/O device finished transfer of data, elapsed time of an event, or power failure. Time-out interrupt may result from a program that is in an endless loop and thus consumes more time its time allocation. Power failure interrupt may have as its service routine a program that transfers the complete state of the CPU into a nondestructive memory in the few milliseconds before power ceases.

Internal interrupts arise when an instruction or data is used illegally or erroneously. These interrupts are also known as traps. Examples of interrupts caused by internal error conditions are register overflow, attempt to divide by zero, an invalid operation code, stack overflow, and protection violation. Occurrence of internal errors is usually a resultant of a premature termination of the instruction execution. Remedial majors to be taken are again determine by service program that processors the internal interrupts.

On the contrary, a **software interrupt** is initiated during execution of an instruction. In precise terms, software interrupt is a special call instruction that behaves like an interrupt rather than a subroutine call. It can be called to function by the programmer to initiate an interrupt procedure at any desired point in the program. Usages of software interrupt is mostly associated with a supervisor call instruction. This instruction is meant for switching from a CPU user mode to the supervisor mode. Certain operations in the computer are privileged to be assigned to the supervisor mode only, as for example, a complex input or output transfer procedure. A program written by a user must run in the user mode. When an input or output transfer is required, the request for the supervisor mode is sent by means of a supervisor call instruction.

Complex Instruction set computer (CISC)

A computer with large no of instruction is classified as complex instruction set computer (CISC). The design of an instruction set for a computer depends on not only machine language constructs, but also on the requirements imposed on the use of high-level programming languages. A compiler program translate high level languages to machine language programs.

The basic reason to a design a complex instruction set is the need to simplify the compilation and enhance the overall computer efficiency. The essential goal of a CISC architecture is to attempt to provide a single machine instruction for each statement that is written in a high-level language.

The major characteristics of CISC architecture are:
1. Large no of instructions typically form hundred to 250 instructions.
2. Same instructions that perform specialized task and are used in frequency.
3. A large variety of addressing modes typically form 2-50 different modes.
4. Variable length instruction format.
5. Instruction that manipulate operands in main memory.

No.	RISC	CISC
1.	Simple instructions taking one cycle.	Complex instructions taking multiple cycles.
2.	Very few instructions refer memory.	Most of instructions may refer memory.
3.	Instructions are executed by hardware.	Instructions are executed by microprogram.
4.	Fixed format instructions.	Variable format instructions.
5.	Few instructions.	Many instructions.
6.	Few addressing modes (3 to 5) and most instructions have register to register addressing mode.	Many addressing modes (12 to 24).
7.	Complex addressing modes are synthesized in software.	Supports complex addressing modes.
8.	Multiple register sets.	Single register set.
9.	Highly pipelined.	Not pipelined or less pipelined.
10.	Complexity is in the complier.	Complexity is in the microprogram.
11.	Clock per instruction < 1.5.	Clock per instruction < 18.

Table: RISC vs. CISC

Reduced instruction set computer (RISC)

In the early 1980's a number of computer designers recommended that computer use fewer instructions with a simple construct so they can be executed much faster within the CPU without having to use memory as often. This type of computer is classified as RISC. The concept of RISC architecture involves attempt to reduce execution time by simplifying the instruction set of the computer.

The major characteristics of RISC processor are:
1. Relatively few instruction.
2. Relatively few addressing mode.
3. Memory access limited to load and store instruction.
4. All operations done within the register of CPU.
5. Fixed length, easily decoded instruction format.
6. Single cycle instruction execution.
7. Hard-wired rather than micro program control

A typical RISC processor architecture includes register-to-register operations, with only simple load and store operations for memory access. Thus the operand is code into a processor register with a load instruction. All computational tasks are performed among the data stored in processor registers and with the help of store instructions results are transferred to the memory. This

architectural feature simplifies the instruction set and encourages the optimization of register manipulation. Almost all instructions have simple register addressing so only a few addressing modes are utilized. Other addressing modes may be included, such as immediate operands and relative mode. An advantage of RISC instruction format is that it is easy to decode.

An important feature of RISC processor is its ability to execute one instruction per clock cycle. This is done by a procedure referred to as pipelining. A load or store instruction may need two clock cycles because access to memory consumes more time than register operations. Other characteristics attributed to RISC architecture are:

8. A relatively large number of register in the processor unit.
9. Use of overlapped register windows to speed-up procedure call and return.
10. Efficient instruction pipeline.
11. Compiler support for efficient translation of high-level language programs into machine language programs.

References:

1. Andrew S. Tanenbuam, "Structured Computer Organization", PHI
2. M. Morris Mano, "Computer System Architecture", Pearson, 3rd Ed, 2004.
3. M. Morris Mano, "Digital Design", Pearson Education, Third Edition
4. M. Morris Mano, "Logic and Computer Design Fundamentals", Pearson Education, 2nd Edition
5. W. Stallings, "Computer Organization and Architecture – Designing for Performance", Prentice Hall of India, 7th Ed, 2007

Assignments:

(1) What are the differences between a branch instructions, a call subroutine instruction, and program interrupt?
(2) What are the major components of a control processing unit?
(3) What do you mean by a control word? Describe the meaning of each field in a control word.
(4) Describe the push and pop examples of a stack with suitable examples.
(5) Give five examples of external interrupts and five examples of internal interrupts.
(6) What is the difference between a software interrupt and a subroutine call?
(7) Mention the type of interrupt and explain it. (T.U. 2066)
(8) Explain the characteristics of RISC and CISC. (T.U. 2066 and 2069)
(9) Write down the following equation in three address, two address and one address instruction. Y = AB + (C X D) + E(F/G) (T.U. 2066)
(10) Explain the different types of addressing modes and compare each other. (T.U. 2066)
(11) Explain with example of Data manipulation instructions. (T.U. 2067 and 2069)
(12) What are the major differences between RISC and CISC architecture? (T.U. 2067)
(13) What do you mean by addressing modes? Differentiate between indexed addressing modes and base register addressing mode. (T.U. 2068)
(14) Explain data transfer instruction with example. (T.U. 2067, 2068, 2069 and 2070)
(15) Differentiate between RISC and CISC processor. (T.U. 2068)
(16) Explain the type of instruction format and compare each of them. (T.U. 2069)
(17) Explain logical and bit manipulation instruction with example. (T.U. 2069)
(18) What do you mean by stack organization? What are the major differences between register stack and memory stack? (T.U. 2069)

(19) What are the typical characteristics of RISC instruction set architecture? Explain. (T.U. 2070)
(20) Write down the code to evaluate.
Y = A(B/C - D) + E for one, two, three instruction format. (T.U. 2070)
(21) An instruction is stored at location 300 with its address field at location 301. The address field has the value 400. A processor register as contains the number 200. Evaluate the effective address if the addressing mode of the instruction is
(a) Direct (b) Immediate (c) Relative (d) Register indirect (e) Index with R1 as the index register
(22) Write a program to evaluate the arithmetic statement?

$$x = \frac{A + B - C \times (D \times F - E)}{G + H \times K}$$

a. Using a general register computer with three address instructions.
b. Using a general register computer with for address instructions.
c. Using an accumulator type computer with on address instructions.
d. Using a stack organized computer with zero-address operation instructions.

A Gentle Advice:

Please go through your text books and reference books for detail study!!! Thank you all.

Notes Compiled By:

Bijay Mishra
biizay.blogspot.com
9813911076 or 9841695

Unit 6 - Fixed Point Computer Arithmetic

Fixed Point Computer Arithmetic	**5 Hrs.**
Addition and Subtraction	**1.5 Hr.**
Introduction	
Addition and Subtraction with Signed Magnitude	
Hardware Implementation	
Hardware Algorithm	
Addition and Subtraction with Signed 2's Complement	
Multiplication	**2 Hrs.**
Introduction	
Hardware Implementation and Algorithm	
Booth Algorithm	
Array Multiplier	
Division Algorithm	**1.5 Hrs.**
Introduction	
Hardware Implementation	
Overflow	
Hardware Algorithm	
Restoring Method	
Comparison and non-restoring Method	

Computer Arithmetic

Data is manipulated by using the arithmetic instructions in digital computers. The Addition, subtraction, multiplication and division are the four basic arithmetic operations. To execute arithmetic operations there is a separate section called ***arithmetic processing unit*** in central processing unit. The arithmetic instructions are performed generally on binary or decimal data. Fixed-point numbers are used to represent integers or fractions. We can have signed or unsigned negative numbers. Fixed-point addition is the simplest arithmetic operation.

In order to solve the computational problems, arithmetic instructions are used in digital computers that manipulate data. These instructions perform arithmetic calculations. And these instructions perform a great activity in processing data in a digital computer. As we already stated that with the four basic arithmetic operations addition, subtraction, multiplication and division, it is possible to derive other arithmetic operations and solve scientific problems by means of numerical analysis methods.

A processor has an arithmetic processor (as a sub part of it) that executes arithmetic operations. The data type, assumed to reside in processor, registers during the execution of an arithmetic instruction. Negative numbers may be in a signed magnitude or signed complement representation. There are three ways of representing negative fixed point - binary numbers signed magnitude, signed 1's complement or signed 2'scomplement. Most computers use the signed magnitude representation for the mantissa.

Addition and Subtraction

Addition and Subtraction with Signed Magnitude Data

We designate the magnitude of the two numbers by A and B. Where the signed numbers are added or subtracted, we find that there are eight different conditions to consider, depending on the sign of the numbers and the operation performed. These conditions are listed in the first column of table below. The other columns in the table show the actual operation to be performed with the magnitude of the numbers.

Operation	Add Magnitudes	Subtract Magnitudes		
		When $A > B$	When $A < B$	When $A = B$
$(+A) + (+B)$	$+(A + B)$			
$(+A) + (-B)$		$+(A - B)$	$-(B - A)$	$+(A - B)$
$(-A) + (+B)$		$-(A - B)$	$+(B - A)$	$+(A - B)$
$(-A) + (-B)$	$-(A + B)$			
$(+A) - (+B)$		$+(A - B)$	$-(B - A)$	$+(A - B)$
$(+A) - (-B)$	$+(A + B)$			
$(-A) - (+B)$	$-(A + B)$			
$(-A) - (-B)$		$-(A - B)$	$+(B - A)$	$+(A - B)$

Table: Addition and Subtraction of Signed Magnitude

The last column is needed to present a negative zero. In other words, when two equal numbers are subtracted, the result should be +0 not -0. The algorithms for addition and subtraction are derived from the table and can be stated as follows (the words parentheses should be used for the subtraction algorithm).

Algorithm

When the signs of A and B are same, add the two magnitudes and attach the sign of result is that of A. When the signs of A and B are not same, compare the magnitudes and subtract the smaller number from the larger. Choose the sign of the result to be the same as A, if A > B or the complement of the sign of A if A < B. If the two magnitudes are equal, subtract B from A and make the sign of the result will be positive.

Hardware Implementation

To implement the two arithmetic operations with hardware, it is first necessary that the two numbers be stored in registers. Let A and B be two registers that keeps the magnitudes of the numbers, and As and Bs be two flip-flops that hold the corresponding signs. The result of the operation may be transferred into A and As. Thus an accumulator register is formed by A and As.

Consider now the hardware implementation of the algorithm above. First, we need a parallel-adder to perform the microoperation A + B. Second, a comparator circuit is needed to establish whether A > B, A = B, or A > B. Third, we need two parallel-subtractor circuits to perform the

b i i z a y @ g m a i l . c o m

microoperations A - B and B - A. The sign relationship can be obtained from an exclusive OR gate with As and Bs as inputs.

Hence we require a magnitude comparator, an adder, and two subtractors. But there is a different procedure that requires less equipment. First, we know that subtraction can be accomplished by means of complement and add. Second, the result of a comparison can be determined from the end carry after subtraction. Careful investigation of the alternatives suggests that the use of 2's complement for subtraction and comparison is an efficient procedure and we require only an adder and a complementer.

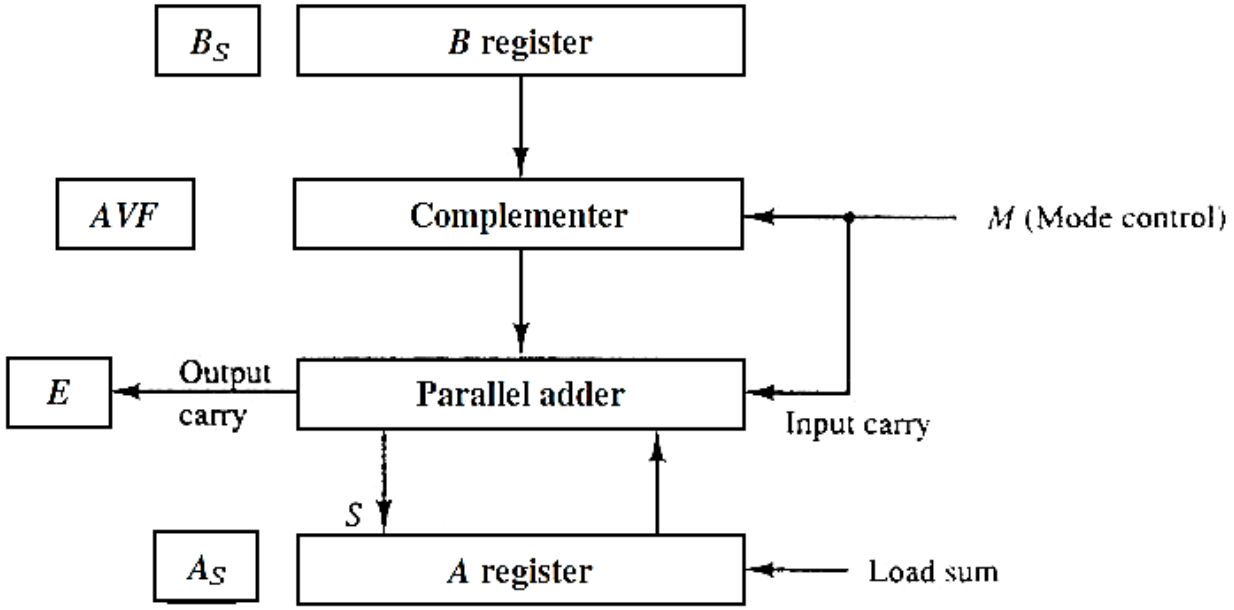


Figure: Hardware for signed magnitude addition and subtraction

The Figure above shows a block diagram of the hardware for implementing the addition and subtraction operations. It has registers A and B and flip-flop As and Bs are used for sign. We perform subtraction by adding A to the 2's complement of B. The output carry is loaded into flip-flop E, where it can be checked to discover the relative magnitudes of the two numbers. The add-overflow flip-flop AVF contains the overflow bit for addition of A and B. The A register provides other micro-operations that may be needed when we specify the sequence of steps in the algorithm.

The operation A+ B is done through the parallel adder. The S (sum) output of the adder becomes to the input of the A register. The complementer gives an output of B or the complement of B depending or the state of the mode control M. The complement consists of XOR gates and the parallel adder consists of full adder circuits.

The M signal is also applied to the input carry of the adder. When M = 0, the output of B is transferred to the adder, the input carry is 0, and the output of the adder is equal to the sum A + B. When M = 1, the 1's complement of B is applied to the adder, the input carry is 1, and output S = A + B + 1. This is equal to A plus the 2's complement of B, which is equivalent to the subtraction A – B.

b i i z a y @ g m a i l . c o m

Hardware Algorithm

We compare the signs of As and Bs by an exclusive-OR gate. If we get 0 output, the signs are identical and if it is 1, the signs are different. For an add operation, identical signs dictate that the magnitudes be added. For a subtract operation, different signs tells that the magnitudes be added. The magnitudes are added with a microoperation EA = A + B, where EA is register that consists of E and A. The carry in E after the addition constitutes an overflow if it is equal to 1. The value of E is transferred into the add-overflow flipflop AVF.

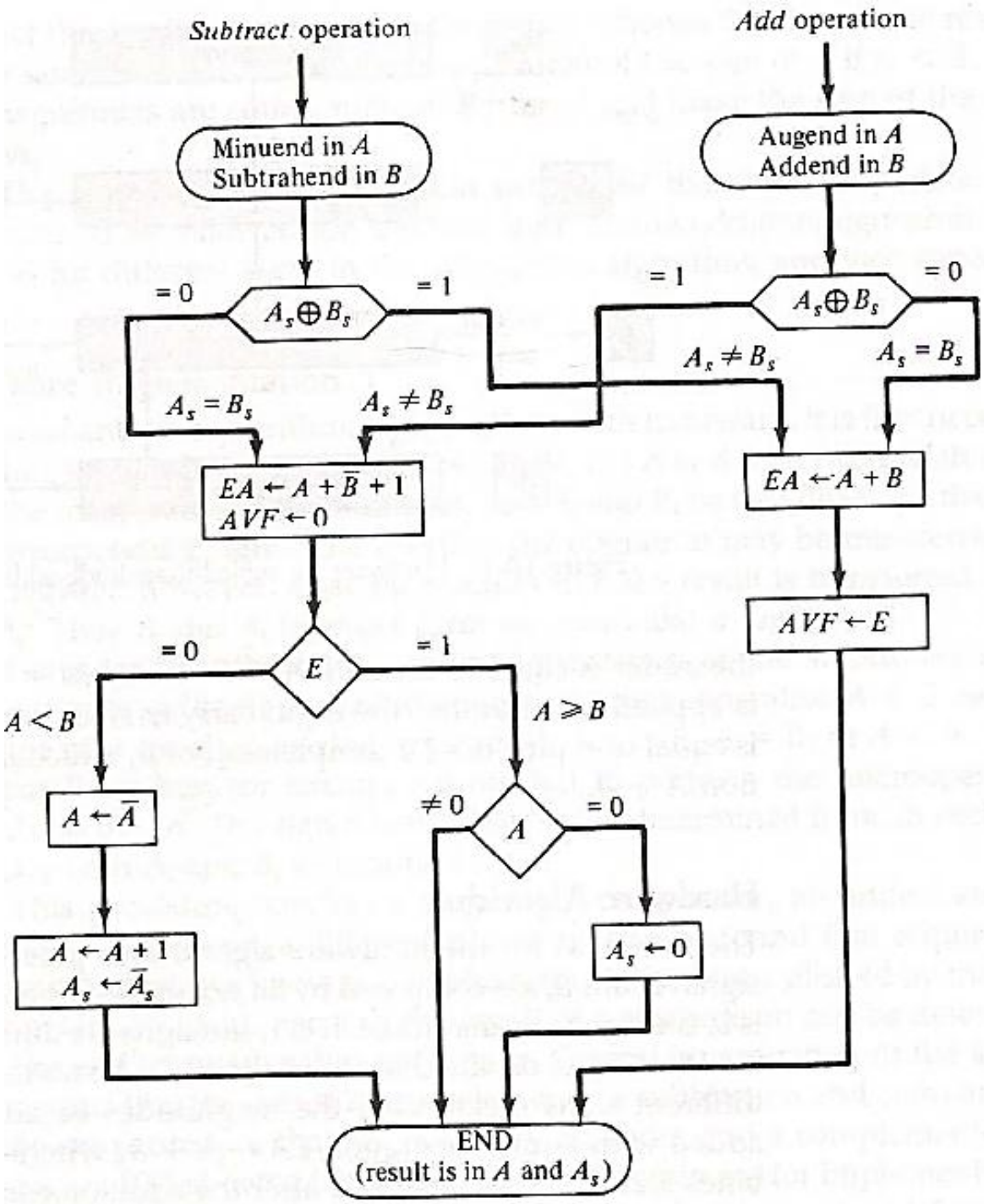


Figure: Flowchart for Add and Subtract Operations

b i i z a y @ g m a i l . c o m

We subtract the two magnitudes if the signs are different for an add operation or identical for a subtract operation. The magnitudes are subtracted by adding A to the 2's complement of B. No overflow can occur if the numbers are subtracted so AVF is cleared to 0. A 1 in E tells that A > B. and the number in A contains the correct result. If A contains a zero, the sign As, must be made positive to avoid a negative zero.

A 0 in E indicates that A < B. For this case it is necessary to take the 2's complement of the value in A. This operation can be done with one microoperation A + 1. However, we assume that the A register has circuits for microoperations complement and increment, so the 2's complement is obtained from these two microoperations.

In other paths of the flowchart, the sign of the result is the same as the sign of A, so no change in As is required. However, when A < B, the sign of the result is the complement of the original sign of A. It is then necessary to complement As, to obtain the correct sign. The final is found in register A and its sign in As. The value in AVF provides an overflow indication. The final value of E is immaterial.

Addition and Subtraction with Signed-2's Complement Data

The signed-2's complement representation of numbers together with arithmetic algorithms for addition and subtraction. They are summarized here. The leftmost bit of a binary number represents the sign: 0 to denote to denote positive and 1 to denote negative. If the sign bit is 1, then we represent number in 2's complement form. Thus + 33 is represented as 00100000 and -33 as 11011110. Note that 11011110 is the 2's complement of 00100000, and vice versa.

The addition of two numbers in signed 2's complement form by adding the numbers with the sign bits treated the same as the other bits of the number. We discard the carry of the sign-bit position. The subtraction consists of first taking the 2's complement of the subtrahend and then adding it to the minuend.

When we add two numbers of n digits then the sum occupies n + 1 digits, in this case an overflow occurs.

The effect of an overflow on the sum of two signed 2's complement numbers has been discussed already. We can detect an overflow by inspecting the last two carries of the addition. When the two carries are applied to an exclusive-OR gate, the overflow is detected when the output of the gate is equal to 1.

The register configuration for the hardware implementation is given in the figure below. We call the A register AC (accumulator) and the B register BR. The two sign bits are added or subtracted together with the other bits in the complementer and parallel adder. The overflow flip-flop V is set to 1 if there is an overflow. The output carry in this case is discarded.

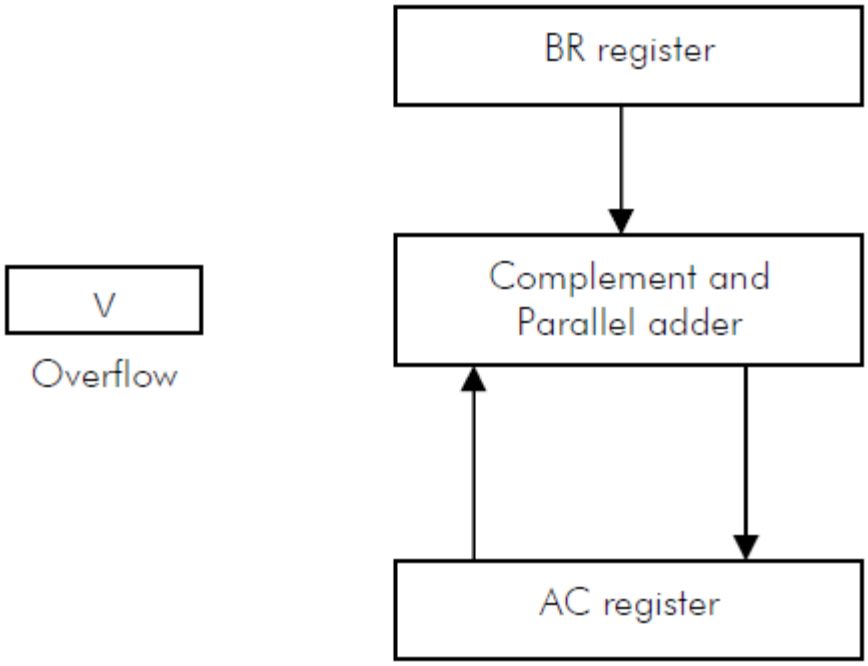


Figure: Hardware for signed-2's complement addition and subtraction

The algorithm for adding and subtracting two binary numbers in signed 2's complement representation is shown in the flowchart of the figure below.

We obtain the sum by adding the contents of AC and BR (including their sign bits). The overflow bit V is set to 1 if the exclusive OR of the last two carries is 1, otherwise it is cleared. The subtraction operation is performed by adding the content of AC to the 2's complement of BR. Taking the 2's complement of BR has the effect of changing a positive number to negative, and vice versa. We have to check an overflow during this operation because the two numbers added may have the same sign. It should be noted that if an overflow occurs, there is an erroneous result in the AC register.

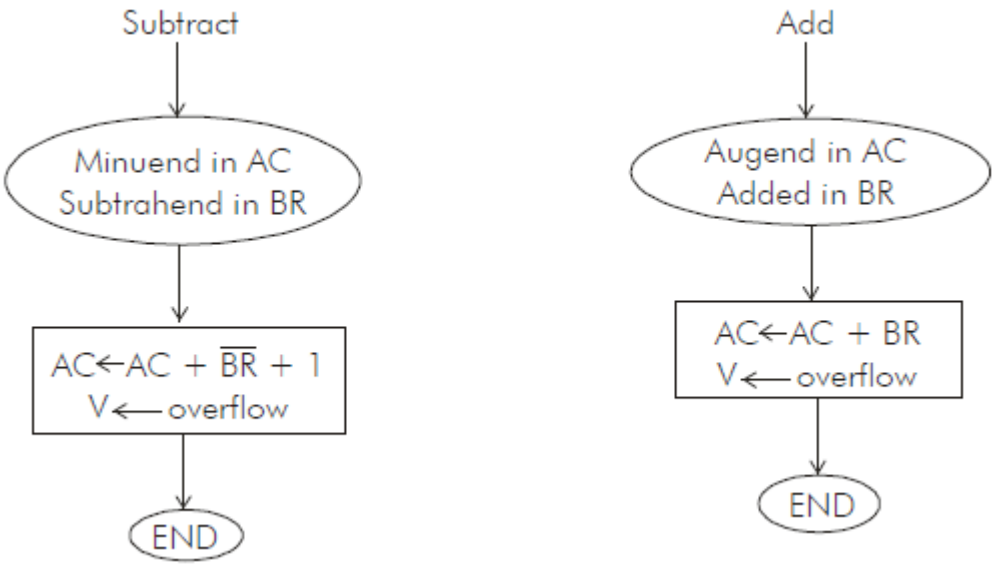


Figure: Flowchart for signed-2's complement addition and subtraction

b i i z a y @ g m a i l . c o m

If we compare this algorithm with its signed-magnitude part, we note that it is much simpler to add and subtract numbers if we keep negative numbers in signed 2's complement representation. Therefore most computers adopt this representation over the more familiar signed-magnitude.

Multiplication Algorithm

Multiplication of two fixed-point binary numbers in signed magnitude representation is done with paper and pencil by a process of successive shift and add operations. This process is best illustrated with a numerical example:

```
    23        10111   Multiplicand
    19   x    10011   Multiplier
  -------
           10111
          10111
         00000
        00000
       10111
  -------------------
  437    110110101 Product
```

This process looks at successive bits of the multiplier, least significant bit first. If the multiplier bit is 1, the multiplicand is copied as it is; otherwise, we copy zeros. Now we shift numbers copied down one position to the left from the previous numbers. Finally, the numbers are added and their sum produces the product.

Hardware Implementation for Signed Magnitude Data

When multiplication is implemented in a digital computer, we change the process slightly. Here, instead of providing registers to store and add simultaneously as many binary numbers as there are bits in the multiplier, it is convenient to provide an adder for the summation of only two binary numbers, and successively accumulate the partial products in a register. Second, instead of shifting the multiplicand to left, the partial product is shifted to the right, which results in leaving the partial product and the multiplicand in the required relative positions. Now, when the corresponding bit of the multiplier is 0, there is no need to add all zeros to the partial product since it will not alter its value.

The hardware for multiplication consists of the equipment given in figure below. The multiplier is stored in the register and its sign in Q_s. The sequence counter SC is initially set bits in the multiplier. After forming each partial product the counter is decremented. When the content of the counter reaches zero, the product is complete and we stop the process.

b i i z a y @ g m a i l . c o m

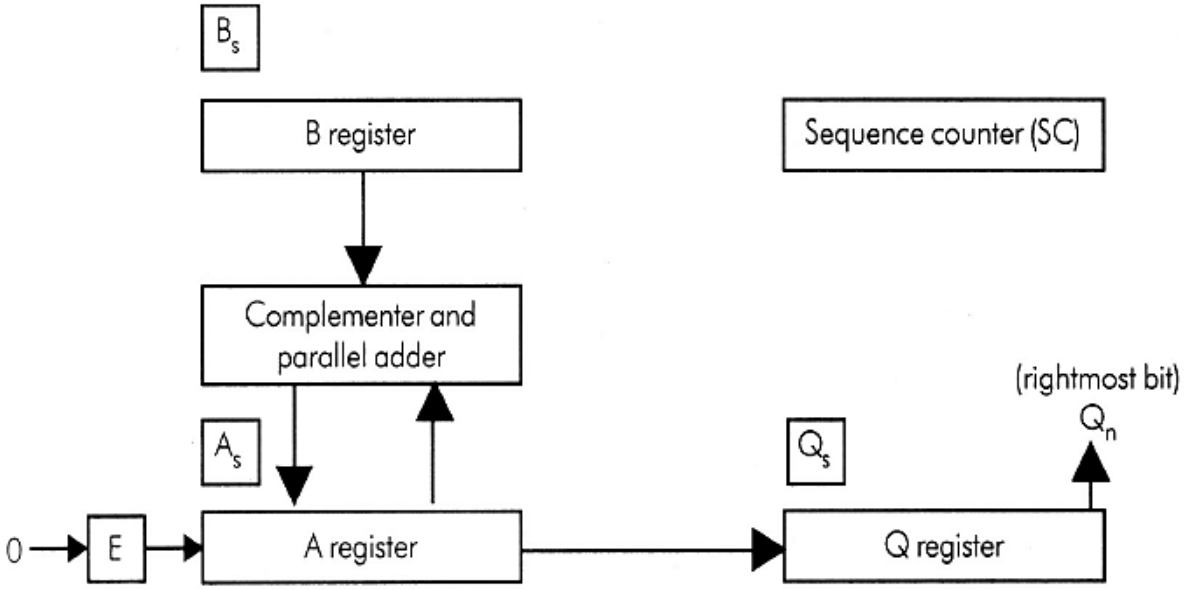


Figure: Hardware for multiply operation

Hardware Algorithm

Figure below is a flowchart of the hardware multiplication algorithm. In the beginning, the multiplicand is in B and the multiplier in Q. Their corresponding signs are in B_s and Q_s respectively.

We compare the signs of both A and Q and set to corresponding sign of the product since a double-length product will be stored in registers A and Q.

Registers A and E are cleared and the sequence counter SC is set to the number of bits of the multiplier. Since an operand must be stored with its sign, one bit of the word will be occupied by the sign and the magnitude will consist of n-1 bits.

Now, the low order bit of the multiplier in Q_n is tested. If it is 1, the multiplicand (B) is added to present partial product (A), 0 otherwise. Register EAQ is then shifted once to the right to form the new partial product. The sequence counter is decremented by 1 and its new value checked. If it is not equal to zero, the process is repeated and a new partial product is formed. When SC = 0 we stops the process.

b i i z a y @ g m a i l . c o m

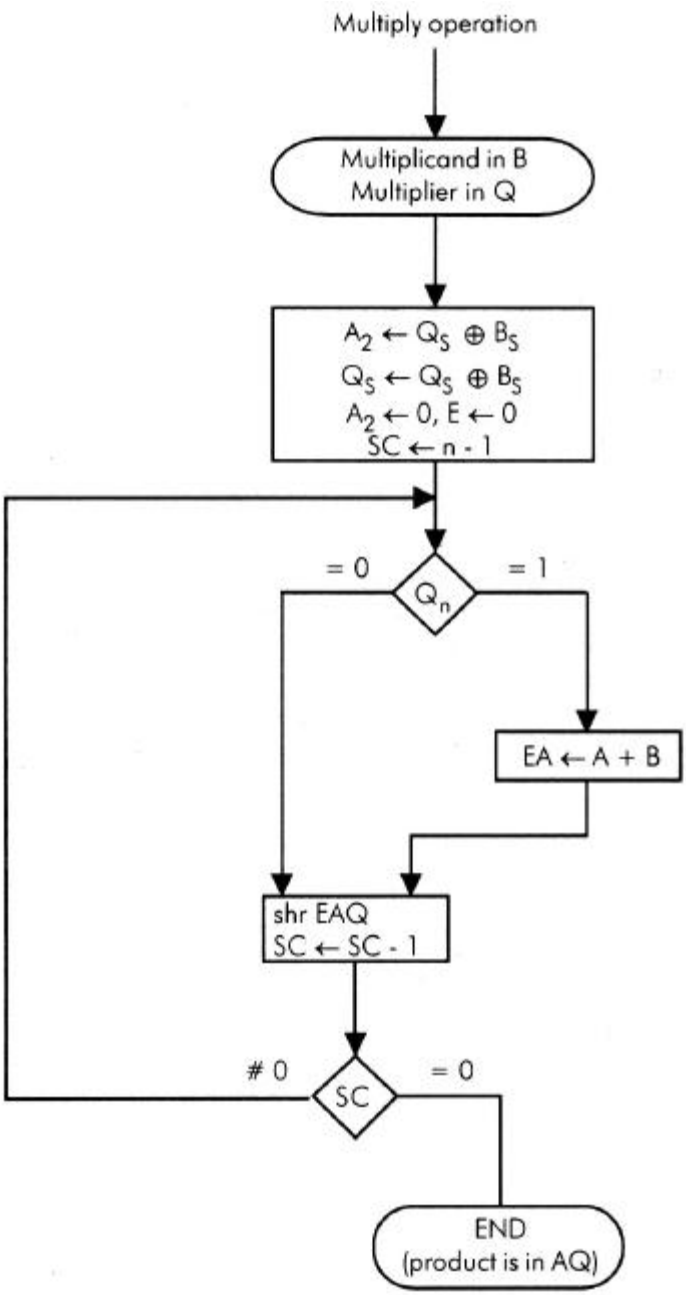


Figure: Flowchart for Multiply operation

Booth Multiplication Algorithm

If the numbers are represented in signed 2's complement then we can multiply them by using Booth algorithm. In fact the strings of 0's in the multiplier need no addition but just shifting, and a string of 1's in the multiplier from bit weight 2^k to weight 2^m can be treated as $2^{k+1} - 2^m$. For example, the binary number 001111 (+15) has a string of 1's from 2^3 to 2^0(k = 3, m = 0).

The number can be represented as $2^{k+1} – 2^m = 2^4 - 2^0 = 16 - 1 = 15$. Therefore, the multiplication M x 14, where M is the multiplicand and 14 the multiplier may be computed as M x 2^4 - M x 2^1. That is, the product can be obtained by shifting the binary multiplicand M four times to the left and subtracting M shifted left once.

b i i z a y @ g m a i l . c o m

Multiplicand $B = 10111$	E	A	Q	SC
Multiplier in Q	0	00000	10011	101
$Q_n = 1$; add B		10111		
First partial product	0	10111		
Shift right EAQ	0	01011	11001	100
$Q_n = 1$; add B		10111		
Second partial product	1	00010		
Shift right EAQ	0	10001	01100	011
$Q_n = 0$; shift right EAQ	0	01000	10110	010
$Q_n = 0$; shift right EAQ	0	00100	01011	001
$Q_n = 1$; add B		10111		
Fifth partial product	0	11011		
Shift right EAQ	0	01101	10101	000
Final product in $AQ = 0110110101$				

Table: Numerical Example for Binary Multiplier

Booth algorithm needs examination of the multiplier bits and shifting of the partial product. Prior to the shifting, the multiplicand added to the partial product, subtracted from the partial product, or left unchanged by the following rules:

1. The multiplicand is subtracted from the partial product when we get the first least significant 1 in a string of 1's in the multiplier.
2. The multiplicand is added to the partial product when we get the first Q (provided that there was a previous 1) in a string of 0's in the multiplier.
3. The partial product does not change when the multiplier bit is the same as the previous multiplier bit.

The algorithm applies to both positive and negative multipliers in 2's complement representation. This is because a negative multiplier ends with a string of 1's and the last operation will be a subtraction of the appropriate weight.

For example, a multiplier equal to -14 is represented in 2's complement as 110010 and is treated as $-2^4 + 2^2 - 2^1 = -14$.

The hardware implementation of Booth algorithm requires the register configuration shown in Figure below. Q_n represents the least significant bit of the multiplier in register QR. An extra flip-flop Q_{n+1} is appended to QR to provide a double bit inspection of the multiplier.

b i i z a y @ g m a i l . c o m

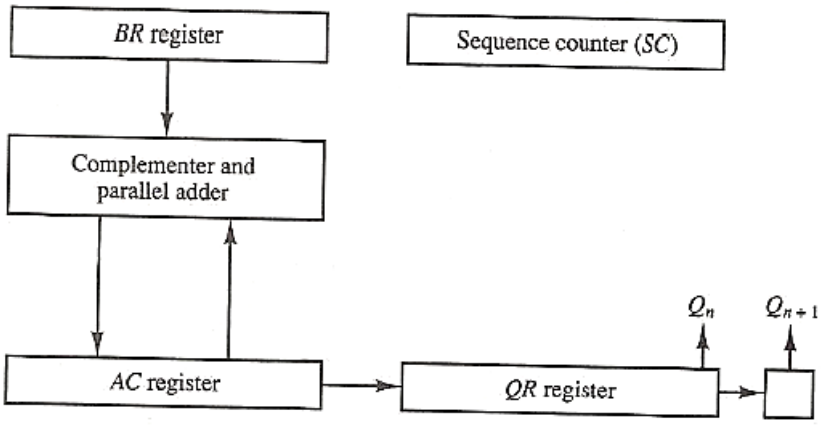


Figure: Hardware for Booth Algorithm

The flowchart for Booth algorithm is shown in Figure below. AC and the appended bit Q_{n+1} are initially set to 0 and the sequence counter SC is set to a number n equal to the number of bits in the multiplier.

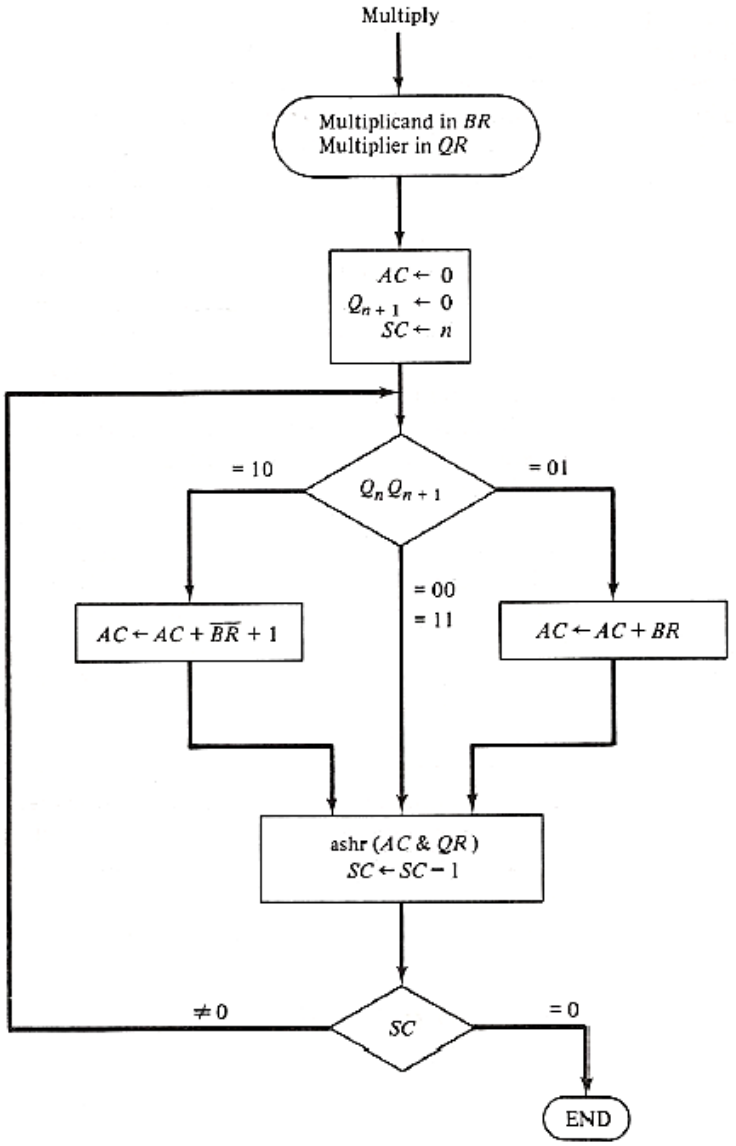


Figure: Flowchart for Booth Algorithm

b i i z a y @ g m a i l . c o m

The two bits of the multiplier in Q_n and Q_{n+1} are inspected.

1. If the two bits are 10, it means that the first 1 in a string of 1's has been encountered. This needs a subtraction of the multiplicand from the partial product in AC.

2. If the two bits are equal to 01. It means that the first 0 in a string of 0's has been encountered. This needs the addition of the multiplicand to the partial product in AC.

3. When the two bits are equal, the partial product does not change. An overflow cannot occur because the addition and subtraction of the multiplicand follow each other. Hence, the two numbers that are added always have opposite sign, a condition that excludes an overflow.

Next step is to shift right the partial product and the multiplier (including bit Q_{n+1}). This is an arithmetic shift right (ashr) operation which shifts AC and QR to the right and leaves the sign bit in AC same. The sequence counter decrements and the computational loop is repeated n times.

A Numerical Example of Booth algorithm

A numerical example of Booth algorithm is given for n = 5. It gives the multiplication of (-9) x (-13) = +117. Note that the multiplier in QR is negative and that the multiplicand in BR is also negative. The 10-bit product appears in AC. The final value of Q_{n+1} is the original sign bit of the multiplier and should not be taken as part of the product.

$Q_n Q_{n+1}$	$BR = 10111$ $\overline{BR} + 1 = 01001$	AC	QR	Q_{n+1}	SC
	Initial	00000	10011	0	101
1 0	Subtract BR	01001			
		01001			
	ashr	00100	11001	1	100
1 1	ashr	00010	01100	1	011
0 1	Add BR	10111			
		11001			
	ashr	11100	10110	0	010
0 0	ashr	11110	01011	0	001
1 0	Subtract BR	01001			
		00111			
	ashr	00011	10101	1	000

Table: Example of Multiplication with Booth Algorithm

b i i z a y @ g m a i l . c o m

Array Multiplier

To check the bits of the multiplier one at a time and forming partial products is a sequential operation requiring a sequence of add and shift micro-operations. The multiplication of two binary numbers can be done with one micro-operation by using combinational circuit that forms the product bits all at once.

This is a fast way since all it takes is the time for the signals to propagate through the gates that form the multiplication array. However, an array multiplier requires a large number of gates, and so it is not an economical unit for the development of ICs.

Now we see how an array multiplier is implemented with a combinational circuit. Consider the multiplication of two 2-bit numbers as shown in the figure below.

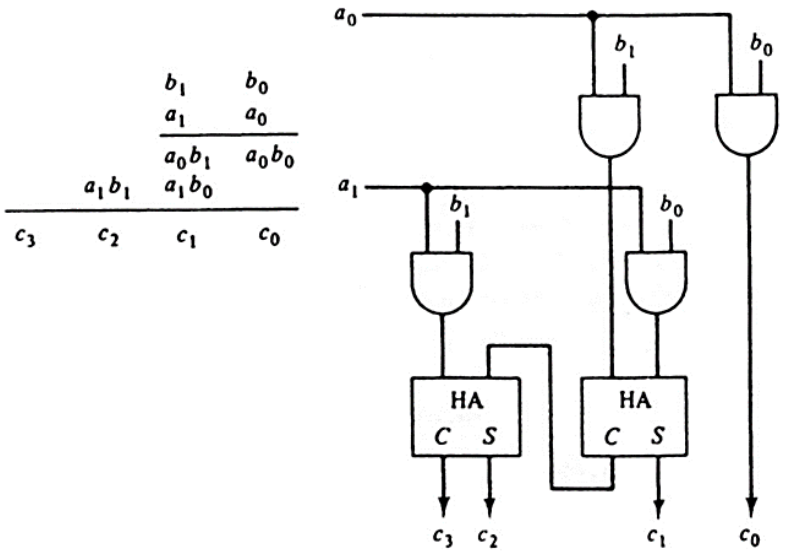


Figure: 2 bit by 2 bit array multiplier

The multiplicand bits are b_1 and b_0, the multiplier bits are a_1 and a_0, and the product is c_3 c_2 c_1 c_0. The first partial product is obtained by multiplying a_0 by b_1b_0. The multiplication of two bits gives a 1 if both bits are 1; otherwise, it produces a 0. This is identical to an AND operation and can we implement it with an AND gate.

As shown in the diagram, the first partial product is formed by means of two AND gates. The second partial product is formed by multiplying a_1 by b_1b_0 and is shifted one position to the left. The two partial products are added with two half-adder (HA) circuits. Usually, there are more bits in the partial products and it will be necessary to use full-adders to produce the sum.

Note that the least significant bit of the product does not have to go through an adder since it is formed by the output of the first AND gate.

A combinational circuit binary multiplier with more bits can be constructed in a similar fashion. A bit of the multiplier is ANDed with each bit of the multiplicand in as many levels as there are bits in the multiplier.

b i i z a y @ g m a i l . c o m

The binary output in each level AND gates is added in parallel with the partial product of the previous level to form a new partial product. The last level produces the product. For j multiplier bits and k multiplicand bits we need (j*k) AND gates and (j - 1) k bits adders to produce a product of j + k bits.

As a second example, consider a multiplier circuit that multiplies a binary number of four bits with a number of three bits. Let the multiplicand be represented by $b_3b_2b_1b_0$ and the multiplier by $a_2a_1a_0$. Since k=4 and j=3, we need 12 AND gates and two 4-bit adders to produce a product of seven bits. The logic diagram of the multiplier is shown in Figure below:

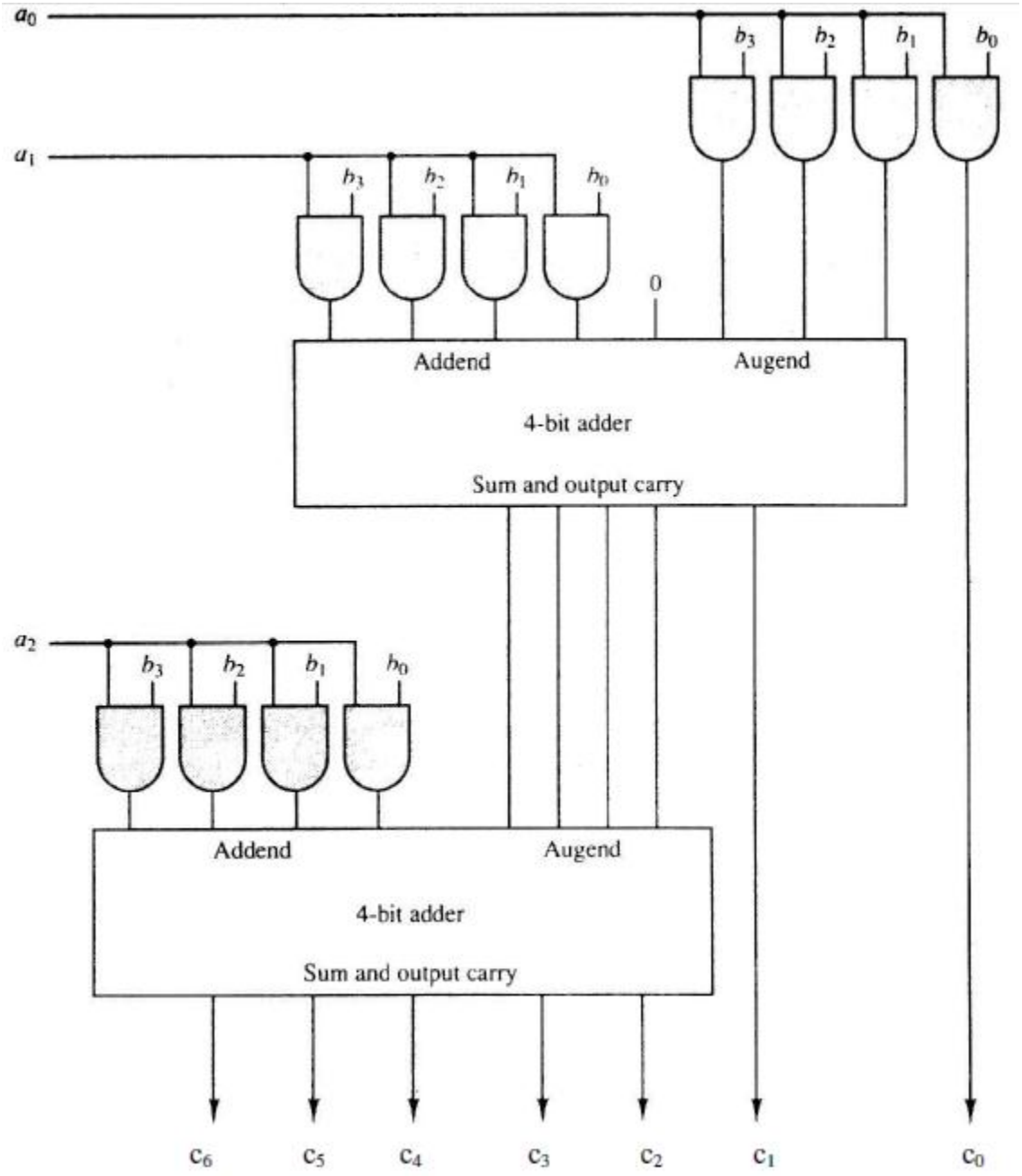


Figure: 4 bit by 3 bit array multiplier

b i i z a y @ g m a i l . c o m

Division Algorithms

Division of two fixed-point binary numbers in signed magnitude representation is performed with paper and pencil by a process of successive compare, shift and subtract operations.

Binary division is much simpler than decimal division because here the quotient digits are either 0 or 1 and there is no need to estimate how many times the dividend or partial remainder fits into the divisor.

The division process is described in the figure below:

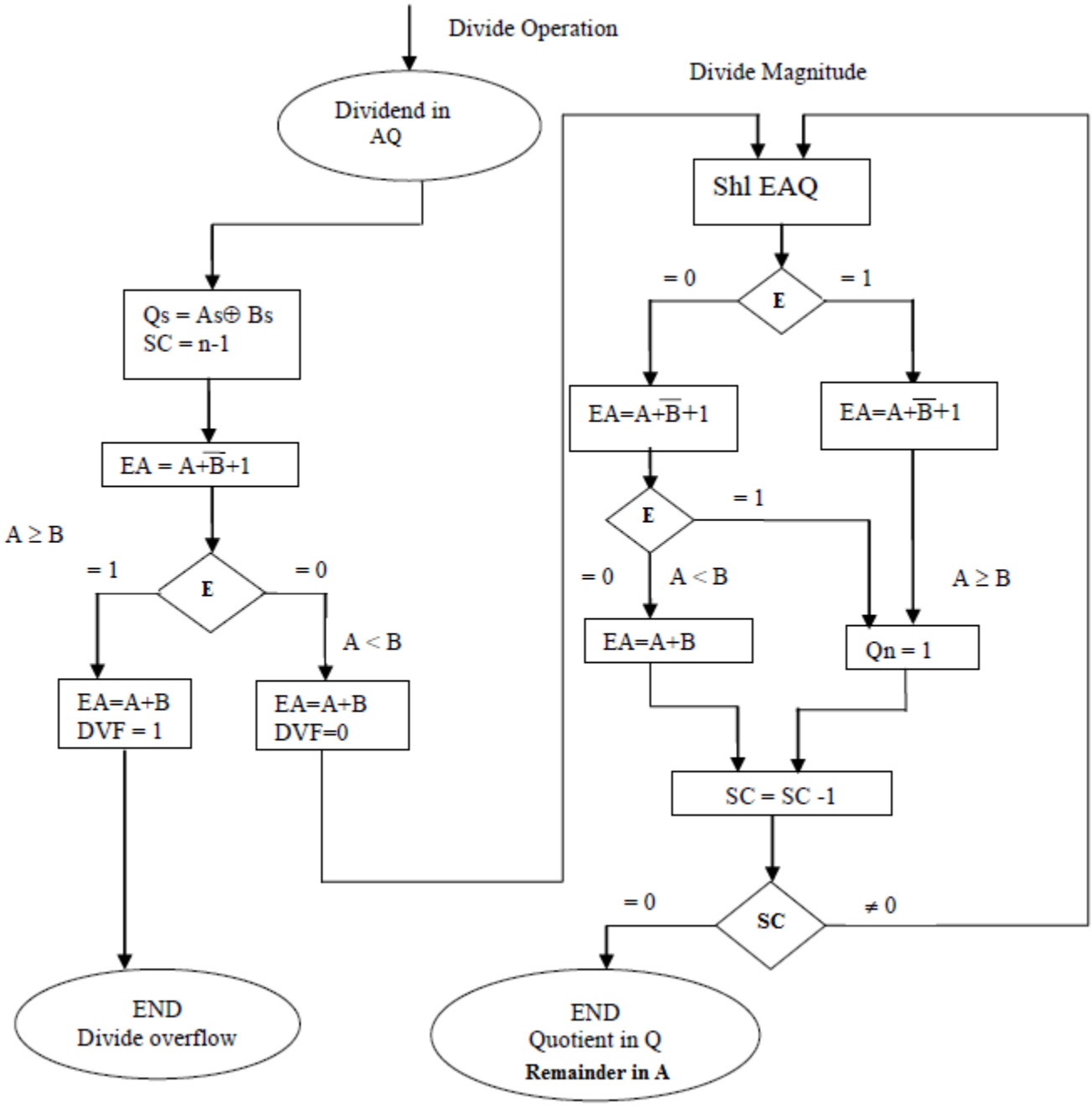


Figure: Flowchart for Division Algorithm for Signed Magnitude Data

b i i z a y @ g m a i l . c o m

Division: B = 10001	11010) 0111000000 01110 011100 - 10001 - 010110 - -10001 - - 001010 - - - 010100 - - - -10001 - - - -000110 - - - - -00110	Quotient = Q Dividend = A 5 bits of A < B, quotient has 5 bits 6 bits of A ≥ B Shift right B and subtract; enter 1 in Q 7 bits of remainder ≥ B Shift right B and subtract; enter 1 in Q Remainder < B; enter 0 in Q; shift right B Remainder ≥ B Shift right B and subtract; enter 1 in Q Remainder < B; enter 0 in Q Final remainder

Figure: Example of Binary Division

The divisor B has five bits and the dividend A has ten. The divisor is compared with the five most significant bits of the dividend. Since the 5-bit number is smaller than B, we again repeat the same process.

Now the 6-bit number is greater than B, so we place a 1 for the quotient bit in the sixth position above the dividend. Now we shift the divisor once to the right and subtract it from the dividend. The difference is known as a partial remainder because the division could have stopped here to obtain a quotient of 1 and a remainder equal to the partial remainder.

Comparing a partial remainder with the divisor continues the process. If the partial remainder is greater than or equal to the divisor, the quotient bit is equal to 1. The divisor is then shifted right and subtracted from the partial remainder.

If the partial remainder is smaller than the divisor, the quotient bit is 0 and no subtraction is needed. The divisor is shifted once to the right in any case. Obviously the result gives both a quotient and a remainder.

Floating Point Representation

The floating point representation of the number has two parts. The first part represents a signed fixed point numbers called ***mantissa***. The second part designates the position of the decimal (or binary) point and is called ***exponent***.

For example the decimal no +6132.789 is represented in floating point with fraction and exponent as follows:

Fraction	***Exponent***
+0.6132789	+04

This representation is equivalent to the scientific notation: $+0.6132789 \times 10^{+4}$

b i i z a y @ g m a i l . c o m

The floating point is always interpreted to represent a number in the following form $\mathbf{m \times r^e}$. Only the mantissa and the exponent e are physically represented in the register (including their sign) .The radix r and the radix point position of the mantissa are always assumed.

A floating point binary no is represented in similar manner except that it uses base 2 for the exponent. For example the binary no +1001.11 is represented with 8 bit fraction and 0 bit exponent as follows:

0.1001110×2^{100}

Fraction	***Exponent***
01001110	000100

The fraction has zero in the leftmost position to denote positive. The floating point number is equivalent to $m \times 2^e = +(0.1001110)_2 \times 2^{+4}$

Floating Point Arithmetic
The basic operation for floating point arithmetic are:

Floating point number	Arithmetic Operations
$X = X_s \times B^{XE}$	$X+Y = (X_s \times B^{XE-YE} + Y_s) \times B^{YE}$
$Y = Y_s \times B^{YE}$	$X-Y = (X_s \times B^{XE-YE} - Y_s) \times B^{YE}$
	$X*Y = (X_s \times Y_s) \times B^{XE+YE}$
	$X/Y = (X_s / Y_s) \times B^{XE-YE}$

For addition and subtraction it is necessary to ensure that both operands have same exponent value. This may require shifting the radix point on one of the operands to achieve alignment. Multiplication and division are both more straight forwards.

The exponent may be represented in biased exponent in this representation, the sign bit is remove from being separate entity. The bias is a positive no i.e. added to the each exponent as floating point no is formed so that internally all exponents are positive. Consider an exponent that ranges form - 50 to 49. It is represented in registers as positive nos. in the range of 0 to 99.

The register organization for floating point operation is shown in figure below.

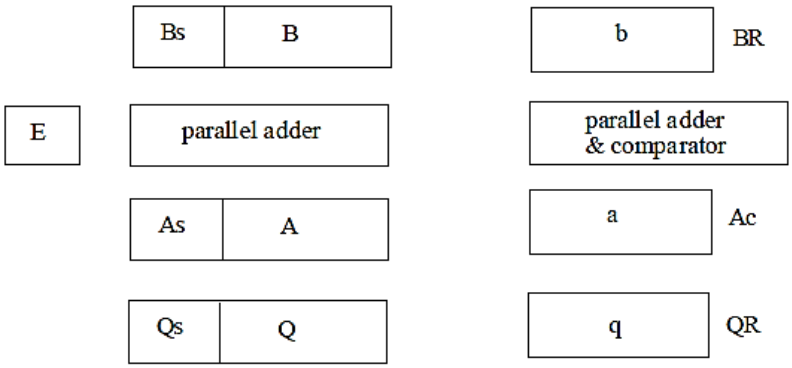


Figure: Register for floating point arithmetic operation

b i i z a y @ g m a i l . c o m

There are two registers BR and AC, each register is subdivided into 2 parts. The mantissa has the uppercase letters symbols and the exponent part uses corresponding lowercase letters symbol. It is assumed that each floating no has mantissa in sign magnitude representation and biased exponent.

Note that the symbol AC represents the entire register that is concatenation of "As", "A" and "a" similarly register BR is subdivide into "B_S", "B" and "b" and QR into "Qs", "Q" and q"".

A parallel adder adds the 2 mantissa and transfer the sum into A and carry into E, a separate parallel adder is used for exponent.

Addition and Subtraction:
During addition and subtraction two floating point operands are in AC and BR. The sums or difference is formed in the AC. The algorithm can be divide into 4 consecutive parts.
1. Check for zeroes.
2. Align the mantissa.
3. Add or subtract the mantissa.
4. Normalize the result.

Multiplication:
The multiplication can be subdivided into 4 parts.
1. Check for zeroes.
2. Add the exponents.
3. Multiply mantissa.
4. Normalize the product.

Division:
The division algorithm can be subdivided into 5 parts.
1. Check for zeroes.
2. Initial registers and evaluate the sign.
3. Align the dividend.
4. Subtract the exponent.
5. Divide the mantissa.

References:

1. M. Morris Mano, "Computer System Architecture", Pearson, 3rd Ed, 2004.
2. M. Morris Mano, "Logic and Computer Design Fundamentals", Pearson Education, 2nd Edition

Assignments:

1. Perform the arithmetic operations with binary numbers given below and with negative numbers in signed 2's complement representation. Use seven bits to accommodate each number together with its sign.
   a. (+ 15) + (+ 14)
   b. (- 15) + (- 14)
   c. (- 15) + (+ 14)
2. Describe the hardware algorithm for addition with an example.

3. Multiply the following numbers using Booth Multiplication algorithm (show all steps)
   a. + 14 * – 13
   b. - 14 * + 13
   Use 5-bit registers
4. Design a 2-bit multiplier by 3 bit array multiplier.
5. Explain the booth algorithm with example. (T.U. 2066)
6. Explain the subtraction algorithm with signed 2's compliment. (T.U. 2067)
7. Explain the non-restoring Division algorithm, flow chart hardware implementation with example. (T.U. 2067, 2069)
8. Explain the Booth algorithm. Multiply 3 x 5 using booth algorithm. (T.U. 2068)
9. Explain the restoring division algorithm with example. (T.U. 2068)
10. Show the steps of multiplication process using Booth algorithm for the following: Y = 8 X 10 (T.U. 2070)
11. Write short notes on the following:
    a. Array Multiplier (T.U. 2069)

A Gentle Advice:

Please go through your text books and reference books for detail study!!! Thank you all.

Notes Compiled By:

Bijay Mishra
biizay.blogspot.com
9813911076 or 9841695609

Unit 7 - Input and Output Organization

Input and Output Organization	**6 Hrs.**
Introduction to Peripheral Devices	**0.5 Hr.**
I/O Subsystem and Peripherals	
I/O Interface	**1.5 Hr.**
Interface I/O Bus and Interface Module Types of I/O Commands I/O and Memory Bus Isolated I/O Memory Mapped I/O I/O Interface Unit	
Direct Memory Access (DMA)	**1.5 Hr.**
Types of I/O, DMA, DMA Transfer	
I/O Processor	**1 Hr.**
I/O Processing CPU-IOP Communication	
Data Communication Processor	**1.5 Hr.**
Serial and Parallel Communication Data Communication Processor Modes of Data Transfer	

Peripheral Devices

The Input/output subsystem of a computer, referred to as I/O, provides an efficient mode of communication between the central system and the output environment. External devices that are under the direct control of the computers are said to be connected on-line. These devices are designed to read information into or out of the memory unit upon command from the CPU and are considered to be part of the computer system. Input/output devices attached to the computer are also called ***peripherals***.

Devices are said to be connected online that are under the direct control of the computer. These devices are designed to read information into or out of the memory unit when the CPU gives a command.

Input or output devices connected to the computer are also called peripherals. Among the most common peripherals are keyboards, display units and printers. Peripherals that provide auxiliary storage for the system are magnetic disks. Other input and output devices are digital incremental plotters, optical and magnetic character readers, analog-to-digital converters etc.

b i i z a y @ g m a i l . c o m

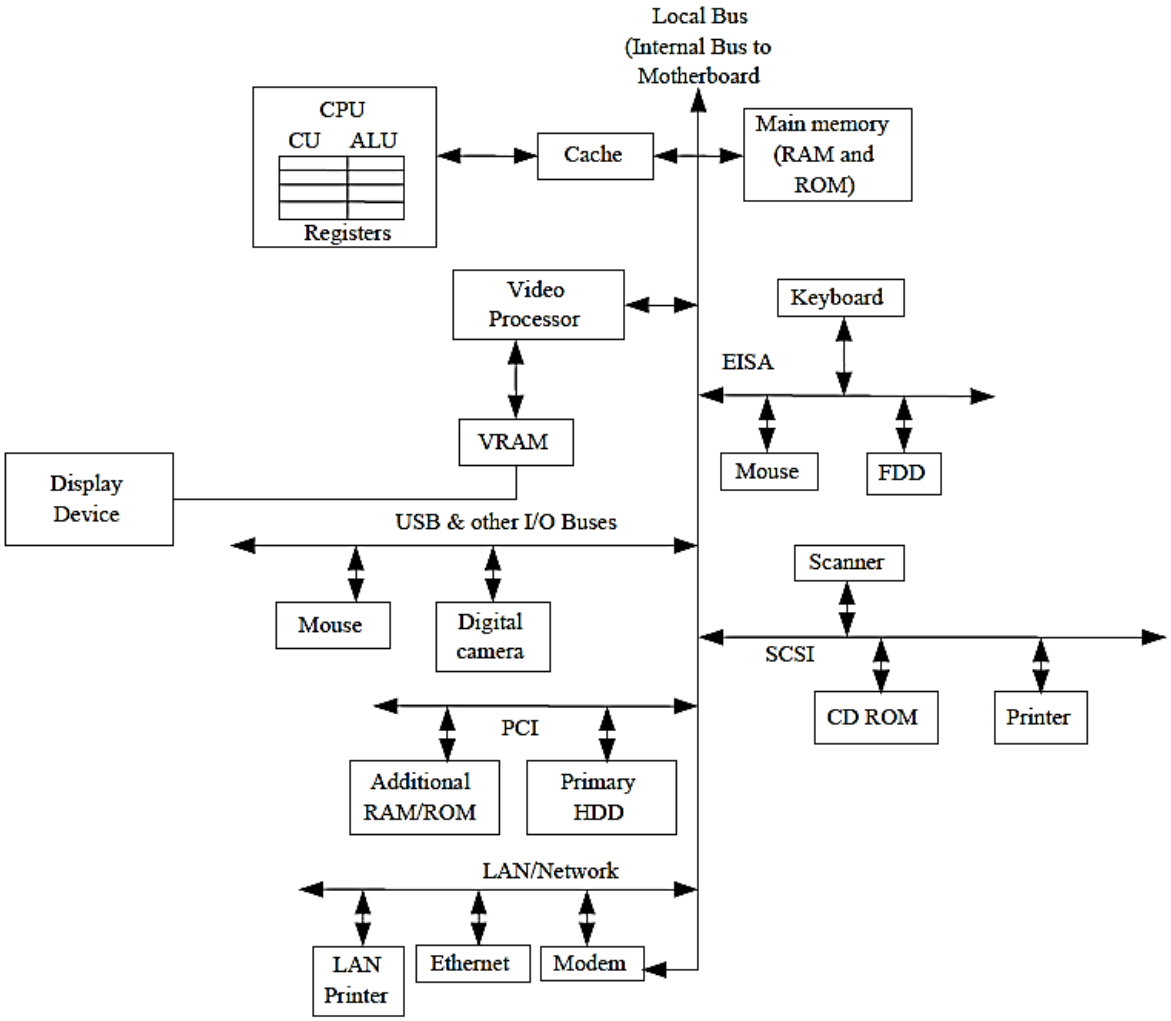


Figure: Block Diagram of a Microcomputer System

Input-Output Interface

Input-output interface gives a method for transferring information between internal memory and I/O devices. Peripherals connected to a computer require special communication links for interfacing them with the central processing unit. The purpose of the communication link is to resolve the differences that exist between the central computer and each peripheral.

The major differences are:

1. Peripherals are electromechanical and electromagnetic devices and their manner of operation is different from the operation of the CPU and memory, which are electronic devices. Therefore a conversion of signal values may be required
2. The data transfer rate of peripherals is usually slower than the transfer rate of the CPU.
3. Data codes and formats in peripherals differ from the word format in the CPU and memory.
4. The operating modes of peripherals are different from each other.

b i i z a y @ g m a i l . c o m

I/O Bus and Interface Modules

A communication link between the processor and several peripherals is represented the following figure below. The I/O bus is made of data lines, address lines and control lines.

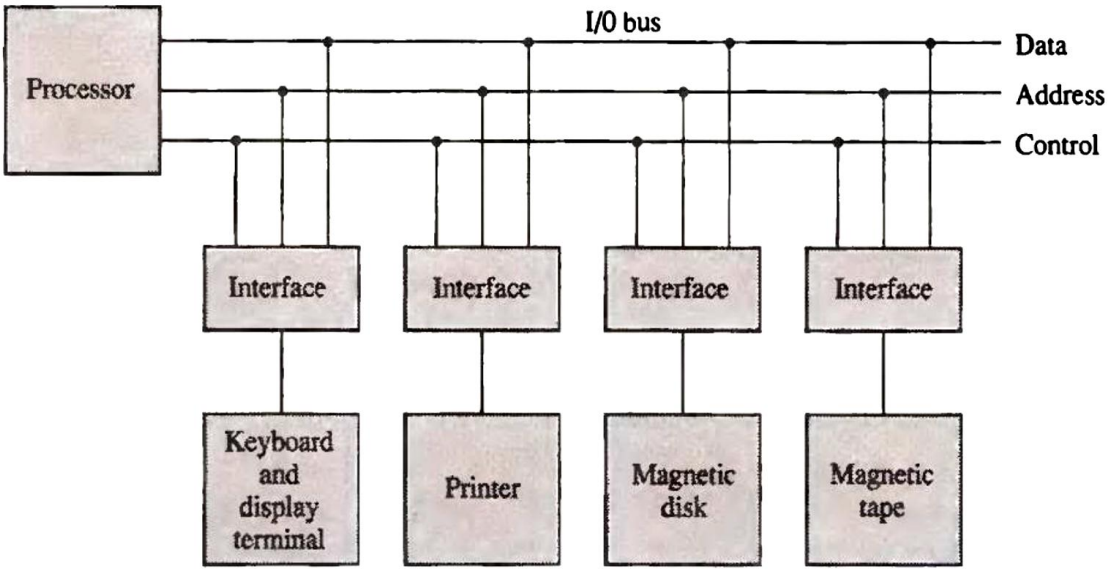


Figure: Connection of I/O Bus to I/O Devices

The magnetic disk, printer and terminal are used in any general-purpose computer. The magnetic tape is used in computers for backup storage. Each peripheral device associated with it by ***interface unit***. Each interface decodes the address and control received from the I/O bus, interprets them for the peripheral and provides signals for the peripheral controller. It also synchronizes the data flow and supervises the transfer between peripheral and processor.

Each peripheral has its own ***controller*** that operates the particular electromechanical device. For example, the printer controller controls the paper motion, the print timing and the selection of printing characters. A controller may be housed separately or may be physically integrated with the peripheral.

The I/O bus from the processor is attached to all peripheral interfaces. To communicate with a particular device, the processor places a device address on the address lines. Each interface attached to the I/O bus contains an address decoder that monitors the address lines. When the interface detects its own address, it activates the path between the bus lines and the device that it controls. All peripherals whose address does not correspond to the address in the bus are disabled by their interface.

The address is made available in the address lines and the processor provides a ***function code*** in the control lines. The interface selected responds to the function code and proceeds to execute it. The function code is referred to as an ***I/O command*** and is in essence an instruction that is executed in the interface and is attached in the peripheral unit. The interpretation of the command depends on the peripheral that the processor is addressing. There are four types of commands that an interface may receive. They are classified as control, status, data output and data input.

We issue a ***control command*** to activate the peripheral. The particular control command issued depends on the peripheral. Each peripheral receives its own distinguished sequence of control commands, depending on its mode of operation.

We use a ***status command*** to test various status conditions in the interface and the peripheral. For example, the computer may wish to check the status of the peripheral before a transfer is initiated. During the transfer, one or more errors may occur which are detected by the interface.

A ***data output command*** causes the interface to respond by transferring data from the bus into one of its registers. Consider an example with a tape unit. The computer starts the tape moving by issuing a control command. Now the processor monitors the status of the tape by means of a status command.

By giving the ***data input command*** the interface receives an item of data from the peripheral and places it in its buffer register. The processor checks if data are available by means of a status command and then issues a data input command. The interface places the data on the data lines, and the processor accepts data.

I/O versus Memory Bus

To communicate with I/O, the processor must communicate with the memory unit. The memory bus contains data, address and read/write control lines. There are three ways to use computer buses to communicate with memory and I/O:

1. Use two separate buses, one for the memory and the other for I/O.
2. Use one common bus for both memory and I/O but have separate control lines for each.
3. Use one common bus for memory and I/O with common control lines.

In the first method, the computer has independent sets of data, address and control buses, one for accessing memory and the other for I/O. This is done in computers having a separate ***I/O processor (IOP)*** in addition to the central processing unit (CPU). The memory communicates with both the CPU and the IOP using a memory bus.

The IOP also communicates with the input and output devices through a separate I/O bus with its own address, data and control lines. The purpose of the IOP is to provide an independent pathway for the transfer of information between external devices and internal memory.

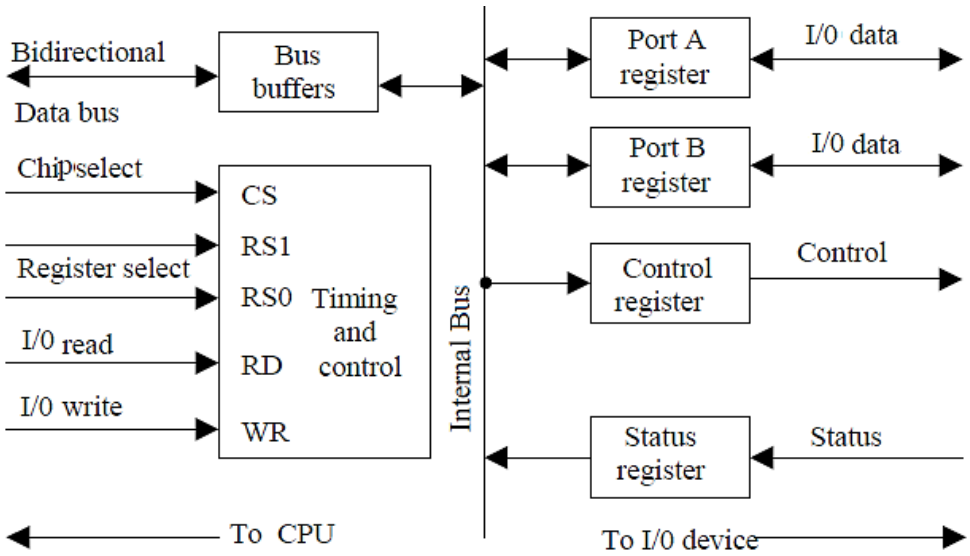
Isolated versus Memory-Mapped I/O

One common bus may be employed to transfer information between memory or I/O and the CPU. Memory transfer and I/O transfer differs in that they use separate read and write lines. The CPU specifies whether the address on the address lines is for a memory word or for an interface register by enabling one of two possible read or write lines.

The I/O read and I/O write control lines are enabled during an ***I/O transfer***. The memory read and memory write control lines are enabled during a ***memory transfer***. This configuration isolates all I/O interface addresses from the addresses assigned to memory and is referred to as the isolated I/O method for assigning addresses in a common bus.

Example of I/O Interface

The figure below shows an example of an I/O interface unit is shown in block diagram.



CS	RS1	RS0	Register selected
0	×	×	None: data bus in high-impedance
1	0	0	Port A register
1	0	1	Port B register
1	1	0	Control register
1	1	1	Status register

Figure: Example of I/O Interface Unit

It has two data registers called ***ports***, a control register, a status register, bus buffers, and timing and control circuits. The interface communicates with the CPU through the data bus. The chip select and register select inputs determine the address assigned to the interface. The I/O read and write are two control lines that specify an input or output, respectively. The four registers communicate directly with the I/O device attached to the interface.

The input-output data to and from the device can be transferred into either port A or port B. The interface may operate with an output device or with an input device, or with a device that requires both input and output. If the interface is connected to a printer, it will only output data, and if it services a character reader, it will only input data. A magnetic disk unit is used to transfer data in both directions but not at the same time, so the interface can use bi-directional lines. A command is passed to the I/O device by sending a word to the appropriate interface register. In a system like this, the function code in the I/O bus is not needed because control is sent to the control register, status information is received from the status register, and data are transferred to and from ports A and B registers. Thus the transfer of data, control, and status information is always via the common data bus. The distinction between data, control, or status information is determined from the particular interface register with which the CPU communicates.

The control register gets control information from the CPU. By loading appropriate bits into the control register, the interface and the I/O device attached to it can be placed in a variety of operating modes. For example, port A may be defined as an input port and port B as an output port. A magnetic tape unit may be instructed to rewind the tape or to start the tape moving in the forward direction. The bits in the status register are used for status conditions and for recording errors that may occur during the data transfer. For example, a status bit may indicate that port-A has received a new data item from the I/O device.

The interface registers uses bi-directional data bus to communicate with the CPU. The address bus selects the interface unit through the chip select and the two register select inputs. A circuit must be provided externally (usually, a decoder) to detect the address assigned to the interface registers. This circuit enables the chip select (CS) input to select the address bus. The two register select-inputs RS1 and RS0 are usually connected to the two least significant lines of the address bus. Those two inputs select one of the four registers in the interface as specified in the table accompanying the diagram. The content of the selected register is transfer into the CPU via the data bus when the I/O read signal is ended. The CPU transfers binary information into the selected register via the data bus when the I/O write input is enabled.

Asynchronous Data Transfer

The internal operations in a digital system are synchronized using a common pulse generator. Clock pulses are used by all registers within a unit and all data transfers among internal registers occur simultaneously during the occurrence of a clock pulse. Two units, such as a CPU and an I/O interface, are independent of each other. If the registers in the interface a common clock with the clock registers, the transfer between the two units is said to be synchronous. In most cases, the internal timing in each unit is independent from the other in that each uses its own private clock for internal registers. Hence, the two units are said to be asynchronous to each other. This approach is widely used in most computer systems.

Strobe Control

The strobe control method uses a single control line to time each transfer. We can activate the strobe by either the source or the destination unit. The figure (a) below shows a source-initiated transfer.

The data bus is used to carry the binary information from source unit to the destination unit. Usually, the bus has multiple lines to transfer an entire byte or word. The strobe is a single line that informs the destination unit when a valid data word is available in the bus.

It is clear from figure (b), that the source unit first places the data on the data bus. After a brief delay to ensure that the data settle to a steady value, the source activates the strobe pulse. The information on the data bus and the strobe signal do not change in the active state for a sufficient time period to allow the destination unit to receive the data. Often, the destination unit uses the falling edge of the strobe pulse to transfer the contents of the data bus into one of its internal registers. The source removes the data from the bus a brief period after it disables its strobe pulse. Actually, the source does not have to change the information in the data bus. The fact that the strobe signal is disabled indicates that the data bus does not have valid data. New valid data will be available only after the strobe is enabled again.

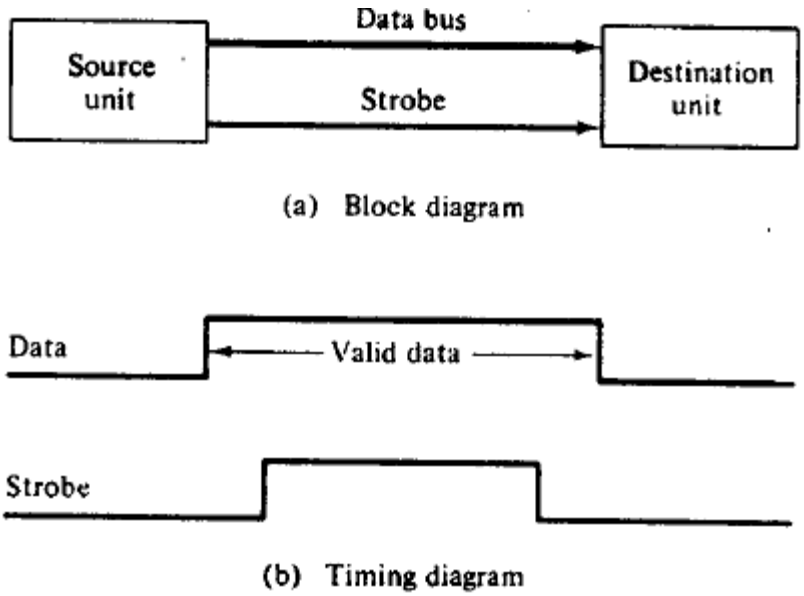


Figure: Source-initiated strobe for data transfer

The figure below describe a data transfer initiated by the destination unit.

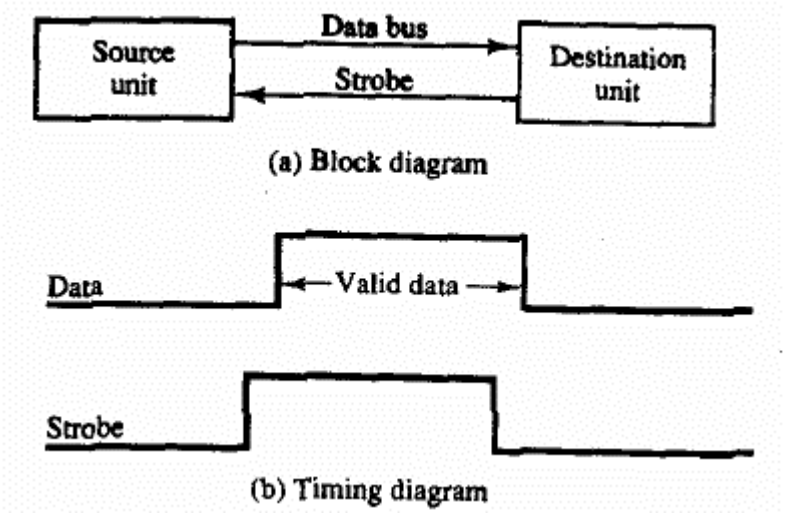


Figure: Destination-initiated strobe for data transfer

In this case the destination unit activates the strobe pulse, informing the source to provide the data. The source unit responds by putting the requested binary information on the data bus.

The data should be valid and remain in the bus long enough for the destination unit to accept it. We can use the falling edge of the strobe pulse again to trigger a destination register. The destination unit then disables the strobe. The source removes the data from the bus after a predetermined time interval.

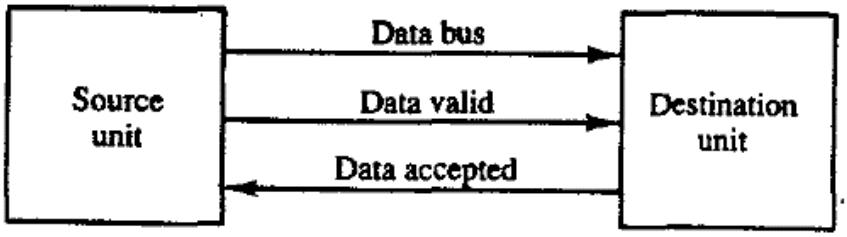
b i i z a y @ g m a i l . c o m

Handshaking

The strobe method has a disadvantage that the source unit that initiates the transfer has no method of knowing whether the destination unit has actually received the data item that was placed in the bus. Similarly, a destination unit that initiates the transfer has no way of knowing whether the source unit has actually placed the data on the bus.

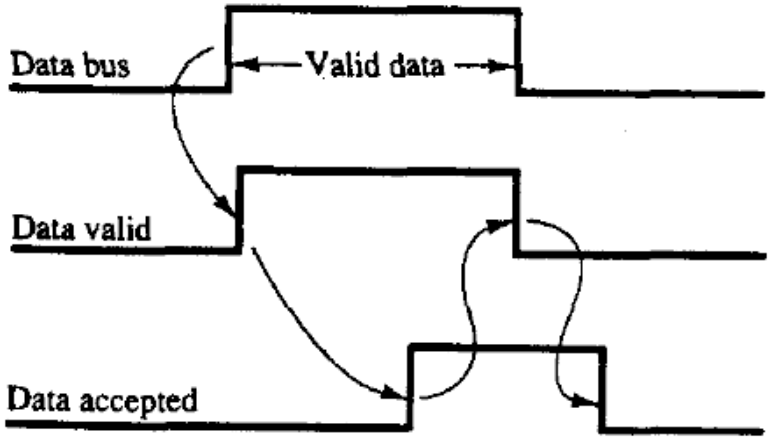
The handshake method solves this problem by introducing a second control signal that provides a reply to the unit that initiates the transfer. The principle of the two-wire handshaking method of data transfer is as follows. One control line is in the same direction as the data low in the bus from the source to the destination. It is used by the source unit to inform the destination unit whether there are valid data in the bus. The other control line is in the other direction from the destination to the source. It is used by the destination unit to inform the source whether it can accept data. The sequence of control during the transfer depends on the unit that initiates the transfer.

The figure below shows the data transfer procedure when source begins it. The two handshaking-lines are ***data valid***, which is generated by the source unit, and ***data accepted***, generated by the destination unit.



(a) Block diagram

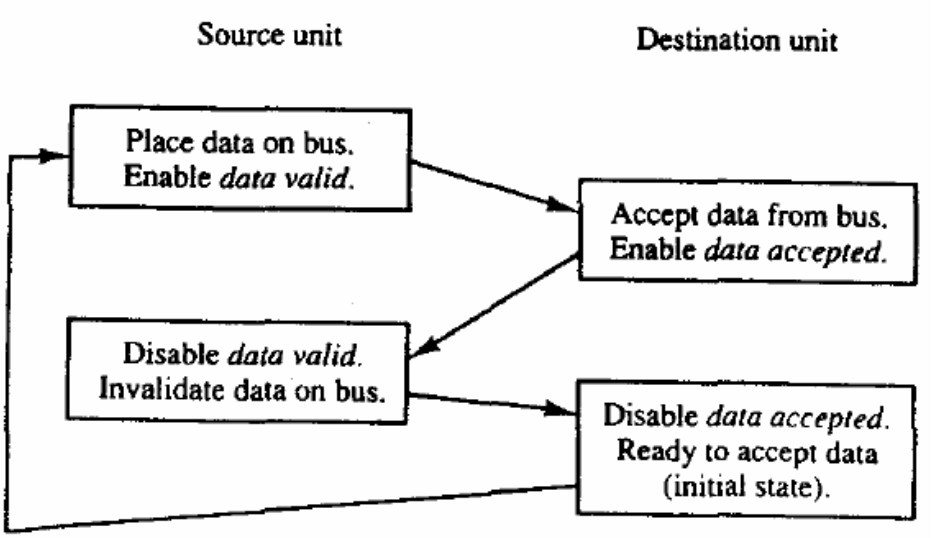
The timing-diagram below describes the exchange of signals between the two units.



(b) Timing diagram

The sequence of events listed in part (c) shows the four possible states that the system can be at any given time.

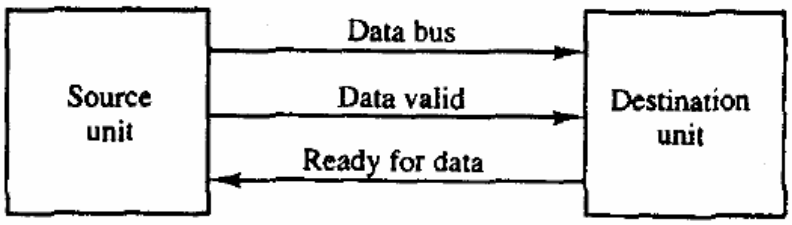
b i i z a y @ g m a i l . c o m



(c) Sequence of events

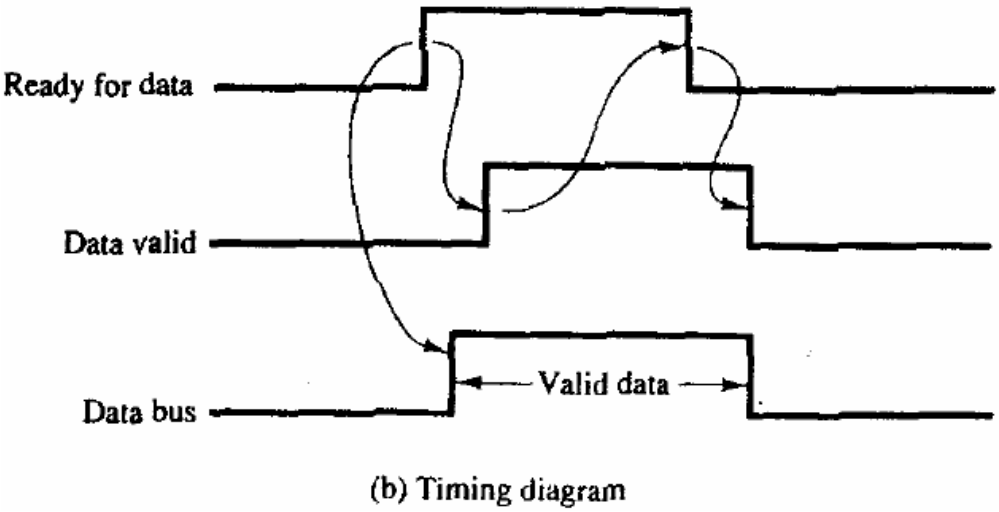
The source unit initiates the transfer by placing the data on the bus and enabling its data-valid signal. The data-accepted signal is activated by the destination unit after it accepts the data from the bus. The source unit then disables its data valid signal, which invalidates the data on the bus. The destination unit then disables its data accepted signal and the system goes into its initial state. The source does not send the next data item until after the destination unit shows its readiness to accept new data by disabling its data accepted signal. So arbitrary delays are allowed from one state to the next, which permits each unit to respond at its own data transfer rate. However, the rate of transfer is determined by the slowest unit.

The destination-initiated transfer using handshaking-lines is shown in the figure below:

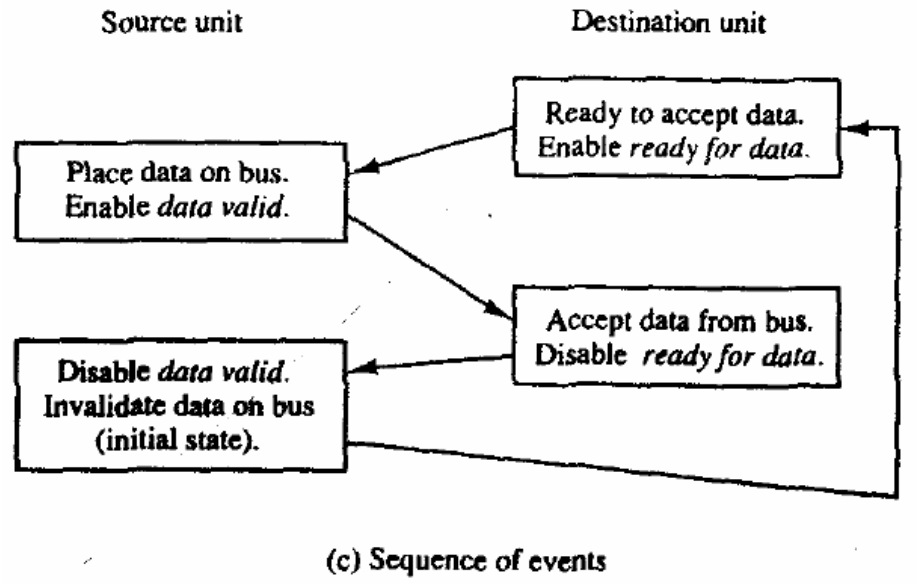


(a) Block diagram

b i i z a y @ g m a i l . c o m



(b) Timing diagram



(c) Sequence of events

Figure: Destination-initiated transfer using handshake

Asynchronous Serial Transfer

We can transfer the data between two units in parallel or serial. In ***parallel data transmission***, each bit has its own path and the total message is transmitted at the same time. In ***serial data transmission***, each bit is sent in sequence one at a time. This method requires the use of one pair of conductors or one conductor and a common ground. Parallel transmission is faster but requires many wires. So we use it for short distances and where speed is important. Serial transmission is slower but is less expensive.

Serial transmission can be synchronous or asynchronous. In ***synchronous transmission***, the two units share a common clock frequency and bits are transmitted continuously at the rate dictated by the clock pulses.

A serial **asynchronous data transmission** technique used in many interactive terminals employs special bits that are inserted at both ends of the character code. With this technique, each character consists of three parts: a start bit, the character bits, and stop bits. The convention is that the transmitter rests at the 12-state when no characters are transmitted. The first bit, called the ***start bit***, is always a 0 and is used to indicate the beginning of a character. The last bit, called the ***stop bit*** is always a 1.

An example of this format is shown in the figure below:

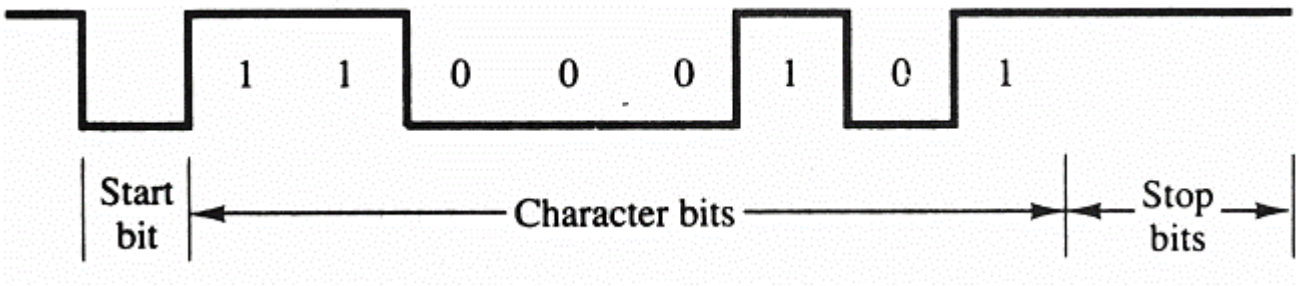


Figure: Asynchronous serial transmission

Receiver can detect a transmitted character from knowledge of the transmission rules:

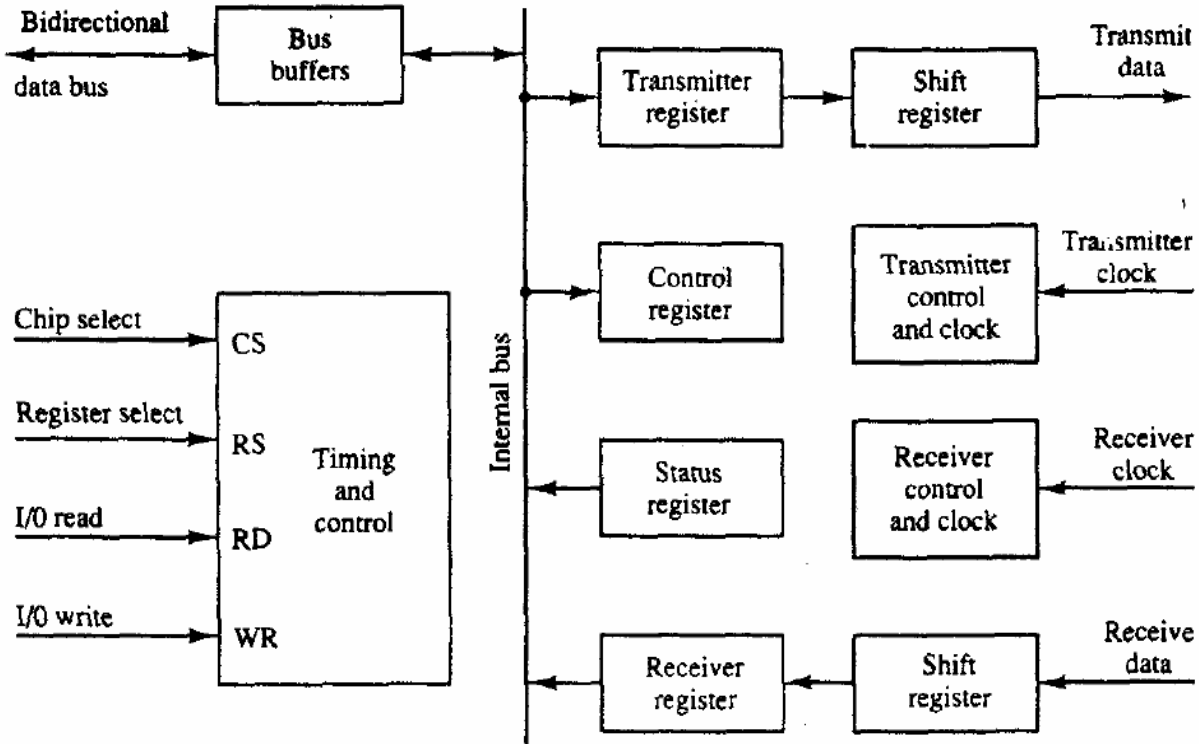
1. When a character is not being sent, the line is in the state 1.
2. The initiation of a character transmission is detected from the start bit, which is always 0.
3. The character bits always follow the start bit.
4. After the last bit of the character is transmitted, a stop bit is detected when the line returns to the 1-state for at least one bit time.

Asynchronous Communication Interface

The interface is initialized for a particular mode of transfer by means of a control byte that is loaded into its control register. The transmitter register accepts a data byte from the CPU through the data bus. This byte is transferred to a shift register for serial transmission. The receiver portion receives serial information into an- other shift register, and when a complete data byte is accumulated, it is transferred to the receiver register.

The CPU can select the receiver register to read the byte through the data bus. The bits in the status register are used for input and output flags and for recording certain errors that may occur during the transmission. The CPU can read the status register to check the status of the flag bits and to determine if any errors have occurred. The chip select and the read and write control lines communicate with the CPU. The chip select (CS) input is used to select the interface through the address bus. The register select (RS) is associated with the read (RD) and write (WR) controls. Two registers are write-only and two are read-only. The register selected is a function of the RS value and the RD and WR status, as listed in the table accompanying the diagram.

The figure below shows the block diagram of an asynchronous communication. It acts as both a transmitter and a receiver.



CS	RS	Operation	Register selected
0	×	×	None: data bus in high-impedance
1	0	WR	Transmitter register
1	1	WR	Control register
1	0	RD	Receiver register
1	1	RD	Status register

Figure: Block diagram of a typical asynchronous communication interface

Modes of Transfer

Binary information received from an external device is usually stored in memory. Information transferred from the central computer into an external device initiates in the memory unit. The CPU only executes the I/O instructions and may accept the data temporarily, but the ultimate source or destination is the memory unit. Data transfer between the central computer and I/O devices may be handled in a variety of modes. Some modes use the CPU as an intermediate path; others transfer the data directly to and from the memory unit.

Data transfer to and from peripherals may be done in either of three possible modes:

1. Programmed I/O
2. Interrupt-initiated I/O
3. Direct memory access (DMA)

b i i z a y @ g m a i l . c o m

Programmed I/O

In this method, the I/O device does not have direct access to memory. A transfer has I/O device to memory needs the execution of several instruction, input CPU, including an input instruction to transfer the data from the device CPU and a store instruction to transfer the data from the CPU to memory instruction may be needed to verify that the data are available from the source and to count the numbers of words transferred.

The figure below shows data transfer from an I/O device through an interface into the CPU. The device transfers bytes of data one at a time, as they are available. When a byte of data is available, the device places it in the I/O bus and enables its data valid line. The interface accepts the byte into its data register and enables the data accepted line. The interface sets a bit in the status register that we will refer to as a "flag" bit. The device can now disable the data valid line, but it will not transfer another byte until the data accepted line is disabled.

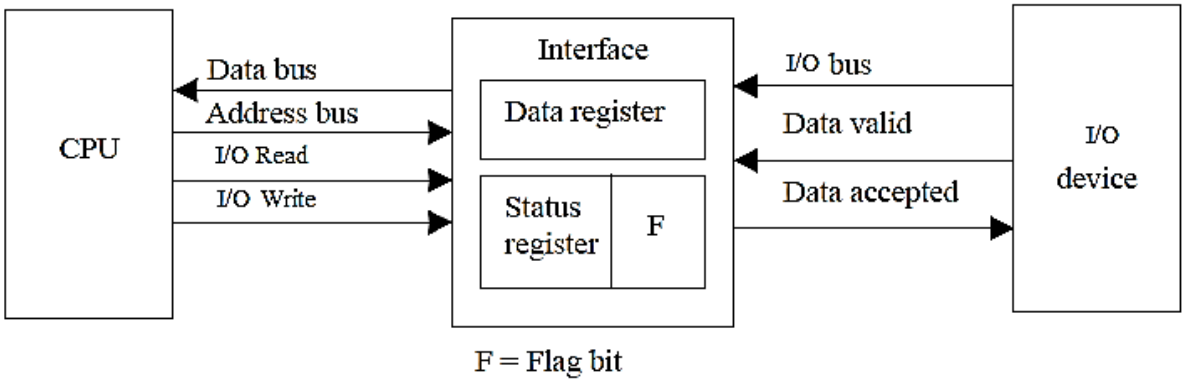


Figure: Data transfer form IO device to CPU

A program is written to check the flag in the status register to determine if a byte has been placed in the data register by the I/O device. This can be done by reading the status register into a CPU register and checking the value of the flag bit. If the flag is equal to 1, the CPU reads the data from the data register. The flag-bit is then cleared to 0 by either the CPU or by the interface, depending on how the interface circuits are designed. Once the flag is cleared, the interface disables the data accepted line and the device can then transfer the next data byte.

A flowchart of the program is shown in the figure below. It is assumed that the device is sending a sequence of bytes that must be stored in memory. The transfer of each byte needs three instructions:

1. Read the status register.
2. Check the status of the flag bit and branch to step I if not set or to step if set.
3. Read the data register.

A CPU register read each byte and then transferred to memory with a store instruction. I/O programming task is to transfer a block of words from an I/O device and store them in a memory buffer.

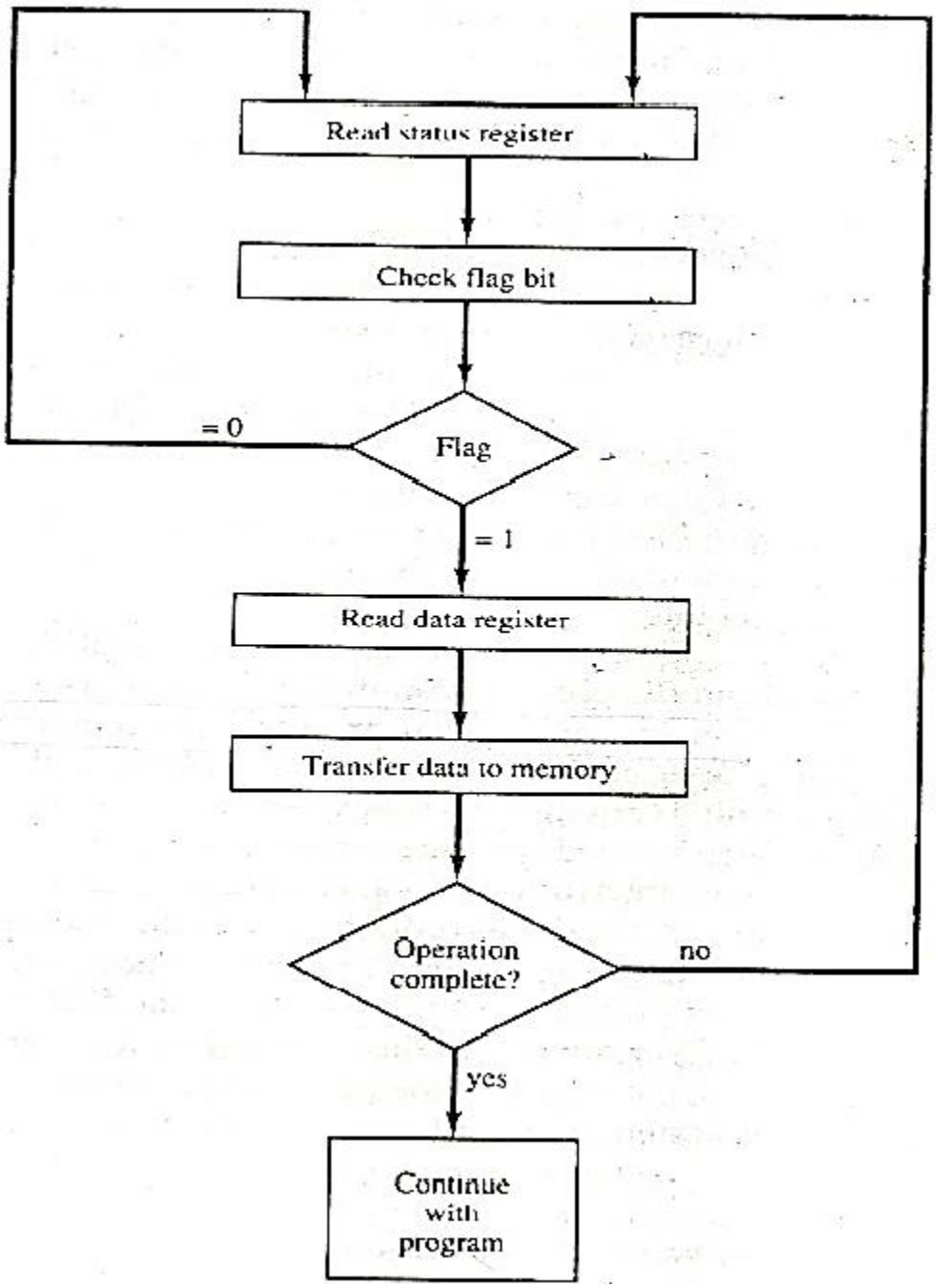


Figure: Flowchart for CPU program to input data

This method is used in small low-speed computers or in systems that are dedicated to monitor a device continuously. The difference in information transfer rate between the CPU and the I/O device makes this type of transfer inefficient.

Interrupt-Initiated I/O

Another way of constantly monitoring the flag is to let the interface inform the computer when it is ready to transfer data. This mode of transfer uses the interrupt facility. While the CPU is running a program, it does not check the flag. However, when the flag is set, the computer is momentarily interrupted from proceeding with the current program and is informed of the fact that the flag has been set. The CPU deviates from what it is doing to take care of the input or output transfer.

The CPU responds to the interrupt signal by storing the return address from the program counter into a memory stack and then control branches to a service routine that processes the required I/O transfer. The way that the processor chooses the branch address of the service routine varies from one unit to another. In principle, there are two methods for accomplishing this. One is called ***vectored interrupt*** and the other, ***non-vectored interrupt***. In a non-vectored interrupt, the branch address is assigned to a fixed location in memory. In vectored interrupt, the source that interrupts supplies the branch information to the computer. This information is called the ***interrupt vector***. In some computers the interrupt vector is the first address of the I/O service routine. In other computers the interrupt vector is an address that points to a location in memory where the beginning address of the I/O service routine is stored.

Priority Interrupt

A priority interrupt is a system that establishes a priority to decide which condition is to be serviced first when two or more requests arrive simultaneously. The system may also determine which conditions are permitted to interrupt the computer while another interrupt is being serviced. Higher-priority interrupt levels are assigned to requests, which if delayed or interrupted, could have serious consequences. Devices with high- speed transfers are given high priority, and slow devices receive low priority. When two devices interrupt the computer at the same time, the computer services the device, with the higher priority first. Establishing the priority of simultaneous interrupts can be done by software or hardware.

We can use a ***polling*** procedure to identify the highest-priority source by the software means. There is one common branch address for all interrupts. The program that takes care of interrupts begins at the branch address and polls the interrupt sources in sequence. The order in which they are tested determines the priority of each interrupt. We test the highest-priority source first, and if its interrupt signal is on, control branches to a service routine for this source. Otherwise, the next-lower priority source is tested, and so on. Thus the initial service routine interrupts consists of a program that tests the interrupt sources in sequence and branches to one of many possible service routines. The particular service routine reached belongs to the highest-priority device among all devices that interrupted the computer.

Example: Daisy-Chaining Priority, Parallel Priority Interrupt, etc.

Direct Memory Access (DMA)

We can transfer data direct to and from memory without the need of the CPU. The transfer of data between a fast storage device such as magnetic disk and memory is often limited by the speed of the CPU. Removing the CPU from the path and letting the peripheral device manager the memory buses directly would improve the speed of transfer. This transfer technique is called ***direct memory access (DMA)***. During DMA transfer, the CPU is idle and has no control of the memory buses. A DMA controller takes over the buses to manage the transfer directly between the I/O device and memory.

The CPU may be in an idle state in a variety of ways. One common method extensively used in microprocessors is to disable the buses through special control signals. Figure below shows two control signals in the CPU that facilitate the DMA transfer.

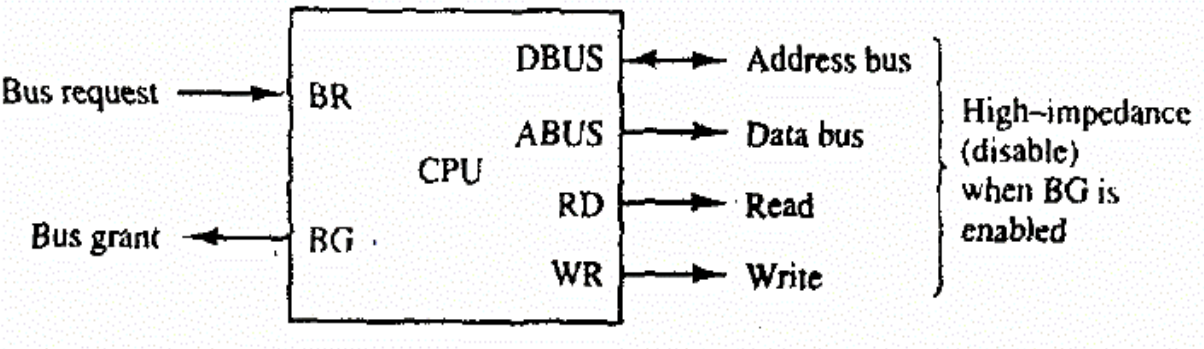


Figure: CPU bus signal for DMA Controller

The ***bus request*** (BR) input is used by the DMA controller to request the CPU to relinquish control of the buses. When this input is active, the CPU terminates the execution of the current instruction and places the address bus, the data bus, and the read and write lines into a high-impedance state. The high-impedance state behaves like an open circuit, which means that the output is disconnected and does not have logic significance.

The CPU activates the ***bus grant*** (BG) output to inform the external DMA that the buses are in the high-impedance state. The DMA that originated the bus request can now take control of the buses to conduct memory transfers without processor intervention. When the DMA terminate the transfer, it disables the bus request line. The CPU disables the bus grant, takes control of the buses, and returns to its normal operation.

DMA communicates directly with the memory, when it takes control of the bus system. We can transfer in several ways. In DMA ***burst transfer***, a block sequence consisting of a number of memory words is transferred in a continuous burst while the DMA controller is master of the memory buses. This mode of transfer is needed for fast devices such as magnetic disks where data transmission cannot be stopped or slowed down until an entire block is transferred.

An alternative technique called ***cycle stealing*** allows the DMA controller to transfer one data word at a time, after which it must return control of the buses to the CPU. The CPU merely delays its operation for one memory cycle to allow the direct memory I/O transfer to "steal" one memory cycle.

DMA Controller

The DMA controller requires the usual circuits of an interface to communicate with the CPU and an I/O device. It also needs an address register, and count register, and a set of address lines. The address register and address line are used for direct communication with the memory. The word count register specifies the number of words that must be transferred. The data transfer may be done directly between the device and memory under control of the DMA.

The unit can communicate with the CPU via the data bus and control lines. The registers in the DMA are selected by the CPU through the address bus by enabling the DS (DMA select) and RS (register select) inputs. The RD (read) and WR (write) inputs are bi-directional.

When the BG=0, the CPU can communicate with the DMA registers through the data bus to read from or write to the DMA registers. When BG = 1, the CPU has relinquished the buses and the DMA can communicate directly with the memory by specifying an address in the address bus and activating the RD or WR control. The DMA communicates with the external peripheral through the request and acknowledge lines by using a prescribed handshaking procedure.

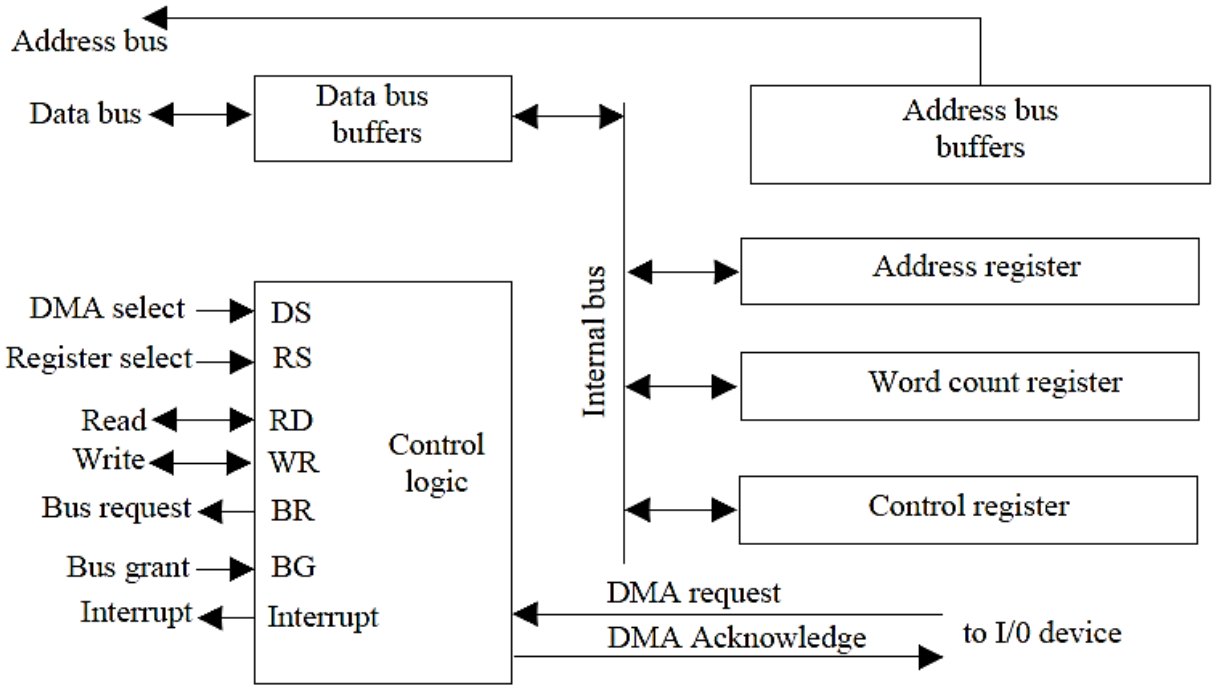


Figure: Block diagram of DMA controller

The DMA controller consists of three registers namely an address register, a word count register, and a control register.

The ***address register*** contains an address to specify the desired location in memory. The address bits go through bus buffers into the address bus. The address register is incremented after each word that is transferred to memory.

The ***word count register*** holds the number of words to be transferred. This register is decremented by one after each word transfer and internally tested for zero.

The ***control register*** specifies the mode of transfer. All registers in the DMA appear to the CPU as I/O interface registers. Thus the CPU can read from or write into the DMA registers under program control via the data bus.

CPU first initializes the DMA. Then the DMA starts and continues to transfer data between memory and peripheral unit until an ending block is transferred. The initialization process is essentially a program consisting of I/O instructions that include the address for selecting particular DMA registers.

The CPU initializes the DMA by sending the following information through the data bus:

1. The starting address of the memory block where data are available (for read) or where data are to be stored (for write)
2. The word count, which is the number of words in the memory block
3. Control to specify the mode of transfer such as read or write
4. A control to start the DMA transfer

The starting address is stored in the address register. The word count is stored in the word count register, and the control information in the control register

DMA Transfer

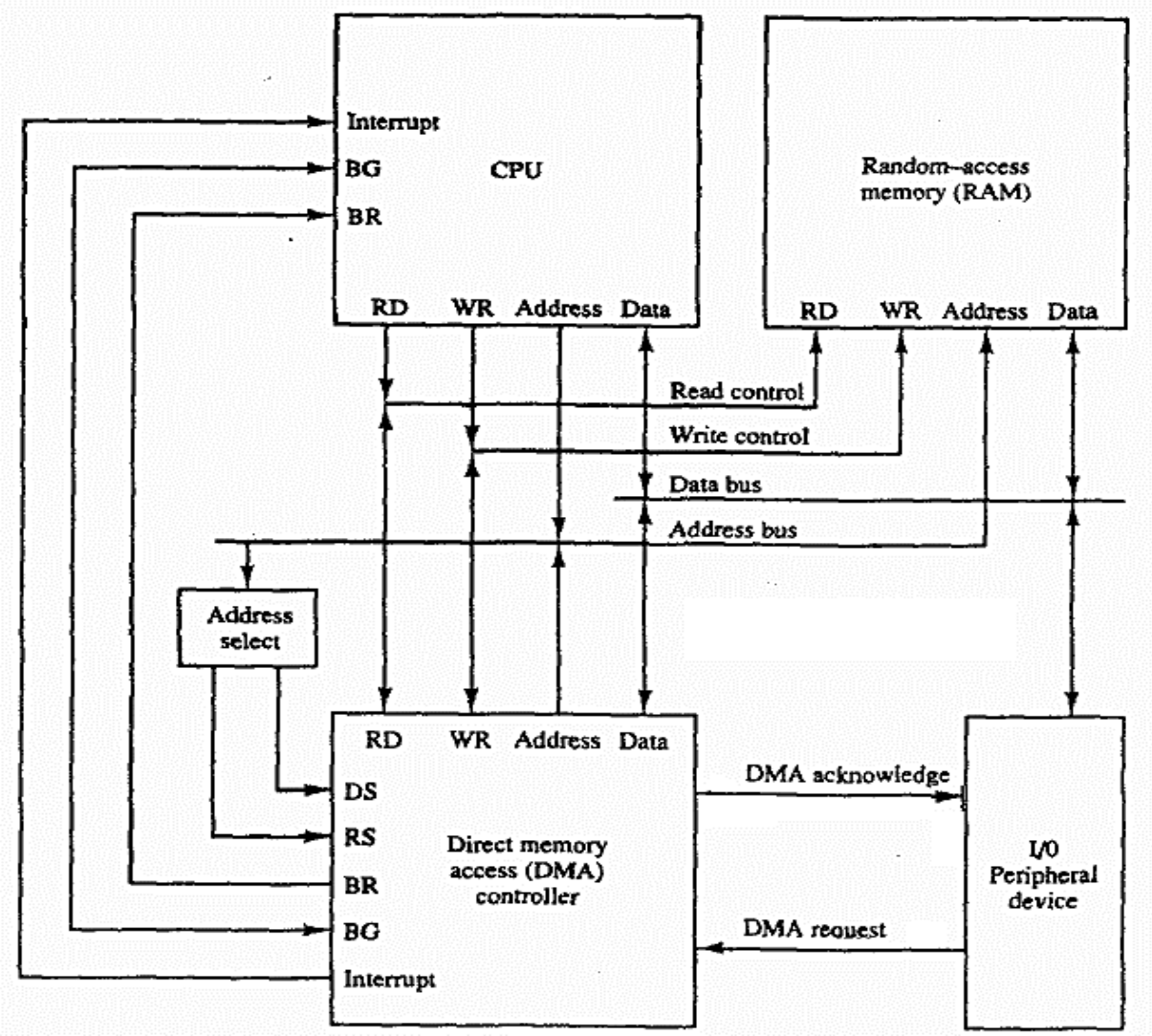


Figure: DMA Transfer

The figure above shows the position of the DMA controller among the other components in a computer system. The CPU communicates with the DMA through the address and data buses as with any interface unit. The DMA has its own address, which activates the DS and RS lines. The CPU initializes the DMA through the data bus. Once the DMA receives the start control command, it can start the transfer between the peripheral device and the memory.

The DMA controller activates the BR line, when the peripheral device sends a DMA request, informing the CPU to relinquish the buses. The CPU responds with its BG line, informing the DMA that its buses are disabled. The DMA then puts the current value of its address register into the address bus, initiates the RD or WR signal, and sends a DMA acknowledge to the peripheral device. Note that the RD and WR lines in the DMA controller are bi-directional. The direction of transfer depends on the status of the BG line. When BG =0, the RD and WR are input lines allowing the CPU to communicate with the internal DMA registers. When BG = 1, the RD and WR are output lines from the DMA controller to the random-access memory to specify the read or write operation for the data.

When the peripheral device gets a DMA acknowledge, it puts a word in the data bus (for write) or receives a word from the data bus (for read). Thus the DMA controls the read or write operations and supplies the address for the memory. The peripheral unit can then communicate with memory through the data bus for direct transfer between the two units while the CPU is momentarily disabled.

The DMA increments its address-register and decrements its word count-register, for each word that is transferred. If the word count does not reach zero, the DMA checks the request line coming from the peripheral. For high-speed device, the line will be active as soon as the previous transfer completed. A second transfer is then initiated, and the process continues until the entire block is transferred. If the peripheral speed is slower, the DMA request line may come somewhat later. In this case the DMA disables the bus request line so that the CPU can continue to execute its program. When the peripheral requests a transfer, the DMA requests the buses again.

If the word count register reaches zero, the DMA stops any further transfer and removes its bus request. It also informs the CPU of the termination by means of an interrupt. When the CPU responds to the interrupt, it reads the content of the word count register. The zero value of this register indicates that all the words were transferred successfully. The CPU can read this register at any time to check the number of words already transferred.

DMA transfer is very useful. It is used for fast transfer of information between magnetic disks and memory. It is also useful for updating the display of the terminal is kept in memory, which can be updated under program control.

Input-Output Processor (IOP)

A computer may incorporate one or more external processors and assign them the task of communicating directly with all I/O devices. An ***input-output processor (IOP)*** may be classified as a processor with direct memory access capability that communicates with I/O devices. In this configuration, the computer system can be divided into a memory unit, and a number of processors comprised of the CPU and one or more IOPs. Each IOP takes care of input and output tasks, relieving the CPU from the housekeeping chores involved in I/O transfers.

The IOP is similar to a CPU except that it is designed to handle the details of I/O processing. Unlike the DMA controller that must be set up entirely by the CPU, the IOP can fetch and execute its own

instructions. IOP instructions are specially designed to facilitate I/O transfers. In addition, the IOP can perform other processing tasks, such as arithmetic, logic, branching, and code translation.

The block diagram of a computer with two processors is shown in figure below. The memory unit occupies a central position and can communicate with each processor by means of direct memory access. The CPU is responsible for processing data needed in the solution of computational tasks. The IOP provides a path for transfer of data between various peripheral devices and the memory unit.

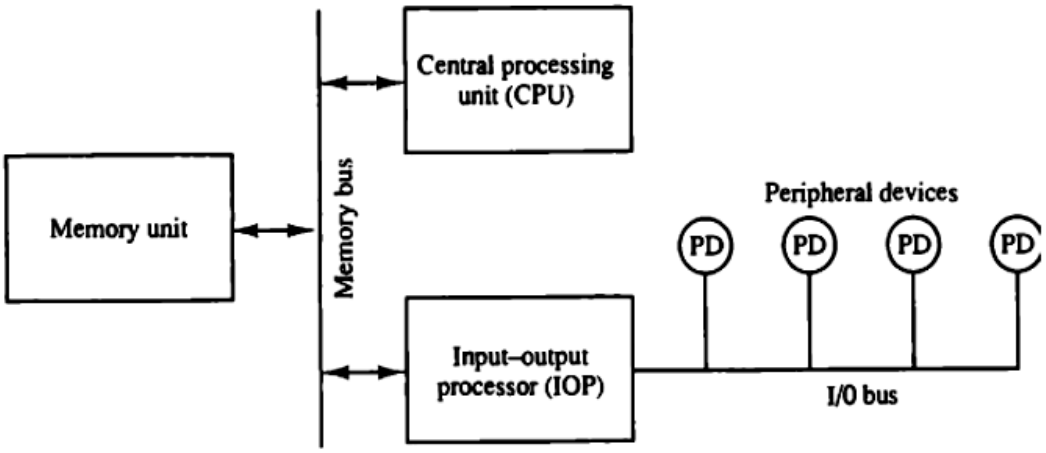


Figure: Block diagram of a computer with I/O Processor

The data formats of peripheral devices differ from memory and CPU data formats. The IOP must structure data words from many different sources. For example, it may be necessary to take four bytes from an input device and pack them into one 32-bit word before the transfer to memory. Data are gathered in the IOP at the device rate and bit capacity while the CPU is executing its own program. After the input data are assembled into a memory word, they are transferred from IOP directly into memory by "stealing" one memory cycle from the CPU. Similarly, an output word transferred from memory to the IOP is directed from the IOP to the output device at the device rate and bit capacity.

The communication between the IOP and the devices attached to it is similar to the program control method of transfer. The way by which the CPU and IOP communicate depends on the level of sophistication included in the system. In most computer systems, the CPU is the master while the IOP is a slave processor. The CPU is assigned the task of initiating all operations, but I/O instructions are execute in the IOP. CPU instructions provide operations to start an I/O transfer and also to test I/O status conditions needed for making decisions on various I/O activities. The IOP, in turn, typically asks for CPU attention by means of an interrupt. It also responds to CPU requests by placing a status word in a prescribed location in memory to be examined later by a CPU program. When an I/O operation is desired, the CPU informs the IOP where to find the I/O program and then leaves the transfer details to the IOP.

Instructions that are read from the memory by an IOP are sometimes called ***commands***, to distinguish them from ***instructions*** that are read by the CPU.

CPU-IOP Communication

The sequence of operations may be carried out as shown in the flowchart of figure below.

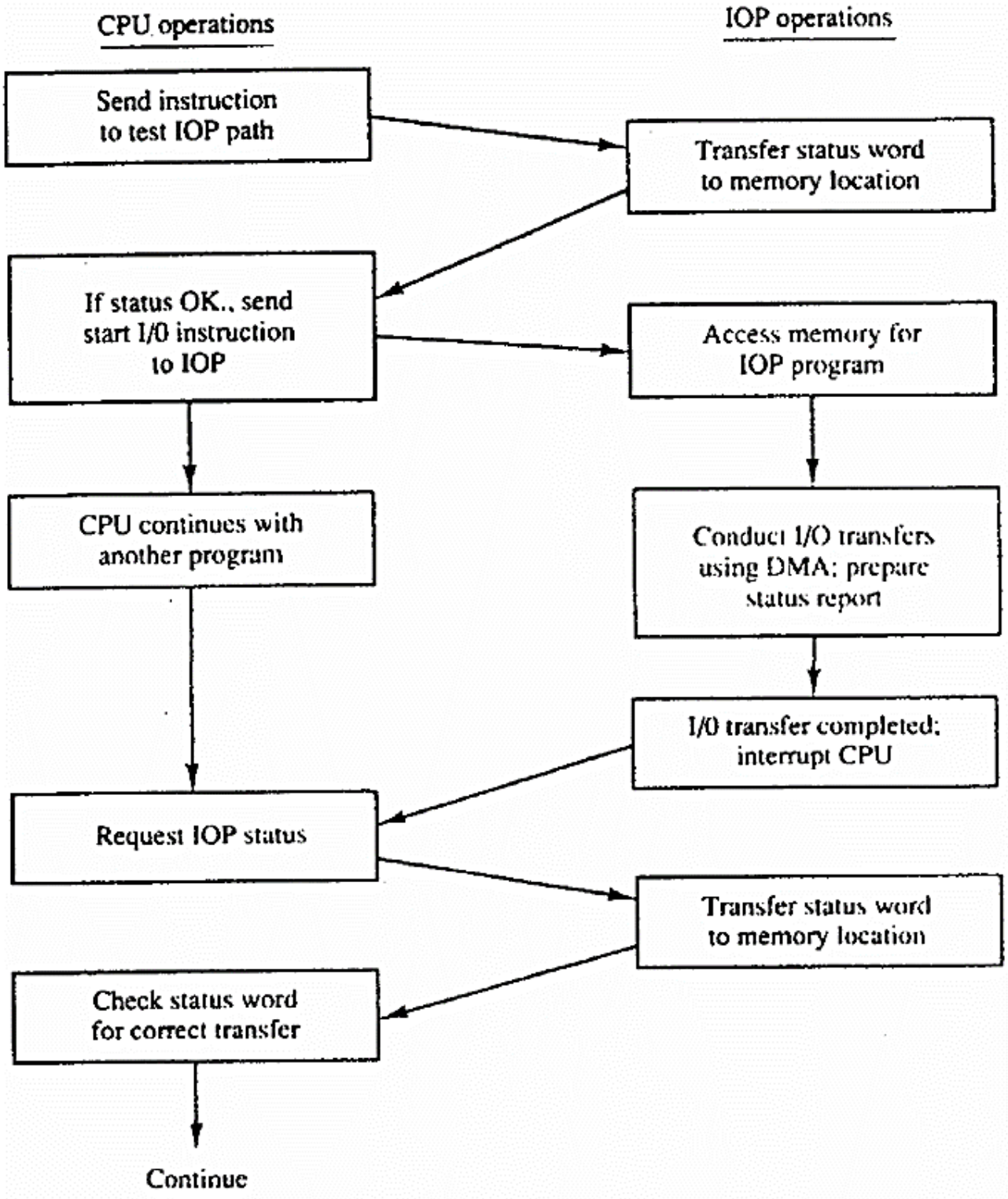


Figure: CPU-IOP Communication

b i i z a y @ g m a i l . c o m

There are many form of the communication between CPU and IOP. These are depending on the particular computer considered. In most cases the memory unit acts as a message center where each processor leaves information for the other. To appreciate the operation of a typical IOP, we will illustrate by a specific example the method by which the CPU and IOP communicate. This is a simplified example that omits many operating details in order to provide an overview of basic concepts.

The CPU sends an instruction to test the IOP path. The IOP responds by inserting a status word in memory for the CPU to check. The bits of the status word indicate the condition of the IOP and I/O device, such as IOP overload condition, device busy with another transfer, or device ready for I/O transfer. The CPU refers to the status word in memory to decide what to do next. If all is in order, the CPU sends the instruction to start I/O transfer. The memory address received with this instruction tells the IOP where to find its program.

The CPU can now continue with another program while the IOP is busy with the I/O program. Both programs refer to memory by means of DMA transfer. When the IOP terminates the execution of its program, it sends an interrupt request to the CPU. The CPU responds to the interrupt by issuing an instruction to read the status from the IOP. The IOP responds by placing the contents of its status report into a specified memory location.

The IOP takes care of all data transfers between several I/O units and the memory while the CPU is processing another program. The IOP and CPU are competing for the use of memory, so the number of devices that can be in operation is limited by the access time of the memory.

Serial Communication

A ***data communication processor (DCP)*** is an I/O processor that distributes and collects data from many remote terminals connected through telephone and other communication lines. It is a specialized I/O processor designed to communicate directly with data communication networks. A communication network may consist of any of a wide variety of devices, such as printers, interactive display devices, digital sensors, or a remote computing facility. With the use of a data communication processor, the computer can service fragments of each network demand in an interspersed manner and thus have the apparent behavior of serving many users at once. In this way the computer is able to operate efficiently in a time-sharing environment.

The main difference between an I/O processor and a data communication processor is in the way the processor communicates with the I/O devices. An I/O processor communicates with the peripherals through a common I/O bus that is comprised of many data and control lines. All peripherals share the common bus and use it to transfer information to and from the I/O processor. A data communication processor communicates with each terminal through a single pair of wires. Both data and control information are transferred in a serial fashion with the result that the transfer rate is much slower. The task of the data communication processor is to transmit and collect digital information to and from each terminal, determine if the information is data or control and respond to all requests according to predetermined established procedures. The processor, obviously, must also communicate, with the CPU and memory in the same manner as any I/O processor.

Synchronous transmission does not use start-stop bits to frame characters and therefore makes more efficient use of the communication link. High-speed devices use synchronous transmission to realize this efficiency. The modems used in synchronous transmission have internal clocks that are set to the frequency that bits are being transmitted in the communication line. For proper operation, it is required that the clocks in the transmitter and receiver modems remain synchronized at all times. The communication line, however, contains only the data bits from which the clock information must be extracted. Frequency synchronization is achieved by the receiving modem from the signal transitions that occur in the received data. Any frequency shift that may occur between the transmitter and receiver clocks is continuously adjusted by maintaining the receiver clock 'at the frequency of the incoming bit stream. The modem transfers the received data together with the clock to the interface unit.

Contrary to asynchronous transmission, where each character can be sent separately with its own start and stop bits, synchronous transmission must send a continuous message in order to maintain synchronism. The message consists of a group of bits transmitted sequentially as a block of data. The entire block is transmitted with special control characters at the beginning and end of the block. The control characters at the beginning of the block supply the information needed to separate the incoming bits into individual characters. In synchronous transmission, where an entire block of characters is transmitted, each character has a parity bit for the receiver to check. After the entire block is sent, the transmitter sends one more character as a parity over the length of the message.

Data can be transmitted between two points in three different modes - simplex, half-duplex, and full duplex.

A ***simplex*** line carries information in one direction only. This mode is seldom used in data communication because the receiver cannot communicate with the transmitter to indicate the occurrence of errors. Examples of simplex transmission are PC to Printer, radio and television broadcasting.

A ***half-duplex*** transmission system is one that is capable of transmitting in both directions but data can be transmitted in only one direction at a time. A pair of wires is needed for this mode. Example: Walkie-Talkie.

A ***full-duplex*** transmission can send and receive data in both directions simultaneously. This can be achieved by means of a four-wire link, with a different pair of wires dedicated to each direction of transmission. Example: Telephone, Mobile Phones, etc.

The communication lines, modems, and other equipment used in the transmission of information between two or more stations is called a data link. The orderly transfer of information in a data link is accomplished by means of a protocol. A data link control protocol is a set of rules that are followed by interconnecting computers and terminals to ensure the orderly transfer of information. The purpose of a data link protocol is to establish and terminate a connection between two stations, to identify the sender and receiver, to ensure that all messages are passed correctly without errors, and to handle all control functions involved in a sequence of data transfers.

References:

1. J. P. Hayes, “Computer Architecture and Organization”, McGraw Hill, 3rd Ed, 1998.
2. M. Morris Mano, “Computer System Architecture”, Pearson, 3rd Ed, 2004.
3. M. Morris Mano, “Digital Design”, Pearson Education, Third Edition
4. M. Morris Mano, “Logic and Computer Design Fundamentals”, Pearson Education, 2nd Edition

Assignments:

(1) What is Handshaking? What advantage it has over strobe control?
(2) Describe strobe control in Asynchronous Data transfer.
(3) What are the different modes of data transfer to and from peripherals?
(4) Draw the flowchart for CPU program to input data and explain it.
(5) What is the need of Priority interrupt?
(6) Discuss the need of Direct Memory Access.
(7) Why does DMA have priority over the CPU when both request a memory transfer?
(8) What are the major differences between I/O bus and interface modules? What are the advantage and disadvantage of each? (T.U. 2066, 2070)
(9) What is the main function of DMA? Mention the three points DMA configurations. (T.U. 2066)
(10) What are the different types of I/O commands? Explain. (T.U. 2066)
(11) What are the three possible modes to transfer the data to and from peripherals? Explain. (T.U. 2066)
(12) What is the role of input-output processor (IOP) in computer system? Explain. (T.U. 2067)
(13) What is DMA transfer? Explain. (T.U. 2067)
(14) Differentiate between isolated I/O and memory mapped I/O. (T.U. 2067, 2068)
(15) Explain the I/O processor with block diagram. (T.U. 2068)
(16) What do you mean by I/O interface? Explain the I/O bus and Interface module. (T.U. 2068)
(17) What do you mean by DMA controller? What are the three registers used in DMA controller? Explain. (T.U. 2069)
(18) What do you mean by interface? What are the major differences between I/O bus and memory bus? (T.U. 2069)
(19) What is input-output processor (IOP)? Why IOP is needed in a computer system? Explain. (T.U. 2070)
(20) Explain the DMA controller with block diagram. How the DMA interacts with I/O devices? Explain. (T.U. 2070)
(21) What are the different type of I/O techniques? Explain. (T.U. 2070)
(22) Differentiate between IOP and DMA. (T.U. 2070)
(23) Write short notes on the following:
    (a) DMA (T.U. 2070)

A Gentle Advice:

Please go through your text books and reference books for detail study!!! Thank you all.

Notes Compiled By:

Bijay Mishra
biizay.blogspot.com
9813911076 or 9841695609

Unit 8 - Memory Organization

Memory Organization	**6 Hrs.**
Hierarchy of Memory System	**1 Hr.**
Types of Memory, Sequential, Random, Memory Hierarchy	
Primary and Secondary Memory	**1.5 Hr.**
Primary memory – RAM, ROM, Bootstrap Loader, RAM and ROM Chips, Memory Address Map, Memory-CPU Connection Auxiliary Memory – Types, Magnetic (Tape, Disk), Optical, Semiconductor	
Virtual Memory	**2.5 Hr.**
Introduction, Address Space, Memory Space, Address Mapping using Pages, Associative Page Table, Page Replacement	
Memory management hardware	**1 Hr.**
Introduction, Segmented Page Mapping, Memory protection	

Memory Hierarchy

Memory in a computer system is required for storage and subsequent retrieval of the instructions and data. A computer system uses a variety of devices for storing these instructions and data that are required for its operation.

Memory hierarchy is to obtain the highest possible access speed while minimizing the total cost of the memory system. The memory hierarchy system consists of all storage devices employed in a computer system from the slow but high capacity auxiliary memory to a relatively faster main memory, to an even smaller and faster cache memory accessible to the high speed registers and processing logic.

The various components are:
Cache Memory: This is the memory which lies in between main memory and CPU. The cache holds those parts of the program and data that are most heavily used

Main Memory: The memory unit that communicates directly with CPU. The programs and data currently needed by the processor reside in main memory (RAM and ROM). It is known as primary memory.

Auxiliary Memory: This is made of devices that provide backup storage. Example: Magnetic tapes, magnetic disks etc. On-line, direct-access secondary storage devices such as magnetic hard disks make up the level of hierarchy just below the main memory. Off-line, direct-access and sequential access secondary storage devices such as magnetic tape, floppy disk, zip disk, WORM disk, etc. fall next in the storage hierarchy. Mass storage devices, often referred to as archival storage, are at the bottom of the storage hierarchy.

b i i z a y @ g m a i l . c o m

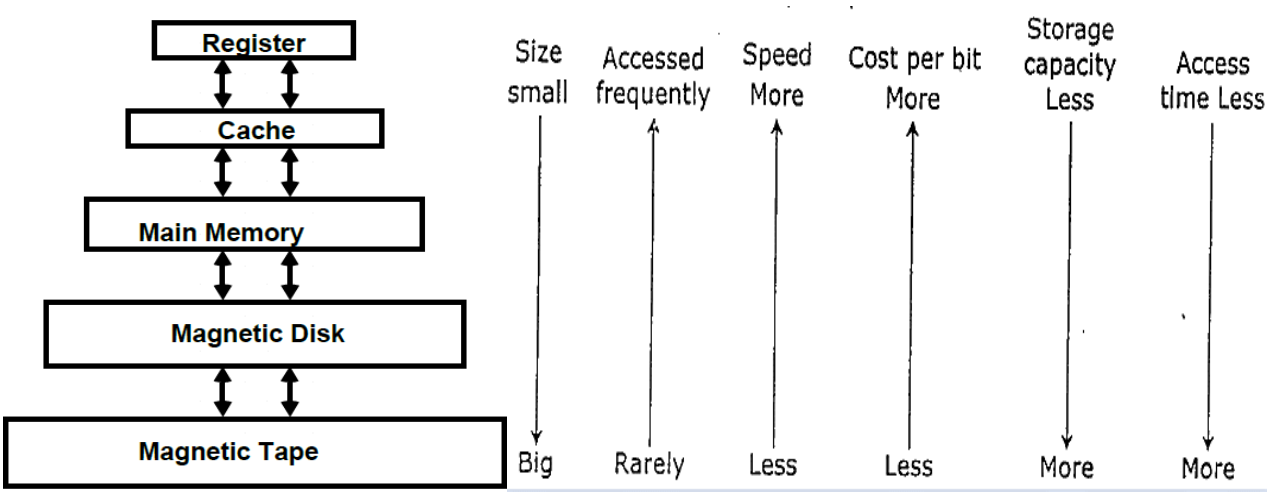
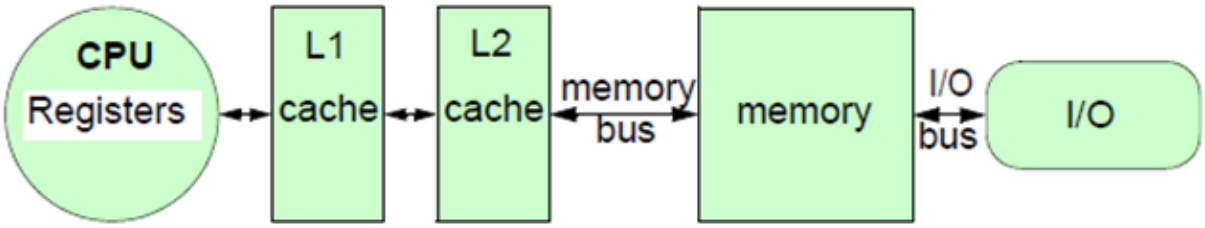


Figure: Memory Hierarchy

In this hierarchy, we have magnetic tapes at the lowest level which means they are very slow and very cheap in nature. Moving on to upper levels, we have main memory in which we get increased speed but with increased cost per bit.

Thus we can conclude as we go towards upper levels:
- Price increases
- Speed increases
- Cost per bit increases
- Access time decreases
- Size decreases



Level	1	2	3	4
Named as	Registers	Cache	memory	disk storage
Typical size	<1 KB	< 4 MB	<2 GB	>2GB
Access time (ns)	2 - 5	3 - 10	80 - 400	5'000'000
Bandwidth(MB/sec)	4000 - 32'000	800 - 5000	400 - 2000	4 - 32
Managed by	Compiler	Hardware	Operating system	Operating system / user

b i i z a y @ g m a i l . c o m

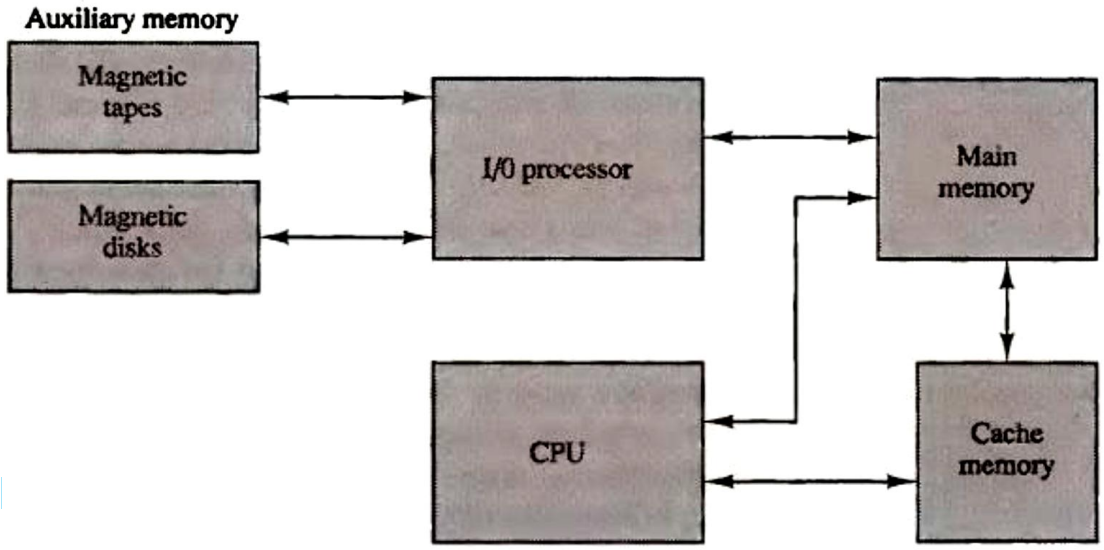


Figure: Memory Hierarchy in a computer system

At the bottom of the hierarchy are the relatively slow magnetic tapes used to store removable files. Next are the magnetic disks which are used as backup storage. The main memory occupies a central position by being able to communicate directly with the CPU and with auxiliary memory devices through an I/O processor. When programs not residing in main memory are needed by the CPU, they are brought in from auxiliary memory. Programs not currently needed in main memory are transferred into auxiliary memory to provide space for currently used programs and data.

A special very-high speed memory called a cache is sometimes used to increase the speed of processing by making current programs and data available to the CPU at a rapid rate. The cache memory is employed in computer systems to compensate for the speed differential between main memory access time and processor logic. CPU logic is usually faster than main memory access time, with the result that processing speed is limited primarily by the speed of main memory. A technique used to compensate for the mismatch in operating speeds is to employ in extremely fast, small cache between the CPU and main memory whose access time is close to processor logic clock cycle time. The cache is used for storing segments of programs currently being executed in the CPU and temporary data frequently needed in the present calculations by making programs and data available at a rapid rate, it is possible to increase the performance rate of the computer.

While the I/O processor manages data transfers between auxiliary memory and main memory, the cache organization is concerned with the transfer of information between main memory and CPU. Thus each is involved with a different level in the memory hierarchy system. The reason for having two or three levels of memory hierarchy is economics. As the storage capacity of the memory increases, the cost per bit for storing binary information decreases and the access time of the memory becomes longer. The auxiliary memory has a large storage capacity, is relatively inexpensive, but has low access speed compared to main memory. The cache memory is very small, relatively expensive, and has very high access speed. Thus as the memory access speed increases, so does its relative cost. The overall goal of using a memory hierarchy is to obtain the highest-possible average access speed while minimizing the total cost of the entire memory system.

b i i z a y @ g m a i l . c o m

Auxiliary and cache memories are used for different purposes. The cache holds those parts of the program and data that are most heavily used, while the auxiliary memory holds those parts that are not presently used by the CPU. Moreover, the CPU has direct access to both cache and main memory but not to auxiliary memory. The transfer from auxiliary to main memory is usually done by means of direct memory access of large blocks of data. The typical access time ratio between cache and main memory is about 1 to 7. For example, a typical cache memory may have an access time of 100ns, while main memory access time may be 700ns. Auxiliary memory average access time is usually 1000 times that of main memory. Block size in auxiliary memory typically ranges from 256 to 2048 words, while cache block size is typically from 1 to 16 words.

Many operating systems are designed to enable the CPU to process a number of independent programs concurrently. This concept, called ***multiprogramming***, refers to the existence of two or more programs indifferent parts of the memory hierarchy at the same time. In this way it is possible to keep all parts of the computer busy by working with several programs in sequence. For example, suppose that a program is being executed in the CPU and an I/O transfer is required. The CPU initiates the I/O processor to start executing the transfer. This leaves the CPU free to execute another program. In a multiprogramming system, when one program is waiting for input or output transfer, there is another program ready to utilize the CPU.

Computer programs are sometimes too long to be accommodated in the total space available in main memory. Moreover, a computer system uses many programs and all the programs cannot reside in main memory at all times. A program with its data normally resides in auxiliary memory. When the program or a segment of the program is to be executed, it is transferred to main memory to be executed by the CPU. Thus one may think of auxiliary memory as containing the totality of information stored in a computer system. It is the task of the operating system to maintain in main memory a portion of this information that is currently active. The part of the computer system that supervises the flow of information between auxiliary memory and main memory is called the ***memory management system.***

Main Memory

The main memory is the central storage unit in a computer system. It is a relatively large and fast memory used to store programs and data during the computer operation. The principal technology used for the main memory is based on semiconductor integrated circuits. Integrated circuit RAM chips are available in two possible operating modes: static and dynamic.

The ***Static RAM*** consists essentially of internal flip-flops that store the binary information. The stored information remains valid as long as power is applied to the unit.

The ***Dynamic RAM*** stores the binary information in the form of electric charges that are applied to capacitors. The capacitors are provided inside the chip by MOS transistors. The stored charge on the capacitors tends to discharge with time and the capacitors must be periodically recharged by refreshing the dynamic memory. Refreshing is done by cycling through the words every few milliseconds to restore the decaying charge. The dynamic RAM offers reduced power consumption and larger storage capacity in a single memory chip. The static RAM is easier to use and has shorter read and write cycles.

b i i z a y @ g m a i l . c o m

Characteristics	SRAM	DRAM
Size	Bigger	Smaller
Cost	More	Less
Speed	Fast	Slow
Bit Density	Less	More
Refreshing	Not required	Required
Construction	0/1 bit is stored in Flip-Flop	0/1 bit is stored as charge on capacitor
Application	Implementing Cache Memory	Implementing Main Memory

SRAM	DRAM
Data is stored in **Flip/Flops**	Data is stored in **charged capacitors**
Faster	**Slower**
Occupies **more space**, hence provides low data storage.	Occupies **less space**, hence provides low data storage.
Consumes **more power** supply	Consumes **less power** supply
More expensive	**Less Expensive**
Does **not require a refreshing** circuit	**Capacitors** need to be **refreshed.**
Used in small quantities in the form of **Cache memory.**	Used in large quantities in the form of **Main Memory.**

RAM and ROM Chips

RAM and ROM chips are available in a variety of sizes. If the memory needed for the computer is larger than the capacity of one chip, it is necessary to combine a number of chips to form the required memory size. A RAM chip is better suited for communication with the CPU if it has one or more control inputs that select the chip only when needed. Another common bidirectional bus feature is a bidirectional data bus that allows the transfer of data either from memory to CPU during a read operation or from CPU to memory during a write operation.

To demonstrate the chip interconnection, we will show an example of a 1024 × 8 memory constructed with 128 × 8 RAM chips and 512 × 8 ROM chips. A RAM chip is better suited for communication with the CPU if it has one or more control inputs that select the chip only when needed. Another common feature is a bidirectional data bus that allows the transfer of data either from memory to CPU during a read operation or from CPU to memory during a write operation. A bidirectional bus can be constructed with three-state buffers. A three-state buffer output can be placed in one of three possible states: a signal equivalent to logic 1, a signal equivalent to logic 0, or a high-impedance state. The logic 1 and 0 are normal digital signals. The high-impedance state behaves like an open circuit, which means that the output does not carry a signal and has no logic significance.

b i i z a y @ g m a i l . c o m

The figure below shows the typical RAM chip with the capacity of the memory 128 words of eight bits (one byte) per word.

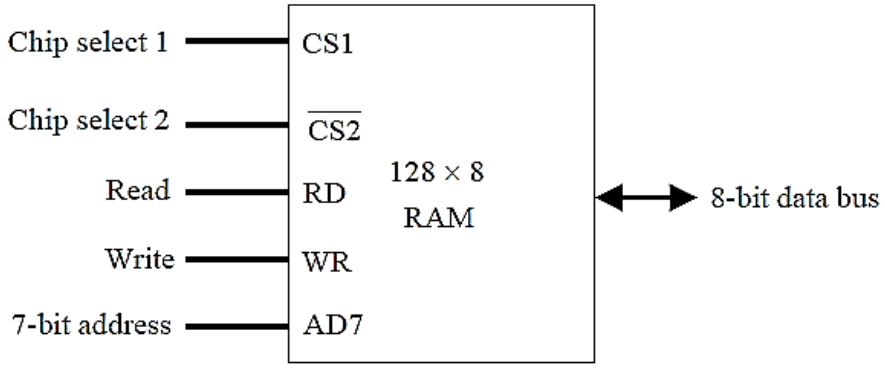


Figure: Block Diagram of RAM

This requires a 7-bit address and an 8-bit bidirectional data bus. The read and write inputs specify the memory operation and the two chips select (CS) control inputs are for enabling the chip only when it is selected by the microprocessor. The availability of more than one control input to select the chip facilitates the decoding of the address lines when multiple chips are used in the microcomputer. The read and write inputs are sometimes combined into one line labeled R/W. When the chip is selected, the two binary states in this line specify the two operations or read or write.

CSI	$\overline{CS2}$	RD	WR	Memory function	State of data bus
0	0	×	×	Inhibit	High-impedance
0	1	×	×	Inhibit	High-impedance
1	0	0	0	Inhibit	High-impedance
1	0	0	1	Write	Input data to RAM
1	0	1	×	Read	Output data from RAM
1	1	×	×	Inhibit	High-impedance

Figure: Function Table

A ROM chip is organized externally in a similar manner. However, since a ROM can only read, the data bus can only be in an output mode. The block diagram of a ROM chip is shown in the figure below.

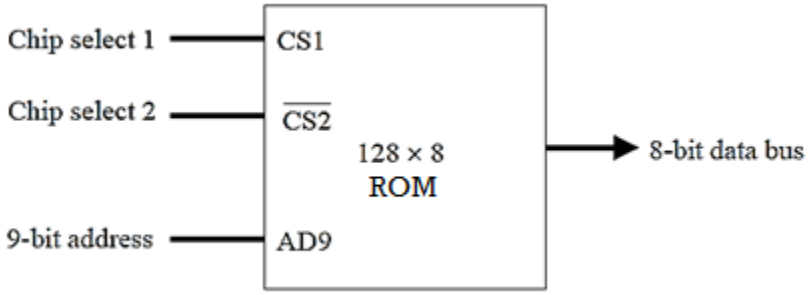


Figure: Typical ROM chip

b i i z a y @ g m a i l . c o m

For the same-size chip, it is possible to have more bits of ROM than of RAM, because the internal binary cells in ROM occupy less space than in RAM. For this reason, the diagram specifies a 512-byte ROM, while the RAM has only 128 bytes. The nine address lines in the ROM chip specify any one of the 512 bytes stored in it. The two chip select inputs must be CS1 = 1 and $\overline{CS2}$ = 0 for the unit to operate. Otherwise, the data bus is in a high-impedance state. There is no need for a read or write control because the unit can only read. Thus when the chip is enabled by the two select inputs, the byte selected by the address lines appears on the data bus.

Memory Address Map

The designer of a computer system must calculate the amount of memory required for the particular application and assign it to either RAM or ROM. The interconnection between memory and processor is then established from knowledge of the size of memory needed and the type of RAM and ROM chips available. The addressing of memory can be established by means of a table that specifies the memory address assigned to each chip. The table, called a ***memory address map***, is a pictorial representation of assigned address space for each chip in the system.

To demonstrate with a particular example, assume that a computer system needs 512 bytes of RAM and 512 bytes of ROM. The component column specifies whether a RAM or a ROM chip is used. The hexadecimal address column assigns a range of hexadecimal equivalent addresses for each chip. The address bus lines are listed in the third column. Although there are 16 lines in the address bus, the table shows only 10 lines because the other 6 are not used in this example and are assumed to be zero. The small x's under the address bus lines designate those lines that must be connected to the address inputs in each chip.

The RAM chips have 128 bytes and need seven address lines. The ROM chip has 512 bytes and needs 9 address lines. The x's are always assigned to the low-order bus lines: lines 1 through 7 for the RAM and lines 1 through 9 for the ROM. It is now necessary to distinguish between four RAM chips by assigning to each a different address. For this particular example we choose bus lines 8 and 9 to represent four distinct binary combinations. Note that any other pair of unused bus lines can be chosen for this purpose.

The table clearly shows that the nine low-order bus lines constitute a memory space for RAM equal to 2^9 = 512 bytes. The distinction between a RAM and ROM address is done with another bus line. Here we choose line 10 for this purpose. When line 10 is 0, the CPU selects a RAM, and when this line is equal to 1, it selects the ROM.

Component	Hexadecimal address	Address bus 10	9	8	7	6	5	4	3	2	1
RAM 1	0000–007F	0	0	0	x	x	x	x	x	x	x
RAM 2	0080–00FF	0	0	1	x	x	x	x	x	x	x
RAM 3	0100–017F	0	1	0	x	x	x	x	x	x	x
RAM 4	0180–01FF	0	1	1	x	x	x	x	x	x	x
ROM	0200–03FF	1	x	x	x	x	x	x	x	x	x

b i i z a y @ g m a i l . c o m

The equivalent hexadecimal address for each chip is obtained from the information under the address bus assignment. The address bus lines are subdivided into groups of four bits each so that each group can be represented with a hexadecimal digit. The first hexadecimal digit represents lines 13 to 16 and is always 0. The next hexadecimal digit represents lines 9 to 12, but lines 11 and 12 are always 0. The range of hexadecimal addresses for each component is determined from the x's associated with it. These x's represent a binary number that can range from an all-0's to an all-1's value.

Memory Connection to CPU

RAM and ROM chips are connected to a CPU through the data and address buses. The low-order lines in the address bus select the byte within the chips and other lines in the address bus select a particular chip through its chip select inputs.

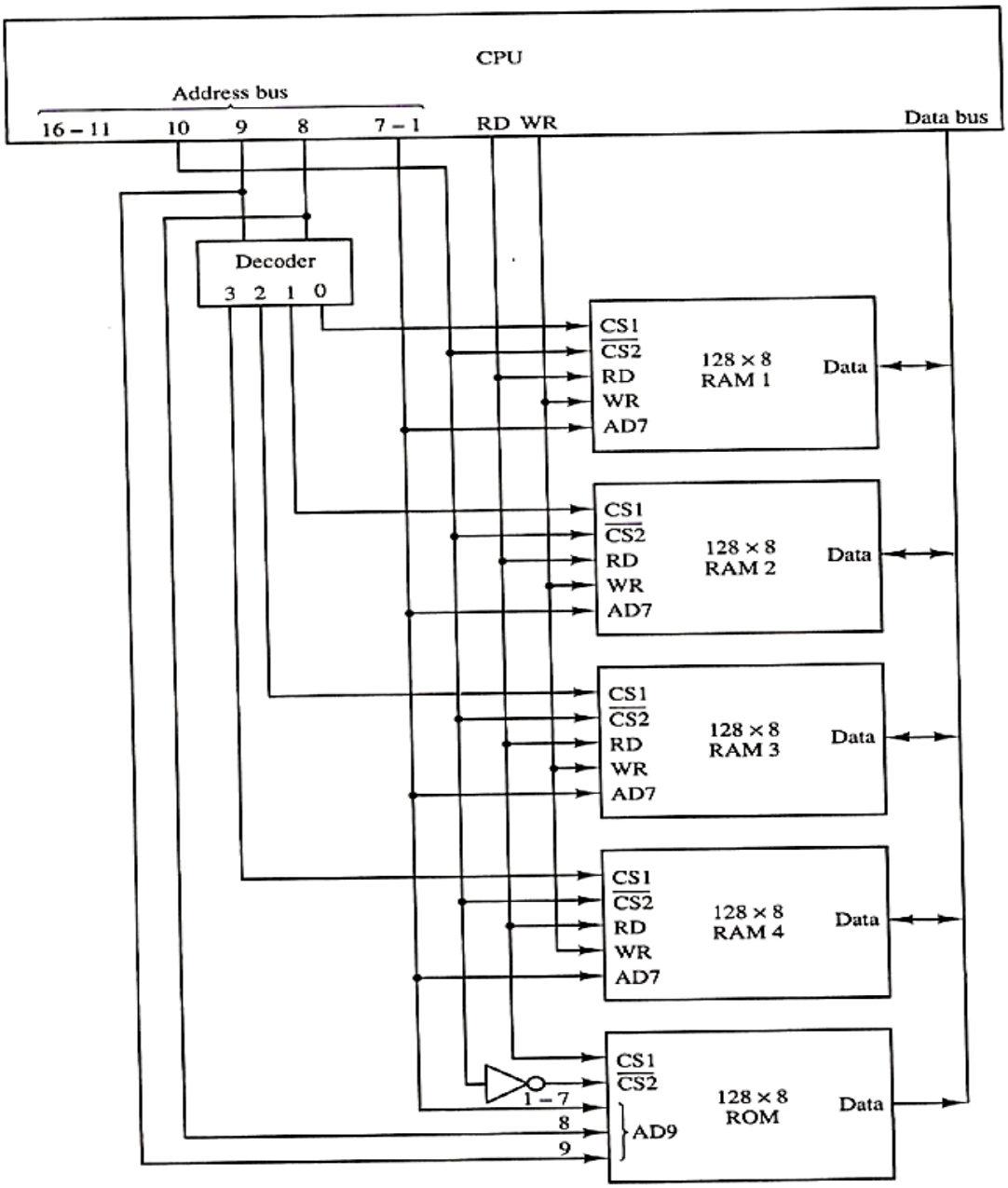


Figure: Memory Connection to the CPU

b i i z a y @ g m a i l . c o m

The connection of memory chips to the CPU is shown in the figure below. This configuration gives a memory capacity of 512 bytes of RAM and 512 bytes of ROM. Each RAM receives the seven low order bits of the address bus to select one of 128 possible bytes. The particular RAM chip selected is determined from lines 8 and 9 in the address bus. This is done through a 2 × 4 decoder whose outputs go to the SCI input in each RAM chip. Thus, when address lines 8 and 9 are equal to 00, the first RAM chip is selected. When 01, the second RAM chip is selected, and so on. The RD and WR outputs from the microprocessor are applied to the inputs of each RAM chip.

The selection between RAM and ROM is achieved through bus line 10. The RAMs are selected when the bit in this line is 0, and the ROM when the bit is 1. The other chip select input in the ROM is connected to the RD control line for the ROM chip to be enabled only during a read operation. Address bus lines 1 to 9 are applied to the input address of ROM without going through the decoder. This assigns addresses 0 to 511 to RAM and 512 to 1023 to ROM. The data bus of the ROM has only an output capability, whereas the data bus connected to the RAMs can transfer information in both directions.

Auxiliary Memory

The most common auxiliary memory devices used in computer systems are magnetic disks and tapes. Other components used, but not as frequently, are magnetic drums, magnetic bubble memory, and optical disks. To understand fully the physical mechanism of auxiliary memory devices one must have a knowledge of magnetics, electronics, and electromechanical systems. Although the physical properties of these storage devices can be quite complex, their logical properties can be characterized and compared by a few parameters. The important characteristics of any device are its access mode, access time, transfer rate, capacity, and cost.

Magnetic Disks

A magnetic disk is a circular plate constructed of metal or plastic coated with magnetized material. Often both sides of the disk are used and several disks may be stacked on one spindle with read/write heads available on each surface. All disks rotate together at high speed and are not stopped or started for access purposes. Bits are stored in the magnetized surface in spots along concentric circles called tracks. The tracks are commonly divided into sections called sectors. In most systems, the minimum quantity of information which can be transferred is a sector.

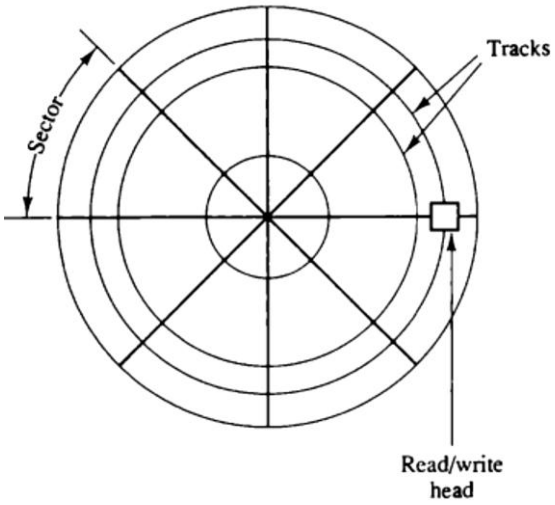


Figure: Magnetic Disk

b i i z a y @ g m a i l . c o m

A track in a given sector near the circumference is longer than a track near the center of the disk. If bits are recorded with equal density, some tracks will contain more recorded bits than others. To make all the records in a sector of equal length, some disks use a variable recording density with higher density on tracks near the center than on tracks near the circumference. This equalizes the number of bits on all tracks of a given sector.

Magnetic Tape

A magnetic tape transport consists of the electrical, mechanical, and electronic components to provide the parts and control mechanism for a magnetic-tape unit. The tape itself is a strip of plastic coated with a magnetic recording medium. Bits are recorded as magnetic spots on the tape along several tracks. Usually, seven or nine bits are recorded simultaneously to form a character together with a parity bit. Read/write heads are mounted one in each track so that data can be recorded and read as a sequence of characters. Magnetic tape units can be stopped, started to move forward or in reverse, or can be rewound. However, they cannot be started or stopped fast enough between individual characters.

Associative Memory

The time required to find an item stored in memory can be reduced considerably if stored data can be identified for access by the content of the data itself rather than by an address. A memory unit accessed by content is called an ***associative memory*** or ***content addressable memory*** (CAM). This type of memory is accessed simultaneously and in parallel on the basis of data content rather than by specific address or location. When a word is written in an associative memory, no address is given. The memory is capable of finding an empty unused location to store the word. When a word is to be read from an associative memory, the content of the word, or part of the word, is specified. The memory locates all words which match the specified content and marks them for reading.

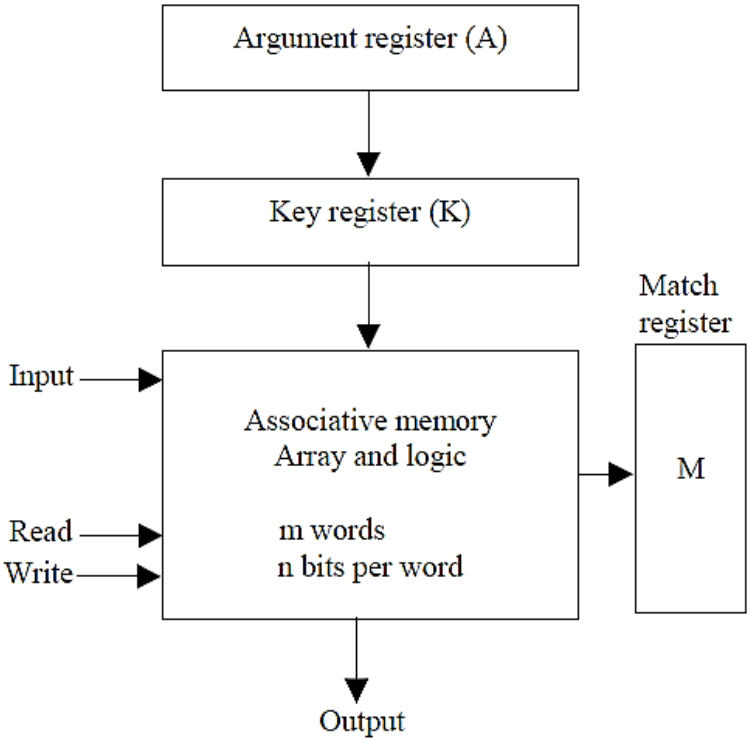


Figure: Block diagram of associative memory

b i i z a y @ g m a i l . c o m

To illustrate with a numerical example, suppose that the argument register A and the key register K have the bit configuration shown below. Only the three leftmost bits of A are compared with memory words because K has 1's in these positions.

A	101	111100	
K	111	000000	
Word 1	100	111100	no match
Word 2	101	000001	match

Word 2 matches the unmasked argument field because the three leftmost bits of the argument and the word are equal. The relation between the memory array and external registers in an associative memory is shown in the figure below.

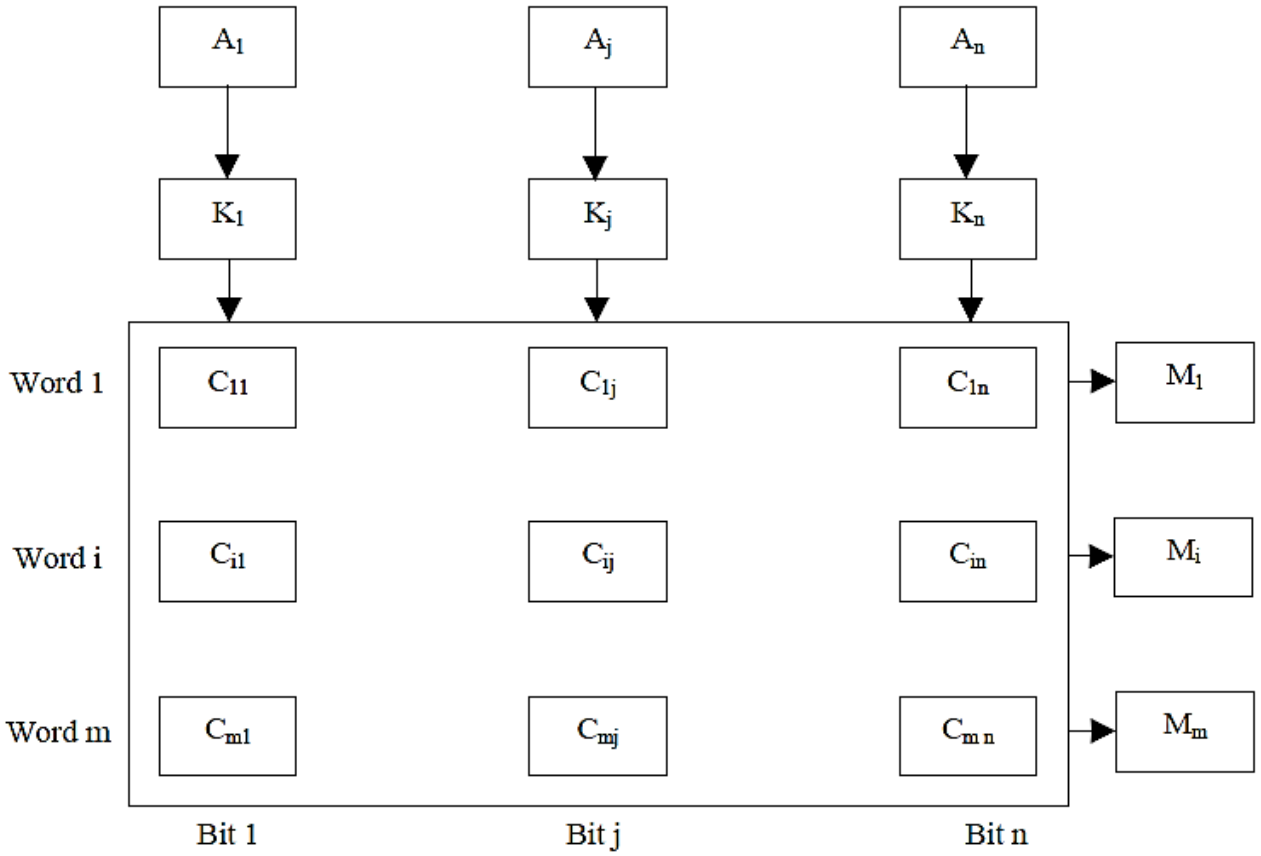


Figure: Associative memory of m words, n cells per word

The cells in the array are marked by the letter C with two subscripts. The first subscript gives the word number and the second specifies the bit position in the word. Thus cell C_{ij} is the cell for bit j in word i. A bit A_j in the argument register is compared with all the bits in column j of the array provided that $K_j = 1$. This is done for all columns j = 1, 2,…,n. If a match occurs between all the unmasked bits of the argument and the bits in word i, the corresponding bit M_i in the match register is set to 1. If one or more unmasked bits of the argument and the word do not match, M_i is cleared to 0.

b i i z a y @ g m a i l . c o m

Cache Memory

Cache memory is a small (in size) and very fast (zero wait state) memory which sits between the CPU and main memory. Unlike normal memory, the bytes appearing within a cache do not have fixed addresses. Instead, cache memory can reassign the address of a data object. This allows the system to keep recently accessed values in the cache.

When a program executes on a computer, most of the memory references are not made uniformly to a small number of locations. Here the Locality of the reference does matter. ***Locality of Reference***, also known as the ***Principle of Locality***, the phenomenon of the same value or related storage locations being frequently accessed. Locality occurs in time (temporal locality) and in space (spatial locality).

Temporal Locality refers to the reuse of specific data and/or resources within relatively small time durations. Spatial Locality refers to the use of data elements within relatively close storage locations. Sequential locality, a special case of spatial locality, occurs when data elements are arranged and accessed linearly. Example: traversing the elements in a one-dimensional array.

When a program loop is executed, the CPU repeatedly refers to the set of instructions in memory that constitute the loop. Every time a given subroutine is called, their sets of instructions are fetched from memory. Thus loops and subroutines tend to localize the references to memory for fetching instructions. To a lesser degree, memory references to data also tend to be localized. Table-lookup procedures repeatedly refer to that portion in memory where the table is stored. Iterative procedures refer to common memory locations and array of numbers are confined within a local portion of memory.

If the active portions of the program and data are placed in a fast small memory, the average memory access time can be reduced, thus reducing the total execution time of the program. Such a fast small memory is referred to as a ***cache memory***. It is placed between the CPU and main memory as illustrated in figure below.

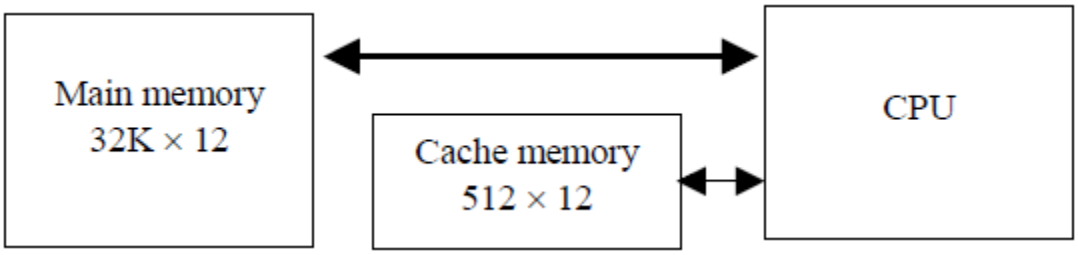


Figure: Example of Cache Memory

The performance of cache memory is frequently measured in terms of a quantity called ***hit ratio***. When the CPU refers to memory and finds the word in cache, it is said to produce a ***hit***. If the word is not found in cache, it is in main memory and it counts as a ***miss***. The ratio of the number of hits divided by the total CPU references to memory (hits plus misses) is the hit ratio.

The hit ratio is best measured experimentally by running representative programs in the computer and measuring the number of hits and misses during a given interval of time. Hit ratios of 0.9 and higher have been reported. This high ratio verifies the validity of the locality of reference property.

The transformation of data from main memory to cache memory is referred to as a mapping process. Three types of mapping procedures are of practical interest when considering the organization of cache memory:

1. Direct mapping
2. Fully-Associative mapping
3. Set-Associative mapping

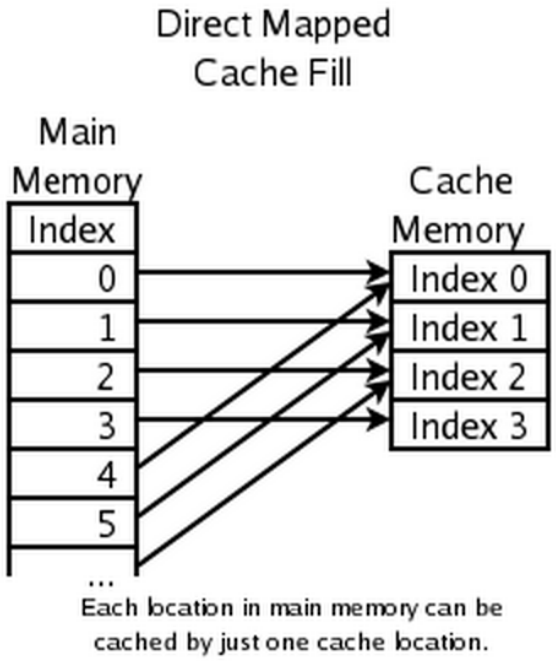
1. Direct Mapping

The simplest technique, known as direct mapping, maps each block of main memory into only one possible cache line. A direct mapped cache could be described as "one-way set associative", i.e. one location in each set.

Direct mapping does not have a replacement policy as such, since there is no choice of which cache entry's contents to evict. This means that if two locations map to the same entry, they may continually knock each other out. Although simpler, a direct-mapped cache needs to be much larger than an associative one to give comparable performance, and is more unpredictable.

Inorder to determine to which Cache line a main memory block is mapped we can use the formula shown below:

Cache Line Number = (Main memory Block number) MOD (Number of Cache lines)



The direct mapping technique is simple and inexpensive to implement. One of the advantages of a direct mapped cache is that it allows simple and fast speculation. Once the address has been computed, the one cache index which might have a copy of that location in memory is known. That cache entry can be read, and the processor can continue to work with that data before it finishes checking that the tag actually matches the requested address.

Its main disadvantage is that there is a fixed cache location for any given block. Different blocks that map into the same line, then the blocks will be continually swapped in the cache, and the hit ratio will be low.

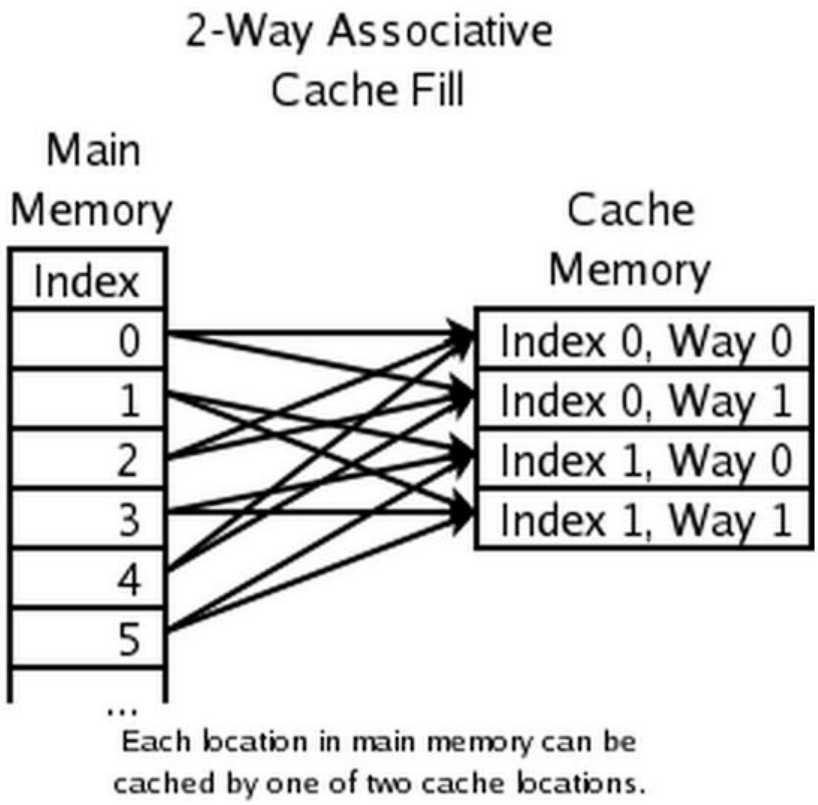
2. Fully-Associative Mapping:

If each location in main memory can be cached in either of two locations in the cache, one logical question is: which one of the two? The simplest and most commonly used scheme is fully-associative mapping. If a main memory block can be placed in any of the cache slots, then the cache is said to be mapped in fully associative. A fully associative cache is N-way associative (where N is the total number of blocks in the cache).

It overcomes the disadvantage of direct mapping by permitting each main memory block to be loaded into any line of the cache. One benefit of this scheme is that the tags stored in the cache do not have to include that part of the main memory address which is implied by the cache memory's index. Since the cache tags have fewer bits, they take less area on the microprocessor chip and can be read and compared faster.

In full-associative mapping:

- A main memory block can load into any line of cache
- Memory address is interpreted as tag and word
- Tag uniquely identifies block of memory
- Every line's tag is examined for a match
- Cache searching gets expensive



b i i z a y @ g m a i l . c o m

With full-associative mapping, there is flexibility as to which block to replace when a new block is read into the cache. The whole address must be used as the tag. All tags must be compared simultaneously (associatively) with the requested address and if one matches then its associated data is accessed. This requires an associative memory to hold the tags which makes this form of cache more expensive.

3. Set-associative Mapping

A set-associative scheme is a hybrid between a fully associative cache, and direct mapped cache. It combines the simplicity of direct mapping with the flexibility of fully associative mapping. The slots are grouped into sets. We find the appropriate set for a given address (which is like the direct mapped scheme), and within the set we find the appropriate slot (which is like the fully associative scheme).

In this mapping mechanism, the cache memory is divided into 'v' sets, each consisting of 'n' cache lines. A block from main memory is first mapped onto a specific cache set, and then it can be placed anywhere within that set. This type of mapping has very efficient ratio between implementation and efficiency.

The set is usually chosen by:

Cache set number = (Main memory block number) MOD (Number of sets in the cache memory)

If there are 'n' cache lines in a set, the cache placement is called n-way set associative i.e. if there are two blocks or cache lines per set, then it is a 2-way set associative cache mapping and four blocks or cache lines per set, then it is a 4-way set associative cache mapping.

Writing into Cache

The simplest and most commonly used procedure is to update main memory with every memory write operation, with cache memory being updated in parallel if it contains the word at the specified address. This is called the ***write-through*** method.

Advantage of write-through:
1. READ miss never results in writes to main memory.
2. Easy to implement
3. Main Memory always has the most current copy of the data (consistent)

Disadvantage of write-through:
1. WRITE operation is slower as we have to update both Main Memory and Cache Memory.
2. Every write needs a main memory access as a result uses more memory bandwidth

The second procedure is called the ***write-back*** method. In this method only the cache location is updated during a write operation. The location is then marked by a flag so that later when the word is removed from the cache it is copied into main memory. The reason for the write-back method is that during the time a word resides in the cache, it may be updated several times; however, as long as the word remains in the cache, it does not matter whether the copy in main memory is out of date, since requests from the word are filled from the cache.

Advantage of write-back:
1. WRITE's occur at the speed of the cache memory.
2. Multiple WRITE's within a block require only one WRITE to main memory as a result uses less memory bandwidth

Disadvantage of write-back:
1. Harder to implement
2. Main Memory is not always consistent with cache reads that result in replacement may cause writes of dirty blocks to main memory.

Cache Initialization
One more aspect of cache organization that must be taken into consideration is the problem of initialization. The cache is initialized when power is applied to the computer or when the main memory is loaded with a complete set of programs from auxiliary memory. After initialization the cache is considered to be empty, but in effect it contains some non-valid data. It is customary to ***valid bit*** include with each word in cache a valid bit to indicate whether or not the word contains valid data.

Virtual Memory

In a memory hierarchy system, programs and data are first stored in auxiliary memory. Portions of a program or data are brought into main memory as they are needed by the CPU. Virtual memory is a concept used in some large computer systems that permit the user to construct programs as though a large memory space were available, equal to the totality of auxiliary memory. Each address that is referenced by the CPU goes through an address mapping from virtual address to a physical address in main memory. Virtual memory is used to give programmers the illusion that they have a very large memory at their disposal, even though the computer actually has a relatively small main memory. A virtual memory system provides a mechanism for translating program-generated addresses into correct main memory locations. This is done dynamically, while programs are being executed in the CPU. The translation or mapping is handled automatically by the hardware by means of a mapping table.

Address Space and Memory Space
An address used by a programmer will be called a virtual address, and the set of such addresses the address space. An address in main memory is called a location or physical address. The set of such locations is called the memory space. Thus the address space is the set of addresses generated by programs as they reference instructions and data; the memory space consists of the actual main memory locations directly addressable for processing. In most computers the address and memory spaces are identical. The address space is allowed to be larger than the memory space in computers with virtual memory.

As an illustration, consider a computer with a main-memory capacity of 32K words (K =1024). Fifteen bits are needed to specify a physical address in memory since $32K = 2^{15}$. Suppose that the computer has available auxiliary memory for storing 2^{20} = 1024K words. Thus auxiliary memory has a capacity for storing information equivalent to the capacity of 32 main memories. Denoting the address space by N and the memory space by M, we then have for this example N = 1024K and M = 32K.

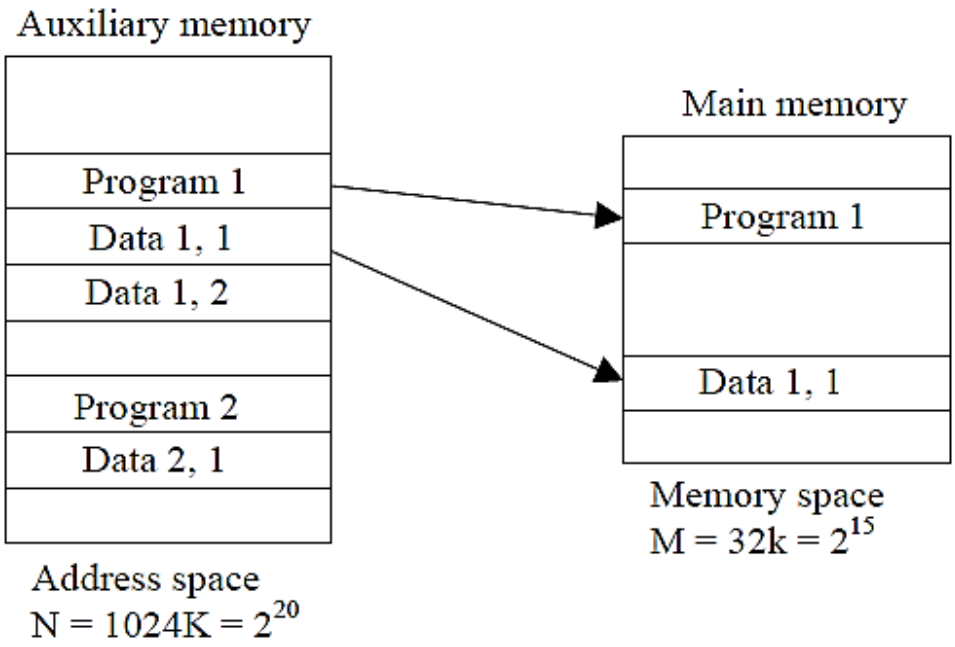


Figure: Relation between address and memory space in a virtual memory system

A table is needed, as shown in figure below to map a virtual address of 20 bits to a physical address of 15 bits. The mapping is a dynamic operation, which means that every address is translated immediately as a word is referenced by CPU. The mapping table may be stored in a separate memory or in main memory. In the first case, an additional memory unit is required as well as one extra memory access time. In the second case, the table takes space from main memory and two accesses to memory are required with the program running at half speed.

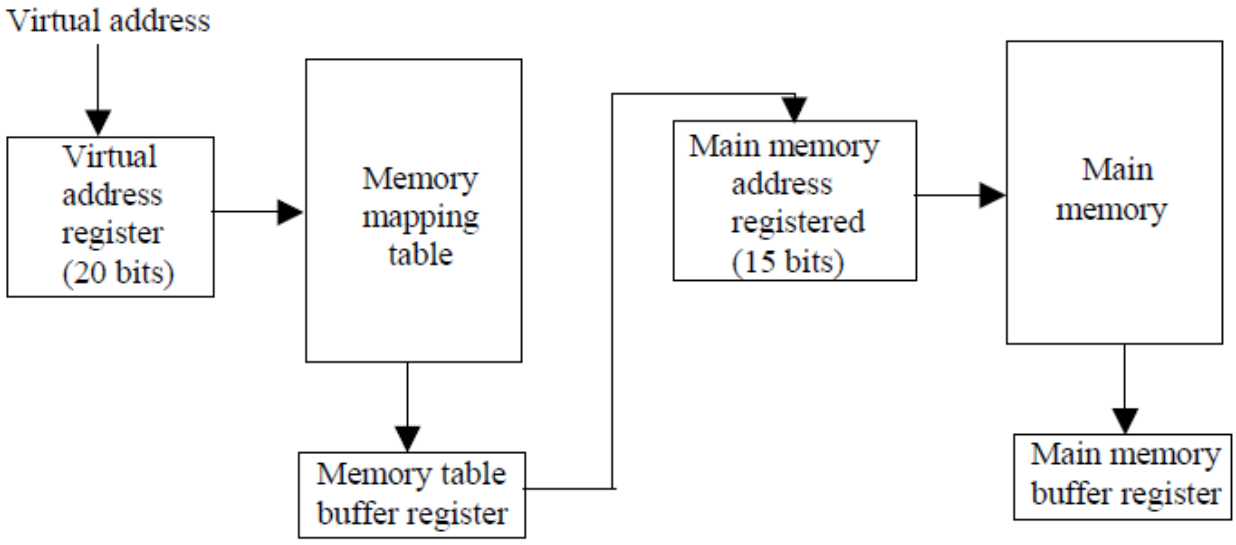


Figure: Memory table for mapping a virtual address

b i i z a y @ g m a i l . c o m

Address Mapping Using Pages

The table implementation of the address mapping is simplified if the information in the address space and the memory space are each divided into groups of fixed size. The physical memory is broken down into groups of equal size called blocks, which may range from 64 to 4096 words each. The term page refers to groups of address space of the same size. For example, if a page or block consists of 1K words, then, using the previous example, address space is divided into 1024 pages and main memory is divided into 32 blocks. Although both a page and a block are split into groups of 1K words, a page refers to the organization of address space, while a block refers to the organization of memory space. The programs are also considered to be split into pages. Portions of programs are moved from auxiliary memory to main memory in records equal to the size of a page. The term "page frame" is sometimes used to denote a block.

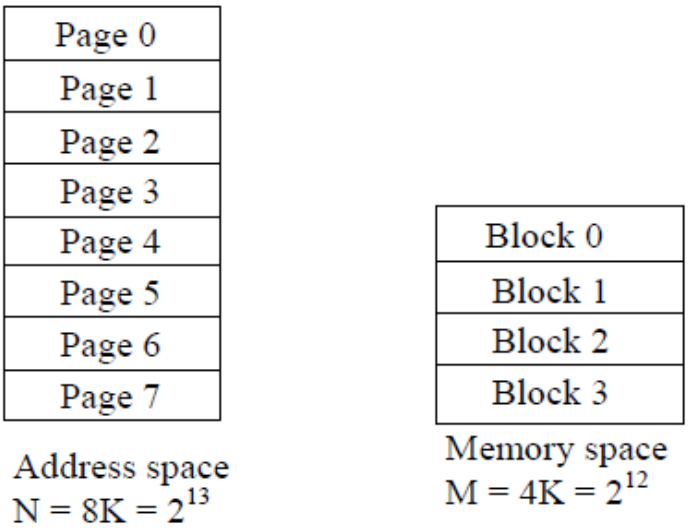


Figure: Address space and Memory space split into groups of 1K words

Consider a computer with an address space of 8K and a memory space of 4K. If we split each into groups of 1K words we obtain eight pages and four blocks as shown in figure above. At any given time, up to four pages of address space may reside in main memory in any one of the four blocks.

The mapping from address space to memory space is facilitated if each virtual address is considered to be represented by two numbers: a page number address and a line within the page. In a computer with 2^p words per page, p bits are used to specify a line address and the remaining high-order bits of the virtual address specify the page number. In the example of figure below, a virtual address has 13 bits. Since each page consists of 2^{10} = 1024 words, the high-order three bits of a virtual address will specify one of the eight pages and the low-order 10 bits give the line address within the page. Note that the line address in address space and memory space is the same; the only mapping required is from a page number to a block number.

The organization of the memory mapping table in a paged system is shown in figure below. The memory-page table consists of eight words, one for each page. The address in the page table denotes the page number and the content of the word gives the block number where that page is stored in main memory.

b i i z a y @ g m a i l . c o m

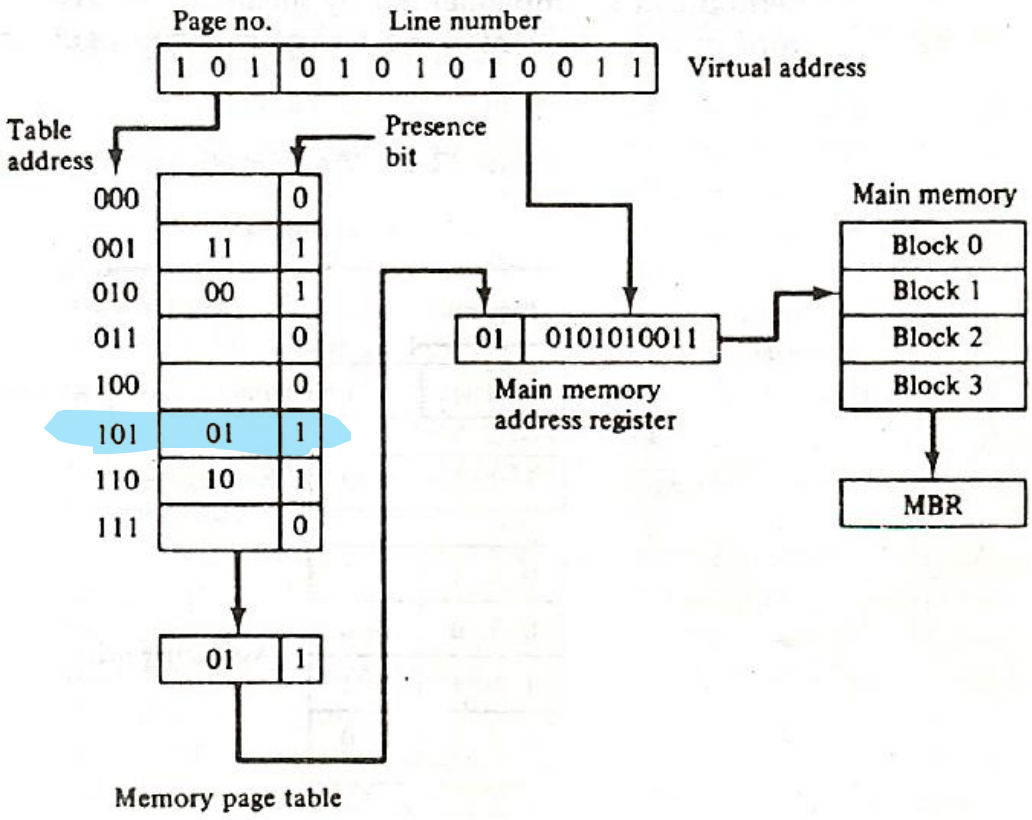


Figure: Memory table in a paged system

The table shows that pages 1, 2, 5 and 6 are now available in main memory in blocks 3, 0, 1, and 2, respectively. A presence bit in each location indicates whether the page has been transferred from auxiliary memory into main memory. A 0 in the presence bit indicates that this page is not available in main memory. The CPU references a word in memory with a virtual address of 13 bits. The three high-order bits of the virtual address specify a page number and also an address for the memory-page table.

The content of the word in the memory page table at the page number address is read out into the memory table buffer register. If the presence bit is a 1, the block number thus read is transferred to the two high-order bits of the main memory address register. The line number from the virtual address is transferred into the 10 low order bits of the memory address register. A read signal to main memory transfers the content of the word to the main memory buffer register ready to be used by the CPU. If the presence bit in the word read from the page table is 0, it signifies that the content of the word referenced by the virtual address does not reside in main memory. A call to the operating system is then generated to fetch the required page from auxiliary memory and place it into main memory before resuming computation.

Associative Memory Page Table

Consider an address space of 1024K words and memory space of 32K words. If each page or block contains 1K words, the number of pages is 1024 and the number of blocks 32. The capacity of the memory-page table must be 1024 words and only 32 locations may have a presence bit equal to 1. At any given time, at least 992 locations will be empty and not in use.

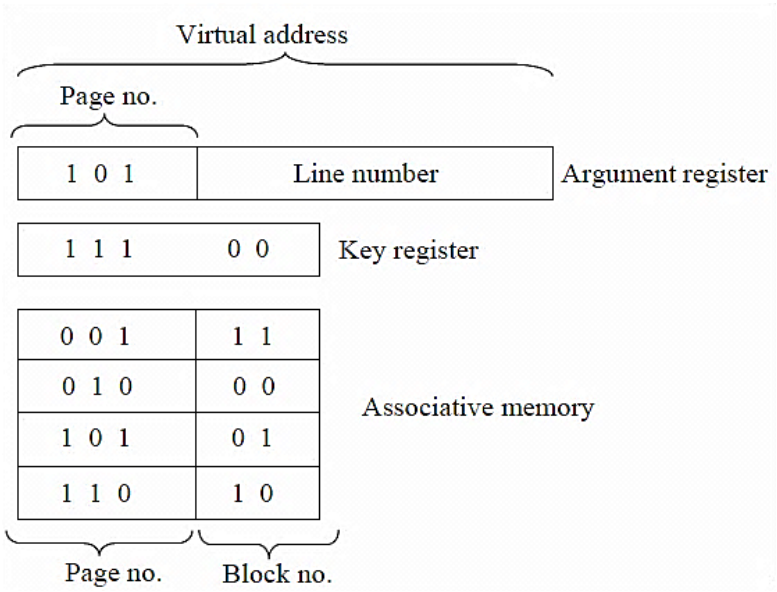


Figure: An Associative memory page table

A more efficient way to organize the page table would be to construct it with a number of words equal to the number of blocks in main memory. In this way the size of the memory is reduced and each location is fully utilized. This method can be implemented by means of an associative memory with each word in memory containing a page number together with its corresponding block number. The page field in each word is compared with the page number in the virtual address. If a match occurs, the word is read from memory and its corresponding block number is extracted.

Each entry in the associative memory array consists of two fields. The first three bits specify a field for storing the page number. The last two bits constitute a field for storing the block number. The virtual address is placed in the argument register. The page number bits in the argument are compared with all page numbers in the page field of the associative memory. If the page number is found, the 5-bit word is read out from memory. The corresponding block number, being in the same word, is transferred to the main memory address register. If no match occurs, a call to the operating system is generated to bring the required page from auxiliary memory.

Page Replacement

A virtual memory system is a combination of hardware and software techniques. The memory management software system handles all the software operations for the efficient utilization of memory space. It must decide:

(1) Which page in main memory ought to be removed to make room for a new page?
(2) When a new page is to be transferred from auxiliary memory to main memory?
(3) Where the page is to be placed in main memory?

The hardware mapping mechanism and the memory management software together constitute the architecture of a virtual memory. When a program starts execution, one or more pages are transferred into main memory and the page table is set to indicate their position. The program is executed from main memory until it attempts to reference a page that is still page fault in auxiliary

memory. This condition is called ***page fault.*** When page fault occurs, the execution of the present program is suspended until the required page is brought into main memory. Since loading a page from auxiliary memory to main memory is basically an I/O operation, the operating system assigns this task to the I/O processor. In the meantime, control is transferred to the next program in memory that is waiting to be processed in the CPU. Later, when the memory block has been assigned and the transfer completed, the original program can resume its operation.

The goal of a replacement policy is to try to remove the page least likely to be referenced in the immediate future. Two of the most common replacement algorithms used is the first-in first-out (FIFO) and the least recently used (LRU). The FIFO algorithm selects for replacement the page that has been in memory the longest time. Each time a page is loaded into memory, its identification number is pushed into a FIFO stack. FIFO will be full whenever memory has no more empty blocks. When a new page must be loaded, the page least recently brought in is removed. The page to be removed is easily determined because its identification number is at the top of the FIFO stack. The FIFO replacement policy has the advantage of being easy to implement. It has the disadvantage that under certain circumstances pages are removed and loaded from memory too frequently.

The LRU policy is more difficult to implement but has been more attractive on the assumption that the least recently used page is a better candidate for removal than the least recently loaded page as in FIFO. The LRU algorithm can be implemented by associating a counter with every page that is in main memory. When a page is referenced, its associated counter is set to zero. At fixed intervals of time, the counters associated with all pages presently in memory are incremented by 1. The least recently used page is the page with the highest count. The counters are often called aging registers, as their count indicates their age, that is, how long ago their associated pages have been referenced.

Memory Management Hardware

A memory management system is a collection of hardware and software procedures for managing the various programs residing in memory. The memory management software is part of an overall operating system available in many computers. Here we are concerned with the hardware unit associated with the memory management system.

The basic components of a memory management unit are:

1. A facility for dynamic storage relocation that maps logical memory references into physical memory addresses
2. A provision for sharing common programs stored in memory by different users
3. Protection of information against unauthorized access between users and preventing users from changing operating system functions

The fixed page size used in the virtual memory system causes certain difficulties with respect to program size and the logical structure of programs. It is more convenient to divide programs and segment data into logical parts called segments. A segment is a set of logically related instructions or data elements associated with a given name. Segments may be generated by the programmer or by the operating system. Examples of segments are a subroutine, an array of data, a table of symbols, or a user's program.

Segmented Page Mapping

The length of each segment is allowed to grow and contract according to the needs of the program being executed. One way of specifying the length of a segment is by associating with it a number of equal-size pages. The logical address is partitioned into three fields. The segment field specifies a segment number. The page field specifies the page within the segment and the word field gives the specific word within the page. A page field of k bits can specify up to 2k pages.

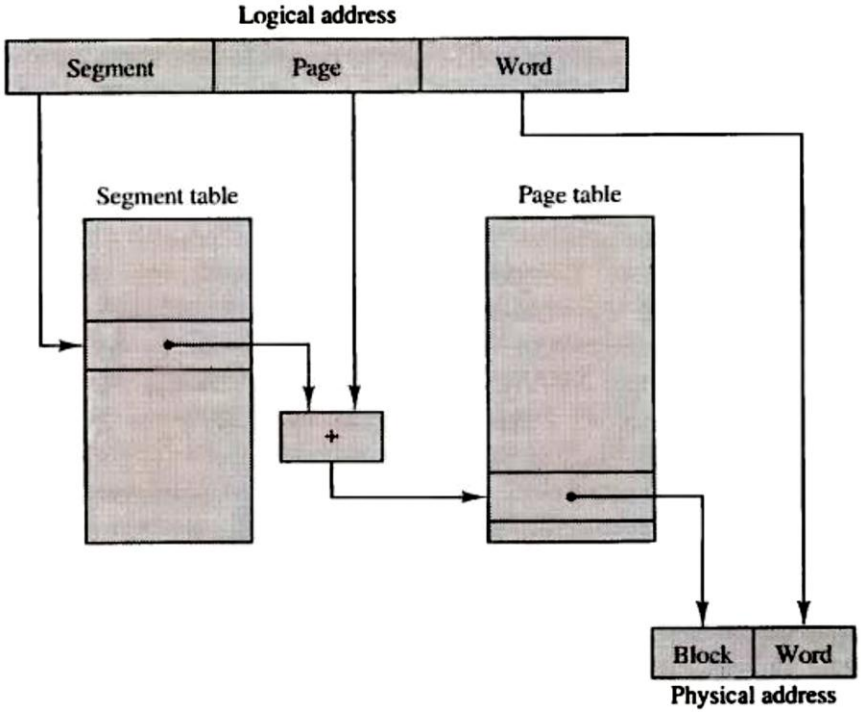


Figure: Logical to Physical Address Mapping

Memory Protection

Memory protection can be assigned to the physical address or the logical address. The protection of memory through the physical address can be done by assigning to each block in memory a number of protection bits that indicate the type of access allowed to its corresponding block. Every time a page is moved from one block to another it would be necessary to update the block protection bits. A much better place to apply protection is in the logical address space rather than the physical address space. This can be done by including protection information within the segment table or segment register of the memory management hardware.

The protection field in a segment descriptor specifies the access rights available to the particular segment. In a segmented-page organization, each entry in the page table may have its own protection field to describe the access rights of each page. The protection information is set into the descriptor by the master control program of the operating system.

Some of the access rights of interest that are used for protecting the programs residing in memory are:

1. Full read and write privileges
2. Read only (write protection)
3. Execute only (program protection)
4. System only (operating system protection)

Full read and write privileges are given to a program when it is executing its own instructions. Write protection is useful for sharing system programs such as utility programs and other library routines. These system programs are stored in an area of memory where they can be shared by many users. They can be read by all programs, but no writing is allowed. This protects them from being changed by other programs.

References:

1. M. Morris Mano, "Computer System Architecture", Pearson, 3rd Ed, 2004.
2. W. Stallings, "Computer Organization and Architecture – Designing for Performance", Prentice Hall of India, 7th Ed, 2007

Assignments:

(1) Explain the memory hierarchy in the computer systems.
(2) What is memory address mapping? Explain the concept with the help of ROM chips.
(3) What is the concept of Associative memory? Explain.
(4) What is cache memory? How it is fast as compared to conventional memory?
(5) What is the concept of page replacement?
(6) Differentiate between associative page table and replacement. (T.U. 2066)
(7) What is memory management hardware? Explain. (T.U. 2067)
(8) What do you mean by memory mapping? Explain. (T.U. 2068)
(9) What do you mean by memory organization? Explain the memory management hardware with example. (T.U. 2068)
(10) What do you mean by memory system? Explain the characteristics of memory systems of computer. (T.U. 2069)
(11) What is virtual memory? What are the major differences between address space and memory space? (T.U. 2069)
(12) What are the key characteristics of computer memory system? Explain. (T.U. 2070)
(13) What is the main role of memory management hardware? Explain. (T.U. 2070)
(14) Write short notes on the following:
   a. Memory space (T.U. 2066)
   b. Address space (T.U. 2066)
   c. Sequential memory hierarchy (T.U. 2067)
   d. Random memory hierarchy (T.U. 2067)
   e. Memory Protection (T.U. 2070)
   f. Address Mapping (T.U. 2070)

A Gentle Advice:

Please go through your text books and reference books for detail study!!! Thank you all.

Notes Compiled By:

Bijay Mishra
biizay.blogspot.com
9813911076 or 9841695609