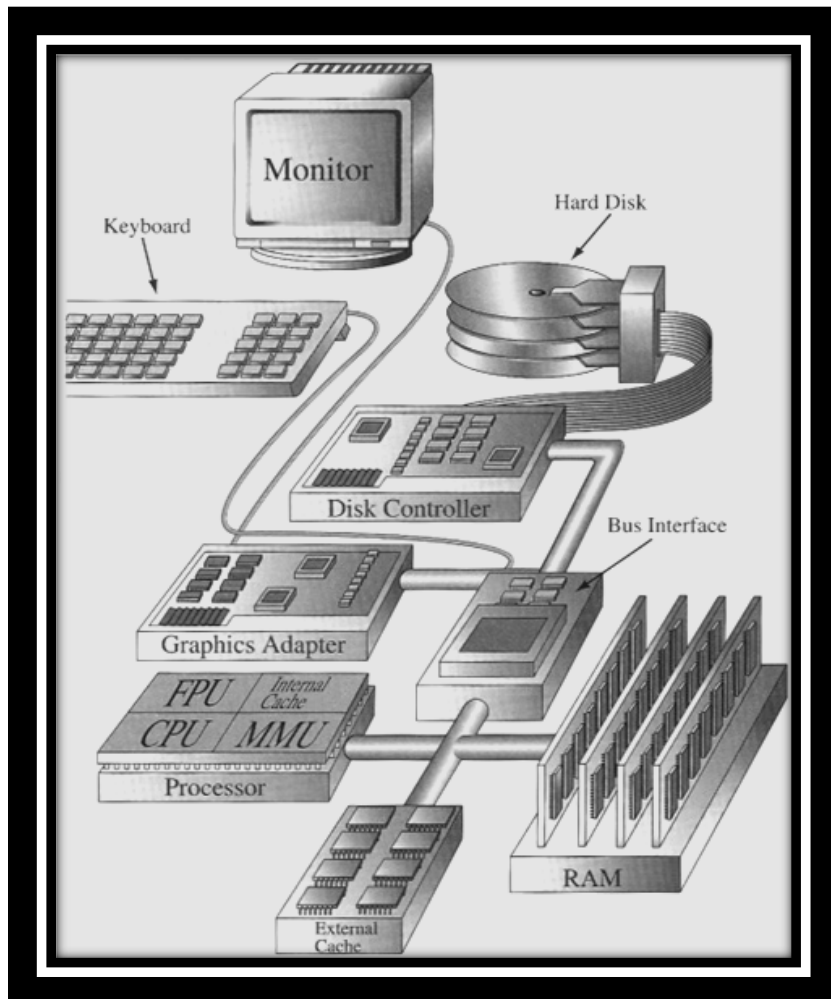


Condensed Notes

On

Computer Architecture

(According to BScCSIT syllabus, TU)



By Bishnu Rawal

More about me >> www.bishnurawal.com.np

Intentionally left blank...

Unit 1

Data Representation

Number System

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)
- Octal Number System (r = 8)
- Decimal Number System (r = 10)
- Hexadecimal Number system (r = 16)

Number system conversions (quite easy guys, do it on your own)

Decimal Representation

We can normally represent decimal numbers in one of following two ways

- By converting into binary
- By using BCD codes

By converting into binary

Advantage

- Arithmetic and logical calculation becomes easy. Negative numbers can be represented easily.

Disadvantage

- At the time of input conversion from decimal to binary is needed and at the time of output conversion from binary to decimal is needed.

Therefore this approach is useful in the systems where there is much calculation than input/output.

By using BCD codes

Decimal number	Binary-coded decimal (BCD) number	
0	0000	↑ Code for one decimal digit ↓
1	0001	
2	0010	
3	0011	
4	0100	
5	0101	
6	0110	
7	0111	
8	1000	
9	1001	
10	0001 0000	
20	0010 0000	
50	0101 0000	
99	1001 1001	
248	0010 0100 1000	

Disadvantage

- Arithmetic and logical calculation becomes difficult to do. Representation of negative numbers is tricky.

Advantage

- At the time of input conversion from decimal to binary and at the time of output conversion from binary to decimal is not needed.

Therefore, this approach is useful in the systems where there is much input/output than arithmetic and logical calculation.

Alphanumeric Representation

Alphanumeric character set is a set of elements that includes the 10 decimal digits, 26 letters of the alphabet and special characters such as \$, %, + etc. The standard alphanumeric binary code is ASCII(American Standard Code for Information Interchange) which uses 7 bits to code 128 characters (both uppercase and lowercase letters, decimal digits and special characters).

Character	Binary code	Character	Binary code
A	100 0001	0	011 0000
B	100 0010	1	011 0001
C	100 0011	2	011 0010
D	100 0100	3	011 0011
E	100 0101	4	011 0100
F	100 0110	5	011 0101
G	100 0111	6	011 0110
H	100 1000	7	011 0111
I	100 1001	8	011 1000
J	100 1010	9	011 1001
K	100 1011		
L	100 1100		
M	100 1101	space	010 0000
N	100 1110	.	010 1110
O	100 1111	(010 1000
P	101 0000	+	010 1011
Q	101 0001	\$	010 0100
R	101 0010	*	010 1010
S	101 0011)	010 1001
T	101 0100	-	010 1101
U	101 0101	/	010 1111
V	101 0110	,	010 1100
W	101 0111	=	011 1101
X	101 1000		
Y	101 1001		
Z	101 1010		

NOTE: Decimal digits in ASCII can be converted to BCD by removing the three higher order bits, 011.

Complements

(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 of 835_{10} is 164_{10} (Rule: $(10^n - 1) - N$)
- 1's complement of 1010_2 is 0101_2 (bit by bit complement operation)

r's Complement

r's complement of a number N is defined as $r^n - N$

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 of 835_{10} is $164_{10} + 1 = 165_{10}$
- 2's complement of 1010_2 is $0101_2 + 1 = 0110_2$

Subtraction of unsigned Numbers (Using complements)

When subtraction is implemented in digital hardware, borrow-method is found to be less efficient than the method that uses complements. The subtraction of two n-digit unsigned numbers $M-N$ ($N \neq 0$) in base r can be done as follows:

1. Add the minuend M to the r 's complement of the subtrahend N . This performs $M + (r^n - N) = M - N + r^n$.
2. If $M \geq N$, the sum will produce an end carry r^n which is discarded, and what is left is the result $M - N$.
3. If $M < N$, the sum does not produce an end carry and is equal to $r^n - (N - M)$, which is the r 's complement of $(N - M)$. To obtain the answer in a familiar form, take the r 's complement of the sum and place a negative sign in front.

Consider, for example, the subtraction $72532 - 13250 = 59282$. The 10's complement of 13250 is 86750. Therefore:

$$\begin{array}{r} M = 72532 \\ 10\text{'s complement of } N = +86750 \\ \text{Sum} = \underline{159282} \\ \text{Discard end carry } 10^5 = -100000 \\ \text{Answer} = \underline{59282} \end{array}$$

Now consider an example with $M < N$. The subtraction $13250 - 72532$ produces negative 59282. Using the procedure with complements, we have

$$\begin{array}{r} M = 13250 \\ 10\text{'s complement of } N = +27468 \\ \text{Sum} = \underline{40718} \end{array}$$

There is no end carry, so answer is negative 59282 = 10's complement of 40718.

Subtraction with complements is done with binary numbers in similar manner using same procedure outlined above.

NOTE: negative numbers are recognized by the absence of the end carry and the complemented result.

Fixed-Point Representation

Positive integers, including 0 can be represented as unsigned numbers. However for negative numbers, we use convention of representing left most bit of a number as a sign-bit: 0 for positive and 1 for negative. In addition, to represent fractions, integers or mixed integer-fraction numbers, number may have a binary (or decimal) point. There are two ways of specifying the position of a binary point in a register:

- by employing a floating-point notation.(discussed later)
- by giving it a fixed position (hence the name)
 - A binary point in the extreme left of the register to make the stored number a fraction.
 - A binary point in the extreme right of a register to make the stored number an integer.

Integer representation

There is only one way of representing positive numbers with sign-bit 0 but when number is negative the sign is represented by 1 and rest of the number may be represented in one of three possible ways:

- Signed magnitude representation

- Signed 1's complement representation
- Signed 2's complement representation

Signed magnitude representation of a negative number consists of the magnitude and a negative sign. In other two representations, the negative number is represented in either 1's or 2's complement of its positive value.

Examples: Representing negative numbers

Signed Magnitude Notation

- Complement only the sign bit
- Example:
 - +9 → 0 001001
 - 9 → 1 001001

Signed 1's complement Notation

- Complement *all* the bits including sign bit.
- Example:
 - +9 → 0 001001
 - 9 → 1 110110

Signed 2's complement Notation

- Take the 2's complement of the number, *including* its sign bit
- Example:
 - +9 → 0 001001
 - 9 → 1 110111

Arithmetic addition and subtraction of signed numbers

Addition

Mostly signed 2's complement system is used. So, in this system only addition and complementation is used. Procedure: add two numbers including sign bit and discard any carry out of the sign bit position. (note: negative numbers initially be in the 2's complement and that if the sum obtained after the addition is negative, it is in 2's complement form).

+6 0000110	-6 1111010
+13 00001101	+13 00001101
+19 00010011	+7 00000111
+6 00000110	-6 1111010
-13 11110011	-13 11110011
-7 11111001	-19 11101101

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.

Subtraction

Subtraction of two signed binary numbers is done as: take the 2's complement of the subtrahend (including the sign bit) and add it to the minuend (including the sign-bit). The carry out of the sign bit position is discarded.

Idea: subtraction operation can be changed to the addition operation if the sign of the subtrahend is changed:

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

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

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

-6 → 11111010
 -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 register 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. For example: two signed binary numbers +70 and +80 are stored in two 8-bit registers.

$$\begin{array}{r} \text{carries: } 0 \ 1 \\ +70 \ 0 \ 1000110 \\ +80 \ 0 \ 1010000 \\ \hline +150 \ 1 \ 0010110 \end{array} \qquad \begin{array}{r} \text{carries: } 1 \ 0 \\ -70 \ 1 \ 0111010 \\ -80 \ 1 \ 0110000 \\ \hline -150 \ 0 \ 1101010 \end{array}$$

Since the sum of numbers 150 exceeds the capacity of the register (since 8-bit register 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.

Hey boys, consider example of above 8-bit register, 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 can not 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 = $f(\text{binary code used to represent each decimal digit})$. 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: $4385 = (0100 \ 0011 \ 1000 \ 0101)_{\text{BCD}}$

While using BCD representation,

Disadvantages:

- wastage of memory (Viz. binary equivalent of 4385 uses less bits than its BCD representation)
- Circuits for decimal arithmetic are quite complex.

Advantages:

- 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]

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

Floating-Point Representation

The floating-point representation of a number has two parts: *mantissa* and *exponent*

Mantissa : represents a signed, fixed-point number. May be a fraction or an integer

Exponent: designates the position of the decimal (or binary) point

Example1: decimal number +6132.789 is represented in floating-point as:

Fraction	exponent
+0.6132789	+04

Floating-point is interpreted to represent a number in the form: $m * r^e$. Only the mantissa m and exponent e are physically represented in registers. The radix r and the radix-point position are always assumed.

Example2: binary number +1001.11 is represented with an 8-bit fraction and 6-bit exponent as,

Fraction	exponent
+01001110	000100

or equivalently,

$$m * 2^e = +(.1001110)_2 * 2^{+4}$$

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.

Other Binary codes

Most common type of binary-coded data found in digital computer is explained before. A few additional binary codes used in digital systems (for special applications) are explained below.

Gray code

The reflected binary or Gray code is used to represent digital data converted from analog information. Gray code changes by only one bit as it sequences from one number to the next.

Binary code	Decimal equivalent	Binary code	Decimal equivalent
0000	0	1100	8
0001	1	1101	9
0011	2	1111	10
0010	3	1110	11
0110	4	1010	12
0111	5	1011	13
0101	6	1001	14
0100	7	1000	15

Table: 4-bit Gray code

Weighted code (2421)

2421 is an example of weighted code. In this, corresponding bits are multiplied by the weights indicated and the sum of the weighted bits gives the decimal digit.

Example: 1101 when weighted by the respective digits 2421 gives $2*1+4*1+2*0+1*1 = 7$.

NOTE: Ladies and gentlemen...☺, you have already studied about BCD codes. BCD can be assigned the weights 8421 and for this reason it is sometimes called 8421 code.

Excess-3 codes

The excess-3 code is a decimal code used in older computers. This is un-weighted code.

Excess-3 code = BCD binary equivalent + 3(0011)

NOTE: excess-n code is possible adding n to the corresponding BCD equivalent.

Excess-3 Gray

In ordinary Gray code, the transition from 9 back to 0 involves a change of three bits (from 1101 to 0000). To overcome this difficulty, we start from third entry 0010 (as first number) up to the twelfth entry 1010, there by change of only one bit is possible upon transition from 1010 to 0010. Since code has been shifted up three numbers, it is called the excess-3 Gray.

Decimal digit	BCD 8421	2421	Excess-3	Excess-3 gray
0	0000	0000	0011	0010
1	0001	0001	0100	0110
2	0010	0010	0101	0111
3	0011	0011	0110	0101
4	0100	0100	0111	0100
5	0101	1011	1000	1100
6	0110	1100	1001	1101
7	0111	1101	1010	1111
8	1000	1110	1011	1110
9	1001	1111	1100	1010
	1010	0101	0000	0000
Unused bit	1011	0110	0001	0001
combi-	1100	0111	0010	0011
nations	1101	1000	1101	1000
	1110	1001	1110	1001
	1111	1010	1111	1011

Table: 4 different binary codes for the decimal digit

Error Detection Codes

Binary information transmitted through some form of communication medium is subject to external noise that could change bits from 1 to 0 and vice versa. An error detection code is a binary code that detects digital errors during transmission. The detected errors can not be corrected but their presence is indicated. The most common error detection code used is the *parity bit*. A parity bit(s) is an extra bit that is added with original message to detect error in the message during data transmission.

Even Parity

One bit is attached to the information so that the total number of 1 bits is an even number.

Message	Parity
1011001	0
1010010	1

Odd Parity

One bit is attached to the information so that the total number of 1 bits is an odd number.

Message	Parity
1011001	1
1010010	0

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

Parity Checker

Considers original message as well as parity bit

$$e = p \oplus x \oplus y \oplus z$$

$e = 1 \Rightarrow$ No. of 1's in pxyz is even \Rightarrow Error in data

$e = 0 \Rightarrow$ No. of 1's in pxyz is odd \Rightarrow Data is error free

Circuit diagram for parity generator and parity checker

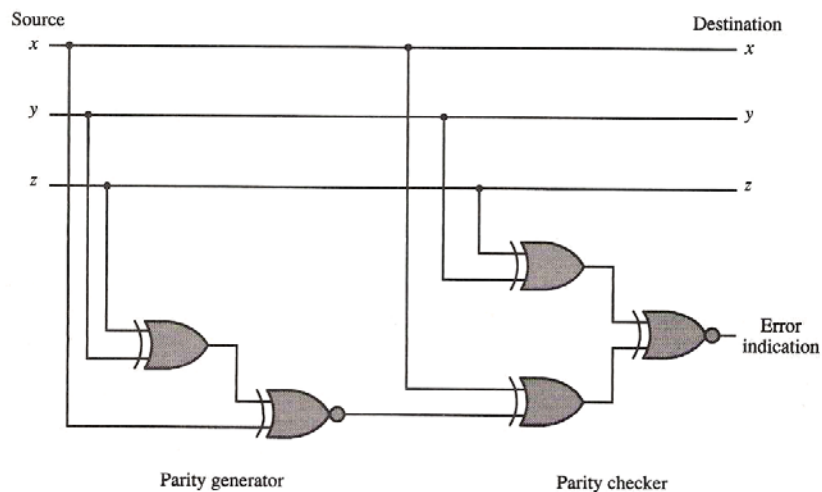


Fig: Error detection with odd parity bit.

EXERCISES: Text Book chapter 3 → 3.15, 3.17, 3.22, 3.26

3.15 (Solution)

(a)	(b)	(c)	(d)
11010	11010	000100	1010100
+10000	10011	010000	0101100
1) 01010	1) 01101	0) 010100	1) 0000000
(26-16=10)	(26-13=13)	-101100 ↓ (4-48=-44)	(84-84=0)

3.17 HINT: see notes

3.22 (Solution)

(a) BCD 1000 0110 0010 0000

(b) XS-3 1011 1001 0101 0011

(c) 2421 1110 1100 0010 0000

(d) Binary 10000110101100 (8192+256+128+32+8+4)

3.26 (Solution)

Same as in Fig. 3-3 but without the complemented circles in the outputs of the gates.

$$P = x \oplus y \oplus z$$

$$\text{Error} = x \oplus y \oplus z \oplus P$$

Unit 2

Microoperations

Combinational and sequential circuits can be used to create simple digital systems. These are the low-level building blocks of a digital computer. The operations on the data in registers are called microoperations. Examples of micro-operations are

- Shift
- Load
- Clear
- Increment

Alternatively we can say that an elementary operation performed during one clock pulse on the information stored in one or more registers is called micro-operation. The result of the operation may replace the previous binary information of the register or may be transferred to another register. Register transfer language can be used to describe the (sequence of) micro-operations.

Microoperation types

The microoperations most often encountered in digital computers are classified into 4 categories:

1. Register transfer microoperations
2. Arithmetic microoperations
3. Logic microoperations
4. Shift microoperations

1. Register transfer microoperations

Registers are designated by capital letters, sometimes followed by numbers (e.g., A, R13, IR). Often the names indicate function:

MAR	memory address register
PC	program counter
IR	instruction register

Information transfer from one register to another is described in symbolic form by replacement operator. The statement " $R2 \leftarrow R1$ " denotes a transfer of the content of the R1 into register R2.

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: $R2 \leftarrow R1$ i.e. if (P = 1) then ($R2 \leftarrow R1$)

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: $R3 \leftarrow R5, MAR \leftarrow IR$

2. Arithmetic microoperations

- The basic arithmetic microoperations are
 - Addition
 - Subtraction
 - Increment
 - Decrement
- The additional arithmetic microoperations are
 - Add with carry
 - Subtract with borrow
 - Transfer/Load

Summary of typical arithmetic microoperations

Symbolic designation	Description
$R3 \leftarrow R1 + R2$	Contents of R1 plus R2 transferred to R3
$R3 \leftarrow R1 - R2$	Contents of R1 minus R2 transferred to R3
$R2 \leftarrow \overline{R2}$	Complement the contents of R2 (1's complement)
$R2 \leftarrow \overline{R2} + 1$	2's complement the contents of R2 (negate)
$R3 \leftarrow R1 + \overline{R2} + 1$	R1 plus the 2's complement of R2 (subtraction)
$R1 \leftarrow R1 + 1$	Increment the contents of R1 by one
$R1 \leftarrow R1 - 1$	Decrement the contents of R1 by one

Binary Adder

To implement the add microoperation with hardware, we need the registers 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.

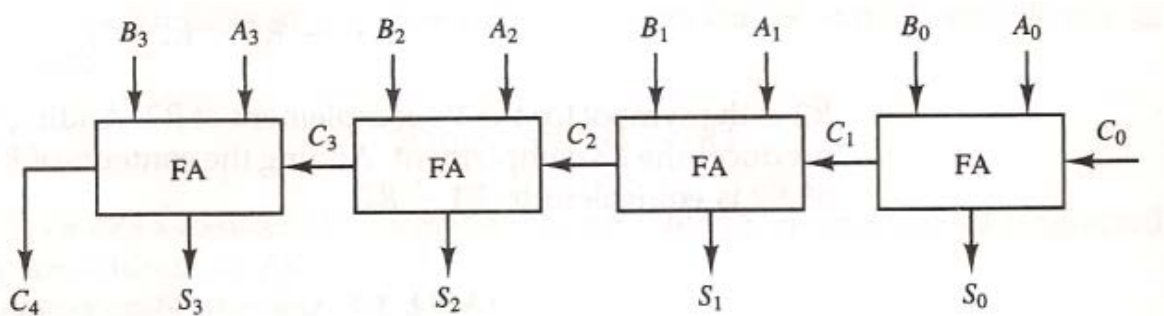


Fig.: 4-bit binary adder

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.

Binary Subtractor

The subtraction $A - B$ can be done by taking the 2's complement of B and adding to A . It means if we use the inverters to make 1's complement of B (connecting each B_i to an inverter) and then add 1 to the least significant bit (by setting carry C_0 to 1) of binary adder, then we can make a binary subtractor.

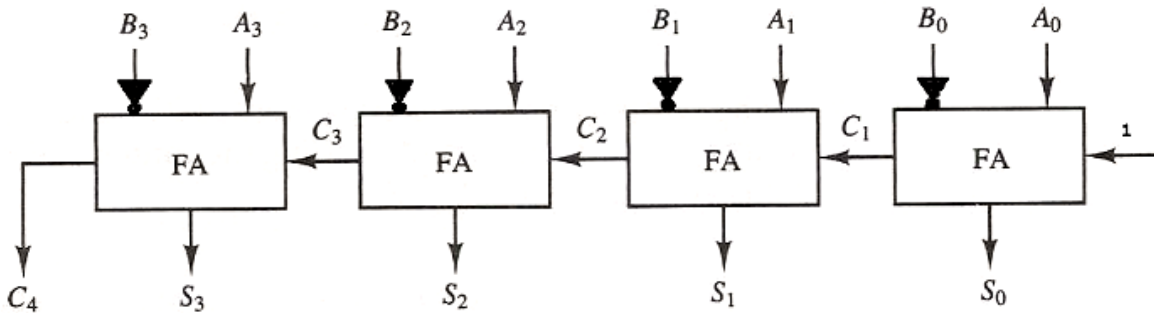


fig.: 4-bit binary subtractor

Binary Adder-Subtractor

Question: How binary adder and subtractor can be accommodated into a single circuit? explain.

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

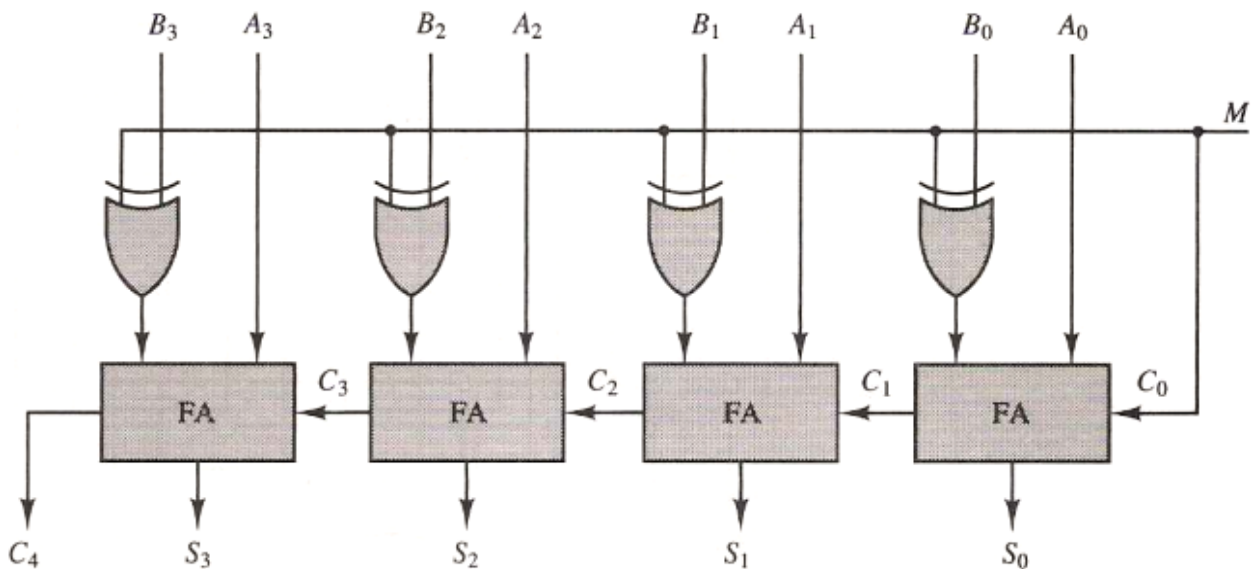


Fig.: 4-bit adder-subtractor

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 .

- 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 $A+B$.
- 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

The increment microoperation 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 microoperation can be done with a combinational circuit (half-adders connected in cascade) independent of a particular register.

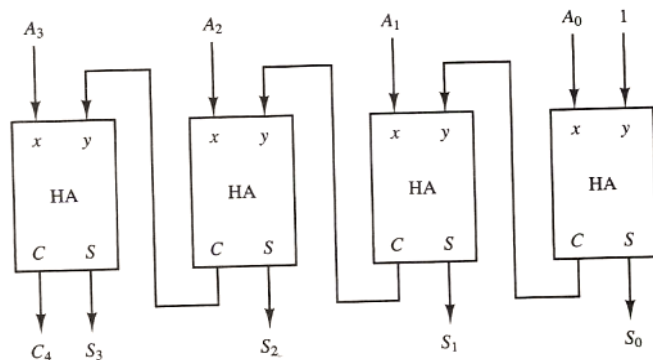


Fig.: 4-bit binary Incrementer

Arithmetic Circuit

The arithmetic microoperations 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 microoperations listed in the table below:

Select			Input Y	Output $D = A + Y + C_{in}$	Microoperation
S ₁	S ₀	C _{in}			
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

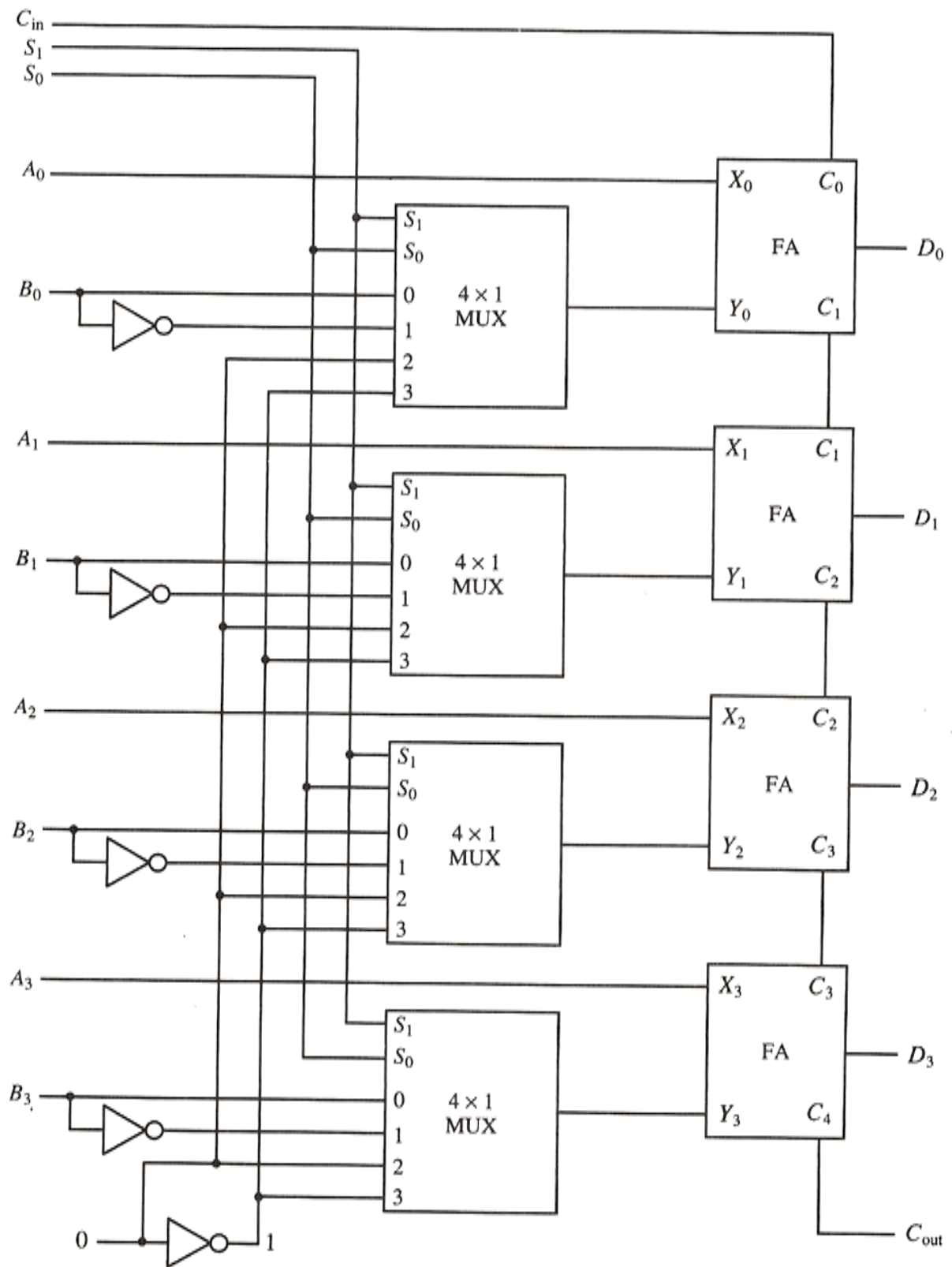


Fig: 4-bit arithmetic circuit

3. Logic microoperations

Question: *What do you mean by Logic microoperations? Explain with its applications.*

Question: *How Logic microoperations can be implemented with hardware?*

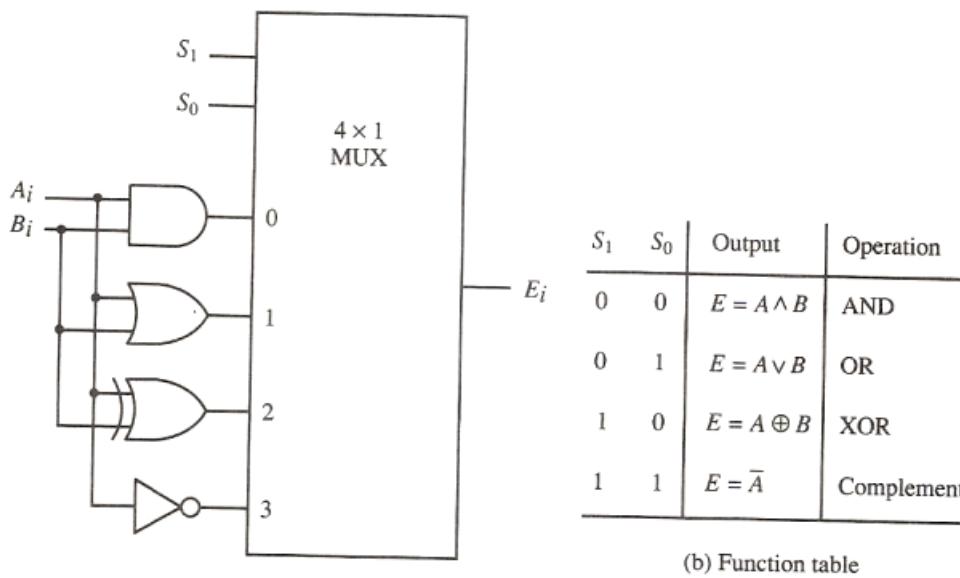
Logic microoperations are bit-wise operations, i.e., they work on the individual bits of data. Useful for bit manipulations on binary data and for making logical decisions based on the bit value. There are, in principle, 16 different logic functions that can be defined over two binary input variables. However, most systems only implement four of these

- AND (\wedge), OR (\vee), XOR (\oplus), Complement/NOT

The others can be created from combination of these four functions.

Hardware implementation

Hardware implementation of logic microoperations requires that logic gates be inserted be each bit or pair of bits in the registers to perform the required logic operation.



(a) Logic diagram

(b) Function table

Applications of Logic Microoperations

Logic microoperations can be used to manipulate individual bits or a portion of a word in a register. Consider the data in a register A. Bits of register B will be used to modify the contents of A.

- | | |
|------------------------|--------------------------------|
| - Selective-set | $A \leftarrow A + B$ |
| - Selective-complement | $A \leftarrow A \oplus B$ |
| - Selective-clear | $A \leftarrow A \cdot B'$ |
| - Mask (Delete) | $A \leftarrow A \cdot B$ |
| - Clear | $A \leftarrow A \oplus B$ |
| - Insert | $A \leftarrow (A \cdot B) + C$ |
| - Compare | $A \leftarrow A \oplus B$ |

Selective-set

In a selective set operation, the bit pattern in B is used to *set* certain bits in A.

$$\begin{array}{r} 1100 \quad A_t \\ 1010 \quad B \\ \hline 1110 \quad A_{t+1} \quad (A \leftarrow A + B) \end{array}$$

Bits in register A are set to 1 when there are corresponding 1's in register B. It does not affect the bit positions that have 0's in B.

Selective-complement

In a selective complement operation, the bit pattern in B is used to *complement* certain bits in A.

$$\begin{array}{r} 1100 \quad A_t \\ 1010 \quad B \\ \hline 0110 \quad A_{t+1} \quad (A \leftarrow A \oplus B) \end{array}$$

If a bit in B is 1, corresponding position in A get complemented from its original value, otherwise it is unchanged.

Selective-clear

In a selective clear operation, the bit pattern in B is used to *clear* certain bits in A.

$$\begin{array}{r} 1100 \quad A_t \\ 1010 \quad B \\ \hline 0100 \quad A_{t+1} \quad (A \leftarrow A \cdot B') \end{array}$$

If a bit in B is 1, corresponding position in A is set to 0, otherwise it is unchanged.

Mask Operation

In a mask operation, the bit pattern in B is used to *clear* certain bits in A.

$$\begin{array}{r} 1100 \quad A_t \\ 1010 \quad B \\ \hline 1000 \quad A_{t+1} \quad (A \leftarrow A \cdot B) \end{array}$$

If a bit in B is 0, corresponding position in A is set to 0, otherwise it is unchanged. This is achieved logically ANDing the corresponding bits of A and B.

Clear Operation

In clear operation, if the bits in the same position in A and B same, that bit in A is cleared (putting 0 there), otherwise same bit in A is set(putting 1 there). This operation is achieved by exclusive-OR microoperation.

$$\begin{array}{r} 1100 \quad A_t \\ 1010 \quad B \\ \hline 0110 \quad A_{t+1} \quad (A \leftarrow A \oplus B) \end{array}$$

Insert Operation

An insert operation is used to introduce a specific bit pattern into A register, leaving the other bit positions unchanged.

This is done as

- A **mask (ANDing)** operation to clear the desired bit positions, followed by
- An **OR** operation to introduce the new bits into the desired positions
- Example
 - » Suppose you want to introduce 1010 into the low order four bits of A:

1101 1000 1011 0001	A (Original)
1101 1000 1011 1010	A (Desired)
1101 1000 1011 0001	A (Original)
1111 1111 1111 0000	B (Mask)

1101 1000 1011 0000	A (Intermediate)
0000 0000 0000 1010	Added bits

1101 1000 1011 1010	A (Desired)

4. Shift microoperations

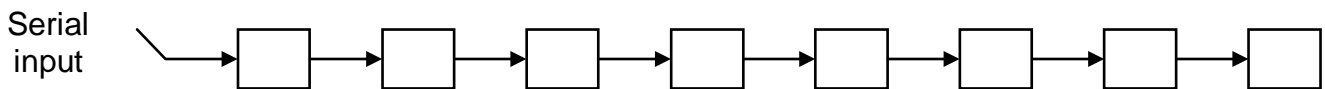
Question: What do you mean by shift microoperations? Explain its types.

Question: Is there a possibility of Overflow during arithmetic shift? If yes, how it can be detected?

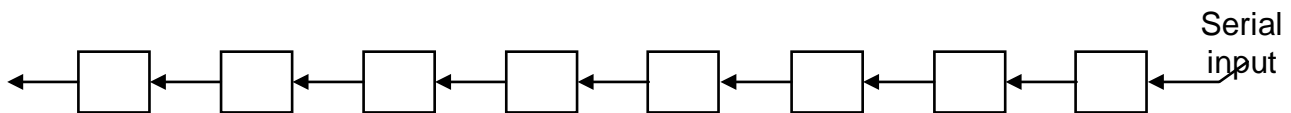
Shift microoperations are used for serial transfer of data. They are also used in conjunction with arithmetic, logic and other data processing operations. The contents of a register can be shifted left or right. There are three types of shifts

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

Right Shift Operation



Left shift operation



1. Logical shift

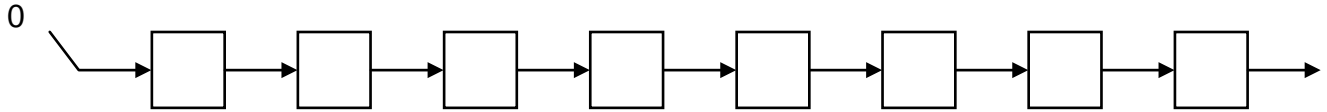
A logical shift is one that transfers 0 through the serial input. In a Register Transfer Language, the following notation is used

- *shl* for a logical shift left
- *shr* for a logical shift right

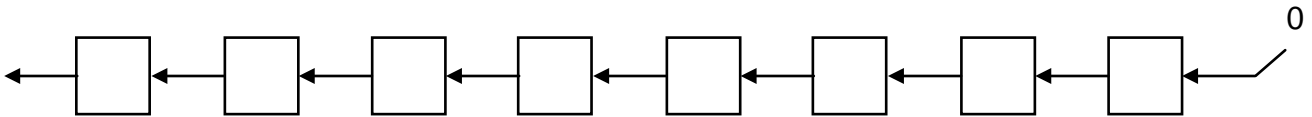
Examples:

$R2 \leftarrow shr R2$

$R3 \leftarrow shl\ R3$



Logical right shift (shr)

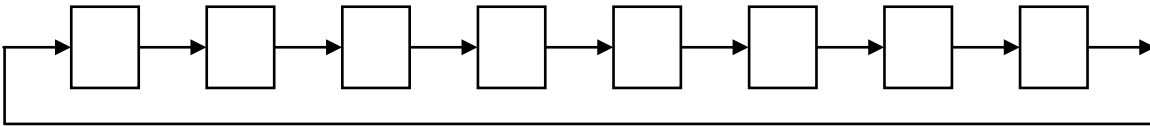


Logical left shift (shl)

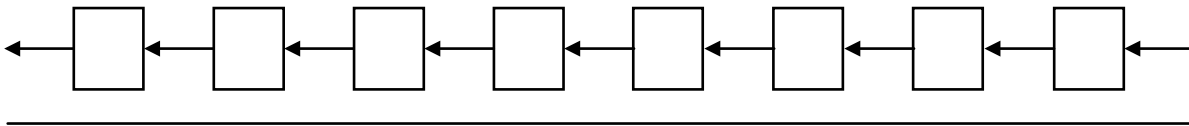
2. Circular Shift (rotate operation)

Circular-shift circulates the bits of the register around the two ends without the loss of information.

Right circular shift operation



Left circular shift operation:



In a RTL, the following notation is used

- *cil* for a circular shift left
- *cir* for a circular shift right
- Examples:

$R2 \leftarrow cir\ R2$

$R3 \leftarrow cil\ R3$

3. Arithmetic shift

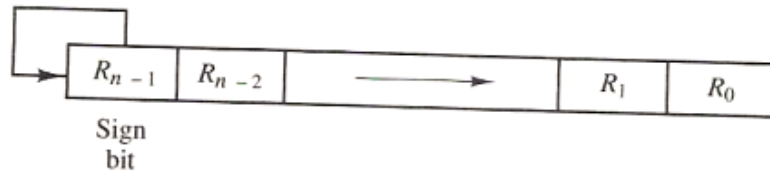
An arithmetic shift is meant for signed binary numbers (integer). An arithmetic left shift multiplies a signed number by 2 and an arithmetic right shift divides a signed number by 2. Arithmetic shifts must leave the sign bit unchanged because the sign of the number remains the same when it is multiplied or divided by 2. The left most bit in a register holds a sign bit and remaining hold the number. Negative numbers are in 2's complement form.

In a Register Transfer Language, the following notation is used

- *ashl* for an arithmetic shift left
- *ashr* for an arithmetic shift right
- Examples:
 - » $R2 \leftarrow ashr\ R2$
 - » $R3 \leftarrow ashl\ R3$

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.



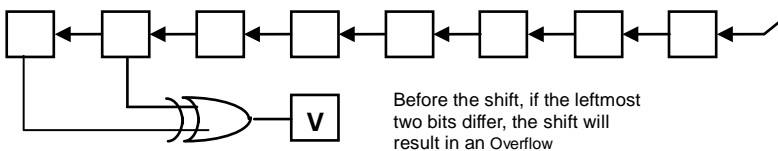
Arithmetic shift-left

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:

If a bit in R_{n-1} changes in value after the shift, sign reversal occurs in the result. This happens if the multiplication by 2 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 $R_{n-1} \neq R_{n-2}$.



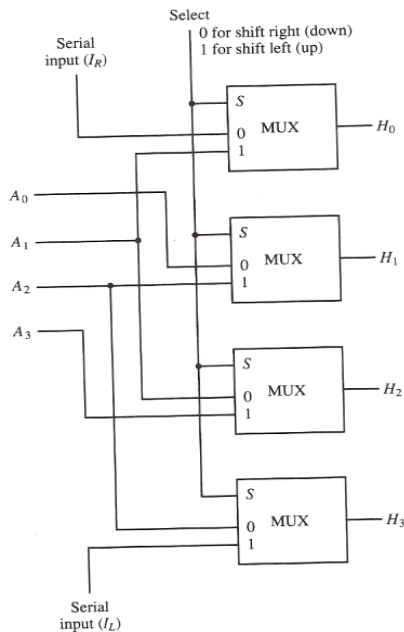
An overflow flip-flop V can be used to detect an arithmetic shift-left overflow.

$$V = R_{n-1} \oplus R_{n-2}$$

If $V = 0$, there is no overflow but if $V = 1$, overflow is detected.

Hardware implementation of shift microoperations

A combinational circuit shifter can be constructed with multiplexers as shown below:



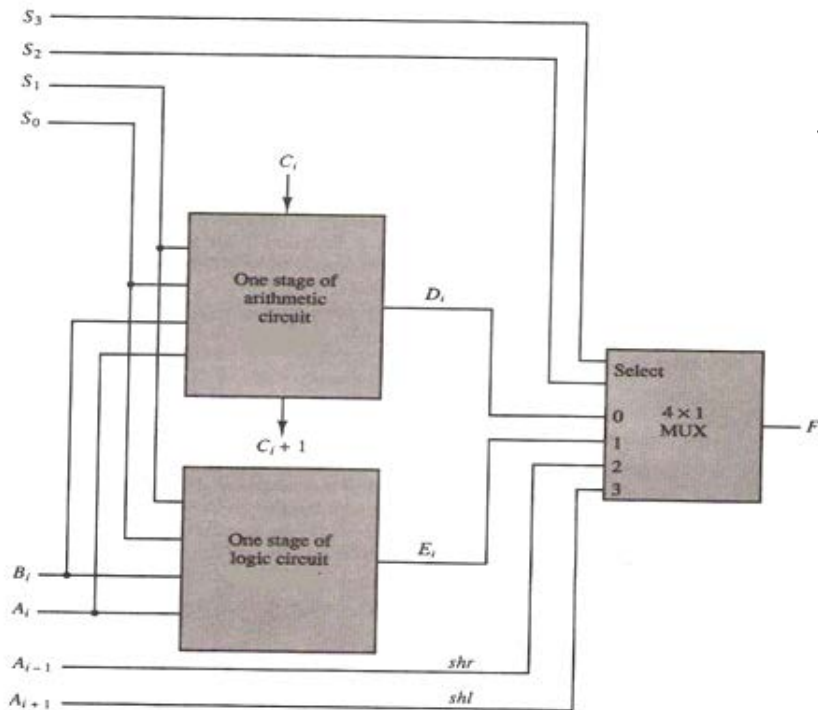
Function table				
Select	Output			
	H_0	H_1	H_2	H_3
0	I_R	A_0	A_1	A_2
1	A_1	A_2	A_3	I_L

- It has 4 data inputs A_0 through A_3 and 4 data outputs H_0 through H_3 .
- There are two serial inputs, one for shift-left (I_L) and other for shift-right (I_R).
- When $S = 0$: input data are shifted right (down in fig).
- When $S = 1$: input data are shifted left (up in fig).

Fig: 4-bit combinational circuit shifter

Arithmetic Logic Shift Unit

This is a common operational unit called arithmetic logic unit (ALU). To perform a microoperation, the contents of specified registers are placed in the inputs of the common ALU. The ALU performs the operation and transfer result to destination register.



- A particular microoperation is selected with inputs s_1 and s_0 .
- A 4x1 MUX at the output chooses between an arithmetic output in D_i and logic output E_i .
- Other two inputs to the MUX receive inputs A_{i-1} for right-shift operation and A_{i+1} for left-shift operation.
- The diagram shows just one typical stage. The circuit must be repeated n times for an n -bit ALU.

This circuit provides 8 arithmetic operations, 4 logic operations and 2 shift operations. Each operation is selected with five variables s_3, s_2, s_1, s_0 and c_{in} . The input carry c_{in} is used for arithmetic operations only. Table below lists the 14 operations of the ALU.

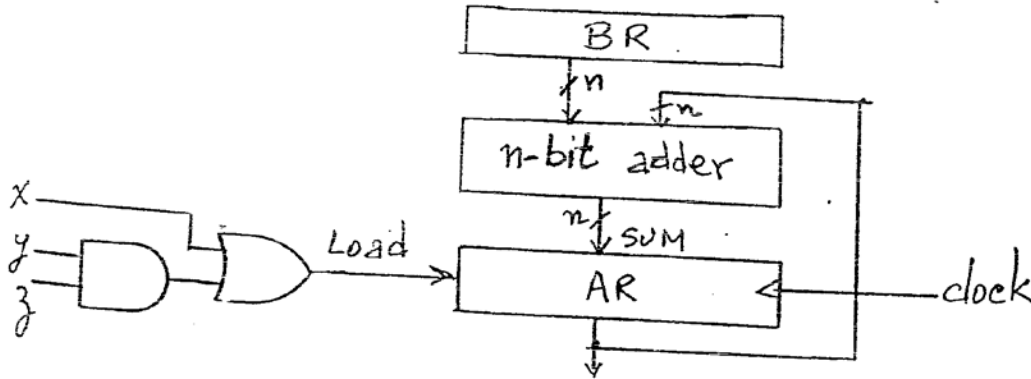
Fig: One stage of arithmetic logic shift unit

Operation select					Operation	Function
s_3	s_2	s_1	s_0	C_{in}		
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	\times	$F = A \wedge B$	AND
0	1	0	1	\times	$F = A \vee B$	OR
0	1	1	0	\times	$F = A \oplus B$	XOR
0	1	1	1	\times	$F = \overline{A}$	Complement A
1	0	\times	\times	\times	$F = shr A$	Shift right A into F
1	1	\times	\times	\times	$F = shl A$	Shift left A into F

Table: Function table for Arithmetic logic shift unit

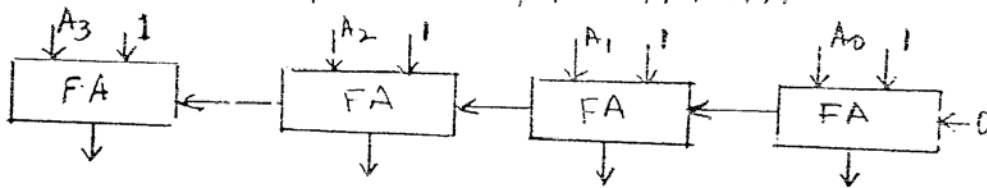
EXERCISES: Textbook chapter 4 → 4.8, 4.13, 4.17, 4.18, 4.19, 4.21

4.8(Solution)

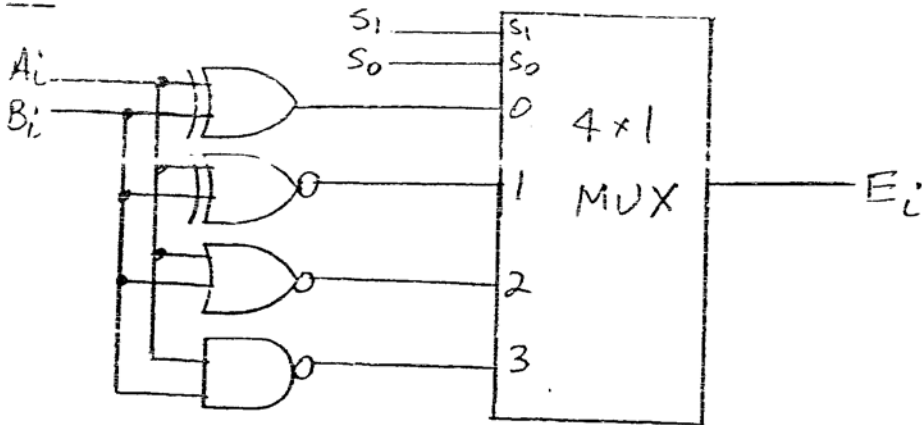


4.13(Solution)

$$A - 1 = A + 2\text{'s complement of } 1 = A + 1111$$



4.17(Solution)



4.18(Solution)

$$\begin{array}{r} \text{(a) } A = 11011001 \\ B = 10110100 \oplus \\ \hline A \leftarrow A \oplus B \quad 01101101 \end{array} \qquad \begin{array}{r} A = 11011001 \\ B = 11111101 \text{ (OR)} \\ \hline 11111101 \quad A \leftarrow A \vee B \end{array}$$

4.19(do it yourself)

4.21(do it too)

Unit 3

Basic Computer Organization and Design

Introduction

We introduce here a basic computer whose operation can be specified by the register transfer statements. Internal organization of the computer is defined by the sequence of microoperations it performs on data stored in its registers. Every different processor type has its own design (different registers, buses, microoperations, machine instructions, etc). Modern processor is a very complex device. It contains:

- Many registers
- Multiple arithmetic units, for both integer and floating point calculations
- The ability to pipeline several consecutive instructions for execution speedup.

However, to understand how processors work, we will start with a simplified processor model. M. Morris Mano introduces a simple processor model; he calls it a "Basic Computer". The Basic Computer has two components, a processor and memory

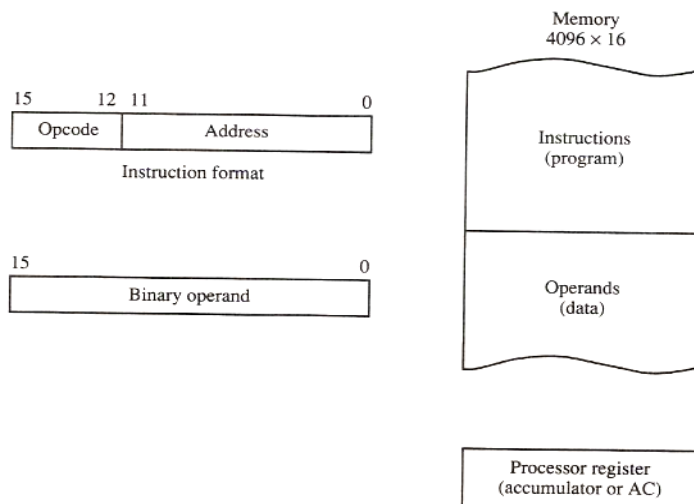
- The memory has 4096 words in it
 - $4096 = 2^{12}$, so it takes 12 bits to select a word in memory
- Each word is 16 bits long

Instruction code and Stored program organization

Question: What do you understand by stored program organization?

Question: What is instruction and instruction format?

Instruction code is a group of bits that instructs the computer to perform a specific operation. It is usually divided into parts. Most basic part is operation (**operation code**). Operation code is group of bits that defines operations as add, subtract, multiply, shift, complement etc. The instructions of a program, along with any needed data are stored in memory. The CPU reads the next instruction from memory. It is placed in an *Instruction Register* (IR). Control circuitry in control unit then translates the instruction into the sequence of microoperations necessary to implement it. Stored program concept is the ability to store and execute instructions.

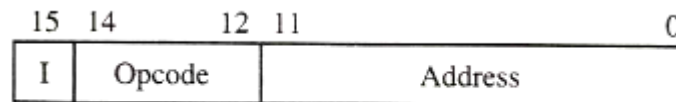


Instruction Format of Basic Computer

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 need 12 bits to specify the memory address that is used by this instruction. 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.

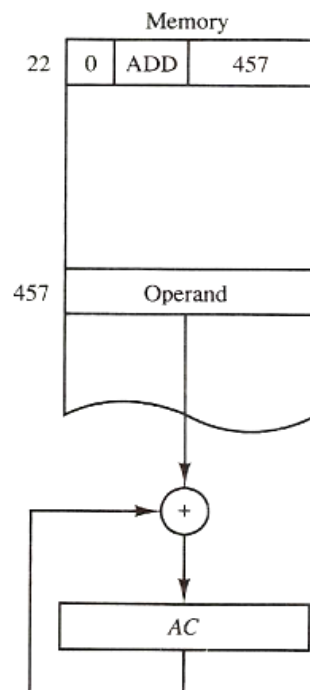


(a) Instruction format

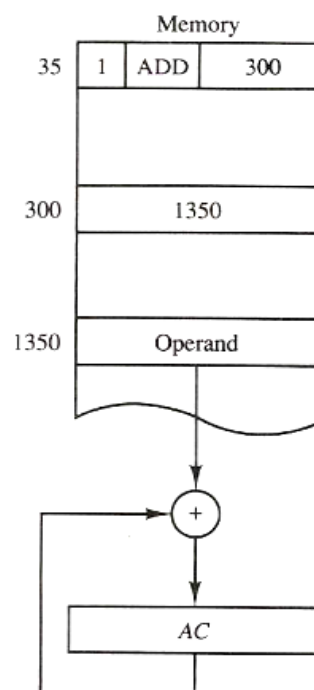
Addressing Modes

The address field of an instruction can represent either

- Direct address: the address operand field is effective address (the address of the operand) or,
- Indirect address: the address in operand field contains the memory address where effective address resides.



(b) Direct address



(c) Indirect address

Effective Address (EA): The address, where actual data resides is called effective address.

Basic Computer Registers

Computer instructions are normally stored in the consecutive memory locations and are executed sequentially one at a time. Thus computer needs processor registers for manipulating data and holding memory address which are shown in the following table:

Symbol	Size	Register Name	Description
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

Since the memory in the Basic Computer only has 4096 ($=2^{12}$) locations, PC and AR only needs 12 bits. Since the word size of Basic Computer only has 16 bit, the DR, AC, IR and TR needs 16 bits. The Basic Computer uses a very simple model of input/output (I/O) operations

- Input devices are considered to send 8 bits of character data to the processor
- The processor can send 8 bits of character data to output devices

The Input Register (INPR) holds an 8 bit character gotten from an input device and the Output Register (OUTR) holds an 8 bit character to be sent to an output device.

Common Bus system of Basic computer

The registers in the Basic Computer are connected using a bus. This gives a savings in circuitry over complete connections between registers. Three control lines, S₂, S₁, and S₀ control which register the bus selects as its input.

S ₂ S ₁ S ₀	Register
0 0 0	X (nothing)
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

Either one of the registers will have its load signal activated, or the memory will have its read signal activated which will determine where the data from the bus gets loaded. 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.

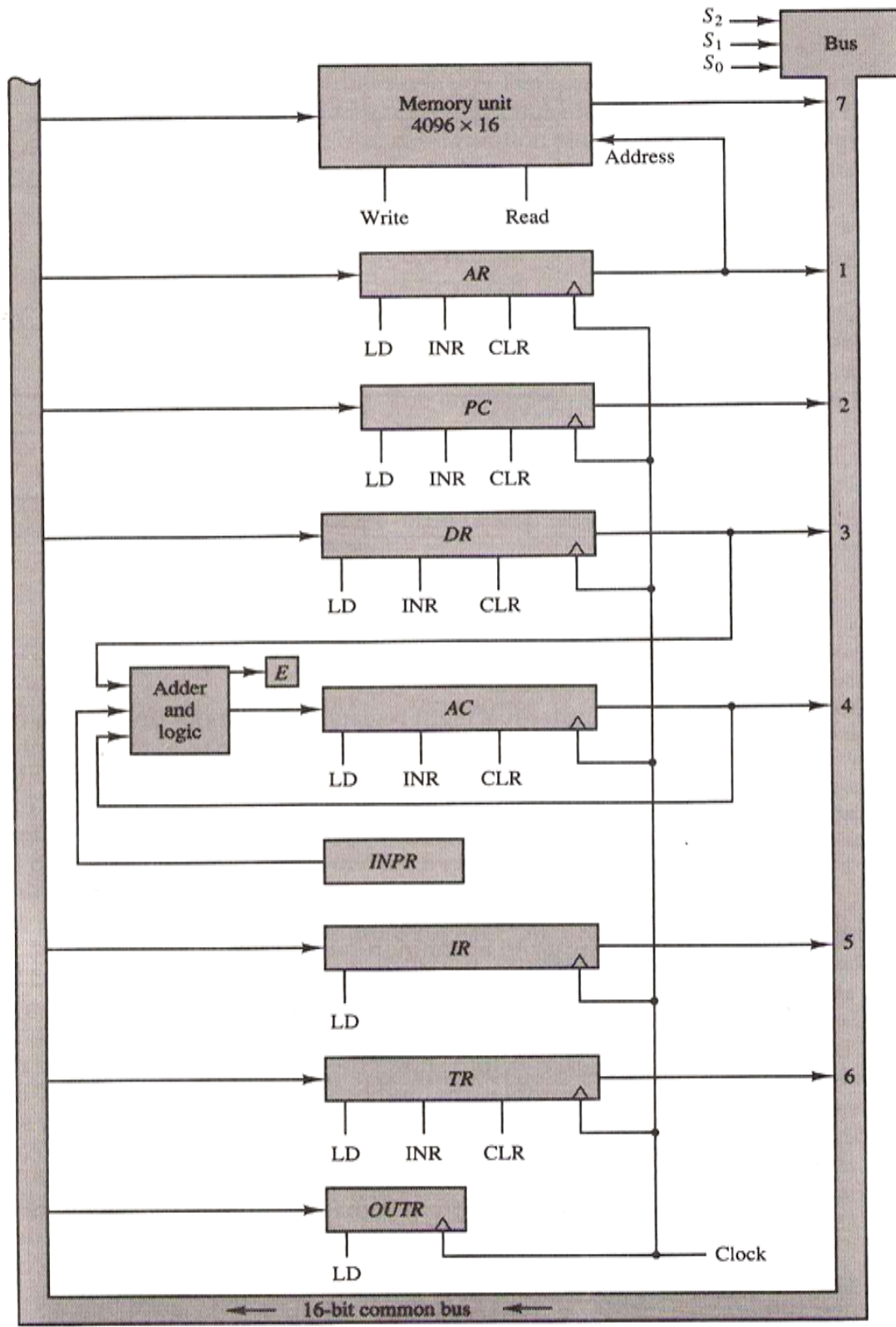


Fig: Basic computer register connected in a common bus.

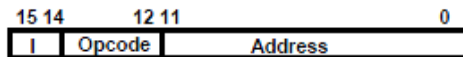
Instruction Formats of Basic Computer

Question: *What are different instruction format used basic computer?*

Question: *What is instruction set completeness? Is instruction set of basic computer complete?*

The basic computer has 3 instruction code formats. Type of the instruction is recognized by the computer control from 4-bit positions 12 through 15 of the instruction.

Memory-Reference Instructions (OP-code = 000 ~ 110)



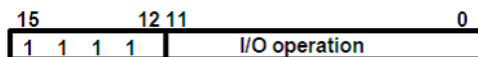
Symbol	Hex Code		Description
	I = 0	I = 1	
AND	0xxx	8xxx	AND memory word to AC
ADD	1xxx	9xxx	Add memory word to AC
LDA	2xxx	Axxx	Load AC from memory
STA	3xxx	Bxxx	Store content of AC into memory
BUN	4xxx	Cxxx	Branch unconditionally
BSA	5xxx	Dxxx	Branch and save return address
ISZ	6xxx	Exxx	Increment and skip if zero

Register-Reference Instructions (OP-code = 111, I = 0)



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

Input-Output Instructions (OP-code = 111, I = 1)



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

Instruction Set Completeness

An instruction set is said to be complete if it contains sufficient instructions to perform operations in following categories:

Functional Instructions

- Arithmetic, logic, and shift instructions
- Examples: ADD, CMA, INC, CIR, CIL, AND, CLA

Transfer Instructions

- Data transfers between the main memory and the processor registers
- Examples: LDA, STA

Control Instructions

- Program sequencing and control
- Examples: BUN, BSA, ISZ

Input/output Instructions

- Input and output
- Examples: INP, OUT

Instruction set of Basic computer is complete because:

- ADD, CMA (complement), INC can be used to perform addition and subtraction and CIR (circular right shift), CIL (circular left shift) instructions can be used to achieve any kind of shift operations. Addition subtraction and shifting can be used together to achieve multiplication and division. AND, CMA and CLA (clear accumulator) can be used to achieve any logical operations.
- LDA instruction moves data from memory to register and STA instruction moves data from register to memory.
- The branch instructions BUN, BSA and ISZ together with skip instruction provide the mechanism of program control and sequencing.
- INP instruction is used to read data from input device and OUT instruction is used to send data from processor to output device.

Instruction Processing & Instruction Cycle (of Basic computer)

Control Unit

Control unit (CU) of a processor translates from machine instructions to the control signals for the microoperations that implement them. There are two types of control organization:

Hardwired Control

- CU is made up of sequential and combinational circuits to generate the control signals.
- If logic is changed we need to change the whole circuitry
- Expensive
- Fast

Microprogrammed Control

- A control memory on the processor contains microprograms that activate the necessary control signals
- If logic is changed we only need to change the microprogram
- Cheap
- Slow

NOTE: Microprogrammed control unit will be discussed in next chapter.

Question: How basic computer translates machine instructions to control signals using hardwired control? Explain with block diagram. (OR Discuss hardwired control unit of basic computer?)

The block diagram of a hardwired control unit is shown below. It consists of two decoders, a sequence counter, and a number of control logic gates.

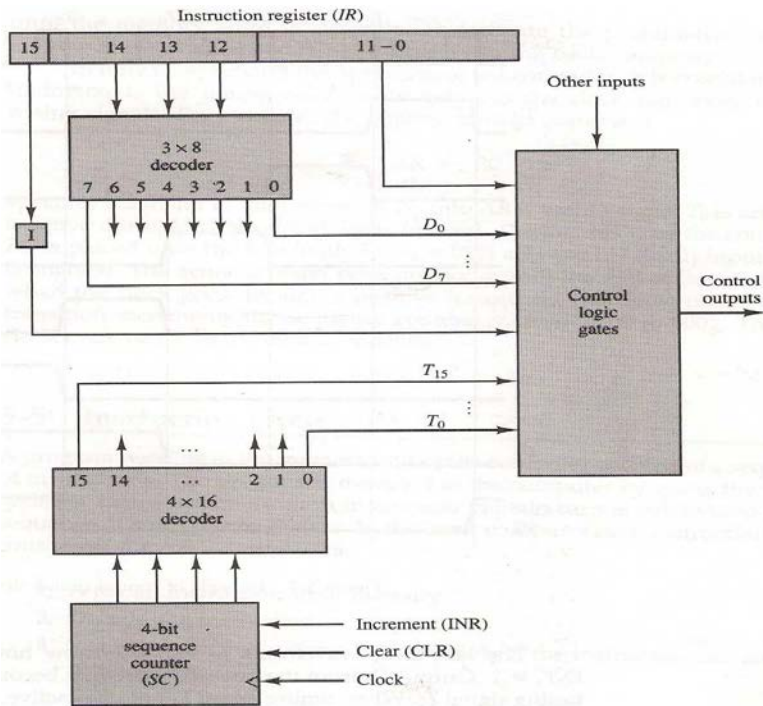


Fig: Control unit of a basic computer

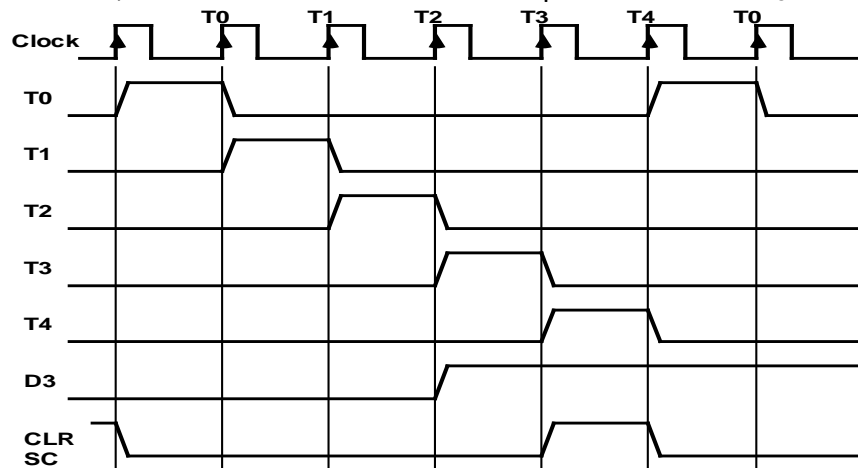
Mechanism:

- An instruction read from memory is placed in the instruction register (IR) where it is decoded into three parts: 1 bit, **operation code** and bits **0 through 11**.
 - The operation code bit is decoded with 3 x 8 decoder producing 8 outputs D_0 through D_7 .
 - Bit 15 of the instruction is transferred to a flip-flop I.
 - And operand bits are applied to control logic gates.
 - The 16 outputs of 4-bit sequence counter (SC) are decoded into 16 timing signals T_0 through T_{15} .
- This means instruction cycle of basic computer can not take more than 16 clock cycles.

Timing signals

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

Assume: At time T_4 , SC is cleared to 0 if decoder output D_3 is active: $D_3 T_4$: SC $\square 0$



Instruction cycle

In Basic Computer, a machine instruction is executed in the following cycle:

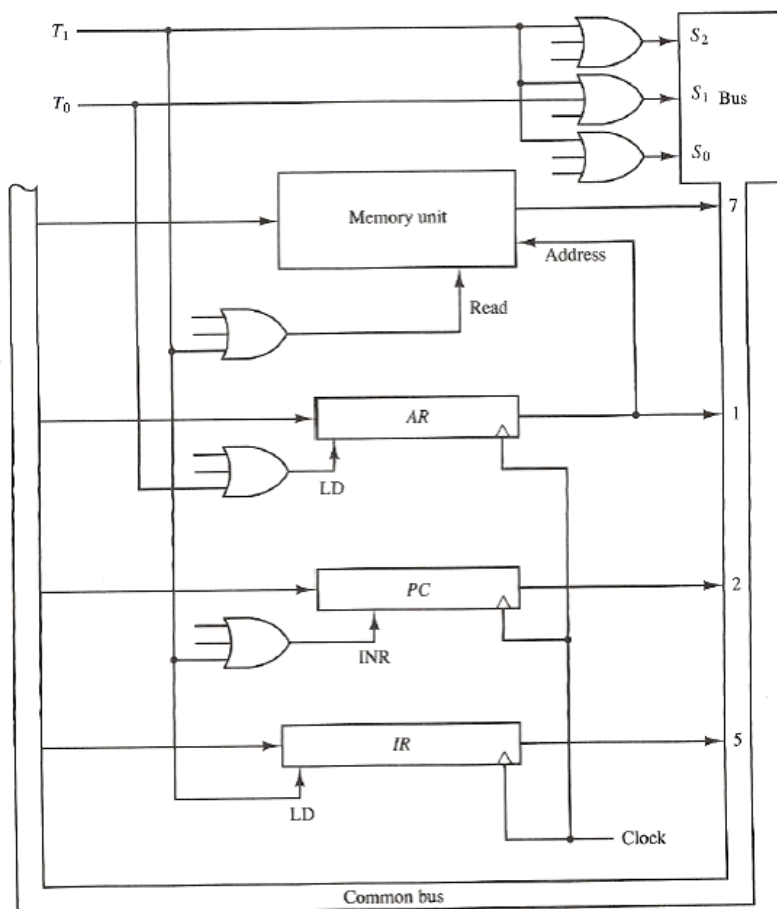
1. Fetch an instruction from memory
2. Decode the instruction
3. Read the effective address from memory if the instruction has an indirect address
4. Execute the instruction

Upon the completion of step 4, control goes back to step 1 to fetch, decode and execute the next instruction. This process is continued indefinitely until HALT instruction is encountered.

Fetch and decode

The microoperations for the fetch and decode phases can be specified by the following register transfer statements:

$T_0: AR \leftarrow PC \quad (S_0S_1S_2=010, T_0=1)$
 $T_1: IR \leftarrow M[AR], PC \leftarrow PC + 1 \quad (S_0S_1S_2=111, T_1=1)$
 $T_2: D_0, \dots, D_7 \leftarrow \text{Decode } IR(12-14), AR \leftarrow IR(0-11), I \leftarrow IR(15)$



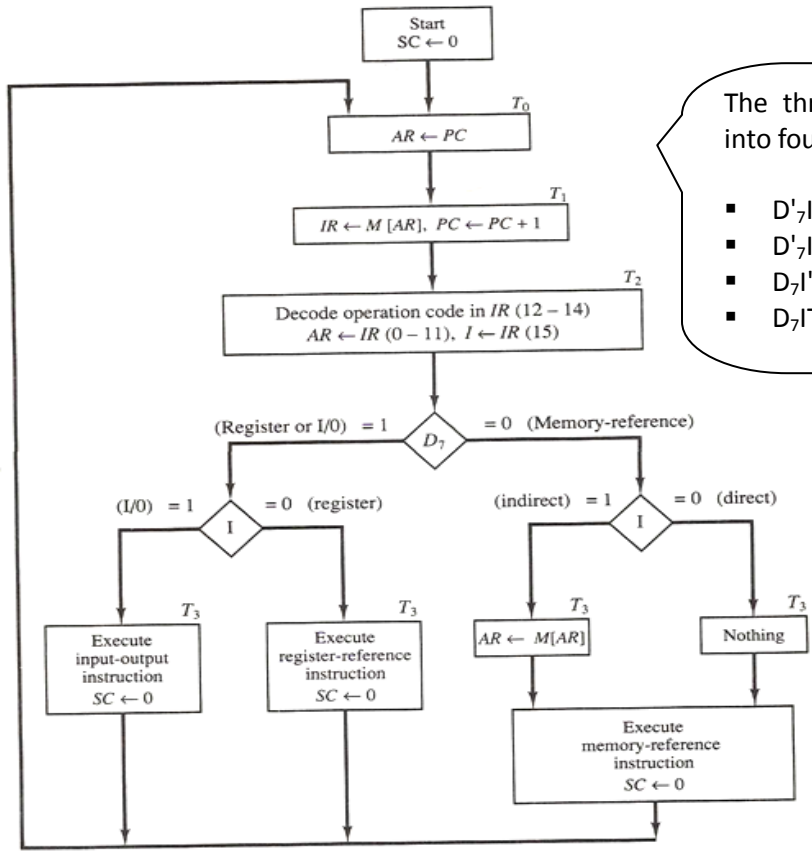
It is necessary to transfer the address from PC to AR during clock transition associated with the timing signal T_0 . The instruction read from memory is then placed in IR with clock transition associated with the timing signal T_1 . At the same time, PC is incremented by one to prepare for the next instruction in the program. At time T_2 , the opcode in IR is decoded, the indirect bit is transferred to flip-flop I, and the address part of the instruction is transferred to AR.

NOTE: SC is incremented after each clock pulse to produce the sequence T_0, T_1 and T_2 .

Fig: Register transfers for the fetch phase

Determine the type of the instruction

The timing signal that is active after decoding is T_3 . During time T_3 , the control unit determines the type of instruction that was just read from memory. Following flowchart presents an initial configuration for the instruction cycle and shows how the control determines the instruction type after decoding.



The three instruction types are subdivided into four separate paths:

- $D_7 I T_3$: $AR \leftarrow M[AR]$ (Indirect address)
- $D_7 I' T_3$: Nothing
- $D_7 I T_3$: Execute register-reference instrs.
- $D_7 I T_3$: Execute input-output instructions.

Fig: Flowchart for instruction cycle (Initial configuration)

Register transfers needed for the execution of register-reference and memory-reference instructions are explained below: (I/O instructions will be discussed later)

Register-reference instructions:

Register Reference Instructions are recognized with

- $D_7 = 1, I = 0$
- Register Ref. Instr. is specified in $b_0 \sim b_{11}$ of IR
- Execution starts with timing signal T_3

Let

$r = D_7 I' T_3 \Rightarrow$ Common to all Register Reference Instruction

$B_i = IR(i), i=0, 1, 2, \dots, 11$. [Bit in IR(0-11) that specifies the operation]

CLA rB_{11} : $AC \leftarrow 0, SC \leftarrow 0$

Clear AC

CLE rB_{10} : $E \leftarrow 0, SC \leftarrow 0$

Clear E

CMA	rB ₉ :	AC ← AC', SC ← 0	Complement AC
CME	rB ₈ :	E ← E', SC ← 0	Complement E
CIR	rB ₇ :	AC ← shr AC, AC(15) ← E, E ← AC(0), SC ← 0	Circulate right
CIL	rB ₆ :	AC ← shl AC, AC(0) ← E, E ← AC(15), SC ← 0	Circulate Left
INC	rB ₅ :	AC ← AC + 1, SC ← 0	Increment AC
SPA	rB ₄ :	if (AC(15) = 0) then (PC ← PC+1), SC ← 0	Skip if positive
SNA	rB ₃ :	if (AC(15) = 1) then (PC ← PC+1), SC ← 0	skip if negative
SZA	rB ₂ :	if (AC = 0) then (PC ← PC+1), SC ← 0	skip if AC zero
SZE	rB ₁ :	if (E = 0) then (PC ← PC+1), SC ← 0	skip if E zero
HLT	rB ₀ :	S ← 0, SC ← 0 (S is a start-stop flip-flop)	Halt computer

Memory-reference instructions

- Once an instruction has been loaded to IR, it may require further access to memory to perform its intended function (direct or indirect).
- The effective address of the instruction is in the AR and was placed their during:
 - Time signal T2 when I = 0 or
 - Time signal T3 when I = 1
- Execution of memory reference instructions starts with the timing signal T4.
- Described symbolically using RTL.

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

AND to AC

This instruction performs the AND logical operation on pairs of bits on AC and the memory word specified by the effective address. The result is transferred to AC. Microoperations that execute these instructions are:

D₀T₄: DR ← M[AR] //Read operand
 D₀T₅: AC ← AC ∧ DR, SC ← 0 //AND with AC

ADD to AC

D₁T₄: DR ← M[AR] //Read operand
 D₁T₅: AC ← AC + DR, E ← C_{out}, SC ← 0 //Add to AC and stores carry in E

LDA: Load to AC

D₂T₄: DR ← M[AR] //Read operand
 D₂T₅: AC ← DR, SC ← 0 //Load AC with DR

STA: Store AC

$D_3T_4: M[AR] \leftarrow AC, SC \leftarrow 0$ // store data into memory location

BUN: Branch Unconditionally

$D_4T_4: PC \leftarrow AR, SC \leftarrow 0$ //Branch to specified address

BSA: Branch and Save Return Address

$D_5T_4: M[AR] \leftarrow PC, AR \leftarrow AR + 1$ // save return address and increment AR

$D_5T_5: PC \leftarrow AR, SC \leftarrow 0$ // load PC with AR

ISZ: Increment and Skip-if-Zero

$D_6T_4: DR \leftarrow M[AR]$ //Load data into DR

$D_6T_5: DR \leftarrow DR + 1$ // Increment the data

$D_6T_4: M[AR] \leftarrow DR, \text{ if } (DR = 0) \text{ then } (PC \leftarrow PC + 1), SC \leftarrow 0$
// if DR=0 skip next instruction by incrementing PC

Input-Output and Interrupt

In computer, instructions and data stored in memory come from some input device and Computational results must be transmitted to the user through some output device.

Input-output configuration

The terminal sends and receives serial information. Each quantity of information has 8 bits of an alphanumeric code. Two basic computer registers INPR and OTR communicate with a communication interfaces.

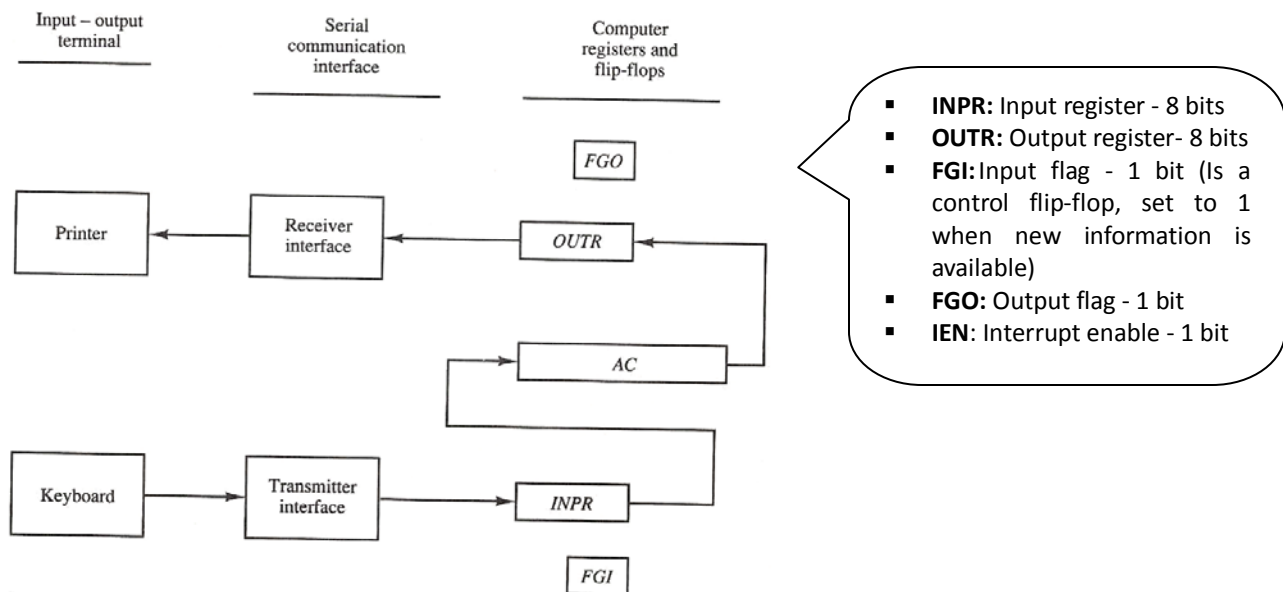


Fig: Input-output configuration

Scenario1: when a key is struck in the keyboard, an 8-bit alphanumeric code is shifted into INPR and the input flag FGI is set to 1. As long as the flag is set, the information in INPR can not be changed by striking another key. The control checks the flag bit, if 1, contents of INPR is transferred in parallel to AC and FGI is cleared to 0. Once the flag is cleared, new information can be shifted into INPR by striking another key.

Scenario2: OUTR works similarly but the direction of information flow is reversed. Initially FGO is set to 1. The computer checks the flag bit; if it is 1, the information is transferred in parallel to OUTR and FGO is cleared to 0. The output device accepts the coded information, prints the corresponding character and when operation is completed, it sets FGO to 1.

Input-output Instructions

I/O instructions are needed to transferring information to and from AC register, for checking the flag bits and for controlling the interrupt facility.

$D_7IT_3 = p$ (common to all input-output instructions)
 $IR(i) = B_i$ [bit in $IR(6-11)$ that specifies the instruction]

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

Program Interrupt

- Input and Output interactions with electromechanical peripheral devices require huge processing times compared with CPU processing times
 - I/O (milliseconds) versus CPU (nano/micro-seconds)
- Interrupts permit other CPU instructions to execute while waiting for I/O to complete
- The I/O interface, instead of the CPU, monitors the I/O device.
- When the interface finds that the I/O device is ready for data transfer, it generates an interrupt request to the CPU
- Upon detecting an interrupt, the CPU stops momentarily the task it is doing, branches to the service routine to process the data transfer, and then returns to the task it was performing.

Scenario3: consider a computer which completes instruction cycle in $1\mu s$. Assume I/O device that can transfer information at the maximum rate of 10 characters/sec. Equivalently, one character every $100000\mu s$. Two instructions are executed when computer checks the flag bit and decides not to transfer information. Which means computer will check the flag 50000 times between each transfer. Computer is wasting time while checking the flag instead of doing some useful processing task.

- IEN (Interrupt-enable flip-flop)
 - can be set and cleared by instructions
 - When cleared, the computer cannot be interrupted

Interrupt cycle

This is a hardware implementation of a branch and save return address operation.

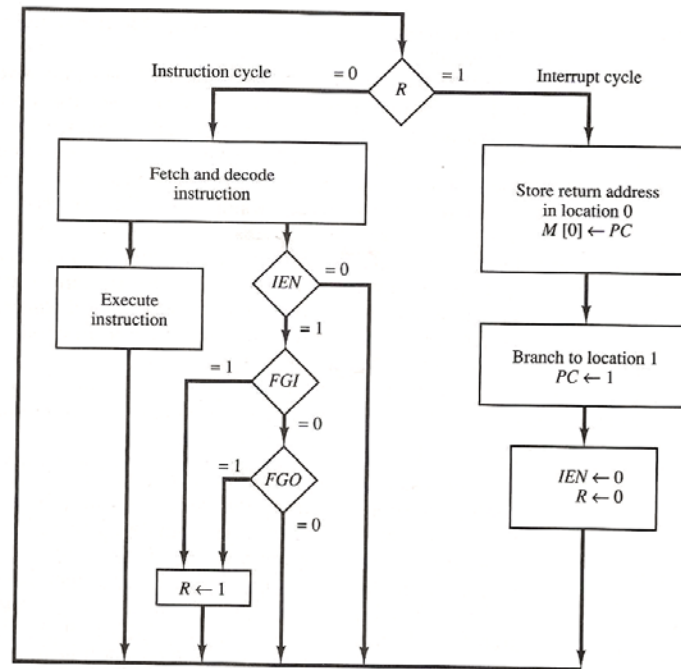


Fig: flowchart of interrupt cycle

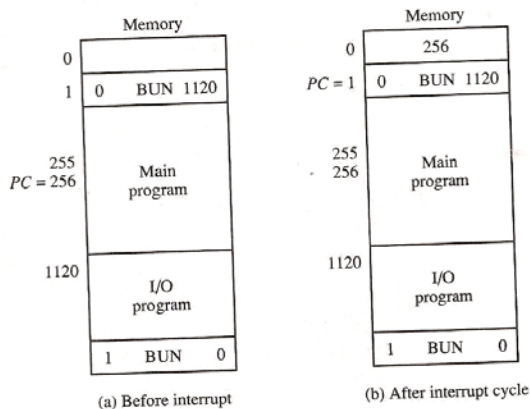


Fig: Demonstration of interrupt cycle

- At the beginning of the 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"

Register transfer operations in interrupt cycle

Register Transfer Statements for Interrupt Cycle

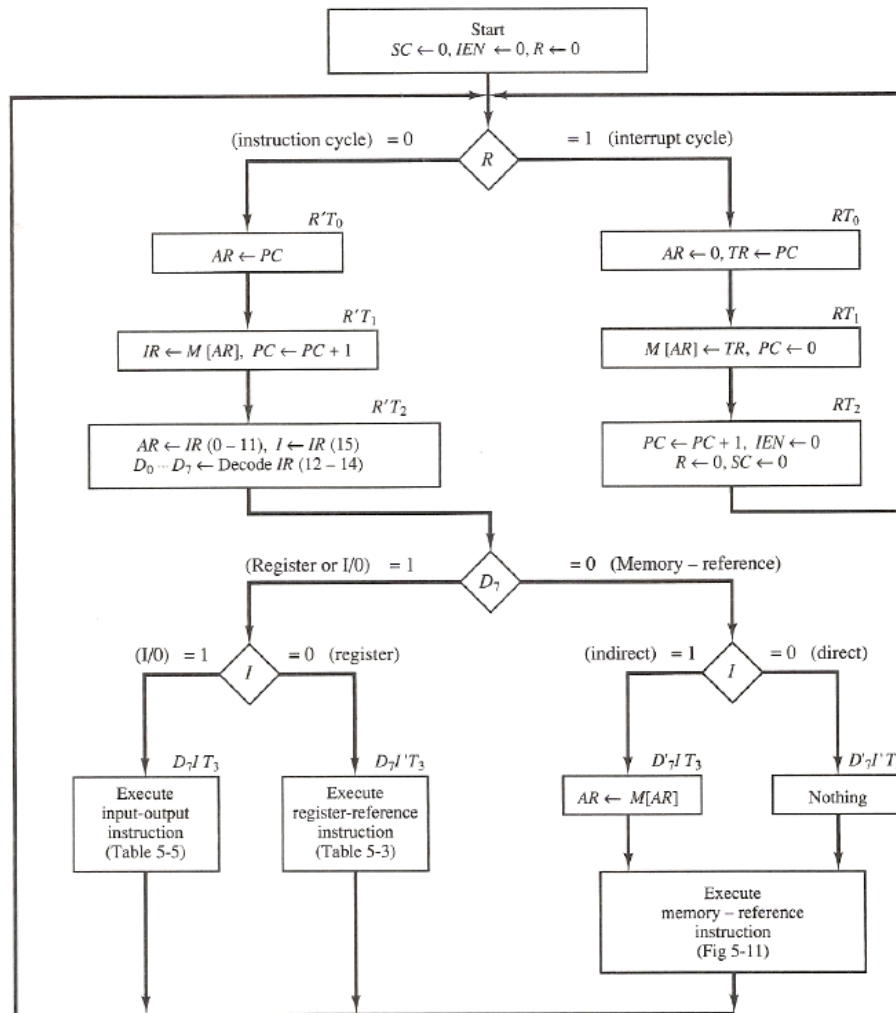
- $R \text{ F/F} \leftarrow 1$ if $IEN (FGI + FGO) T_0' T_1' T_2' \leftrightarrow T_0 T_1 T_2 (IEN) (FGI + FGO): R \leftarrow 1$

- The fetch and decode phases of the instruction cycle must be modified: Replace T_0, T_1, T_2 with $R'T_0, R'T_1, R'T_2$
- The interrupt cycle : $RT_0: AR \leftarrow 0, TR \leftarrow PC$
 $RT_1: M[AR] \leftarrow TR, PC \leftarrow 0$
 $RT_2: PC \leftarrow PC + 1, IEN \leftarrow 0, R \leftarrow 0, SC \leftarrow 0$

Complete computer description

Flowchart

This is the final flowchart of the instruction cycle including interrupt cycle for the basic computer.



Microoperations

Fetch	R'T0:	AR ← PC
	R'T1:	IR ← M[AR], PC ← PC + 1
Decode	R'T2:	D0, ..., D7 ← Decode IR(12 ~ 14), AR ← IR(0 ~ 11), I ← IR(15)
Indirect	D7'I T3:	AR ← M[AR]
Interrupt	T0'T1'T2'(IEN)(FGI + FGO):	R ← 1
	RT0:	AR ← 0, TR ← PC
	RT1:	M[AR] ← TR, PC ← 0
	RT2:	PC ← PC + 1, IEN ← 0, R ← 0, SC ← 0
Memory-Reference		
AND	D0T4:	DR ← M[AR]
	D0T5:	AC ← AC . DR, SC ← 0
ADD	D1T4:	DR ← M[AR]
	D1T5:	AC ← AC + DR, E ← Cout, SC ← 0
LDA	D2T4:	DR ← M[AR]
	D2T5:	AC ← DR, SC ← 0
STA	D3T4:	M[AR] ← AC, SC ← 0
BUN	D4T4:	PC ← AR, SC ← 0
BSA	D5T4:	M[AR] ← PC, AR ← AR + 1
	D5T5:	PC ← AR, SC ← 0
ISZ	D6T4:	DR ← M[AR]
	D6T5:	DR ← DR + 1
	D6T6:	M[AR] ← DR, if(DR=0) then (PC ← PC + 1), SC ← 0

Register-Reference		
	D7'I T3 = r	(Common to all register-reference instr)
	IR(i) = Bi	(i = 0,1,2, ..., 11)
	r:	SC <- 0
CLA	rB11:	AC <- 0
CLE	rB10:	E <- 0
CMA	rB9:	AC <- AC'
CME	rB8:	E <- E'
CIR	rB7:	AC <- shr AC, AC(15) <- E, E <- AC(0)
CIL	rB6:	AC <- shl AC, AC(0) <- E, E <- AC(15)
INC	rB5:	AC <- AC + 1
SPA	rB4:	If(AC(15) = 0) then (PC <- PC + 1)
SNA	rB3:	If(AC(15) = 1) then (PC <- PC + 1)
SZA	rB2:	If(AC = 0) then (PC <- PC + 1)
SZE	rB1:	If(E=0) then (PC <- PC + 1)
HLT	rB0:	S <- 0
Input-Output		
	D7'I T3 = p	(Common to all input-output instructions)
	IR(i) = Bi	(i = 6,7,8,9,10,11)
	p:	SC <- 0
INP	pB11:	AC(0-7) <- INPR, FGI <- 0
OUT	pB10:	OUTR <- AC(0-7), FGO <- 0
SKI	pB9:	If(FGI=1) then (PC <- PC + 1)
SKO	pB8:	If(FGO=1) then (PC <- PC + 1)
ION	pB7:	IEN <- 1
IOF	pB6:	IEN <- 0

Design of Basic Computer (BC)

Hardware Components of BC

1. A memory unit: 4096 x 16.
2. Registers:
AR, PC, DR, AC, IR, TR, OUTR, INPR, and SC
3. Flip-Flops(Status):
I, S, E, R, IEN, FGI, and FGO
4. Decoders: A 3x8 Opcode decoder
A 4x16 timing decoder
5. Common bus: 16 bits
6. Control logic gates
7. Adder and Logic circuit: Connected to AC

Control Logic Gates

Inputs:

1. Two decoder outputs
2. I flip-flop
3. IR(0-11)
4. AC(0-15)
 - To check if AC = 0
 - To detect sign bit AC(15)
5. DR(0-15)
 - To check if DR = 0
6. Values of seven flip-flops

Outputs:

1. Input Controls of the nine registers
2. Read and Write Controls of memory
3. Set, Clear, or Complement Controls of the flip-flops
4. S₂, S₁, S₀ Controls to select a register for the bus
5. AC, and Adder and Logic circuit

Control of registers and memory

The control inputs of the registers are LD (load), INR (increment) and CLR (clear).

- Address Register (AR)

To derive the gate structure associated with the control inputs of AR: we find all the statements that change the contents of AR.

$R'T_0$:	$AR \leftarrow PC$	LD(AR)
$R'T_2$:	$AR \leftarrow IR(0-11)$	LD(AR)
$D'I_7T_3$:	$AR \leftarrow M[AR]$	LD(AR)
RT_0 :	$AR \leftarrow 0$	CLR(AR)
D_5T_4 :	$AR \leftarrow AR + 1$	INR(AR)



$LD(AR) = R'T_0 + R'T_2 + D'I_7T_3$
$CLR(AR) = RT_0$
$INR(AR) = D_5T_4$

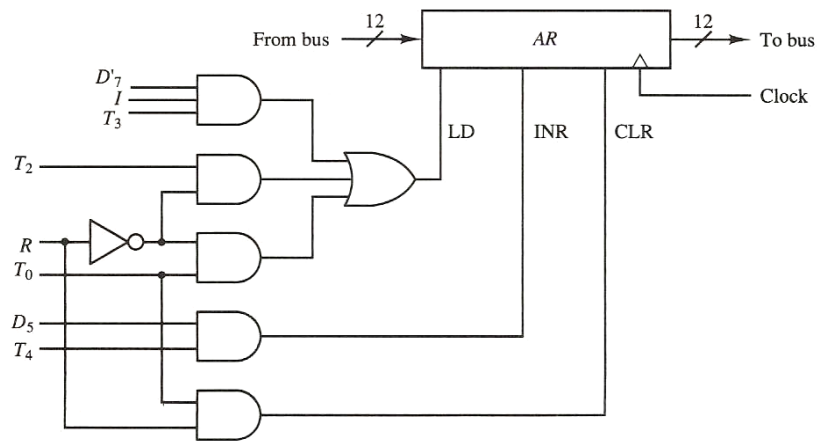


Fig: Control gates associated with AR

Similarly, control gates for the other registers as well as the read and write inputs of memory can be derived. Viz. the logic gates associated with the read inputs of memory is derived by scanning all statements that contain a read operation. (Read operation is recognized by the symbol $\leftarrow M[AR]$).

$$\text{Read} = R'T_1 + D'I_7T_3 + (D_0 + D_1 + D_2 + D_6)T_4$$

The output of the logic gates that implement the Boolean expression above must be connected to the read input of memory.

Control of flip-flops

The control gates for the seven flip-flops can be determined in a similar manner.

Example:

- IEN(Interrupt Enable Flag)

pB7: $IEN \leftarrow 1$ (I/O Instruction)

pB6: $IEN \leftarrow 0$ (I/O Instruction)

RT_2 : $IEN \leftarrow 0$ (Interrupt)

These three instructions can cause IEN flag to change its value.

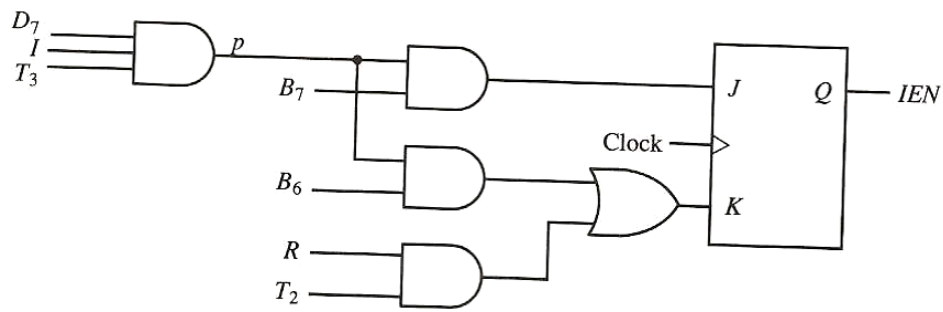


Fig: control inputs for IEN

Control of Common Bus

The 16-bit common bus is controlled by the selection inputs S_2 , S_1 and S_0 . Binary numbers for $S_2S_1S_0$ is associated with a Boolean variable x_1 through x_7 , which must be active in order to select the register or memory for the bus.

Inputs							Outputs			Register selected for bus
x_1	x_2	x_3	x_4	x_5	x_6	x_7	S_2	S_1	S_0	
0	0	0	0	0	0	0	0	0	0	None
1	0	0	0	0	0	0	0	0	1	AR
0	1	0	0	0	0	0	0	1	0	PC
0	0	1	0	0	0	0	0	1	1	DR
0	0	0	1	0	0	0	1	0	0	AC
0	0	0	0	1	0	0	1	0	1	IR
0	0	0	0	0	1	0	1	1	0	TR
0	0	0	0	0	0	1	1	1	1	Memory

Fig: Encoder for Bus Selection Circuit

Example: when $x_1 = 1$, $S_2S_1S_0$ must be 001 and thus output of AR will be selected for the bus.

To determine the logic for each encoder input, it is necessary to find the control functions that place the corresponding register onto the bus.

Example: to find the logic that makes $x_1 = 1$, we scan all register transfer statements that have AR as a source.

$$D_4T_4: PC \leftarrow AR$$

$$D_5T_5: PC \leftarrow AR$$

Therefore the Boolean function for x_1 is,

$$x_1 = D_4T_4 + D_5T_5$$

Similarly, for memory read operation,

$$x_7 = R'T_1 + D_7IT_3 + (D_0 + D_1 + D_2 + D_6)T_4$$

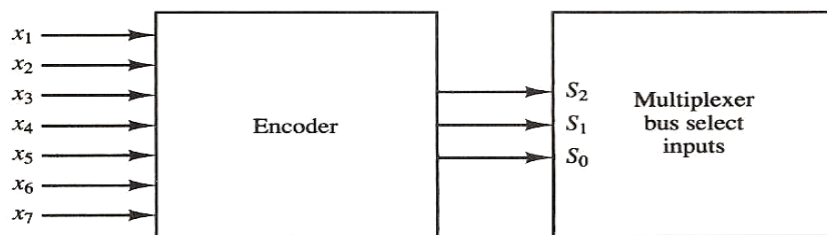


Fig: Encoder for bus selection inputs

Design of Accumulator Logic

To design the logic associated with AC, we extract all register transfer statements that change the contents of AC. The circuit associated with the AC register is shown below:

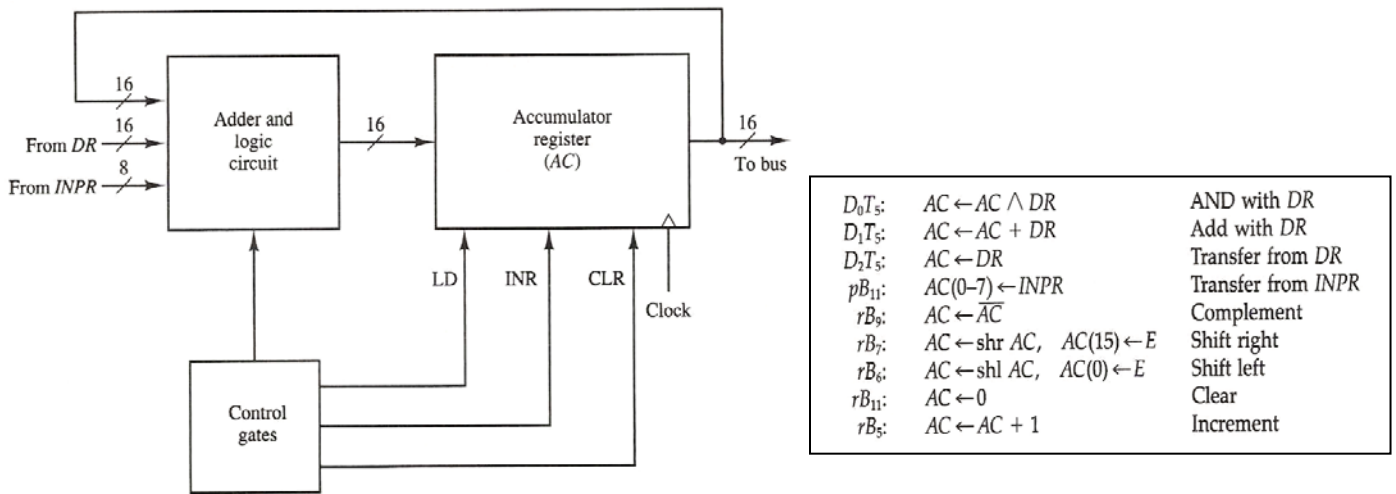


Fig: circuits associated with AC

Control of AC Resister

The gate structure that controls the LD, INR and CLR inputs of AC is shown below:

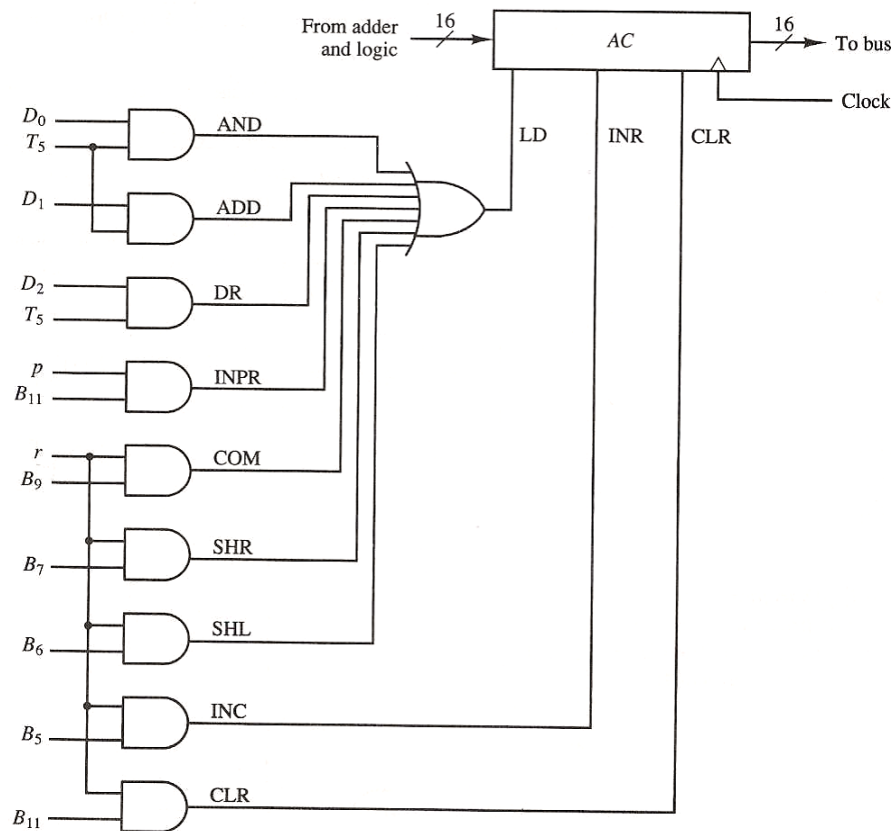


Fig: Gate structure for controlling LD, INR and CLR of AC

Adder and Logic Circuit

The adder and logic circuit can be subdivided into 16 stages, with each bit corresponding to one bit of AC.

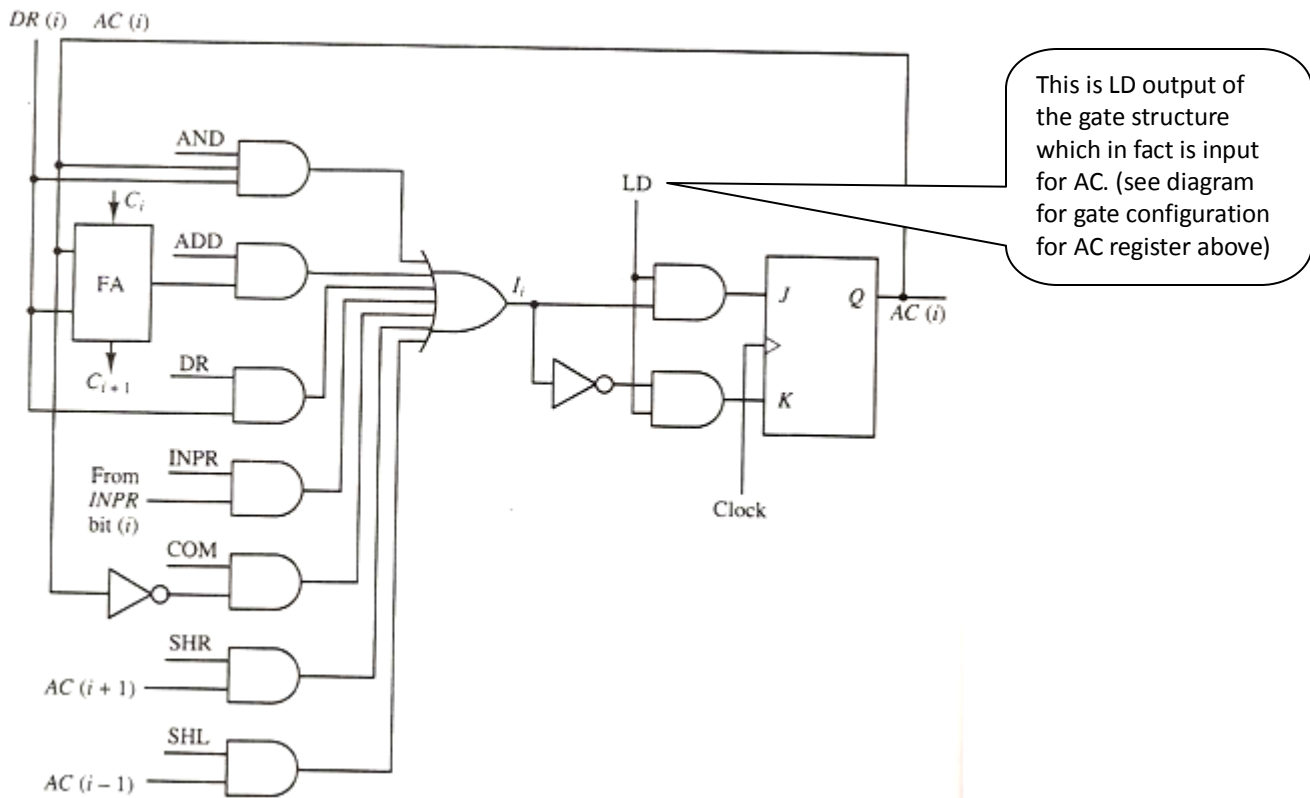


Fig: One stage of adder and logic circuit

- One stage of the adder and logic circuit consists of seven AND gates, one OR gate and a full adder (FA) as shown above.
- The input is labeled I_i output $AC(i)$.
- When LD input is enabled, the 16 inputs I_i for $i = 0, 1, 2, \dots, 15$ are transferred to $AC(i)$.
- The AND operation is achieved by ANDing $AC(i)$ with the corresponding bit in $DR(i)$.
- The transfer from INPR to AC is only for bits 0 through 7.
- The complement microoperation is obtained by inverting the bit value in AC.
- Shift-right operation transfers bit from $AC(i+1)$ and shift-left operation transfers the bit from $AC(i-1)$.

HEY! : The complete adder and logic circuit consists of 16 stages connected together.

EXERCISES: Textbook chapter 5 → 5.1, 5.2, 5.10, 5.23

5.1(solution)

$$256K = 2^8 \times 2^{10} = 2^{18}$$

$$64 = 2^6$$

(a) Address: 18 bits
 Register code: 6 bits
 Indirect bit: 1 bit
 $\frac{25}{25}$ $32 - 25 = 7$ bits for opcode.

(b) $\frac{1}{I} \quad \frac{7}{opcode} \quad \frac{6}{Register} \quad \frac{18}{Address} = 32$ bits

I	opcode	Register	Address
---	--------	----------	---------

(c) Data: 32 bits; address: 18 bits.

5-2

A direct address instruction needs two references to memory: (1) Read instruction; (2) Read operand.

An indirect address instruction needs three references to memory: (1) Read instruction; (2) Read effective address; (3) Read operand.

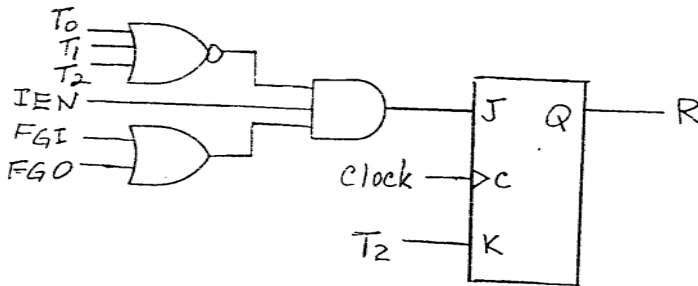
5.10 (Solution)

	PC	AR	DR	AC	IR
Initial	021	—	—	A937	—
AND	022	083	B8F2	A832	0083
ADD	022	083	B8F2	6229	1083
LDA	022	083	B8F2	B8F2	2083
STA	022	083	—	A937	3083
BUN	083	083	—	A937	4083
BSA	084	084	—	A937	5083
ISZ	022	083	B8F3	A937	6083

5.23 (Solution)

$$(T_0 + T_1 + T_2)' (IEN) (FGI + FGO) ; R \leftarrow 1$$

$$RT_2 ; R \leftarrow 0$$



Unit 4

Control Unit

In digital computer, function of control unit is to initiate sequences of microoperations. Types of microoperations for particular system are finite. The complexity of digital system is dependent on the number of sequences of microoperations that are performed. Two complementary techniques used for implementing control unit: hardwired and micro programmed.

Hardwired control

When the control signals are generated by hardware using conventional logic design techniques, the control unit is said to be *hardwired*. We have already studied about the hardwired control unit of basic computer and timing signals associated with it, so guys, turn back to unit3 (textbook, chapter-5) for this portion.

Microprogrammed control

Basic terminologies:

Control Memory (Control Storage: CS)

- ✓ Storage in the microprogrammed control unit to store the microprogram.

Control word

- ✓ It is a string of control variables (0's and 1's) occupying a word in control memory.

Microprogram

- ✓ Program stored in control memory that generates all the control signals required to execute the instruction set correctly
- ✓ Consists of microinstructions

Microinstruction

- ✓ Contains a control word and a sequencing word
- ✓ Control Word – contains all the control information required for one clock cycle
- ✓ Sequencing Word - Contains information needed to decide the next microinstruction address

Writable Control Memory (Writable Control Storage: WCS)

- ✓ CS whose contents can be modified:
 - Microprogram can be changed
 - Instruction set can be changed or modified

A computer that employs a microprogrammed control unit will have two separate memories: main memory and a control memory. The user's program in main memory consists of machine instructions and data whereas control memory holds a fixed micro program that cannot be altered by the user. Each machine instruction initiates a series of microinstructions in control memory.

The general configuration of a microprogrammed control unit is demonstrated in the following block diagram:

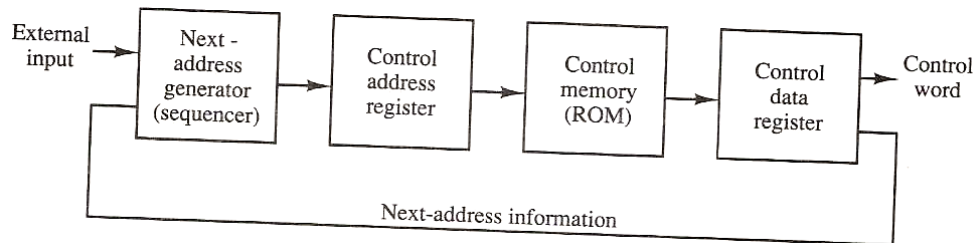


Fig: Microprogrammed control organization

Dynamic Microprogramming

- ✓ Computer system whose control unit is implemented with a microprogram in WCS.
- ✓ Microprogram can be changed by a systems programmer or a user

Control Address Register: Control address register contains address of microinstruction.

Control Data Register: Control data register contains microinstruction.

Sequencer

- ✓ The device or program that generates address of next microinstruction to be executed is called sequencer.

Address Sequencing

Each computer instruction has its own microprogram routine in control memory to generate the microoperations that execute the instruction. Process of finding address of next micro-instruction 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 op-code to address in control memory.

Following is the block diagram for control memory and the associated hardware needed for selecting the next microinstruction address.

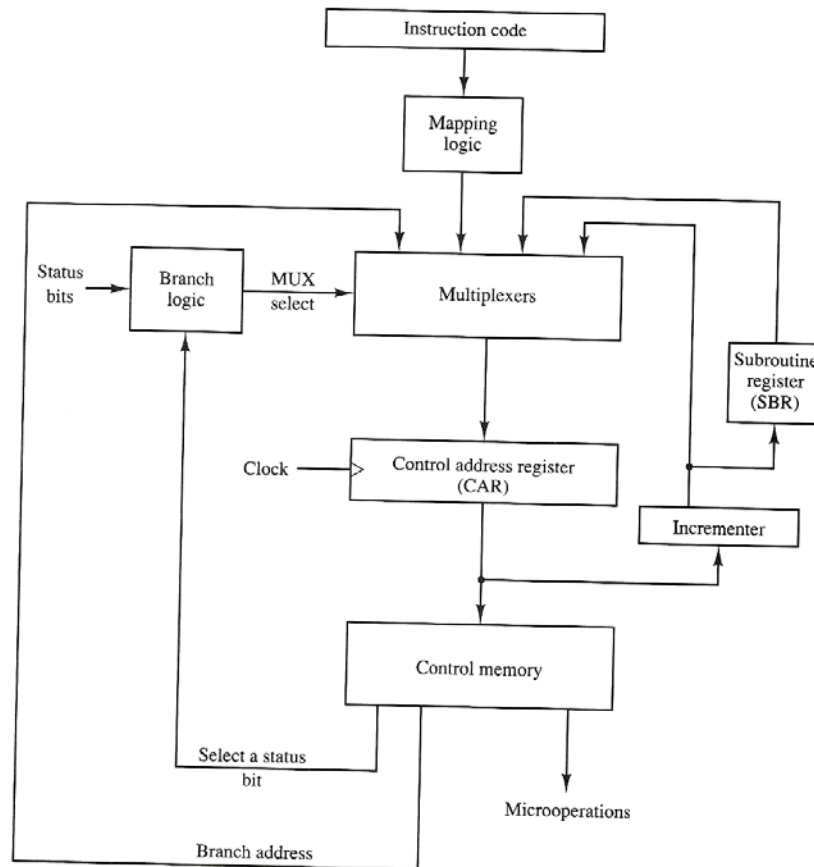


Fig: Block diagram of address sequencer.

- ✓ Control address register receives address of next micro instruction from different sources.
- ✓ Incrementer simply increments the address by one
- ✓ In case of branching branch address is specified in one of the field of microinstruction.
- ✓ In case of subroutine call return address is stored in the register SBR which is used when returning from called subroutine.

Conditional Branch

Simplest way of implementing branch logic hardware is to test the specified condition and branch to the indicated address if condition is met otherwise address register is simply incremented. If Condition is true, h/w set the appropriate field of status register to 1. Conditions are tested for O (overflow), N (negative), Z (zero), C (carry), etc.

Unconditional Branch

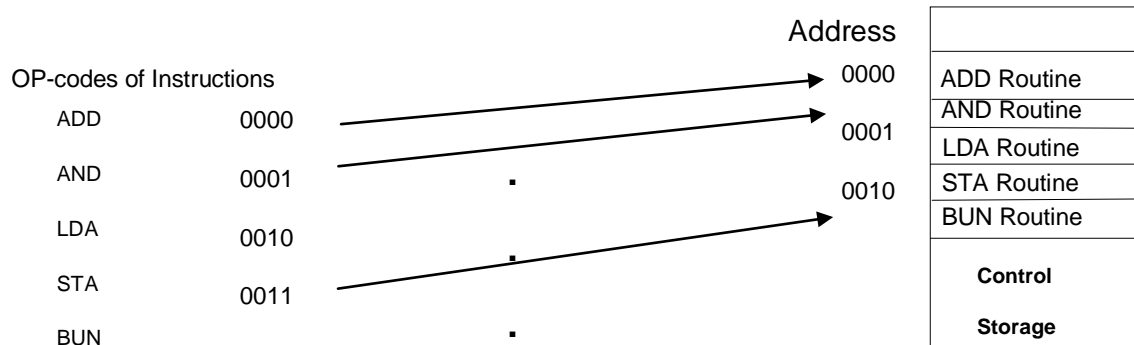
Fix the value of one status bit at the input of the multiplexer to 1. So that, branching can always be done.

Mapping

Assuming operation code of 4-bits which can specify 16 (2^4) distinct instructions. Assume further and control memory has 128 words, requiring an address of 7-bits. Now we have to map 4-bit operation code into 7-bit control memory address. Thus, we have to map Op-code of an instruction to the address of the Microinstruction which is the starting microinstruction of its subroutine in memory.

Direct mapping:

Directly use opcode as address of Control memory



Another approach of direct mapping:

Transfer Opcode bits to use it as an address of control memory.

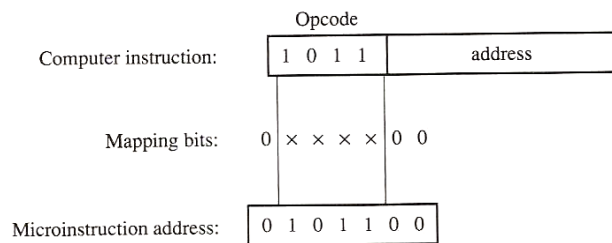
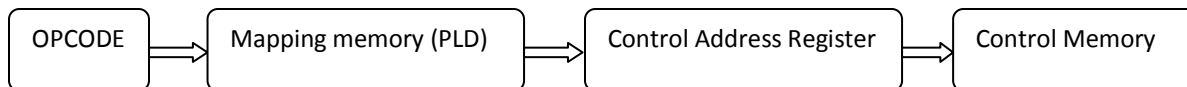


Fig: mapping from instruction code to microinstruction address

Extended idea: Mapping function implemented by ROM or PLD(Programmable Logic Device)

Use opcode as address of ROM where address of control memory is stored and then use that address as an address of control memory. This provides flexibility to add instructions for control memory as the need arises.



Subroutines

Subroutines are programs that are used by another program to accomplish a particular task. Microinstructions can be saved by employing subroutines that use common sections of micro code.

Example: the sequence of microoperations needed to generate the effective address is common to all memory reference instructions. Thus, this sequence could be a subroutine that is called from within many other routines to execute the effective address computation.

Subroutine register is used to save a return address during a subroutine call which is organized in LIFO (last in, first out) stack.

Microprogram (An example)

Once we have a configuration of a computer and its microprogrammed 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. We assume here a simple digital computer similar (but not identical) to Manos' basic computer.

Computer configuration

Block diagram is shown below; it consists of two memory units: a main memory for storing instructions and data, and a control memory for storing the microprogram. 4 registers are with processor unit and 2 registers with the control unit.

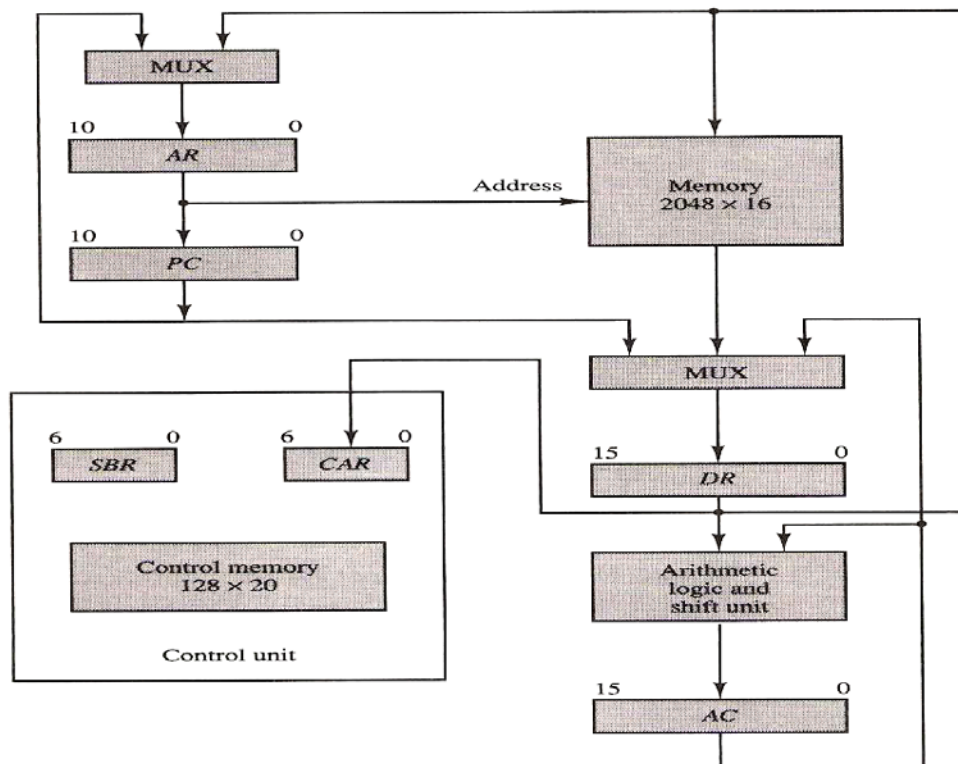


Fig: Computer hardware configuration

Microinstruction Format

We know the computer instruction format (explained in unit3) for different set of instruction in main memory. Similarly, microinstruction in control memory has 20-bit format divided into 4 functional parts as shown below.

3	3	3	2	2	7
F1	F2	F3	CD	BR	AD

F1, F2, F3: Microoperation fields

CD: Condition for branching

BR: Branch field

AD: Address field

Each microoperation below is defined using register transfer statements and is assigned a symbol for use in symbolic microprogram.

Description of CD

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

Description of BR

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	$CAR(2-5) \leftarrow DR(11-14), CAR(0,1,6) \leftarrow 0$

CD (condition) field consists of two bits representing 4 status bits and BR (branch) field (2-bits) used together with address field AD, to choose the address of the next microinstruction.

Microinstruction fields (F1, F2, F3)

F1	Microoperation	Symbol	F2	Microoperation	Symbol	F3	Microoperation	Symbol
000	None	NOP	000	None	NOP	000	None	NOP
001	$AC \leftarrow AC + DR$	ADD	001	$AC \leftarrow AC - DR$	SUB	001	$AC \leftarrow AC \oplus DR$	XOR
010	$AC \leftarrow 0$	CLRAC	010	$AC \leftarrow AC \vee DR$	OR	010	$AC \leftarrow \overline{AC}$	COM
011	$AC \leftarrow AC + 1$	INCAC	011	$AC \leftarrow AC \wedge DR$	AND	011	$AC \leftarrow \text{shl } AC$	SHL
100	$AC \leftarrow DR$	DRTAC	100	$DR \leftarrow M[AR]$	READ	100	$AC \leftarrow \text{shr } AC$	SHR
101	$AR \leftarrow DR(0-10)$	DRTAR	101	$DR \leftarrow AC$	ACTDR	101	$PC \leftarrow PC + 1$	INCPC
110	$AR \leftarrow PC$	PCTAR	110	$DR \leftarrow DR + 1$	INCDR	110	$PC \leftarrow AR$	ARTPC
111	$M[AR] \leftarrow DR$	WRITE	111	$DR(0-10) \leftarrow PC$	PCTDR	111	Reserved	

Here, microoperations are subdivided into three fields of 3-bits each. These 3 bits are used to encode 7 different microoperations. No more than 3 microoperations can be chosen for a microinstruction, one for each field. If fewer than 3 microoperations are used, one or more fields will contain 000 for no operation.

Symbolic Microinstructions

Symbols are used in microinstructions as in assembly language. A symbolic microprogram can be translated into its binary equivalent by a microprogram assembler.

Format of Microinstruction:

Contains five fields: label; micro-ops; CD; BR; AD

Label: may be empty or may specify a symbolic address terminated with a colon

Micro-ops: consists of one, two, or three symbols separated by commas

CD: one of {U, I, S, Z},

Where U: Unconditional Branch

I: Indirect address bit

S: Sign of AC

Z: Zero value in AC

BR: one of {JMP, CALL, RET, MAP}

AD: one of {Symbolic address, NEXT, empty (in case of MAP and RET)}

Symbolic Microprogram (example)

FETCH Routine: During FETCH Read an instruction from memory and decode the instruction and update PC

Sequence of microoperations in the *fetch cycle*:

$AR \leftarrow PC$
$DR \leftarrow M[AR], PC \leftarrow PC + 1$
$AR \leftarrow DR(0-10), CAR(2-5) \leftarrow DR(11-14), CAR(0,1,6) \leftarrow 0$

Symbolic microprogram for the fetch cycle:

	ORG 64			
FETCH:	PCTAR	U	JMP	NEXT
	READ, INCPC	U	JMP	NEXT
	DRTAR	U	MAP	

- Control Storage: 128 20-bit words
- The first 64 words: Routines for the 16 machine instructions
- The last 64 words: Used for other purpose (e.g., fetch routine and other subroutines)
- Mapping: OP-code XXXX into 0XXXX00, the first address for the 16 routines are 0(0 0000 00), 4(0 0001 00), 8, 12, 16, 20, ..., 60

Partial Symbolic Microprogram

Label	Microoperations	CD	BR	AD
ADD:	ORG 0			
	NOP	I	CALL	INDRCT
	READ	U	JMP	NEXT
	ADD	U	JMP	FETCH
BRANCH:	ORG 4			
	NOP	S	JMP	OVER
OVER:	NOP	U	JMP	FETCH
	ARTPC	I	CALL	INDRCT
		U	JMP	FETCH
STORE:	ORG 8			
	NOP	I	CALL	INDRCT
	ACTDR	U	JMP	NEXT
	WRITE	U	JMP	FETCH
EXCHANGE:	ORG 12			
	NOP	I	CALL	INDRCT
	READ	U	JMP	NEXT
	ACTDR, DRTAC	U	JMP	NEXT
	WRITE	U	JMP	FETCH
FETCH:	ORG 64			
	PCTAR	U	JMP	NEXT
	READ, INCPC	U	JMP	NEXT
INDRCT:	DRTAR	U	MAP	
	READ	U	JMP	NEXT
	DRTAR	U	RET	

E.g. the execution of ADD instruction is carried out by the microinstructions at addresses 1 and 2. The first microinstruction reads operand from into DR. The second microinstruction performs an add microoperation with the content of DR AC and then jumps back to the beginning of the fetch routine.

Binary Microprogram

Symbolic microprogram is a convenient form for writing microprograms in a way that people can understand. But this is not a way that the microprogram is stored in memory. It must be translated into binary by means of assembler.

Binary equivalent of a microprogram translated by an assembler for fetch cycle:

Binary address	F1	F2	F3	CD	BR	AD
1000000	110	000	000	00	00	1000001
1000001	000	100	101	00	00	1000010
1000010	101	000	000	00	11	0000000

Binary program for control memory

Micro Routine	Address		Binary Microinstruction					
	Decimal	Binary	F1	F2	F3	CD	BR	AD
ADD	0	0000000	000	000	000	01	01	1000011
	1	0000001	000	100	000	00	00	0000010
	2	0000010	001	000	000	00	00	1000000
	3	0000011	000	000	000	00	00	1000000
BRANCH	4	0000100	000	000	000	10	00	0000110
	5	0000101	000	000	000	00	00	1000000
	6	0000110	000	000	000	01	01	1000011
	7	0000111	000	000	110	00	00	1000000
STORE	8	0001000	000	000	000	01	01	1000011
	9	0001001	000	101	000	00	00	0001010
	10	0001010	111	000	000	00	00	1000000
	11	0001011	000	000	000	00	00	1000000
EXCHANGE	12	0001100	000	000	000	01	01	1000011
	13	0001101	001	000	000	00	00	0001110
	14	0001110	100	101	000	00	00	0001111
	15	0001111	111	000	000	00	00	1000000
FETCH	64	1000000	110	000	000	00	00	1000001
	65	1000001	000	100	101	00	00	1000010
	66	1000010	101	000	000	00	11	0000000
INDRCT	67	1000011	000	100	000	00	00	1000100
	68	1000100	101	000	000	00	10	0000000

Design of Control Unit

F-field decoding

The 9-bits of the microoperation field are divided into 3 subfields of 3 bits each. The control memory output of each subfield must be decoded to provide distinct microoperations. The outputs of the decoders are connected to the appropriate inputs in the processor unit.

Fig below shows 3 decoders and connections that must be made from their outputs.

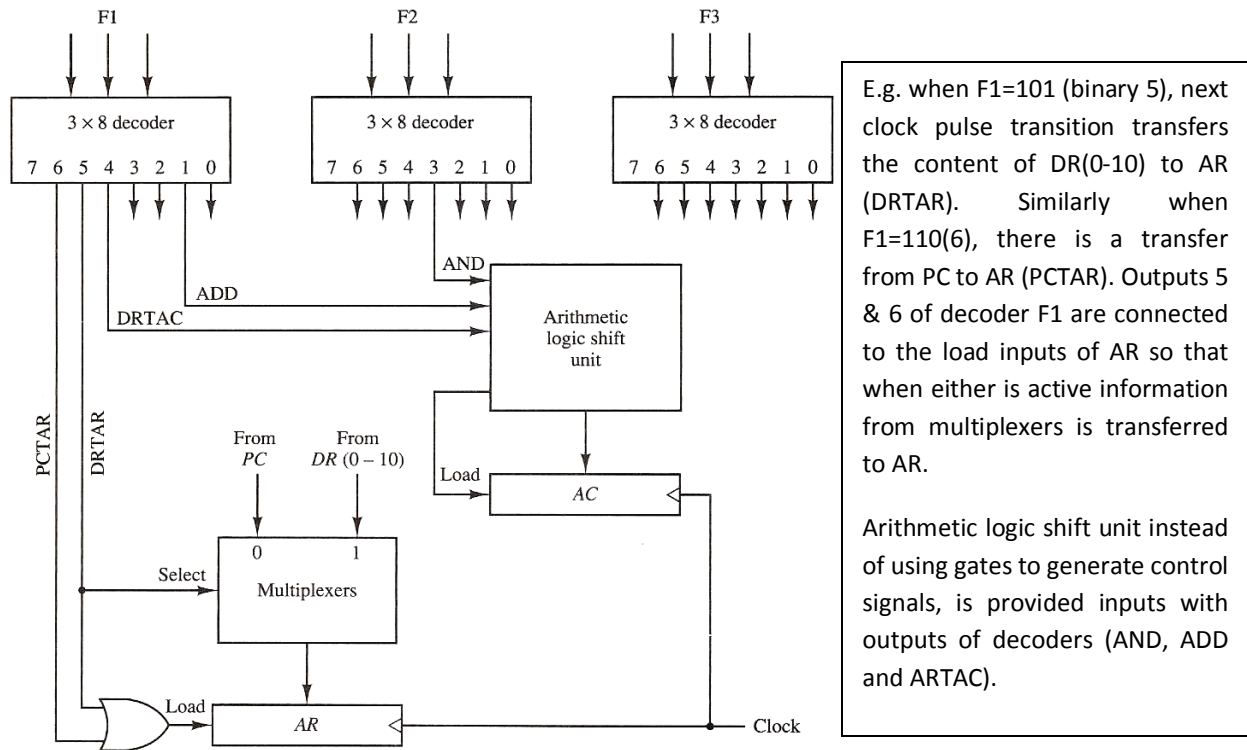


Fig: Decoding of microoperation fields

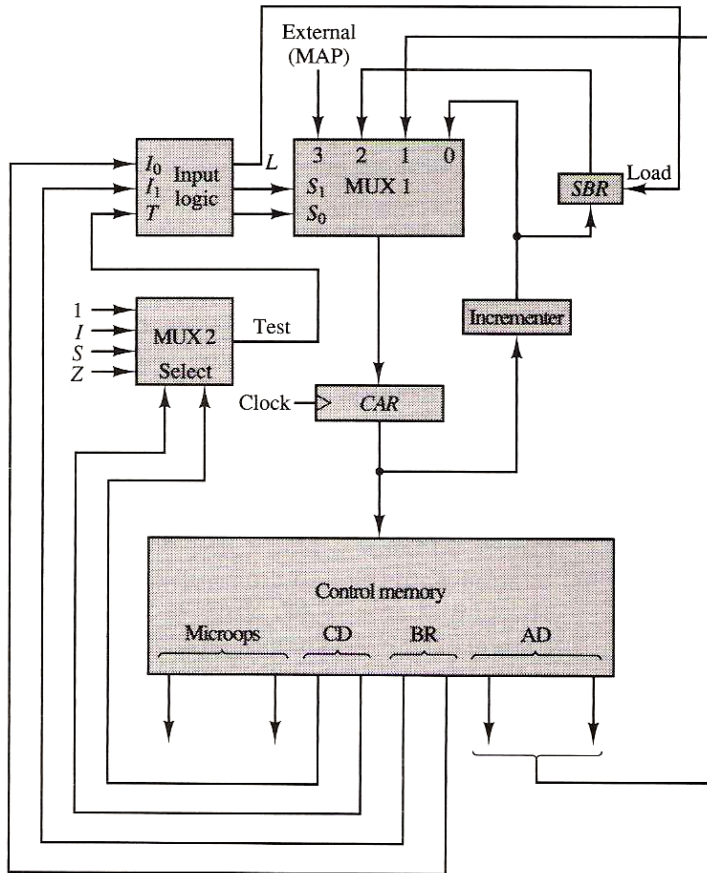
Microprogram Sequencer

Basic components of a microprogrammed control unit are control memory and the circuits that select the next address. This address selection part is called a microprogram sequencer. The purpose of microprogram sequencer is to load CAR so that microinstruction may be read and executed. Commercial sequencers include within the unit an internal register stack to store addresses during microprogram looping and subroutine calls.

Internal structure of a typical microprogram sequencer is shown below in the diagram. It consists of input logic circuit having following truth table.

BR Field	Input			MUX 1		Load <i>SBR</i> <i>L</i>
	<i>I</i> ₁	<i>I</i> ₀	<i>T</i>	<i>S</i> ₁	<i>S</i> ₀	
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

Fig: Input Logic Truth for Microprogram Sequencer



- MUX1 selects an address from one of four sources of and routes it into CAR.
- MUX2 tests the value of selected status bit and result is applied to input logic circuit.
- Output of CAR provides address for the control memory
- Input logic circuit has 3 inputs I_0 , I_1 and T and 3 outputs S_0 , S_1 and L . variables S_0 and S_1 select one of the source addresses for CAR. L enables load input of SBR.
- e.g. when $S_1S_0=10$, MUX input number 2 is selected and establishes a transfer path from SBR to CAR.

Fig: Microprogram sequencer for a control memory

Unit 5

Central Processing Unit (CPU)

Introduction

Part of the computer that performs the bulk of data-processing operations is called the central processing unit (CPU). It consists of 3 major parts:

- **Register set:** stores intermediate data during execution of an instruction
- **ALU:** performs various microoperations required
- **Control unit:** supervises register transfers and instructs ALU

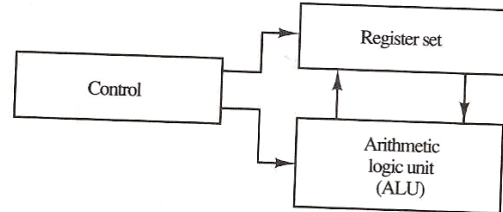
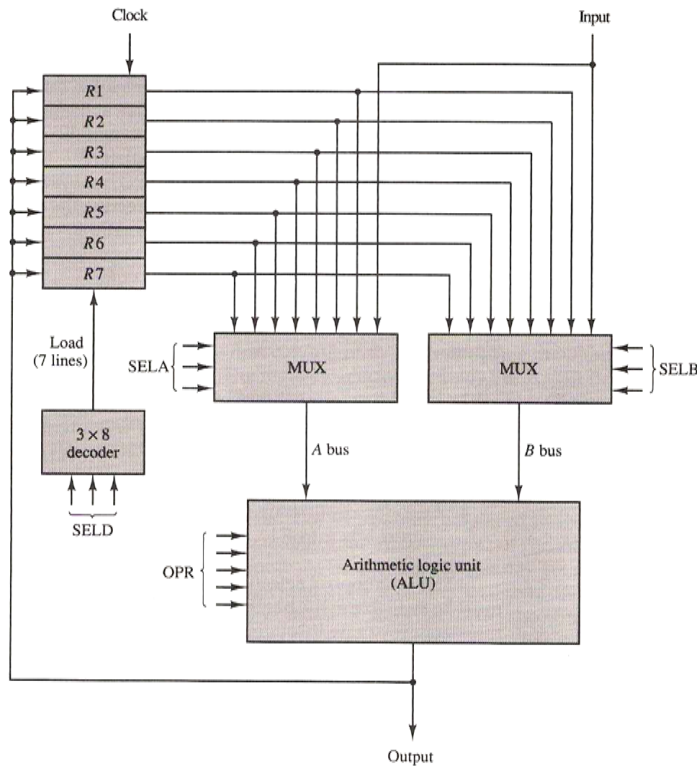


Fig: Major components of CPU

Here, we will proceed from programmer's point of view (as we know CA is the study of computer structure and behavior as seen by the programmer) which includes the instruction formats, addressing modes, instruction set and general organization of CPU registers.

General Register Organization

A bus organization of seven CPU registers is shown below:



(a) Block diagram (register organization)

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.

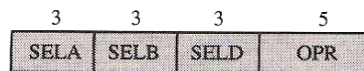
All registers are connected to two multiplexers (MUX) that select the registers for bus A and bus B. Registers selected by multiplexers are sent to ALU. Another selector (OPR) connected to ALU selects the operation for the ALU. Output produced by ALU is stored in some register and this destination register for storing the result is activated by the destination decoder (SELD).

Example: $R1 \leftarrow R2 + R3$

- MUX selector (SELA): $BUS\ A \leftarrow R2$
- MUX selector (SELB): $BUS\ B \leftarrow R3$
- ALU operation selector (OPR): ALU to ADD
- Decoder destination selector (SELD): $R1 \leftarrow Out\ Bus$

Control word

Combination of all selection bits of a processing unit is called control word. Control Word for above CPU is as below:



The 14 bit control word when applied to the selection inputs specify a particular microoperation. Encoding of the register selection fields and ALU operations is given below:

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

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

Example: $R1 \leftarrow R2 - R3$

This microoperation specifies R2 for A input of the ALU, R3 for the B input of the ALU, R1 for the destination register and ALU operation to subtract A-B. Binary control word for this microoperation statement is:

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

Examples of different microoperations are shown below:

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 $\leftarrow R2$	R2	—	None	TSFA	010 000 000 00000
Output \leftarrow Input	Input	—	None	TSFA	000 000 000 00000
$R4 \leftarrow sh1\ R4$	R4	—	R4	SHLA	100 000 100 11000
$R5 \leftarrow 0$	R5	R5	R5	XOR	101 101 101 01100

Stack Organization

This is useful *last-in, first-out* (LIFO) list (actually storage device) included in most CPU's. Stack in digital computers is essentially a memory unit with a stack pointer (SP). SP is simply an address register that points stack top. Two operations of a stack are the insertion (push) and deletion (pop) of items. In a computer stack, nothing is pushed or popped; these operations are simulated by incrementing or decrementing the SP register.

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.

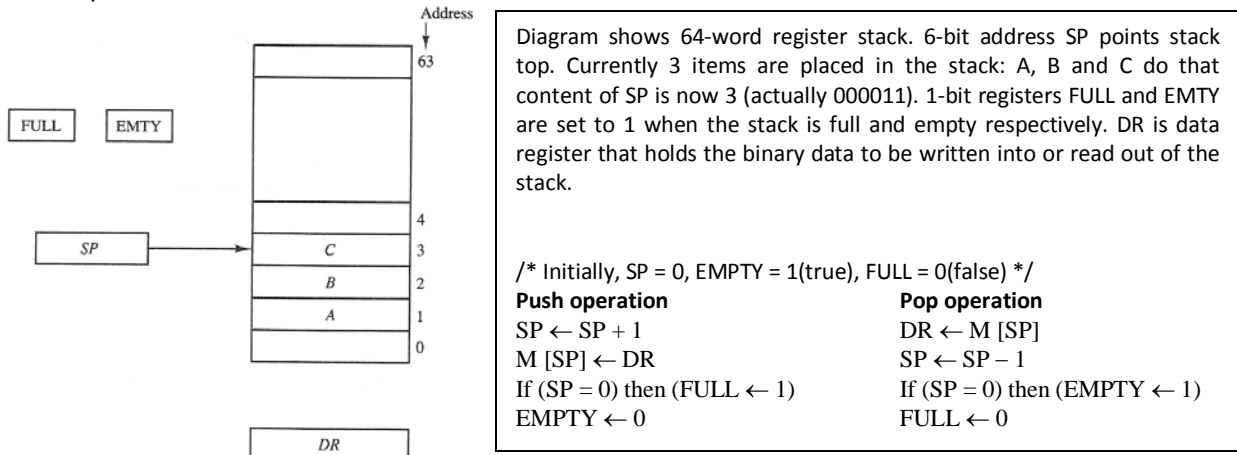
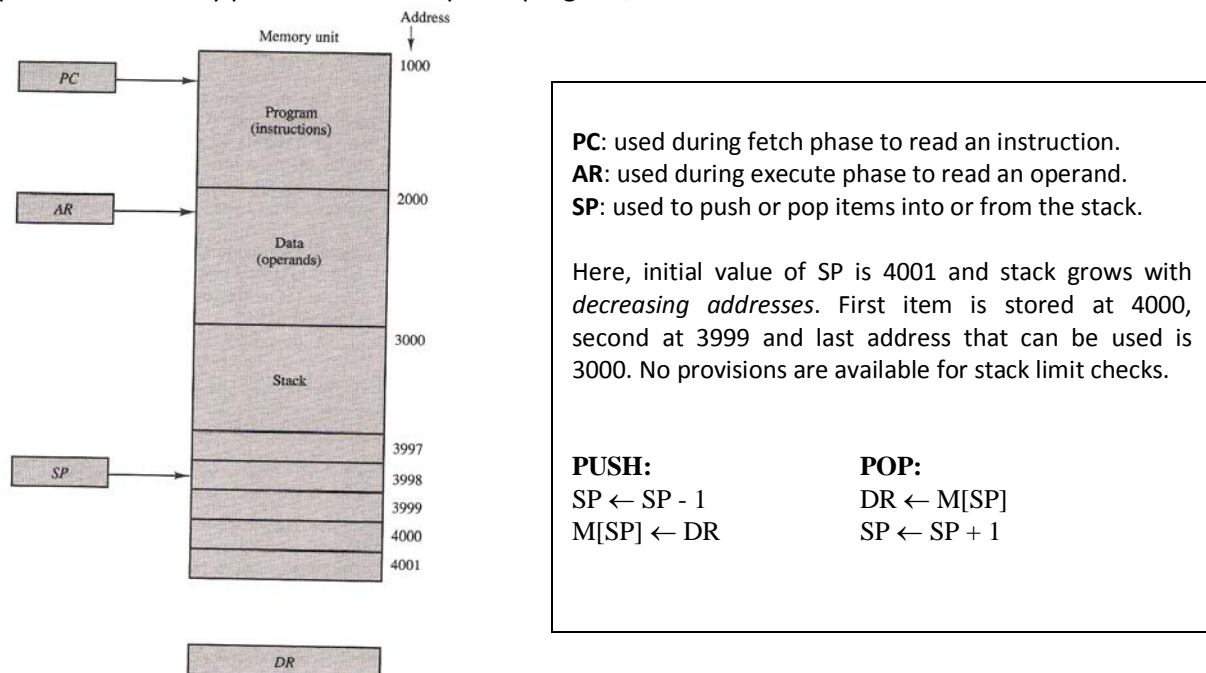


Fig: Block diagram of a 64-word stack

Memory stack

A portion of memory can be used as a stack with a processor register as a SP. Figure below shows a portion of memory partitioned into 3 parts: program, data and stack.



Processor Organization

In general, most processors are organized in one of 3 ways:

1. Single register (Accumulator) organization

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

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

2. General register organization

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

```
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]
```

3. Stack organization

- 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.

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

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.

Assembly language program to evaluate $X = (A + B) * (C + D)$:

```
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
```

- Results in short programs
- Instruction becomes long (many bits)

• Two-Address Instructions

These instructions are most common in commercial computers.

Program to evaluate $X = (A + B) * (C + D)$:

```

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

```

- Tries to minimize the size of instruction
- Size of program is relatively larger.

- **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.

Program to evaluate $X = (A + B) * (C + D)$:

```

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

```

- Memory access is only limited to load and store
- Large program size

- **Zero-Address Instructions**

A stack-organized computer uses this type of instructions.

Program to evaluate $X = (A + B) * (C + D)$:

```

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

```

The name “zero-address” is given to this type of computer because of the absence of an address field in the computational instructions.

Addressing Modes

I am repeating it again guys: "Operation field of an instruction specifies the operation that must be executed on some data stored in computer register or memory words". The way operands (data) are chosen during program execution depends on the addressing mode of the instruction. So, *addressing mode* specifies a rule for interpreting or modifying the address field of the instruction before the operand is actually referenced.

We use variety of addressing modes to accommodate one or both of following provisions:

- To give programming versatility to the user (by providing facilities as: pointers to memory, counters for loop control, indexing of data and program relocation)
- To use the bits in the address field of the instruction efficiently

Types of addressing modes

▪ **Implied Mode**

Address of the operands is specified implicitly in the definition of the instruction.

- No need to specify address in the instruction
- Examples from Basic Computer CLA, CME, INP
ADD X;
PUSH Y;

▪ **Immediate Mode**

Instead of specifying the address of the operand, operand itself is specified in the instruction.

- No need to specify address in the instruction
- However, operand itself needs to be specified
- Sometimes, require more bits than the address
- Fast to acquire an operand

▪ **Register Mode**

Address specified in the instruction is the address of a register

- Designated operand need to be in a register
- Shorter address than the memory address
- A k-bit address field can specify one of 2^k registers.
- Faster to acquire an operand than the memory addressing

▪ **Register Indirect Mode**

Instruction specifies a register which contains the memory address of the operand.

- Saving instruction bits since register address is shorter than the memory address
- Slower to acquire an operand than both the register addressing or memory addressing
- EA (effective address) = content of R.

▪ **Autoincrement or Autodecrement Mode**

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**

Instruction specifies the memory address which can be used directly to access the memory

- Faster than the other memory addressing modes
- Too many bits are needed to specify the address for a large physical memory Space
- EA= IR(address)

▪ **Indirect Addressing Mode**

- The address field of an instruction specifies the address of a memory location that contains the address of the operand
- When the abbreviated address is used large physical memory can be addressed with a relatively small number of bits
- Slow to acquire an operand because of an additional memory access
- $EA = M[IR(\text{address})]$

▪ **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

3 different Relative Addressing Modes:

* **PC Relative Addressing Mode:**

- $EA = PC + IR(\text{address})$

* **Indexed Addressing Mode**

- $EA = IX + IR(\text{address})$ { IX is index register }

* **Base Register Addressing Mode**

- $EA = BAR + IR(\text{address})$

Numerical Example (Addressing modes)

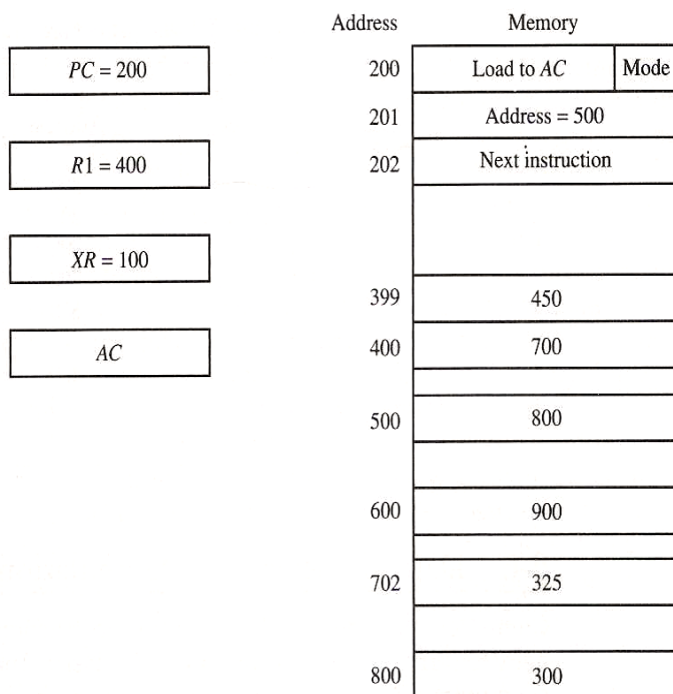


Fig: numerical example of addressing modes

→ 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 here).

→ 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.

→ **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.

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

Direct address	EA = 500 AC content = 800	// AC ← M[500]
Immediate operand	EA = 201 AC content = 500	// AC ← 500
Indirect address	EA = 500 AC content = 300	// AC ← M[M[500]]
Relative address	EA = 500 AC content = 325	// AC ← M[PC+500]
Indexed address	EA = 500 AC content = 900	// AC ← (IX+500)
Register	EA = 500 AC content = 400	// AC ← R1
Register indirect	EA = 400 AC content = 700	// AC ← M[R1]
Autoincrement	EA = 500 AC content = 700	// AC ← (R1)
Autodecrement	EA = 399 AC content = 450	//AC ← -(R)

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.

HEY!, different computer use different mnemonics for the same instruction name.

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)+	$AC \leftarrow M[R1], R1 \leftarrow R1 + 1$

Table: Recommended assembly language conventions for load instruction in different addressing modes

Data manipulation Instructions

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

4. Arithmetic instructions
5. Logical and bit manipulation instructions
6. Shift instructions

These instructions are similar to the microoperations in unit3. But actually; each instruction when executed must go through the *fetch phase* to read its binary code value from memory. The operands must also be brought into registers according to the rules of different addressing mode. And the last step of executing instruction is implemented by means of microoperations listed in unit 3.

Arithmetic instructions

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

- Increment (decrement) instr. adds 1 to (subtracts 1 from) the register or memory word value.
- Add, subtract, multiply and divide instructions may operate on different data types (fixed-point or floating-point, binary or decimal).

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

- Clear instr. causes specified operand to be replaced by 0's.
- Complement instr. produces the 1's complement.
- AND, OR and XOR instructions produce the corresponding logical operations on individual bits of the operands.

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:

- Logical shifts
- Arithmetic shifts
- Rotate-type operations

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

- Table lists 4 types of shift instructions.
- 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. But sometimes it is necessary to condition the data processing instructions which change the PC value accidentally causing a break in the instruction execution and branching to different program segments.

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.

RISC and CISC

An important aspect of computer architecture is the design of the instruction set for the processor. Early computers had small and simple instruction sets, forced mainly by the need to minimize the hardware used to implement them. As digital hardware became cheaper with the advent of ICs, computer instructions tended to increase both in number and complexity. Many computers have instruction sets that include 100-200 instructions employing variety of data types and large number of addressing modes and are classified as **Complex Instruction Set Computer (CISC)**. In early 1980s, a number of computer designers recommended that computers use fewer instructions with simple constructs so as to execute them faster with in CPU without using memory as often. This type of computer is classified as a **Reduced Instruction Set Computer (RISC)**.

CISC

One reason to provide a complex instruction set is the desire to simplify the compilation (done by compilers to convert high level constructs to machine instructions) and improve the overall computer performance.

Essential goal: Provide a single machine instruction for each statement in high level language.

Examples: Digital Equipment Corporation VAX computer and IBM 370 computer.

Characteristics:

1. A large no of instructions - typically from 100 to 250 instructions.
2. A large variety of addressing modes – typically form 5 to 20.
3. Variable-length instruction formats
4. Instructions that manipulate operands in memory

RISC

Main Concept: Attempt to reduce execution time by simplifying the instruction set of the computer.

Characteristics:

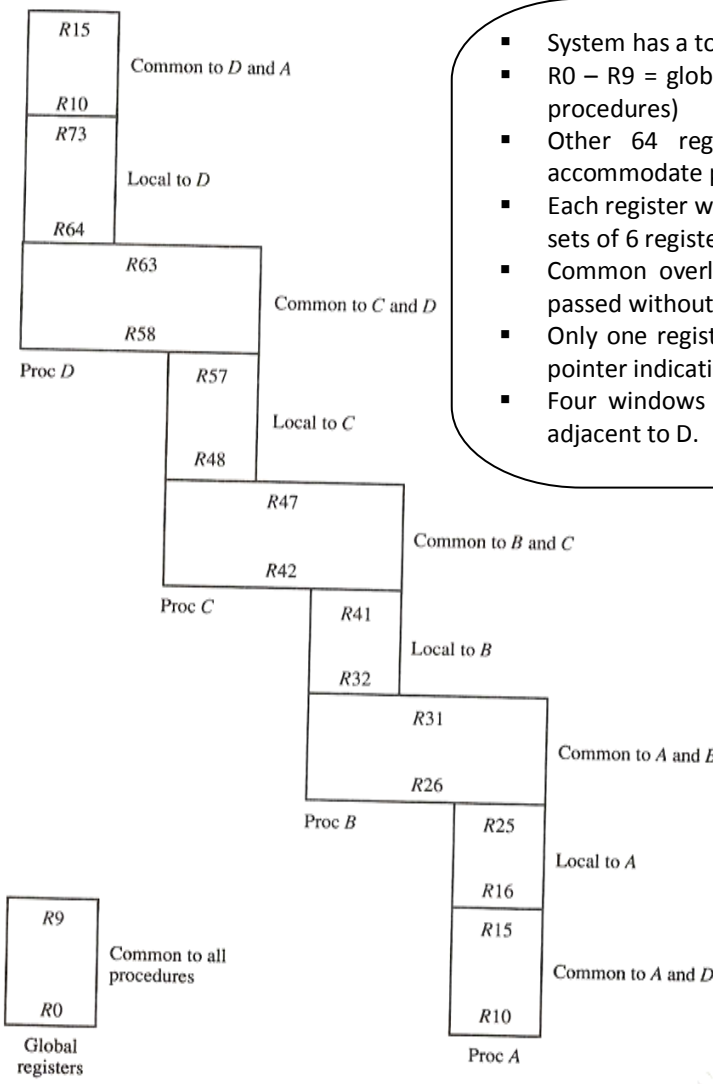
1. Relatively few instructions and addressing modes.
2. Memory access limited to load and store instructions
3. All operations done with in CPU registers (relatively large no of registers)
4. Fixed-length, easily decoded instruction format

5. Single cycle instruction execution
6. Hardwired rather than Microprogrammed control
7. Use of overlapped-register windows to speed procedure call and return
8. Efficient instruction pipeline

Overlapped Resister Windows

Procedure call and return occurs quite often in high-level programming languages. When translated into machine language, procedure call produces a sequence of instructions that **save register values, pass parameters** needed for the procedure and then **calls a subroutine** to execute the body of the procedure. After a procedure return, the program restores the old register values, passes results to the calling program and returns from the subroutine. Saving & restoring registers and passing of parameters & results involve time consuming operations.

A characteristic of some RISC processors is use of overlapped register windows to provide the passing of parameters and avoid need for saving & restoring register values. The concept of overlapped register windows is illustrated below:



- System has a total of 74 registers (Just an example)
- R0 – R9 = global registers (hold parameters shared by all procedures)
- Other 64 registers are divided into 4 windows to accommodate procedures A, B, C and D.
- Each register window consists of 10 local registers and two sets of 6 registers common to adjacent windows.
- Common overlapped registers permit parameters to be passed without the actual movement of data
- Only one register window is activated at any time with a pointer indicating the active window.
- Four windows have a circular organization with A being adjacent to D.

Example: Procedure A calls B

- Registers R26 to R31 are common to both procedures and therefore procedure A stores the parameters for procedure B in these registers.
- B uses local registers R32 through R41 for local variable storage.
- When B is ready to return at the end of its computation, programs stores results in registers R26-R31 and transfers back to the register window of procedure A.

Fig: Overlapped Resister Windows

In general, the organization of register windows will have following relationships:

- Number of global registers = G
- Number of local register in each window = L
- Number of registers common to windows = C
- Number of windows = W

Now,

- Window size = $L + 2C + G$
- Register file = $(L+C)W + G$ (total number of register needed in the processor)

Example: In above fig, $G = 10$, $L = 10$, $C = 6$ and $W = 4$. Thus window size = $10+12+10 = 32$ registers and register file consists of $(10+6)*4+10 = 74$ registers.

Exercises: textbook chapter 8 → 8.12 (do it yourself)

Unit 6

Fixed point Computer Arithmetic

Arithmetic instructions manipulate data to produce solution for computational problems. The 4 basic arithmetic operations are addition, subtraction, multiplication and division. From these 4, it is possible to formulate other scientific problems by means of numerical analysis methods. Here, we'll discuss these 4 operations only on fixed-point binary data (there are other types too, viz. floating point binary data, binary-coded decimal data) and hence the unit named.

Addition and Subtraction

There are 3 ways of representing negative fixe-point binary numbers: signed magnitude, signed 1's complement or signed 2's complement. Singed 2's complemented form used most but occasionally we deal with signed magnitude representation.

Addition and Subtraction with signed-magnitude data

Everyday arithmetic calculations with paper and pencil for signed binary numbers are straight forward and are helpful on deriving hardware algorithm. When two signed numbers A and B are added are added are subtracted, we find 8 different conditions to consider as described in following table:

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

Note: Brackets () for subtraction

Addition (subtraction) algorithm:
 when the signs of A and B are identical (different), add magnitudes and attach the sign of A to result. When the signs of A and b are different (identical), compare the magnitudes and subtract the smaller form larger.

Table: addition and subtraction of signed-magnitude numbers

Hardware Implementation

To implement the two arithmetic operations with hardware, we have to store numbers into two register A and B. let A_s and B_s be two flip-flops that holds corresponding signs. The result is transferred to A and A_s . A and A_s together form a accumulator.

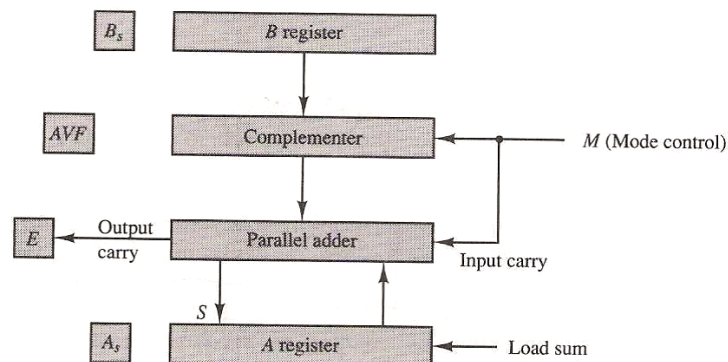


Fig: hardware for signed-magnitude addition and subtraction

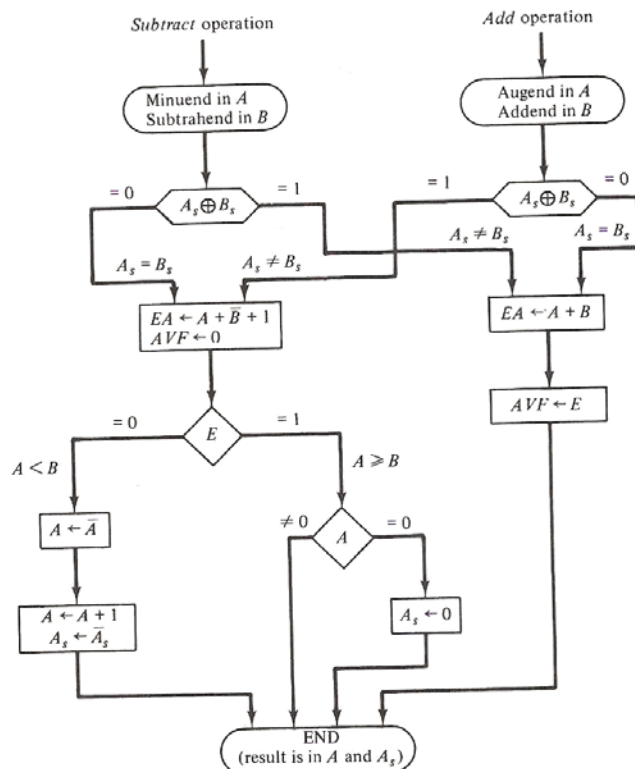
We need:

- Consists of two registers A and B and sign flip-flops A_s and B_s .
- **A magnitude comparator:** to check if $A > B$, $A < B$ or $A = B$.
- **A parallel adder:** to perform $A + B$
- **Two parallel subtractors:** for $A - B$ and $B - A$
- The **sign** relationships are determined from an exclusive-OR gate with A_s and B_s as inputs.

Block Diagram Description: hardware above consists of registers A and B and sign flip-flops A_s and B_s . subtraction is done by adding A to the 2's complement of B. Output carry is transferred to flip-flop E, where it can be checked to determine the relative magnitude of two numbers. Add-overflow flip-flop AVF holds overflow bit when A and B are added. Addition of A and B is done through the parallel adder. The S output of adder is applied to A again. The complemeter provides an output of B or B' depending on mode input M. Recalling unit 2, when $M = 0$, the output of B is transferred to the adder, the input carry is 0 and thus output of adder is $A+B$. when $M=1$, 1's complement of B is applied to the adder, input carry is 1 and output is $S = A+B'+1$ (i.e. $A-B$).

Hardware Algorithm

The flowchart for the H/W algorithm is given below:



→ A_s and B_s are compared by an exclusive-OR gate. If output = 0, signs are identical, if 1 signs are different.

→ For *add* operation identical signs dictate addition of magnitudes and for *subtraction*, different magnitudes dictate magnitudes be **added**. Magnitudes are added with a microoperation $EA \leftarrow A+B$ (EA is a register that combines A and E). if $E = 1$, overflow occurs and is transferred to AVF.

→ Two magnitudes are subtracted if signs are different for add operation and identical for subtract operation. Magnitudes are subtracted with a microoperation $EA \leftarrow A+B'+1$. No overflow occurs if the numbers are subtracted so AVF is cleared to 0. $E=1$ indicates $A \geq B$ and number (this number is checked again for 0 to make positive 0 [$A_s=0$]) in A is correct result. $E=0$ indicates $A < B$, so we take 2's complement of A.

Fig: flowchart for add and subtract operations

Addition and Subtraction with signed 2's complement data

Guys, refer unit 1 once, addition and subtraction with signed 2's complement data are introduced there. Anyway, in signed 2's complement representation, the leftmost bit represents sign (0-positive and 1-negative). If sign bit is 1, entire number is represented in 2's complement form ($+33=00100001$ and $-33=2's \text{ complement of } 00100001 = 11011111$).

Addition: sign bits treated as other bits of the number. Carry out of the sign bit is discarded.

Subtraction: consists of first taking 2's complement of the subtrahend and then adding it to minuend.

When two numbers of n-digits each are added and the sum occupies n+1 bits, overflow occurs which is detected by applying last two carries out of the addition to XOR gate. The overflow occurs when output of the gate is 1.

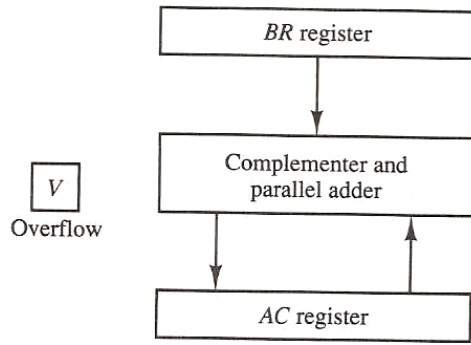


Fig: hardware for signed-2's complement addition and subtraction

→ Register configuration is same as signed-magnitude representation except sign bits are not separated. The leftmost bits in AC and BR represent sign bits.

→ Significant difference: sign bits are added are subtracted together with the other bits in complementer and parallel adder. The overflow flip-flop V is set to 1 if there is an overflow. Output carry in this case is discarded.

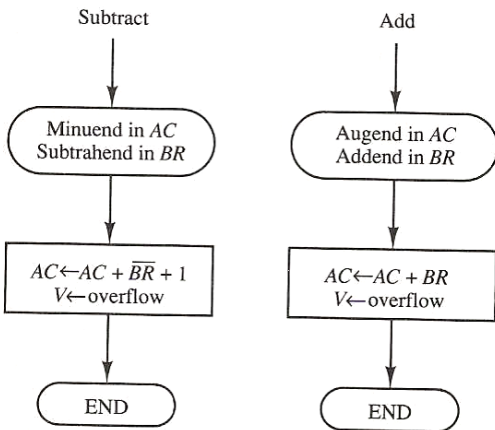


Fig: algorithm for addition & subtraction of numbers in signed-2's complement representation

Example: $33 + (-35)$
 $AC = 33 = 00100001$
 $BR = -35 = 2's \text{ complement of } 35 = 11011101$
 $AC + BR = 11111110 = -2$ which is the result

Comparing this algorithm with its signed-magnitude counterpart, it is much easier to add and subtract numbers. For this reason most computers adopt this representation over the more familiar signed-magnitude.

Multiplication

Signed-magnitude representation

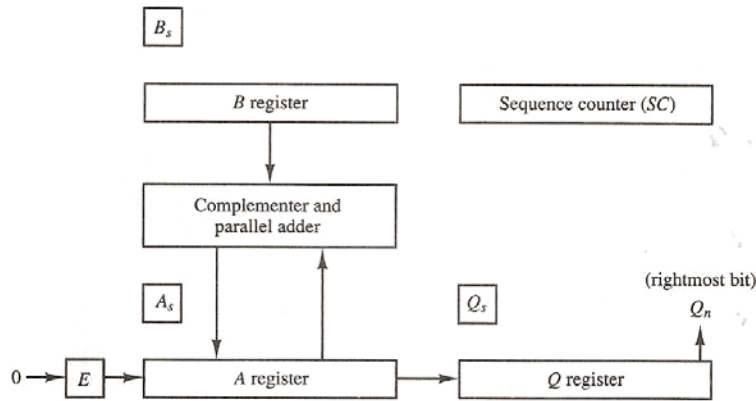
For this representation, multiplication is done by a process of successive shift and adds operations. As an example:

23	10111	Multiplicand
19	× 10011	Multiplier
	10111	
	10111	
	00000	+
	00000	
	10111	
437	110110101	Product

Process consists of looking successive bits of the multiplier, least significant bits first. If the multiplier bit is 1, the multiplicand is copied down; otherwise, zeros are copied down. Numbers copied down in successive lines are shifted one position. Shifted left one position. Finally, numbers are added to form a product.

Hardware implementation for signed-magnitude data

It needs same hardware as that of addition and subtraction of signed-magnitude. In addition it needs two more registers Q and SC.

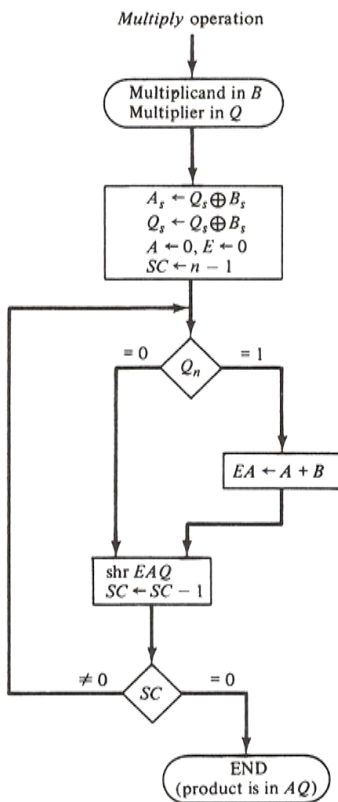


- Successively accumulate partial products and shift it right.
- $Q \leftarrow$ multiplier and $Q_s \leftarrow$ sign.
- $SC \leftarrow$ no. of bits in multiplier (magnitude only).
- SC is decremented after forming each partial product. When SC is 0, process halts and final product is formed.
- $B \leftarrow$ multiplicand, $B_s \leftarrow$ sign
- Sum of A and B forms a partial product

Fig: Hardware for multiply operation

Hardware Algorithm

Flowchart below shows a hardware multiply algorithm.



Example: $B = 10111$ (Multiplicand)
 $Q = 10011$ (Multiplier)

Operation	E	A	Q	SC
Initial conf.	0	00000	10011	101
Iteration 1 ($Q_n = 1$) $EA \leftarrow A+B$ PP1-->	0	00000 + 10111 ----- 10111		
shr EAQ, $SC \leftarrow SC-1$	0	01011	11001	100
Iteration 2 ($Q_n = 1$) $EA \leftarrow A+B$ PP2-->	1	01011 + 10111 ----- 00010	11001	
shr EAQ, $SC \leftarrow SC-1$	0	10001	01100	011
Iteration 3 ($Q_n = 0$) $EA \leftarrow A+B$ PP3-->	0	01000	10110	010
shr EAQ, $SC \leftarrow SC-1$	0	00100	01011	001
Iteration 4 ($Q_n = 0$) $EA \leftarrow A+B$ PP3-->	0	00100 + 10111 ----- 11011	01011	
shr EAQ, $SC \leftarrow SC-1$	0	01101	10101	000
Final Product in AQ		0110110101		

Signed 2's complement representation

Booth multiplication Algorithm

Booth algorithm gives a procedure for multiplying binary integers in signed 2's complement notation.

Inspiration: String of 1's in the multiplier from bit weight 2^k to weight 2^m can be treated as $2^{k+1}-2^m$. As an example, binary number 001110 (+14) has string of 1's from 2^3 to 2^1 ($k=3, m=1$). So, this number can be represented as $2^{k+1} - 2^m = 2^4 - 2^1 = 16 - 2 = 14$ (case is similar for -14 (110010) = $-2^4+2^2-2^1$). Thus, $M * 14 = M * 2^4 - M * 2^1$; product can be obtained by shifting multiplicand M four times left and subtracted M shifted left once.

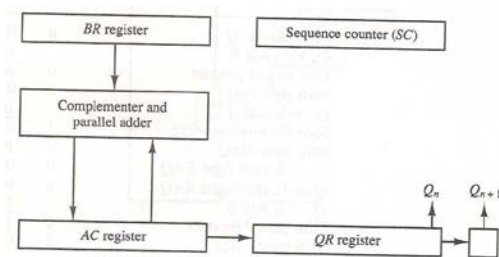
As in other multiplication schemes, Booth algorithm also requires examination of multiplier bits and shifting of the partial product. Prior to shifting multiplicand may be:

Subtracted <-- upon the encountering first least significant 1 in the string of 1's in the multiplier.

Added <-- upon encountering first 0 (left of it must be 1) in string of 0's in the multiplier.

Unchanged <-- when multiplier bit (Q_n) is identical to previous multiplier bit (Q_{n+1})

Hardware for Booth algorithm



- Here, sign bits are not separated.
- Registers A, B and Q are renamed to AC, BR and QR.
- Extra flip-flop Q_{n+1} appended to QR is needed to store almost lost right shifted bit of the multiplier (which along with current Q_n gives information about bit sequencing of multiplier, in fact no. of 1's gathered together).
- Pair $Q_n Q_{n+1}$ inspect double bits of the multiplier.

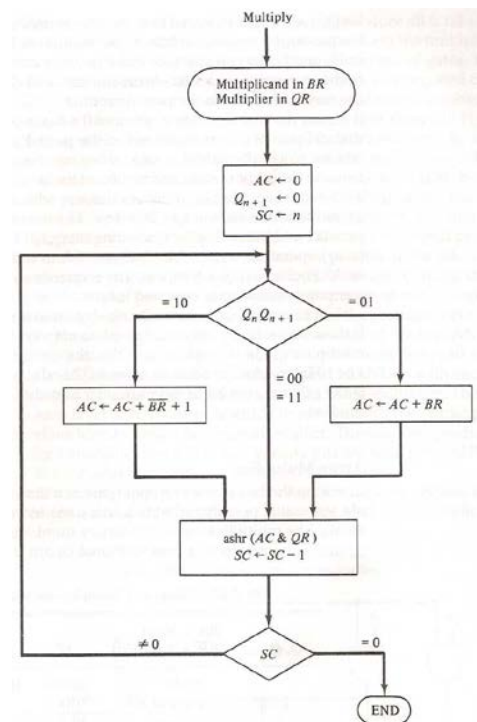
Hardware Booth algorithm

Numerical Example: Booth algorithm

BR = 10111 (Multiplicand)

QR = 10011 (Multiplier)

$Q_n Q_{n+1}$	BR = 10111 $\overline{BR} + 1 = 01001$	AC	QR	Q_{n+1}	SC
	Initial	00000	10011	0	101
1 0	Subtract BR	$\begin{array}{r} 01001 \\ -01001 \\ \hline 00100 \end{array}$			
	ashr	00100	11001	1	100
1 1	ashr	00010	01100	1	011
0 1	Add BR	$\begin{array}{r} 01111 \\ +01100 \\ \hline 11001 \end{array}$			
	ashr	11100	10110	0	010
0 0	ashr	11110	01011	0	001
1 0	Subtract BR	$\begin{array}{r} 01001 \\ -01001 \\ \hline 00111 \end{array}$			
	ashr	00011	10101	1	000



Array Multiplier

Checking the bits of the multiplier one at a time and forming partial products is a sequential operation requiring sequence of add and shift microoperations. The multiplication of two binary numbers can be done with one microoperation by using combinational circuit that forms product bits all at once. This is a fast way of multiplying two numbers since all it takes is the time to propagate through the gates that form the **multiplication array**.

Consider multiplication of two 2-bit numbers: Multiplicand = $\mathbf{b_1b_0}$, Multiplier = $\mathbf{a_1a_0}$, Product = $\mathbf{c_3c_2c_1c_0}$

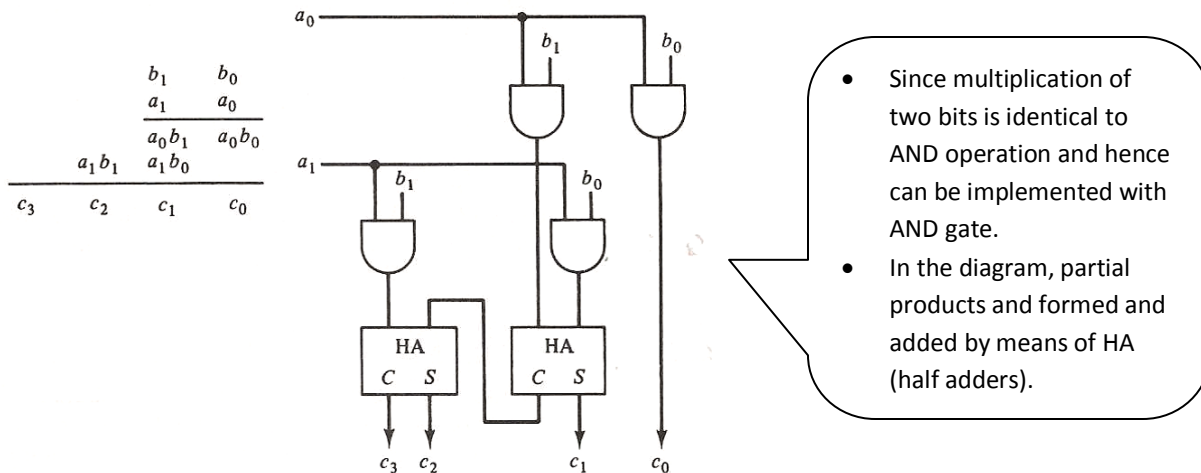


Fig: 2-bit by 2-bit array multiplier

A combinational circuit binary multiplier with more bits can be constructed in similar fashion. For j multiplier bits and k multiplicand bits, we need $j \cdot k$ AND gates and $(j-1)$ k -bit adders to produce a product of $j+k$ bits.

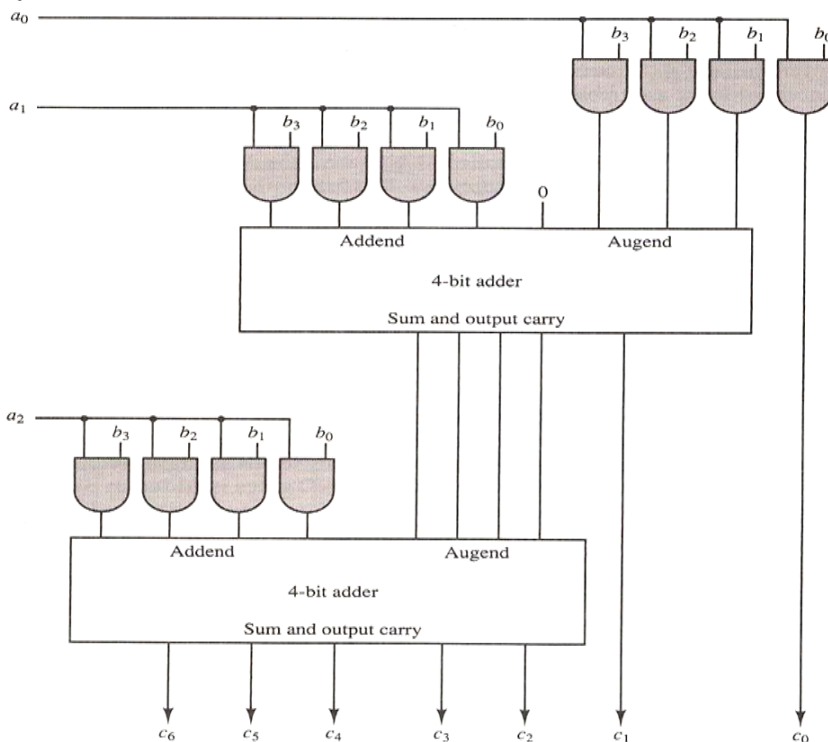


Fig: 4-bit by 3-bit array multiplier

Division Algorithms

Division of fixed-point binary numbers in signed-magnitude representation is done with successive compare, shift and subtract operations.

Example:

Divisor:	11010	Quotient = Q
$B = 10001$)0111000000	Dividend = A
	01110	5 bits of $A < B$, quotient has 5 bits
	011100	6 bits of $A \geq B$
	-10001	Shift right B and subtract; enter 1 in Q
	-010110	7 bits of remainder $\geq B$
	--10001	Shift right B and subtract; enter 1 in Q
	--001010	Remainder $< B$; enter 0 in Q ; shift right B
	---010100	Remainder $\geq B$
	----10001	Shift right B and subtract; enter 1 in Q
	----000110	Remainder $< B$; enter 0 in Q
	-----00110	Final remainder

- Easier than decimal since quotient digits are 0 or 1.
- $B \leftarrow$ divisor, $A \leftarrow$ dividend, $Q \leftarrow$ Quotient
- Process consists of comparing a **partial remainder** with a divisor.

Hardware Implementation for Signed-Magnitude Data

While implementing division in digital system, we adopt slightly different approach. Instead of shifting divisor right, the partial remainder (or dividend) is shifted left. Hardware is similar to multiplication algorithm (not booth). Register EAQ is now shifted left with 0 inserted into Q_n (Obviously, previous value of E is lost). (I am not redrawing the diagram guys, it's all same as multiplication but EAQ is shifted left so change the direction of arrows at bottom).

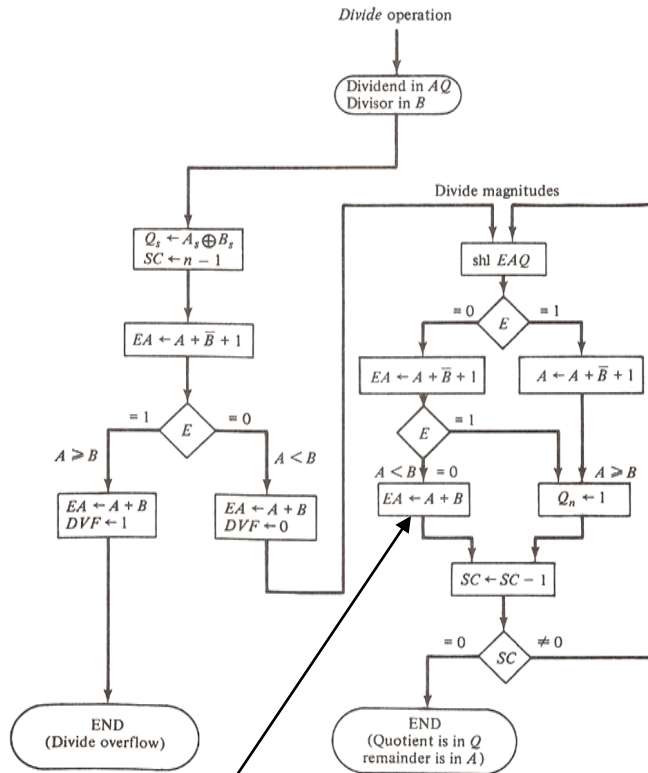
Divide Overflow

- Division operation may result in a quotient with an overflow when working with finite size registers.
- Storing divisor in n -bit register and dividend in $2n$ -bit registers, then if quotient occupies $n+1$ bits, we say divide-overflow has occurred (since $n+1$ bit quotient can not be stored in standard n -bit Q -register and/or memory word).
- Talking about special case: **size** (dividend) = $2 * \text{size}$ (divisor). Divide-overflow condition will occur if *high-order half bits of the dividend* \geq divisor. This condition is detected by DVF (**D**ivide-**o**verflow **F**lip-flop).

Handling of overflow: its programmer's responsibility to detect DVF and take corrective measure. The best way is to use floating point data.

Hardware algorithm (Restoring algorithm)

Flowchart for hardware algorithm is shown below:



- B: Divisor, AQ: Dividend
- If $A \geq B$ (oh yes, magnitudes are compared subtracting one from another and testing E flip-flop), DVF is set and operation is terminated prematurely. If $A < B$, no overflow and dividend is restored by adding B to A (since B was subtracted previously to compare magnitudes).
- Division starts by left shifting AQ (dividend) with high order bit shifted to E. Then $E=1$, $EA > B$ so B is subtracted from EA and Q_n is set to 1. If $E=0$, result of subtraction is stored in EA, again E is tested. $E=1$ signifies $A \geq B$, thus Q_n is set to 1 and $E=0$ denotes $A < B$, so original number is **restored** by adding B to A and we leave 0 in Q_n .
- Process is repeated again with register A holding partial remainder. After $n-1$ times Q contains magnitude of Quotient and A contains remainder. Quotient sign in Q_s and remainder sign in A_s .

Fig: flow chart for divide operation

This is the restoring step. Different variant of division algorithm only have distinction at this step.

HEY! You may face Nonrestoring or comparison methods as long questions. Don't blame me for that since everything (hardware implementation and hardware algorithm) is same. Only difference is at this step.

!!!HEY: In each iteration, just after left-shifting EAQ, we test it for 0 or 1 and proceed accordingly which is not noted in example (Example is taken such that E is always 0 just after shifting).

Numerical Example: Binary division with digital hardware

	E	A	Q	SC
Divisor $B = 10001$,				
$\bar{B} + 1 = 01111$				
Dividend:		01110	00000	5
shl EAQ	0	11100	00000	
add $\bar{B} + 1$		01111		
$E = 1$	1	01011		
Set $Q_n = 1$	1	01011	00001	4
shl EAQ	0	10110	00010	
Add $\bar{B} + 1$		01111		
$E = 1$	1	00101		
Set $Q_n = 1$	1	00101	00011	3
shl EAQ	0	01010	00110	
Add $\bar{B} + 1$		01111		
$E = 0$; leave $Q_n = 0$	0	11001	00110	
Add B		10001		
Restore remainder	1	01010		2
shl EAQ	0	10100	01100	
Add $\bar{B} + 1$		01111		
$E = 1$	1	00011		
Set $Q_n = 1$	1	00011	01101	1
shl EAQ	0	00110	11010	
Add $\bar{B} + 1$		01111		
$E = 0$; leave $Q_n = 0$	0	10101	11010	
Add B		10001		
Restore remainder	1	00110	11010	0
Neglect E				
Remainder in A:		00110		
Quotient in Q:			11010	

Other division algorithms

Method described above is **restoring method** in which *partial remainder* is restored by adding the divisor to the negative result. Other methods:

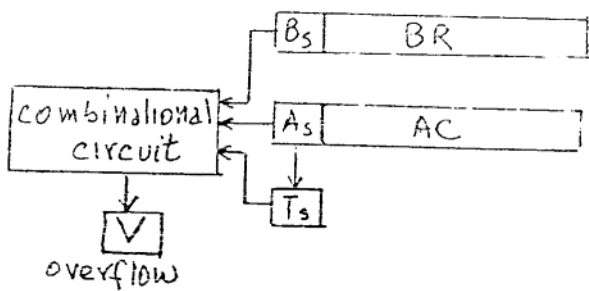
Comparison method: A and B are compared prior to subtraction. Then if $A \geq B$, B is subtracted from A. if $A < B$ nothing is done. The partial remainder is then shifted left and numbers are compared again. Comparison inspects end-carry out of the parallel adder before transferring to E.

Nonrestoring method: In contrast to restoring method, when $A - B$ is negative, B is not added to restore A but instead, negative difference is shifted left and then B is added. How is it possible? Let's argue:

- In flowchart for restoring method, when $A < B$, we restore A by operation $A - B + B$. Next time in a loop, this number is shifted left (multiplied by 2) and B subtracted again, which gives: $2(A - B + B) - B = 2A - B$.
- In Nonrestoring method, we leave $A - B$ as it is. Next time around the loop, the number is shifted left and B is added: $2(A - B) + B = 2A - B$ (same as above).

Exercises: textbook ch 10 → 10.5, 10.9, 10.10, 10.15

10.5 solution



Boolean function for circuit:
 $V = T_s' B_s' A_s + T_s B_s A_s'$

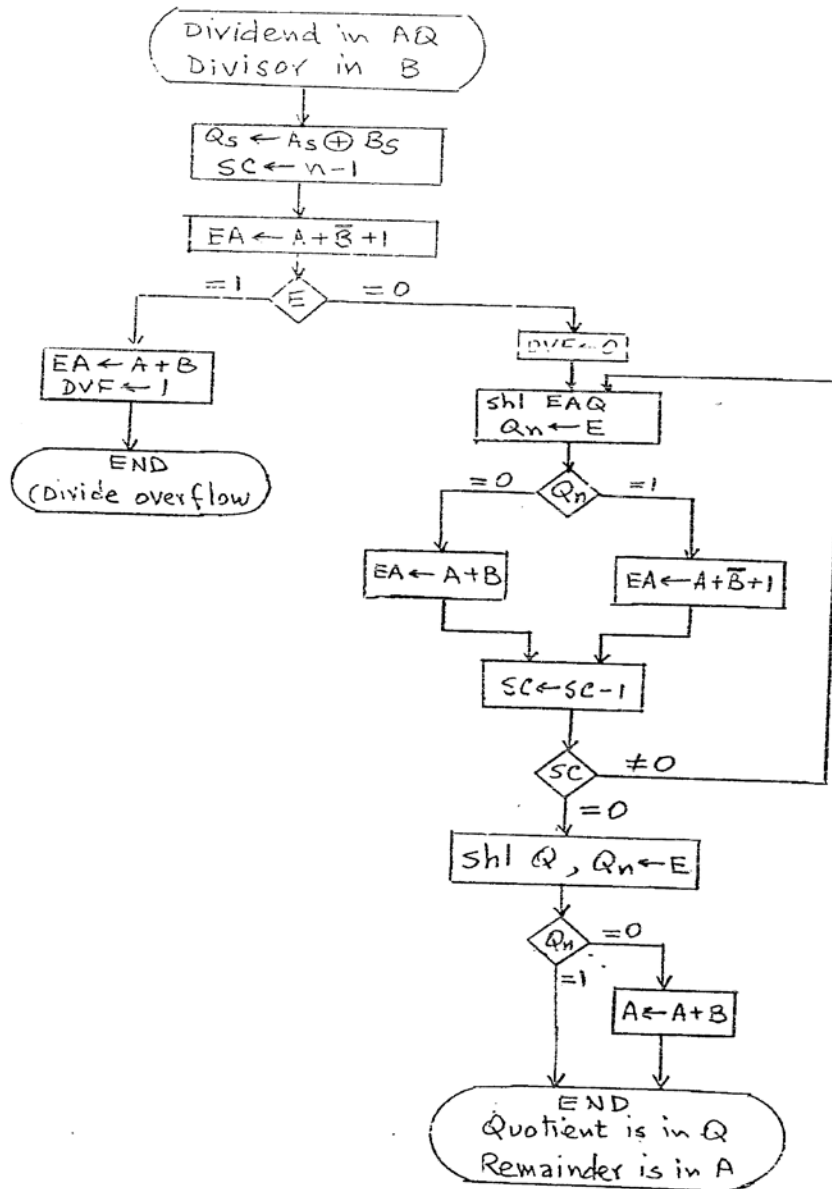
Transfer Augend sign into T_s .
 Then add: $AC \leftarrow AC + BR$
 A_s will have sign of sum.

Truth Table for combin. circuit

T_s	B_s	A_s	V	
0	0	0	0	
0	0	1	1	change of sign quantities subtracted
0	1	0	0	
0	1	1	0	
1	0	0	0	
1	0	1	0	
1	1	0	1	change of sign
1	1	1	0	

10.9 and 10.10 solution: do it yourself

10.15 solution:



Unit 7

Input Output Organization

I/O plays a crucial role in any modern computer system. Therefore, a clear understanding and appreciation of the fundamentals of I/O operations, devices, and interfaces are of great importance.

I/O subsystem

The input-output subsystem of a computer, referred as I/O, provides an efficient mode of communication between the central system and outside environment. Data and programs must be entered into the computer memory for processing and result of computations must be recorded or displayed for the user.

Peripheral devices

Input or output devices attached to the computer are called *peripherals*. Keyboards, display units and printers are most common peripheral devices. Magnetic disks, tapes are also peripherals which provide auxiliary storage for the system.

Input Devices

- Keyboard and mouse
- Touch screen
- Light pen
- Auxiliary storage
- Card reader
- Optical and magnetic character readers
- Data acquisition equipments

Output Devices

- CRT
- Printer (Impact, Ink Jet, Laser, Dot Matrix)
- Digital incremental Plotters
- Auxiliary storage

Not all input comes from people and not all intended for people. In various real time processes as machine tooling, assembly line procedures and chemical & industrial processes, various processes communicate with each other providing input and/or outputs to other processes.

- I/O organization of a computer is a function of size of the computer and the devices connected to it. In other words, amount of hardware computer possesses to communicate with no. of peripheral units, differentiate between small and large system.
- IO devices communicating with people and computer usually transfer alphanumeric information using ASCII binary encoding.

Input-Output Interface

Input-output interface provides a method for transferring information between internal storage and external I/O devices. It resolves the *differences* between the computer and peripheral devices. The major differences are:

- Peripherals are **electromechanical and electromagnetic** devices and manner of operation is different from that of CPU which is **electronic** component.
- **Data transfer rate** of peripherals is slower than that of CPU. So some synchronization mechanism may be needed.
- **Data codes and formats** in peripherals differ from the word format in CPU and memory.
- **Operating modes of peripherals** are different from each other and each must be controlled so as not to disturb other.

To resolve these differences, computer system usually include special hardware unit between CPU and peripherals to supervise and synchronize I/O transfers, which are called **Interface units** since they interface processor bus and peripherals.

I/O Bus and Interface Modules

Peripherals connected to a computer need special communication link to interface with CPU. This special link is called **I/O bus**. Fig below clears the idea:

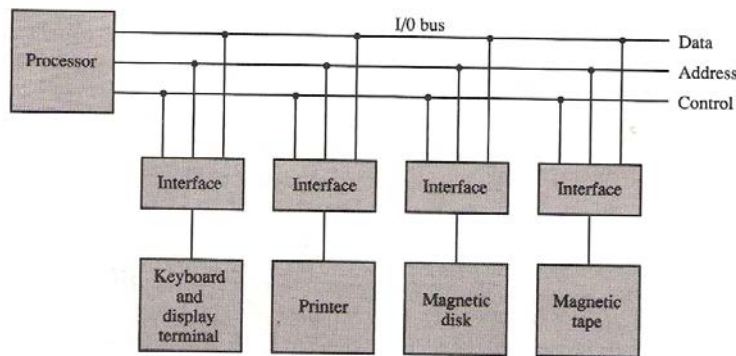


Fig: Connection of I/O bus to I/O devices

- I/O bus from the processor is attached to all peripheral interfaces.
- I/O bus consists of **Data lines, address and control lines**.
- To communicate with a particular device, the processor places a device address on the address lines. Each peripheral has an **interface module** associated with its interface.

Functions of an interface are as below:

- Decodes the device address (device code)
- Decodes the I/O commands (operation or function code) in control lines.
- Provides signals for the peripheral controller
- Synchronizes the data flow
- Supervises the transfer rate between peripheral and CPU or Memory

I/O commands

The function code provided by processor in control line is called *I/O command*. The interpretation of command depends on the peripheral that the processor is addressing. There are **4 types** of commands that an interface may receive:

- Control command:** Issued to activate the peripheral and to inform it what to do? E.g. a magnetic tape unit may be instructed to backspace tape by one record.
- Status command:** Used to check the various status conditions of the interface before a transfer is initiated.
- Data input command:** Causes the interface to read the data from the peripheral and places it into the interface buffer. [HEY! Processor checks if data are available using status command and then issues a data input command. The interface places the data on data lines, where they are accepted by the processor]

- d) Data output command: Causes the interface to read the data from the bus and saves it into the interface buffer.

I/O Bus versus Memory Bus

In addition to communicating with I/O, processor also has to work with memory unit. Like I/O bus, memory bus contains data, address and read/write control lines. 3 physical organizations, the computer buses can be used to communicate with memory and I/O:

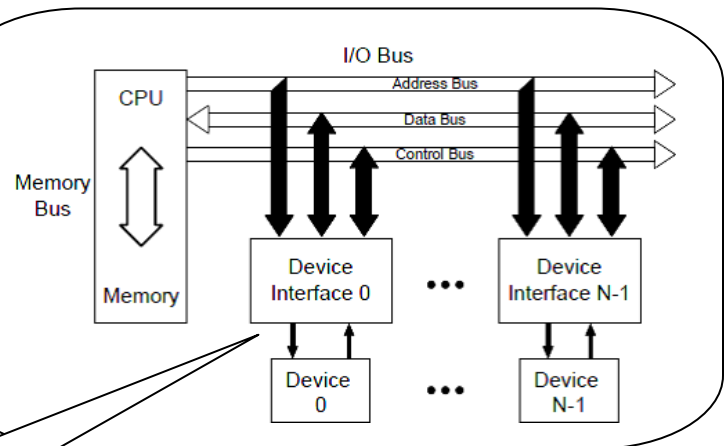
- Use two separate buses, one for memory and other for I/O: Computer has independent sets of data, address and control buses, one for accessing memory and other for I/O. usually employed in a computer that has separate IOP (Input Output Processor).
- Use one common bus for both memory and I/O having separate control lines
- Use one common bus for memory and I/O with common control lines

Isolated I/O versus Memory-Mapped I/O

Question: Differentiate between isolated I/O and memory-mapped I/O.

Isolated I/O Configuration

- Two functionally and physically separate buses
 - Memory bus
 - I/O bus
- Each consists of the same three main groupings of wires
 - Address (example might be 6 wires, or up to $2^6 = 64$ devices)
 - Data
 - Control

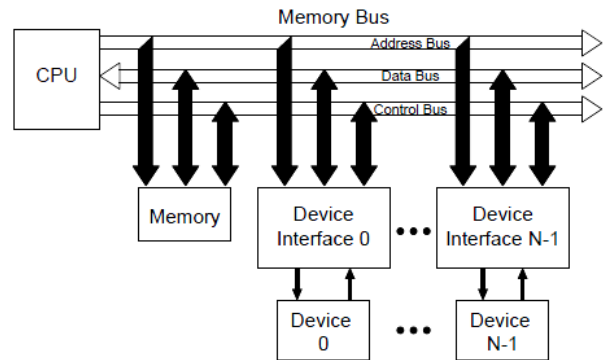


- Each device interface has a port which consists of a minimum of:
 - Control register
 - Status register
 - Data-in (Read) register (or buffer)
 - Data-out (Write) register (or buffer)

- CPU can execute instructions to manipulate the I/O bus separate from the memory bus
- Now only used in very high performance systems

Memory-Mapped I/O configuration

- The memory bus is the *only* bus in the system
- Device interfaces assigned to the *address space* of the CPU or processing element
- Most common way of interfacing devices to computer systems
- CPU can manipulate I/O data residing in interface registers with same instructions that are used to access memory words.
- Typically, a segment of total address space is reserved for interface registers.



In this case, Memory space is not only ordinary system memory. It can refer to all the addresses that the programmer may specify. These addresses correspond to all the possible valid addresses that the CPU may place on its memory bus address lines.

- Diagram shows a hypothetical example of a 10,000 byte memory space
 - Shows the principal regions of the memory Space of a computer system
- Random Access Memory (RAM) includes both:
 - ROM: Read-only memory (0000-0999)
 - RWM: Read-write memory (5000-9999)
- Unused memory space
 - No devices connected to these addresses
 - If CPU tries to access, causes a hardware or *bus* error

Example Usage	Address	CPU Address Space	Generic Usage
Memory (ROM)	0000	[CPU Address Space]	Interrupt Vector Table
	0001		
	0999		
	0999		
Device Interface Registers (Not Memory)	1000	Data-in (Read)	Disk Device Interface
	1001	Data-out (Write)	
	1002	Status	
	1003	Control	
Neither Devices Nor Memory (Unused)	1004	[CPU Address Space]	
	4999		
Memory (RWM)	5000	[CPU Address Space]	Program Code, Data, and Stack
	5001		
	9999		
	9999		

I/O interface Unit (an example)

I/O interface unit is shown in the block diagram below, it consists:

- Two data registers called *ports*.
- A control register
- A status register
- Bus buffers
- Timing and control circuits

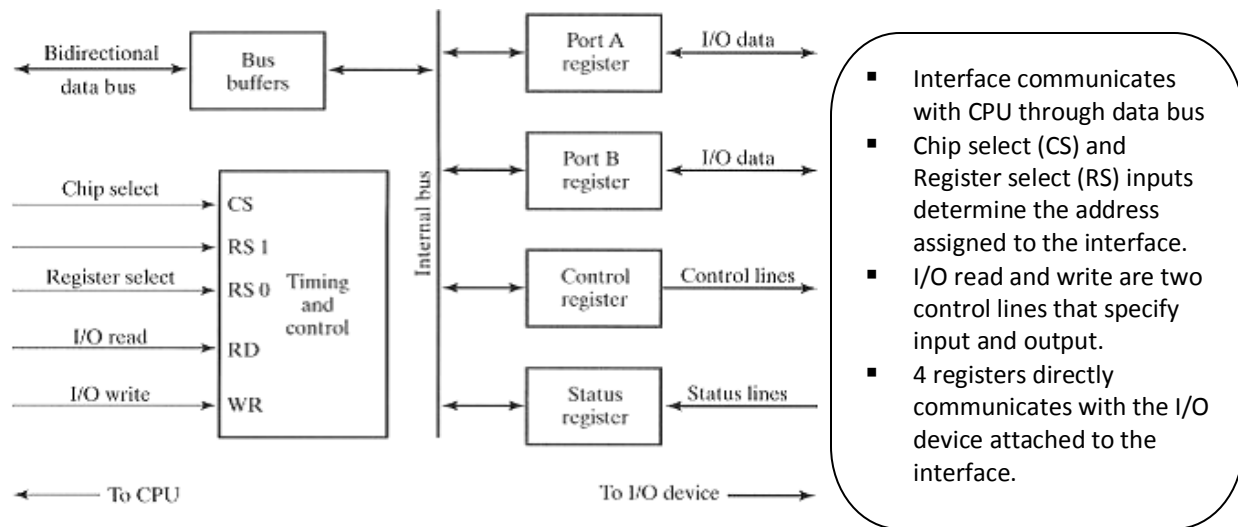


Fig: I/O interface unit

CS	RS1	RS0	Register selected
0	x	x	None: data bus in high-impedance state
1	0	0	Port A register
1	0	1	Port B register
1	1	0	Control register
1	1	1	Status register

- Address bus selects the interface unit through CS and RS1 & RS0.
- Particular interface is selected by the circuit (decoder) enabling CS.
- RS1 and RS0 select one of 4 registers.

Modes of I/O transfer (Types of I/O)

Binary information received from an external device is usually stored in memory for later processing. CPU merely executes I/O instructions and may accept data from memory unit (which in fact is ultimate source or destination). Data transfer between the central computer and I/O devices may be handled in one 3 modes:

- Programmed I/O
- Interrupt-initiated I/O
- Direct memory access (DMA)

Programmed I/O

Programmed I/O operations are the result of I/O instructions written in the computer program. Each data item transfer is initiated by an instruction in the program. Usually, the transfer is to and from a CPU register and peripheral. Other instructions are needed to transfer the data to and from CPU and memory. Transferring data under program control requires constant monitoring of the peripheral by the CPU. Once a data transfer is initiated, the CPU is required to monitor the interface to see when a transfer can again be made. It is up to the **programmed instructions** executed in the CPU to keep close tabs on everything that is taking place in the interface unit and the I/O device. In programmed I/O method, I/O device does not have direct access to memory. Transfer from peripheral to memory/ CPU requires the execution of several I/O instructions by CPU.

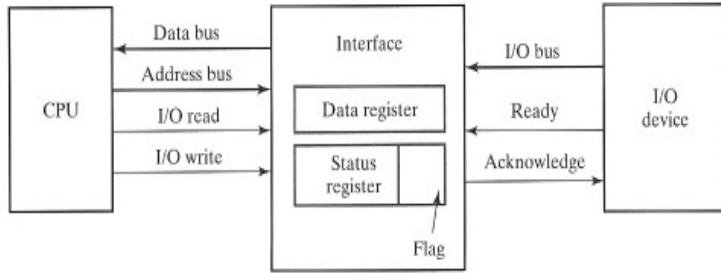
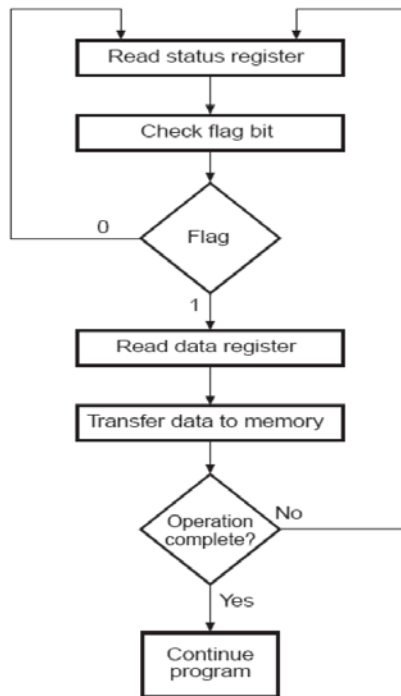


Diagram shows data transfer from I/O device to CPU. Device transfers bytes of data one at a time as they are available. When a byte of data is available, the device places it in the I/O bus and enables its data valid line. The interface accepts the byte into its data register and enables the data accepted line. The interface sets a bit in the status register that we will refer to as an F or "flag" bit.

Now for programmed I/O, a program is written for the computer to check the flag bit to determine if I/O device has put byte of data in data register of interface.



Flowchart of the program that must be written to the CPU is shown here. The transfer of each byte (assuming device is sending sequence of bytes) requires 3 instructions:

- Read status register
- Check F bit. If not set branch to a) and if set branch to c).
- Read data register

Interrupt-initiated I/O

Since polling (constantly monitoring the flag F) takes valuable CPU time, alternative for CPU 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. After the transfer is completed, the computer returns to the previous program to continue what it was doing before the interrupt.

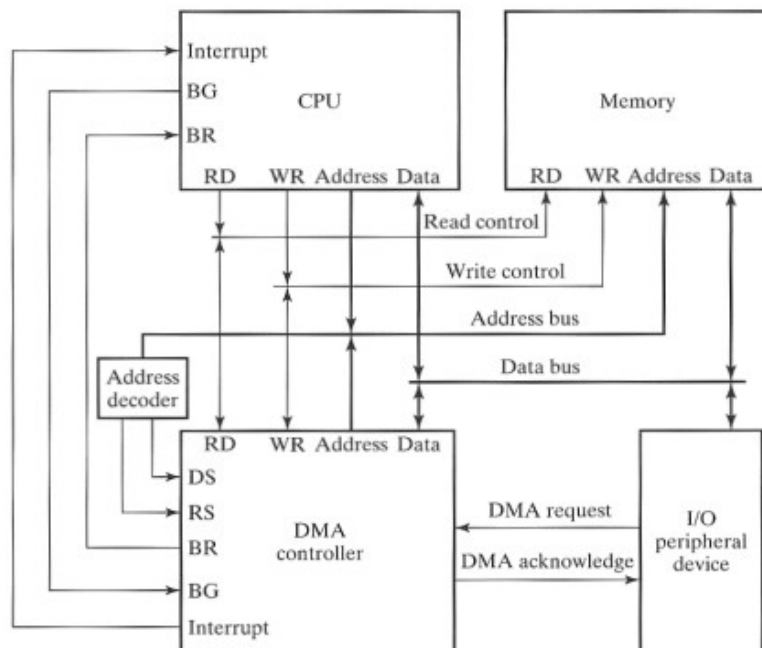
The CPU **responds to the interrupt signal** by storing the return address from the program counter into a memory stack and then control branches to a service routine that processes the required I/O transfer.

Direct Memory Access (DMA)

- **What is DMA?** - DMA is a sophisticated I/O technique in which a DMA controller replaces the CPU and takes care of the access of both, the I/O device and the memory, for fast data transfers. Using DMA you get the fastest data transfer rates possible.
- **Momentum behind DMA:** Interrupt driven and programmed I/O require active CPU intervention (All data must pass through CPU). Transfer rate is limited by processor's ability to service the device and hence CPU is tied up managing I/O transfer. Removing CPU from the path and letting the peripheral device manage the memory buses directly would improve the speed of transfer.
- Extensively used method to **capture buses** is through special control signals:
 - **Bus request (BR):** used by DMA controller to request the CPU for buses. When this input is active, CPU terminates the execution the current instruction and places the address bus, data bus and read & write lines into a high impedance state (open circuit).
 - **Bus grant (BG):** CPU activates BG output to inform DMA that buses are available (in high impedance state). DMA now take control over buses to conduct memory transfers without processor intervention. When DMA terminates the transfer, it disables the BR line and CPU disables BG and returns to normal operation.
- When DMA takes control of bus system, the **transfer with the memory** can be made in following two ways:
 - **Burst transfer:** A block sequence consisting of a number of memory words is transferred in continuous burst. Needed for fast devices as magnetic disks where data transmission can not be stopped (or slowed down) until whole block is transferred.
 - **Cycle stealing:** This allows 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 DMA to “steal” one memory cycle.

DMA Transfer

Question: what is DMA transfer? Explain.



- CPU communicates with the DMA through address and data buses.
- DMA has its own address which activates RS (Register select) and DS (DMA select) lines.
- When a peripheral device sends a DMA request, the DMA controller activates the BR line, informing CPU to leave buses. The CPU responds with its BG line.
- DMA then puts current value of its address register into the address bus, initiates RD or WR signal, and sends a DMA acknowledge to the peripheral devices.
- When BG=0, RD & WR allow CPU to communicate with internal DMA registers and when BG=1, DMA communicates with RAM through RD & WR lines.

Fig: DMA transfer in a computer system

Input-Output Processor (IOP)

- IOP is 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 CPU and one or more IOPs.
- IOP is similar to CPU except that it is designed to handle the details of **I/O processing**.
- Unlike DMA controller (which is set up completely by the CPU), IOP can fetch and execute its own instructions. IOP instructions are designed specifically to facilitate I/O transfers.
- Instructions that are read from memory by an IOP are called **commands** to differ them from instructions read by CPU. The *command words* constitute the program for the IOP. The CPU informs the IOP where to find commands in memory when it is time to execute the I/O program.

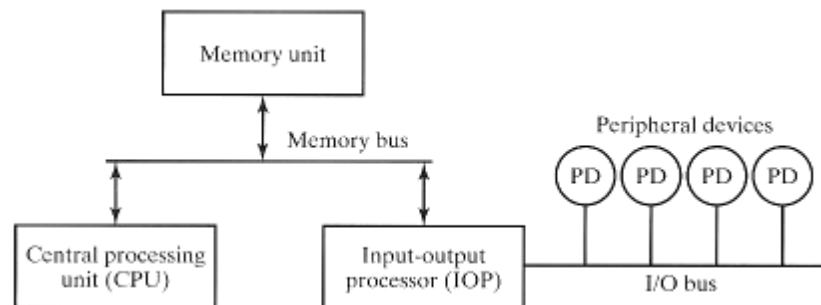


Fig: Block diagram of computer with I/O processor

The memory occupies a central position and can communicate with each processor by means of DMA. CPU is usually assigned the task of initiating the I/O program, from then on; IOP operates independent of the CPU and continues to transfer data from external devices and memory.

CPU-IOP communication

Communication between the CPU and IOP may take different forms depending on the particular computer used. Mostly, memory unit acts as a memory center where each processor leaves information for the other.

Mechanism: 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 IOP and I/O device ("IOP overload condition", "device busy with another transfer" etc). CPU then checks status word to decide what to do next. If all is in order, CPU sends the instruction to start the I/O transfer. The memory address received with this instruction tells the IOP where to find its program. CPU may continue with another program while the IOP is busy with the I/O program. When IOP terminates the transfer (using DMA), it sends an interrupt request to CPU. The CPU responds by issuing an instruction to read the status from the IOP and IOP then answers by placing the status report into specified memory location. By inspecting the bits in the status word, CPU determines whether the I/O operation was completed satisfactorily and the process is repeated again.

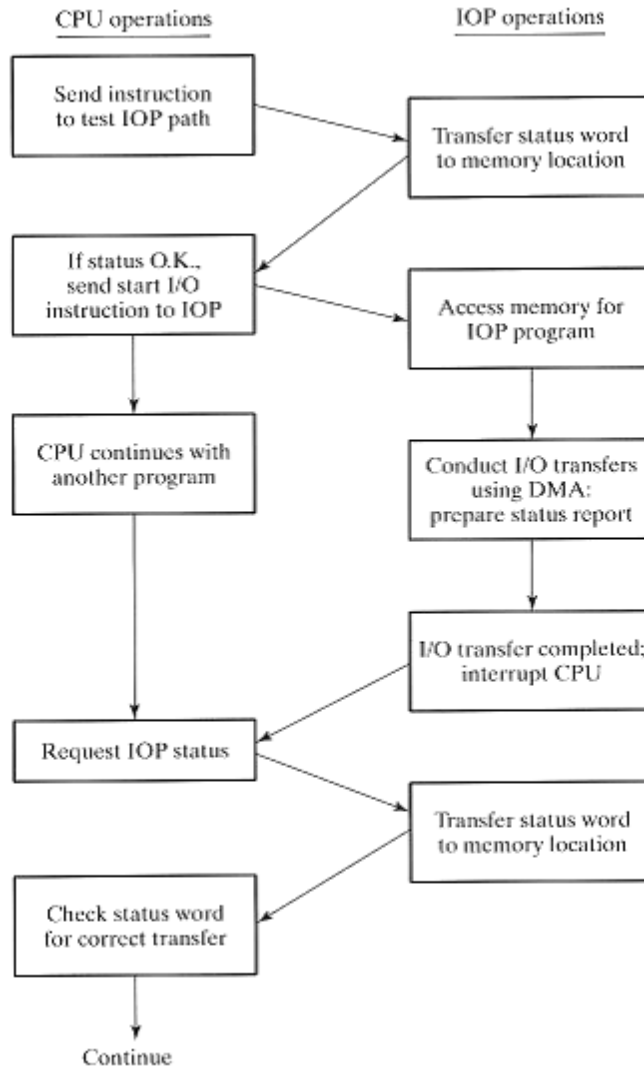


Fig: CPU-IOP communication

Data Communication Processor (DCP)

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 (which may consists of wide variety of devices as printers, displays, sensors etc.). So DCP makes possible to operate efficiently in a time-sharing environment.

Difference between IOP and DCP: Is the way processor communicates with I/O devices.

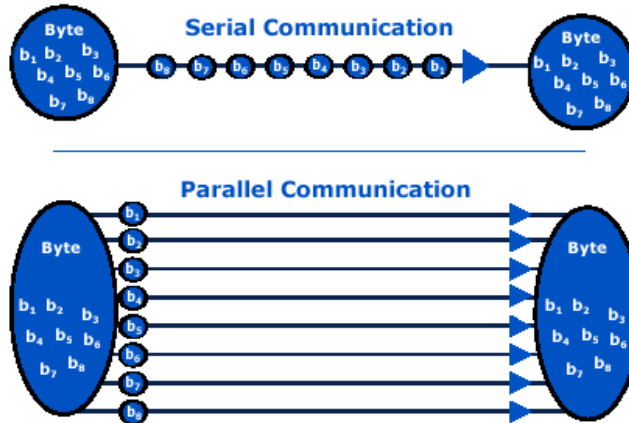
- An I/O processor communicates with the peripherals through a common I/O bus i.e. all peripherals share common bus and use to transfer information to and from I/O processor.
- DCP communicates with each terminal through a single pair of wires. Both data and control information are transferred in serial fashion.

DCP must also communicate with the CPU and memory in the same manner as any I/O processor.

Serial and parallel communication

Serial: Serial communication is the process of sending data one bit at a time, sequentially, over a communication channel or computer bus. This is in contrast to parallel communication.

Parallel: Parallel communication is a method of sending several data signals simultaneously over several parallel channels. It contrasts with serial communication; this distinction is one way of characterizing a communications link.

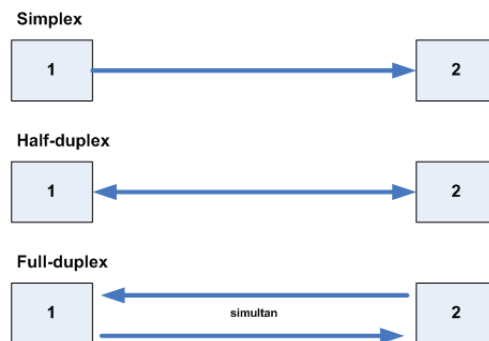
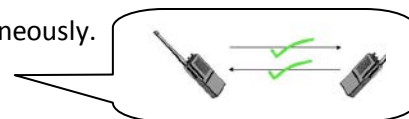
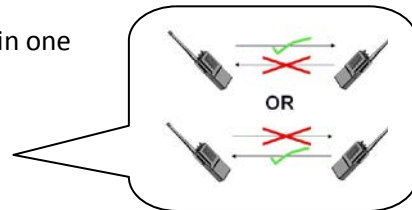


Modes of data transfer

Question: What are 3 possible modes of transfer data to and from peripherals? Explain.

Data can be transmitted in between two points in 3 different modes:

- Simplex:
 - Carries information in one direction only.
 - Seldom used
 - Example: PC to printer, radio and TV broadcasting
- Half-duplex:
 - Capable of transmitting in both directions but only in one direction at a time.
 - Turnaround time: time to switch a half-duplex line from one direction to other.
 - Ex: walkie-talkie" style two-way radio
- Full duplex:
 - Can send and receive data in both directions simultaneously.
 - Example: Telephone, Mobile Phone, etc



Protocol

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.

Purpose of data link protocol:

- To establish and terminate a connection between two stations
- To identify the sender and receiver
- To identify errors
- To handle all control functions

Two major categories according to the message-framing technique used:

- Character-oriented protocol
- Bit-oriented protocol

Character-oriented protocol

It is based on the binary code of the character set (e.g. ASCII). ASCII communication control characters are used for the purpose of routing data, arranging the text in desired format and for the layout of the printed page.

Code	Symbol	Meaning	Function
0010110	SYN	Synchronous idle	Establishes synchronism
0000001	SOH	Start of heading	Heading of block message
0000010	STX	Start of text	Precedes block of text
0000011	ETX	End of text	Terminates block of text
0000100	EOT	End of transmission	Concludes transmission
0000110	ACK	Acknowledge	Affirmative acknowledgement
0010101	NAK	Negative acknowledge	Negative acknowledgement
0000101	ENQ	Inquiry	Inquire if terminal is on
0010111	ETB	End of transmission block	End of block of data
0010000	DLE	Data link escape	Special control character

Table: ASCII communication control characters

Here is the typical example to appreciate the function of the DCP:

SYN	SYN	SOH	Header	STX	Text	ETX	BCC
-----	-----	-----	--------	-----	------	-----	-----

Fig: message format

Typical message format that might be sent from a terminal to the processor is shown above. It contains following portions:

Code	Symbol	Comments
0001 0110	SYN	First sync character
0001 0110	SYN	Second sync character
0000 0001	SOH	Start of heading
0101 0100	T	Address of terminal is T4
0011 0100	4	
0000 0010	STX	Start of text transmission
0101 0010		
0100 0101	request	Text sent is a request to respond with the balance of
.	balance	account number 1234
.	of account	
.	No. 1234	
1011 0011		
0011 0100		
1000 0011	ETX	End of text transmission
0111 0000	LRC	Longitudinal parity character

Bit-oriented protocol

It allows the transmission of serial bit stream of any length without the implication of character boundaries. Messages are organized in a *frame*. In addition to the information field, a frame contains address, control and error-checking fields.

Flag 01111110	Address 8 bits	Control 8 bits	Information any number of bits	Frame check 16 bits	Flag 01111110
------------------	-------------------	-------------------	-----------------------------------	------------------------	------------------

Fig: Frame format for bit-oriented protocol

A frame starts with a 8-bit flag 01111110 followed by an address and control sequence. The information field can be of any length. The frame check field CRC (cyclic redundancy check) detects errors in transmission. The ending flag represents the receiving station.

Unit 8

Memory Organization

Introduction

Memory unit is an essential component in any general purpose computer since it is needed to store programs and data. The memory unit that communicates directly with the CPU is called the **main memory** and devices that provide backup storage are called **auxiliary memory**.

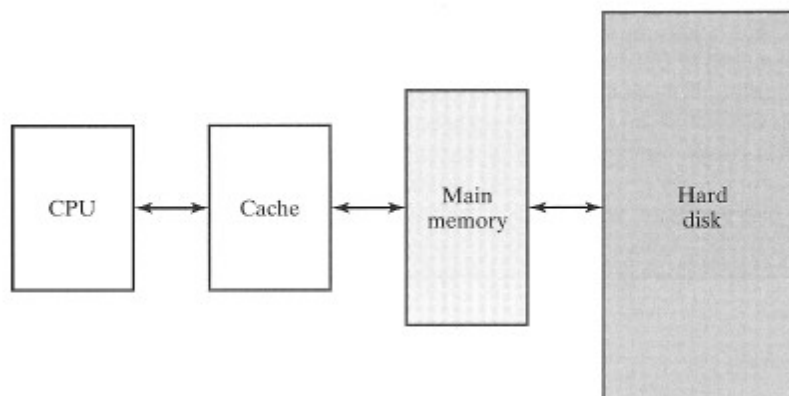
NOTE: Auxiliary memory devices such as magnetic disk and tapes are used to store system programs, large data files and other backup information. Only programs and data currently needed by the processor reside in main memory. All other information is stored in main memory and transferred to main memory when needed.

Memory Types

- **Sequential Access Memory (SAM):** In computing, **SAM** is a class of data storage devices that read their data in sequence. This is in contrast to random access memory (RAM) where data can be accessed in any order. Sequential access devices are usually a form of magnetic memory. Magnetic sequential access memory is typically used for secondary storage in general-purpose computers due to their higher density at lower cost compared to RAM, as well as resistance to wear and non-volatility. Examples of SAM devices still in use include hard disks, CD-ROMs and magnetic tapes. Historically, drum memory has also been used.
- **Random Access Memory (RAM):** **RAM** is a form of computer data storage. Today, it takes the form of integrated circuits that allow stored data to be accessed in any order with a worst case performance of constant time. Strictly speaking, modern types of DRAM are therefore not random access, as data is read in bursts, although the name DRAM / RAM has stuck. However, many types of SRAM, ROM and NOR flash are still random access even in a strict sense. RAM is often associated with volatile types of memory, where its stored information is lost if the power is removed. The first RAM modules to come into the market were created in 1951 and were sold until the late 1960s and early 1970s.

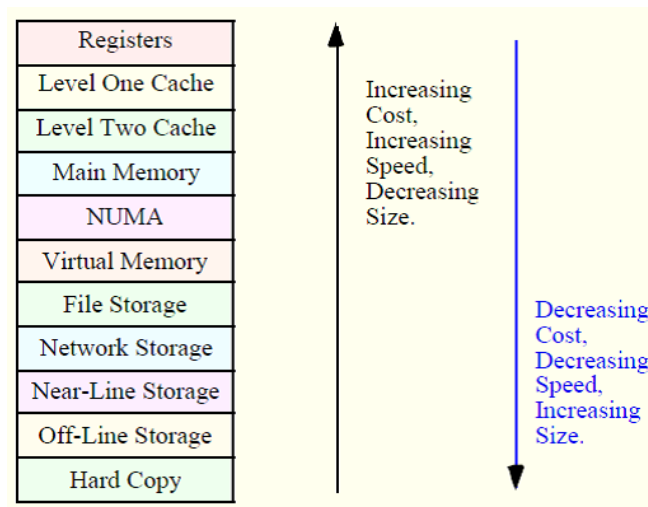
Memory hierarchy

Block diagram below shows the generic memory hierarchy.



Talking roughly, **lowest level** of hierarchy is small, fast memory called *cache* where anticipated CPU instructions and data resides. At the **next level** upward in the hierarchy is *main memory*. The main memory serves CPU instruction fetches not satisfied by cache. At the **top level** of the hierarchy is the *hard disk* which is accessed rarely only when CPU instruction fetch is not found even in main memory.

Example: Memory hierarchy in Intel 80x86 processor family:



- At the top level of the memory hierarchy are the CPU's general purpose registers.
- Level One on-chip Cache system is the next highest performance subsystem.
- Next is expandable level two cache
- Next, main memory (usually DRAM) subsystem. Relatively low-cost memory found in most computer systems.
- Next is NUMA (Non Uniform Memory Access)
- Next is Virtual Memory scheme that simulate main memory using storage on a disk drive.
- Next in hierarchy is File Storage which uses disk media to store program data.
- Below is Network Storage stores data in distributed fashion.
- Near-Line Storage uses the same media as Off-Line Storage; the difference is that the system holds the media in a special robotic jukebox device that can automatically mount the desired media when some program requests it.
- Next is Off-Line Storage that includes magnetic tapes, disk cartridges, optical disks, floppy diskettes and USBs.
- Hardcopy..! I don't think I have to explain it.

Primary and Secondary Memory

Primary (Main) Memory

It is a relatively large and fast memory used to store programs and data during the computer operation. Semiconductor integrated circuit is the principle technology used for main memory.

Random Access Memory (RAM): RAM chips are available in two possible modes, *static* and *dynamic*.

Static RAM: consists of internal flip-flops to store binary information. It is easier to use and has shorter read/write cycles.

Dynamic RAM: stores binary information in the form of electric charges in capacitors. The stored charge tends to discharge with time, so DRAM words are refreshed every few milliseconds to restore the decaying charge. DRAM offers reduced power consumption and larger storage capacity in a single memory chip.

Read-Only Memory (ROM): Random access ROM chips are used for storing programs that are permanently resident in computer and for tables of constants that do not change once computer is manufactured. The contents of ROM remain unchanged after power is turned off and on again.

Bootstrap loader: It is initial program whose function is to start the computer operating system when power is turned on and is stored in ROM portion of the main memory.

Computer startup: The startup of a computer consists of turning the power on and starting the execution of an initial program. Thus when power is turned on, the hardware of the computer sets the PC to the first address of the bootstrap loader. The bootstrap program loads the portion of the OS from the disk to main memory and control is then transferred to the OS, which prepares the computer for general use.

RAM and ROM Chips

RAM and ROM chips are available in a variety of sizes. If we larger memory for the system, it is necessary to combine a number of chips to form the required memory size.

RAM Chips

A RAM chip is better suited to communicate with CPU if it has one or more control inputs that select the chip only when needed. The block diagram of a RAM chip is shown below:

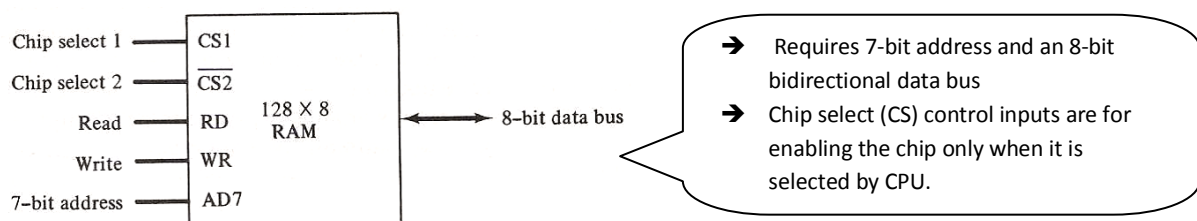


Fig: Typical RAM chip (128 words of eight bits each)

CS1	$\overline{CS2}$	RD	WR	Memory function	State of data bus
0	0	x	x	Inhibit	High-impedance
0	1	x	x	Inhibit	High-impedance
1	0	0	0	Inhibit	High-impedance
1	0	0	1	Write	Input data to RAM
1	0	1	x	Read	Output data from RAM
1	1	x	x	Inhibit	High-impedance

A callout box explains that the unit is in operation only when CS1=1 and (CS2)'=0, and that high impedance state indicates an open circuit (i.e., output does not carry a signal and has no logic significance).

Fig: Function table for RAM chip

ROM Chips

Since a ROM chip can only read, data bus is unidirectional (output mode only).

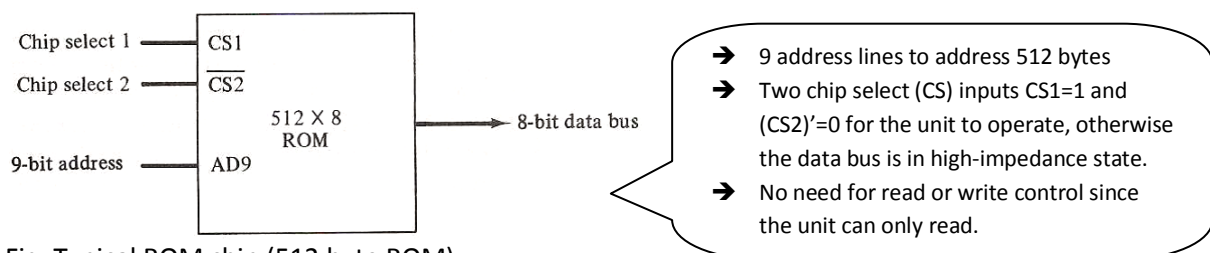


Fig: Typical ROM chip (512 byte ROM)

Memory Address Map

The addressing of memory can be established by means of a table that specifies the memory address assigned to each RAM or ROM chip. This table is called memory address map and is a pictorial representation of assigned address space for particular chip.

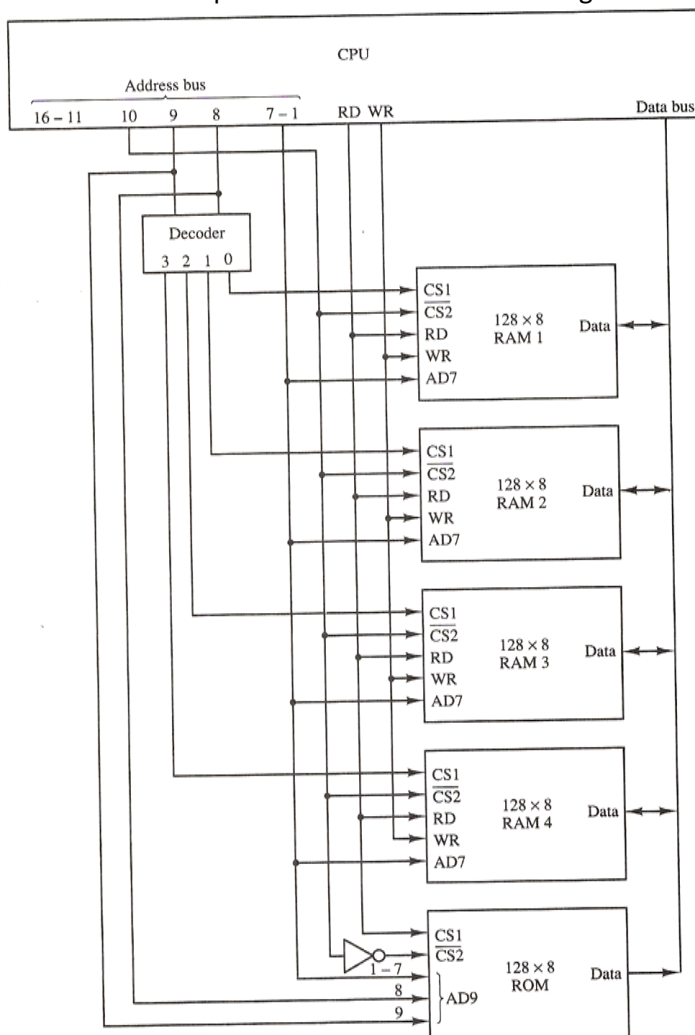
Example: Suppose computer system needs 512 bytes of RAM and 512 bytes of ROM.

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

- ➔ Component column specifies RAM or ROM chip. We use four 128 words RAM to make 512 byte size.
- ➔ Hexadecimal address column assigns a range of addresses for each chip.
- ➔ 10 lines in address bus column: lines 1 through 7 for RAM and 1 through 9 for ROM. Distinction between RAM and ROM chip is made by line 10. When line 10 is 1, it selects ROM and when it is 0, CPU selects RAM.
- ➔ X's represents a binary number ranging from all-0's to all-1's.

Memory-CPU Connection

RAM and ROM chips are connected to CPU through data and address buses.



Example gives an indication of the interconnection complexity that can exist between memory chips and CPU. More the chips, more external decoders are required for selection among the chips.

- ➔ This configuration gives 512 bytes of RAM and 512 bytes of ROM
- ➔ Each RAM receives 7 low-order bits of the address bus to select a byte.
- ➔ RAM chips are selected with decoder with selection input of line 8 and 9.
- ➔ The selection between RAM and ROM is done by line 10. When 0, RAMs are selected and when 1 ROM get selected.

Auxiliary (Secondary) Memory

The most common auxiliary memory devices used in computer systems are **magnetic disks**, **magnetic tapes** and **optical disks**. To understand fully the physical mechanism of auxiliary memory devices, we should have knowledge of magnetics, electronics and electromechanical systems.

HEY! Read yourself about these three devices... I hope u guys have studied in your OS course.

Virtual Memory

- A virtual memory system attempts to optimize the use of the main memory (the higher speed portion) with the hard disk (the lower speed portion). In effect, **virtual memory** is a technique for using the secondary storage to extend the apparent limited size of the physical memory beyond its actual physical size. It is usually the case that the available physical memory space will not be enough to host all the parts of a given active program.
- Virtual memory gives programmers the illusion that they have a very large memory and provides mechanism for dynamically translating program-generated addresses into correct main memory locations. 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 the programmer is a virtual address (virtual memory addresses) and the set of such addresses is 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 the programs as they reference instructions and data; the memory space consists of actual main memory locations directly addressable for processing. Generally, the address space is larger than the memory space.

Example: consider main memory: 32K words ($K = 1024$) = 2^{15} and auxiliary memory 1024K words = 2^{20} . Thus we need 15 bits to address physical memory and 20 bits for virtual memory (virtual memory can be as large as we have auxiliary storage).

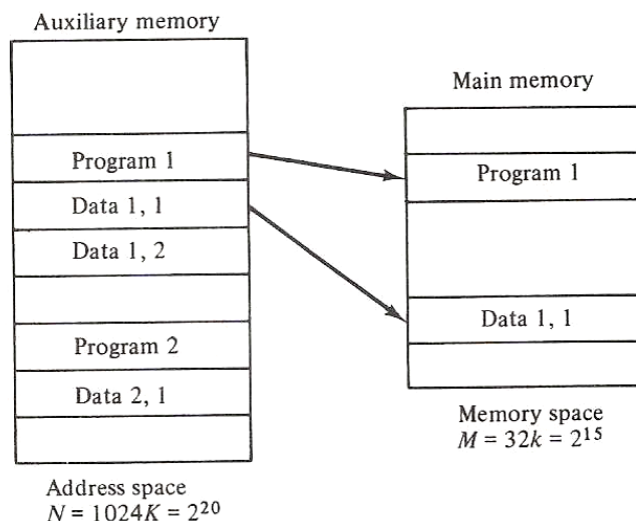


Fig: Relation between address and memory space in a virtual memory system

- ➔ Here auxiliary memory has the capacity of storing information equivalent to 32 main memories.
- ➔ Address space $N = 1024K$
- ➔ Memory space $M = 32K$
- ➔ In multiprogram computer system, programs and data are transferred to and from auxiliary memory and main memory based on the demands imposed by the CPU.

In virtual memory system, address field of an instruction code has a sufficient number of bits to specify all virtual addresses. In our example above we have 20-bit address of an instruction (to refer 20-bit virtual address) but physical memory addresses are specified with 15-bits. So a table is needed

to map a virtual address of 20-bits to a physical address of 15-bits. Mapping is a dynamic operation, which means that every address is translated immediately as a word is referenced by CPU.

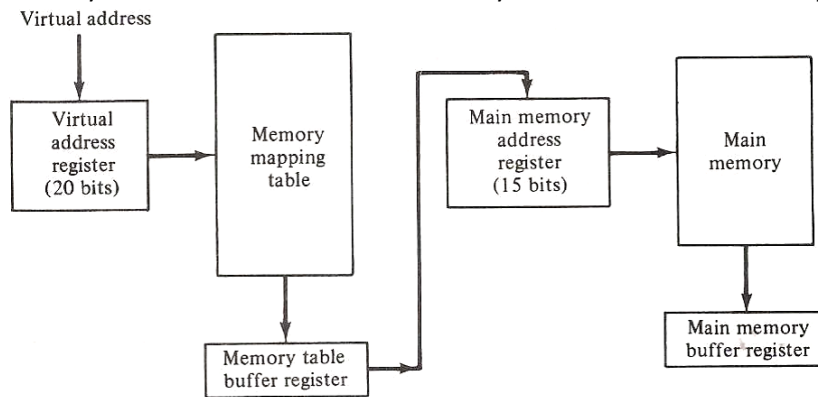


Fig: Memory table for mapping a virtual address

Address Mapping using Pages

Above memory table implementation of address mapping is simplified if the information in address space and memory space are each divided into groups of fixed size.

Blocks (or page frame): The physical memory is broken down into groups of equal size called blocks, which may range from 64 to 4096 words each.

Pages: refers to a portion of subdivided virtual memory having same size as blocks i.e. groups of address space.

Example: consider computer with address space = 8K and memory space = 4K.

Page 0
Page 1
Page 2
Page 3
Page 4
Page 5
Page 6
Page 7

Address space
 $N = 8K = 2^{13}$

Block 0
Block 1
Block 2
Block 3

Memory space
 $M = 4K = 2^{12}$

- ➔ If we split both spaces into groups of 1K words, we obtain 8 pages and 4 blocks.
- ➔ Virtual address has 13 bits. Since each page consists of $2^{10} = 1024$ words, high-order 3 bits will specify one of 8 pages and low-order 10 bits give the line address within the pages.

The mapping from address space to memory space becomes easy if virtual address is 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 remaining high-order bits of the virtual address specify the **page number**.

NOTE: line address in address space and memory space is same; only mapping required is from page number to a block number.

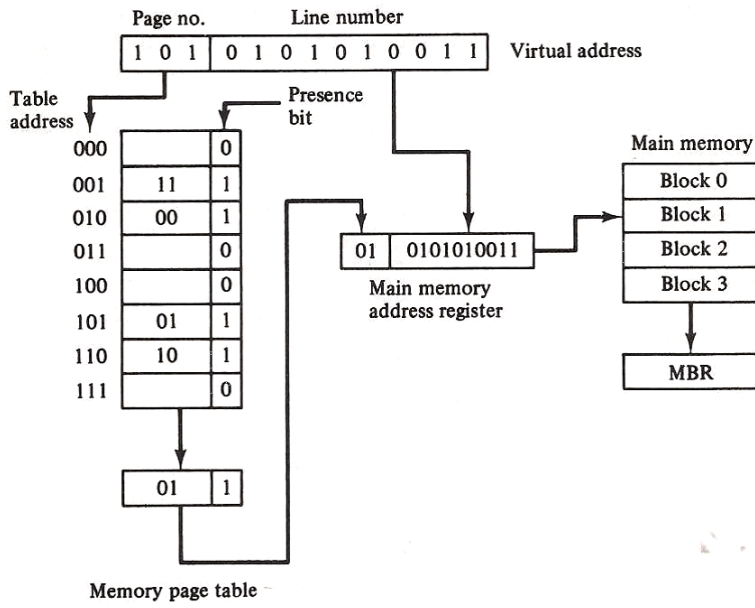
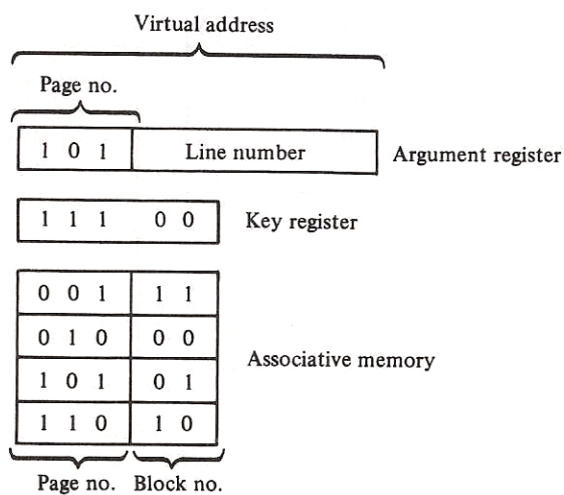


Fig: Memory table in paged system

- ➔ The memory-page table consists of 8 words, one for each page.
- ➔ The address in the page table denotes page number and the content of the word gives the block number where the page is stored in main memory.
- ➔ Presence bit when 0 indicates page is not available in main memory and when 1 says that page has been transferred to main memory.
- ➔ Table shows that pages 1,2,5 and 6 are now available in main memory in blocks 3,0,1 and 2 respectively.

Associative Memory Page table

- ➔ In above figure, we use random-access page table which is inefficient with respect to storage utilization. For example: consider address space = 1024K words and memory space = 32K words. If each page or block contains 1K words, the number of pages is 1024 and number of blocks 32. The capacity of the memory page table must be 1024 words and only 32 locations have presence bit equal to 1. At any given time, at least 992 locations will be empty and not in use.
- ➔ What about making page table with number of words equal to the number of blocks in main memory? Obviously this is an efficient approach since size of memory is reduced and each location is fully utilized.
- ➔ This method can be implemented by means of an **associative memory** in which each word in memory containing a page number with its corresponding block number.



- ➔ The page field in each **associative memory table** word is compared with page number bits in an argument register (which contains page number in the virtual address), if match occurs, the word is read from memory and its corresponding block number is extracted.
- ➔ This is the associative page table for same example in previous section where we were using random-access page table.

Page Replacement

A virtual memory system is a combination of hardware and software techniques. A memory management software system handles:

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?

Mechanism: 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 in auxiliary memory. This condition is called **page fault**. When page fault occurs, the execution of the present program is suspended until required page is brought into memory. Since loading a page from auxiliary memory to main memory is basically an I/O operation, OS assigns this task to I/O processor. In the mean time, control is transferred to the next program in memory that is waiting to be processed in the CPU. Later, when memory block has been assigned, the original program can resume its operation.

When a **page fault occurs** in a virtual memory system, it signifies that the page referenced by the program is not in main memory. A new page is then transferred from auxiliary memory to main memory. If main memory is full, it would be necessary to remove a page from a memory block to make a room for a new page. The **policy for choosing pages to remove is determined from the replacement algorithm** that is used.

GOAL: try to remove the page least likely to be referenced by in the immediate future.

There are numerous page replacement algorithms, two of which are:

- First-in First-out (FIFO): replaces a page that has been in memory longest time.
- Least Recently Used (LRU): assumes that least recently used page is the better candidate for removal than the least recently loaded page.

Memory Management Hardware

A memory management system is a collection of hardware and software procedures for managing various programs (effect of multiprogramming support) residing in memory. Basic components of memory management unit (MMU) are:

- A facility for dynamic storage relocation that maps logical memory references into physical memory addresses.
- A provision for sharing common programs by multiple users
- Protection of information against unauthorized access.

The dynamic storage relocation hardware is a mapping process similar to paging system.

Segment: It is more convenient to divide programs and data into logical parts called segments despite of fixed-size pages. A **segment** is a set of logically related instructions or data elements. Segments may be generated by the programmer or by OS. Examples are: a subroutine, an array of data, a table of symbols or user's program.

Logical address: The address generated by the segmented program is called a *logical address*. This is similar to virtual address except that logical address space is associated with variable-length segments rather than fixed-length pages.

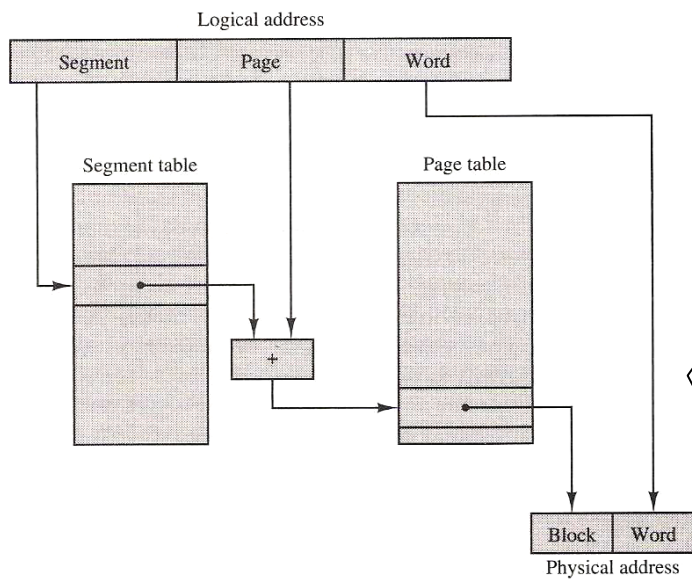
Segmented-Page Mapping

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-sized pages.

Consider diagram below:

Logical address = Segment + page + Word

Where **segment** specifies segment number, **page** field specifies page within the segment and **word** field specifies specific word within the page.



- Mapping of logical address to physical address is done by using two tables: segment and page table.
- The entry in the segment table is a pointer address for the page table base, which is then added to page number (given in logical address). The sum points to some entry in page table and content of that page is the address of physical block. The concatenation of block field with the word field produces final **physical mapped address**.

Fig: Logical to physical address mapping

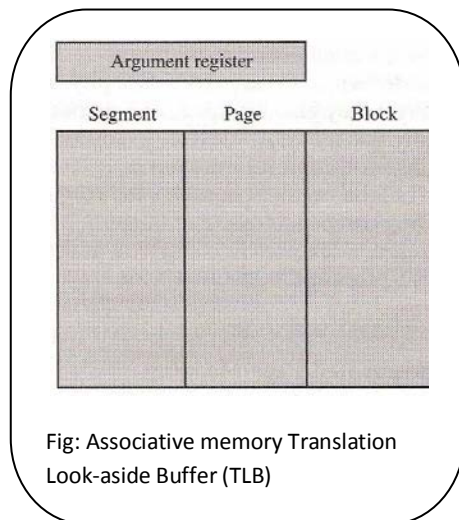


Fig: Associative memory Translation Look-aside Buffer (TLB)

- This is a fast associative memory (TLB) and holds most recently referenced entries. (Alternatively we could store above two tables: segment table and page table, in two separate small memories which really increases the CPU access time)

HEY..! See Numerical example to clear the concept of MMU (page no. 497, Morris Mano 3rd edition Computer System Architecture)

Memory Protection

- Memory protection is concerned with protecting one program from unwanted interaction with another and preventing the occasional user performing OS functions.
- Memory protection can be assigned to the physical address or the logical address.
 - Through physical address: assign each block in memory a number of protection bits.
 - Through logical address: better idea is to apply protection bits in logical address and can be done by including protection information within the segment table or segment register.

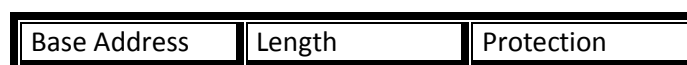


Fig: format of typical segment descriptor

Where

- **Base address field** gives the base of the page table address in segmented-page organization.

- **Length field** gives the segment size (in number of pages)
- The **protection field** specifies access rights available to a particular segment. The protection information is set into the descriptor by the master control program of the OS. Some of the access rights are:
 - Full read and write privileges
 - Read only (Write protection)
 - Execute only (Program protection)
 - System only (OS protection)