SYLLABUS

Unit I:

Introduction: Objective, scope and outcome of the course.

Unit II:

Computer Data Representation: Basic computer data types, Complements, Fixed point representation, Floating point representation.

Register Transfer and Micro-operations: Register Transfer language, Register Transfer, Bus and Memory Transfers (Three-State Bus Buffers, Memory Transfer), Arithmetic Micro-Operations, Logic Micro-Operations, Shift Micro-Operations, Arithmetic logical shift unit.

Basic Computer Organization and Design: Instruction codes, Computer registers, computer instructions, Timing and Control, Instruction cycle, Memory-Reference Instructions, Input-output and interrupt, Complete computer description, Design of Basic computer, design of Accumulator Unit.

Unit III:

Programming The Basic Computer: Introduction, Machine Language, Assembly Language, assembler, Program loops, Programming Arithmetic and logic operations, subroutines, I-O Programming.

Micro programmed Control: Control Memory, Address sequencing, Micro program Example, design of control Unit.

Unit IV:

Central Processing Unit: Introduction, General Register Organization, Stack Organization, Instruction format, Addressing Modes, data transfer and manipulation, Program Control, Reduced Instruction Set Computer (RISC)

Pipeline And Vector Processing: Flynn's taxonomy, Parallel Processing, Pipelining, Arithmetic Pipeline, Instruction, Pipeline, RISC Pipeline, Vector Processing, Array Processors.

Unit V:

Computer Arithmetic: Introduction, Addition and subtraction, Multiplication Algorithms (Booth Multiplication Algorithm), Division Algorithms, Floating Point Arithmetic operations, Decimal Arithmetic Unit.

Input-Output Organization, Input-Output Interface, Asynchronous Data Transfer, Modes Of Transfer, Priority Interrupt, DMA, Input-Output Processor (IOP), CPU-IOP Communication, Serial communication.

Unit VI: Memory Organization: Memory Hierarchy, Main Memory, Auxiliary Memory, Associative Memory, Cache Memory, Virtual Memory.

Multiprocessors: Characteristics of Multiprocessors, Interconnection Structures, Inter-processor Arbitration, Inter- processor Communication and Synchronization, Cache Coherence, Shared Memory Multiprocessors.

Text Books:

1. "Computer Organization and Architecture", William Stallings (Pearson Education India)
2. "Computer Organization and Architecture", John P. Hayes (McGraw Hill)
3. "Computer Organization", V. Carl. Hamacher (McGraw Hill)

UNIT 1

1.1 Course Description: Computer Architecture

This course introduces the principles of computer organization and the basic architecture concepts. The course emphasizes performance, instruction set design, pipelining, memory technology, memory hierarchy, virtual memory management, and I/O systems and design of various microprocessors.

1.2 Course Objectives

On successful completion of this course students should be able:

- To understand the structure, function and characteristics of computer systems.
- To understand the design of the various functional units and components of computers.
- To explain the function of each element of a memory hierarchy.
- To identify and compare different methods for computer I/O.
- To identify the elements of modern instructions sets and their impact on processor design.

1.3 Course Scope

- Computer architecture and organization knowledge helps to have peripheral knowledge as there are various aspects to processor design.
- Very broadly, there is an architecture aspect, a circuit aspect, and a process aspect to designing a CPU.
- The process engineers at the fabrication centers for manufacturing the CPU in real silicon need know-how in semiconductor physics, fabrication technology, and possibly material science. It all depends on how deep you want to go down the rabbit hole.
- A lot of ideas from OS, networks, compiler design, and distributed systems can be applied here, with a singular purpose of delivering the maximum performance at the lowest cost.

- However, we get to know everything about how a CPU works. And that is a good direction to go, because having knowledge of supporting technology will also help you to become a better computer architect or supporting roles.

1.4 Course Outcomes

On successful completion of this course students will be able to:

- Understand the impact of instruction set architecture on cost-performance of computer design.
- Identify microprocessor designs and various design techniques employed.
- Examine the design process of a computer and critical elements in each step.
- Understand memory hierarchy and its impact on computer cost/performance.

Unit-2

2.1 Computer Data Representation

2.1.1 Computer Data Representation

Data refers to the symbols that represent people, events, things, and ideas. Data can be a name, a number, the colors in a photograph, or the notes in a musical composition.

Data Representation refers to the form in which data is stored, processed, and transmitted.

Devices such as smart-phones, iPods, and computers store data in digital formats that can be handled by electronic circuitry

Representing Numbers

DECIMAL (BASE 10)	BINARY (BASE 2)
0	0
1	1
2	10
3	11
4	100
5	101
6	110
7	111
8	1000
9	1001
10	1010
11	1011
1000	1111101000

Representing Text

00100000	Space	00110011	3	01000110	F	01011001	Y	01101100	l
00100001	!	00110100	4	01000111	G	01011010	Z	01101101	m
00100010	"	00110101	5	01001000	H	01011011	[	01101110	n
00100011	#	00110110	6	01001001	I	01011100	\	01101111	o
00100100	$	00110111	7	01001010	J	01011101	]	01110000	p
00100101	%	00111000	8	01001011	K	01011110	^	01110001	q
00100110	&	00111001	9	01001100	L	01011111	_	01110010	r
00100111	'	00111010	:	01001101	M	01100000	`	01110011	s
00101000	(	00111011	;	01001110	N	01100001	a	01110100	t
00101001	)	00111100	<	01001111	O	01100010	b	01110101	u
00101010	*	00111101	=	01010000	P	01100011	c	01110110	v
00101011	+	00111110	>	01010001	Q	01100100	d	01110111	w
00101100	,	00111111	?	01010010	R	01100101	e	01111000	x
00101101	-	01000000	@	01010011	S	01100110	f	01111001	y
00101110	.	01000001	A	01010100	T	01100111	g	01111010	z
00101111	/	01000010	B	01010101	U	01101000	h	01111011	{
00110000	0	01000011	C	01010110	V	01101001	i	01111100	|
00110001	1	01000100	D	01010111	W	01101010	j	01111101	}
00110010	2	01000101	E	01011000	X	01101011	k	01111110	~

Bites and Bytes

Bit	One binary digit
Byte	8 bits
Kilobit	1,024 or 2^{10} bits
Kilobyte	1,024 or 2^{10} bytes
Megabit	1,048,576 or 2^{20} bits
Megabyte	1,048,576 or 2^{20} bytes
Gigabit	2^{30} bits
Gigabyte	2^{30} bytes
Terabyte	2^{40} bytes
Petabyte	2^{50} bytes
Exabyte	2^{60} bytes

2.1.2 Basic Computer Data Types

- Registers contain either data or control information
- Control information is a bit or group of bits used to specify the sequence of command signals needed for data manipulation
- Data are numbers and other binary-coded information that are operated on
- Possible data types in registers:
    - Numbers used in computations
    - Letters of the alphabet used in data processing
    - Other discrete symbols used for specific purposes
- All types of data, except binary numbers, are represented in binary-coded form
- A number system of *base*, or *radix*, *r* is a system that uses distinct symbols for *r* digits
- Numbers are represented by a string of digit symbols
- The string of digits 724.5 represents the quantity

  $7 \times 10^2 + 2 \times 10^1 + 4 \times 10^0 + 5 \times 10^{-1}$
- The string of digits 101101 in the binary number system represents the quantity

  $1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 45$
- $(101101)_2 = (45)_{10}$
- We will also use the octal (radix 8) and hexadecimal (radix 16) number systems $(736.4)_8 = 7 \times 8^2 + 3 \times 8^1 + 6 \times 8^0 + 4 \times 8^{-1} = (478.5)_{10}$

  $(F3)_{16} = F \times 16^1 + 3 \times 16^0 = (243)_{10}$
- Conversion from decimal to radix *r* system is carried out by separating the number into its integer and fraction parts and converting each part separately
- Divide the integer successively by *r* and accumulate the remainders

- Multiply the fraction successively by *r* until the fraction becomes zero

Conversion of decimal 41.6875 into binary.

Integer = 41		Fraction = 0.6875
41		0.6875
20	1	2
10	0	1.3750
5	0	x 2
2	1	0.7500
1	0	x 2
0	1	1.5000
		x 2
		1.0000

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

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

- Each octal digit corresponds to three binary digits
- Each hexadecimal digit corresponds to four binary digits
- Rather than specifying numbers in binary form, refer to them in octal or hexadecimal and reduce the number of digits by 1/3 or ¼, respectively

1	2	7	5	4	3	Octal
1 0 1 0	1 1 1 1		0 1 1 0	0 0 1 1		Binary
A	F		6	3		Hexadecimal

Figure 3-2 Binary, octal, and hexadecimal conversion.

Binary-Coded Octal Numbers

Octal number	Binary-coded octal	Decimal equivalent	
0	000	0	↑
1	001	1	
2	010	2	Code
3	011	3	for one
4	100	4	octal
5	101	5	digit
6	110	6	
7	111	7	↓
10	001 000	8	
11	001 001	9	
12	001 010	10	
24	010 100	20	
62	110 010	50	
143	001 100 011	99	
370	011 111 000	248	

Binary-Coded Hexadecimal Numbers

Hexadecimal number	Binary-coded hexadecimal	Decimal equivalent	
0	0000	0	
1	0001	1	
2	0010	2	
3	0011	3	
4	0100	4	
5	0101	5	
6	0110	6	Code
7	0111	7	for one
8	1000	8	hexadecimal
9	1001	9	digit
A	1010	10	
B	1011	11	
C	1100	12	
D	1101	13	
E	1110	14	
F	1111	15	
14	0001 0100	20	
32	0011 0010	50	
63	0110 0011	99	
F8	1111 1000	248	

- A binary code is a group of *n* bits that assume up to 2^n distinct combinations
- A four bit code is necessary to represent the ten decimal digits – 6 are unused
- The most popular decimal code is called *binary-coded decimal* (BCD)
- BCD is different from converting a decimal number to binary
- For example 99, when converted to binary, is 1100011
- 99 when represented in BCD is 1001 1001

Binary-Coded Decimal (BCD) Numbers

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

- The standard alphanumeric binary code is ASCII
- This uses seven bits to code 128 characters
- Binary codes are required since registers can hold binary information only

American Standard Code for Information Interchange (ASCII)

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		

2.1.3 Complements

- Complements are used in digital computers for simplifying subtraction and logical manipulation
- Two types of complements for each base r system: r's complement and $(r-1)$'s complement
- Given a number N in base r having n digits, the $(r-1)$'s complement of N is defined as $(r^n - 1) - N$

- For decimal, the 9's complement of N is $(10^n - 1) - N$
- The 9's complement of 546700 is 999999 – 546700 = 453299
- The 9's complement of 453299 is 999999 – 453299 = 546700
- For binary, the 1's complement of N is $(2^n - 1) - N$
- The 1's complement of 1011001 is 1111111 – 1011001 = 0100110
- The 1's complement is the true complement of the number – just toggle all bits

- The r's complement of an n-digit number N in base r is defined as $r^n - N$
- This is the same as adding 1 to the $(r-1)$'s complement

- The 10's complement of 2389 is 7610 + 1 = 7611
- The 2's complement of 101100 is 010011 + 1 = 010100
- Subtraction of unsigned *n*-digit numbers: $M - N$
    - Add M to the r's complement of N – this results in
      $M + (r^n - N) = M - N + r^n$
    - If $M \geq N$, the sum will produce an end carry r^n which is discarded
    - 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.

Example: 72532 – 13250 = 59282. The 10's complement of 13250 is 86750.

M	= 72352
10's comp. of N	= +86750
Sum	= 159282
Discard end carry	= -100000
Answer	= 59282

Example for M < N: 13250 – 72532 = -59282

M	= 13250
10's comp. of N	= +27468
Sum	= 40718
No end carry	
Answer	= -59282 (10's comp. of 40718)

Example for X = 1010100 and Y = 1000011

X	= 1010100
2's comp. of Y	= +0111101
Sum	= 10010001
Discard end carry	= -10000000
Answer X – Y	= 0010001

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

2.2 Fixed-Point & Floating Point Representation

2.2.1 Fixed-Point Representation:

This representation has fixed number of bits for integer part and for fractional part. For example, if given fixed-point representation is IIII.FFFF, then you can store minimum value is 0000.0001 and maximum value is 9999.9999. There are three parts of a fixed-point number representation: the sign field, integer field, and fractional field.

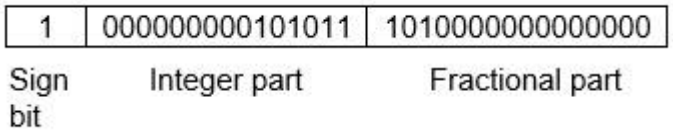
We can represent these numbers using:

- Signed representation: range from $-(2^{(k-1)}-1)$ to $(2^{(k-1)}-1)$, for k bits.
- 1's complement representation: range from $-(2^{(k-1)}-1)$ to $(2^{(k-1)}-1)$, for k bits.
- 2's complementation representation: range from $-(2^{(k-1)})$ to $(2^{(k-1)}-1)$, for k bits.

2's complementation representation is preferred in computer system because of unambiguous property and easier for arithmetic operations.

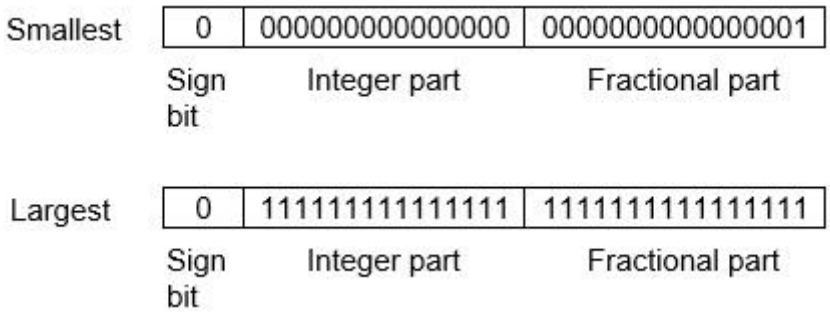
Example: Assume number is using 32-bit format which reserve 1 bit for the sign, 15 bits for the integer part and 16 bits for the fractional part.

Then, -43.625 is represented as following:



Where, 0 is used to represent + and 1 is used to represent. 000000000101011 is 15 bit binary value for decimal 43 and 1010000000000000 is 16 bit binary value for fractional 0.625.

The advantage of using a fixed-point representation is performance and disadvantage is relatively limited range of values that they can represent. So, it is usually inadequate for numerical analysis as it does not allow enough numbers and accuracy. A number whose representation exceeds 32 bits would have to be stored inexactly.



These are above smallest positive number and largest positive number which can be store in 32-bit representation as given above format. Therefore, the smallest positive number is $2^{-16} \approx 0.000015$ approximate and the largest positive number is $(2^{15}-1)+(1-2^{-16})=2^{15}(1-2^{-16})=32768$, and gap between these numbers is 2^{-16}.

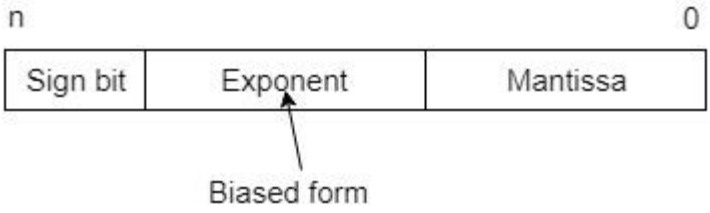
We can move the radix point either left or right with the help of only integer field is 1.

2.2.2 Floating-Point Representation:

This representation does not reserve a specific number of bits for the integer part or the fractional part. Instead it reserves a certain number of bits for the number (called the mantissa or significand) and a certain number of bits to say where within that number the decimal place sits (called the exponent).

The floating number representation of a number has two part: the first part represents a signed fixed point number called mantissa. The second part of designates the position of the decimal (or binary) point and is called the exponent. The fixed point mantissa may be fraction or an integer. Floating -point is always interpreted to represent a number in the following form: Mxr^e.

Only the mantissa m and the exponent e are physically represented in the register (including their sign). A floating-point binary number is represented in a similar manner except that is uses base 2 for the exponent. A floating-point number is said to be normalized if the most significant digit of the mantissa is 1.



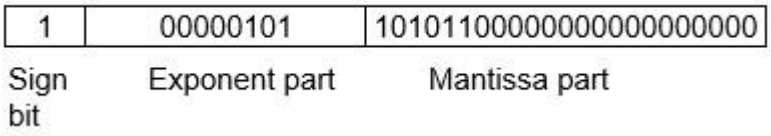
So, actual number is $(-1)^s(1+m)x2^{(e-Bias)}$, where *s* is the sign bit, *m* is the mantissa, *e* is the exponent value, and *Bias* is the bias number.

Note that signed integers and exponent are represented by either sign representation, or one's complement representation, or two's complement representation.

The floating point representation is more flexible. Any non-zero number can be represented in the normalized form of $\pm(1.b_1b_2b_3 ...)_2x2^n$ This is normalized form of a number x.

Example: Suppose number is using 32-bit format: the 1 bit sign bit, 8 bits for signed exponent, and 23 bits for the fractional part. The leading bit 1 is not stored (as it is always 1 for a normalized number) and is referred to as a *"hidden bit*".

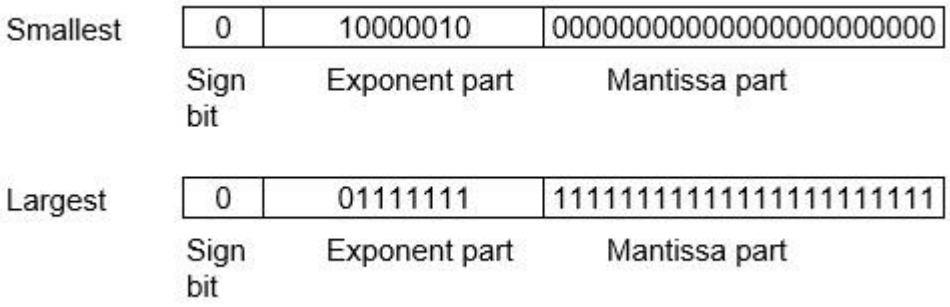
Then −53.5 is normalized as $-53.5=(-110101.1)_2=(-1.101011)x2^5$, which is represented as following below,



Where 00000101 is the 8-bit binary value of exponent value +5.

Note that 8-bit exponent field is used to store integer exponents $-126 \leq n \leq 127$.

The smallest normalized positive number that fits into 32 bits is $(1.00000000000000000000000)_2 x 2^{-126} = 2^{-126} \approx 1.18 x 10^{-38}$, and largest normalized positive number that fits into 32 bits is $(1.11111111111111111111111)_2 x 2^{127} = (2^{24}-1) x 2^{104} \approx 3.40 x 10^{38}$. These numbers are represented as following below,



The precision of a floating-point format is the number of positions reserved for binary digits plus one (for the hidden bit). In the examples considered here the precision is 23+1=24.

The gap between 1 and the next normalized floating-point number is known as machine epsilon. the gap is $(1+2^{-23})-1=2^{-23}$ for above example, but this is same as the smallest positive floating-point number because of non-uniform spacing unlike in the fixed-point scenario.

Note that non-terminating binary numbers can be represented in floating point representation, e.g., $1/3 = (0.010101 \ldots)_2$ cannot be a floating-point number as its binary representation is non-terminating.

2.3 Register Transfer and Micro-operations:

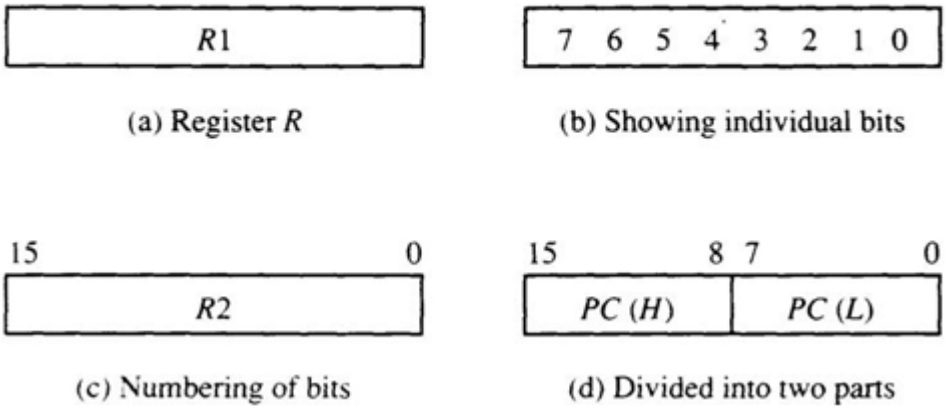
2.3.1 Introduction to Register Transfer

- Digital systems are composed of modules that are constructed from digital components, such as registers, decoders, arithmetic elements, and control logic
- The modules are interconnected with common data and control paths to form a digital computer system
- The operations executed on data stored in registers are called *microoperations*
- A microoperation is an elementary operation performed on the information stored in one or more registers
- Examples are shift, count, clear, and load
- Some of the digital components from before are registers that implement microoperations
- The internal hardware organization of a digital computer is best defined by specifying
    - The set of registers it contains and their functions
    - The sequence of microoperations performed on the binary information stored
    - The control that initiates the sequence of microoperations
- Use symbols, rather than words, to specify the sequence of microoperations
- The symbolic notation used is called a *register transfer language*
- A programming language is a procedure for writing symbols to specify a given computational process
- Define symbols for various types of microoperations and describe associated hardware that can implement the microoperations

Register Transfer

- Designate computer registers by capital letters to denote its function
- The register that holds an address for the memory unit is called MAR
- The program counter register is called PC
- IR is the instruction register and R1 is a processor register
- The individual flip-flops in an *n*-bit register are numbered in sequence from 0 to *n*-1
- Refer to Figure 4.1 for the different representations of a register

Block diagram of register.



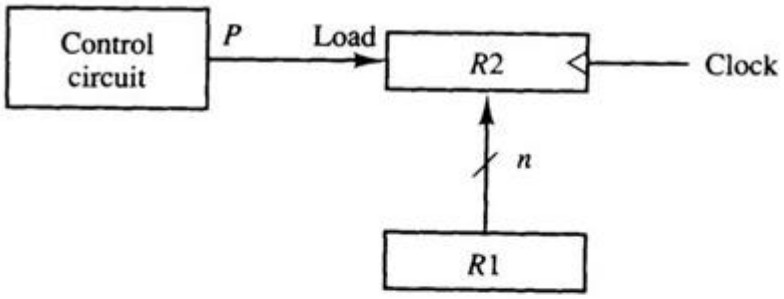
- Designate information transfer from one register to another by
  R2 □ R1

- This statement implies that the hardware is available
  - The outputs of the source must have a path to the inputs of the destination
  - The destination register has a parallel load capability

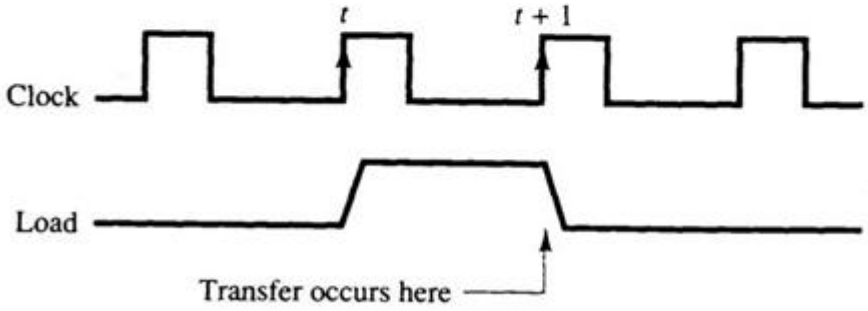
- If the transfer is to occur only under a predetermined control condition, designate it by
  If (P = 1) *then* (R2 □ R1)
  or,
  P: R2 □ R1,
  where P is a control function that can be either 0 or 1

- Every statement written in register transfer notation implies the presence of the required hardware construction

Transfer from $R1$ to $R2$ when $P = 1$.



(a) Block diagram



(b) Timing diagram

- It is assumed that all transfers occur during a clock edge transition
- All microoperations written on a single line are to be executed at the same time
  T: R2 □ R1, R1 □ R2

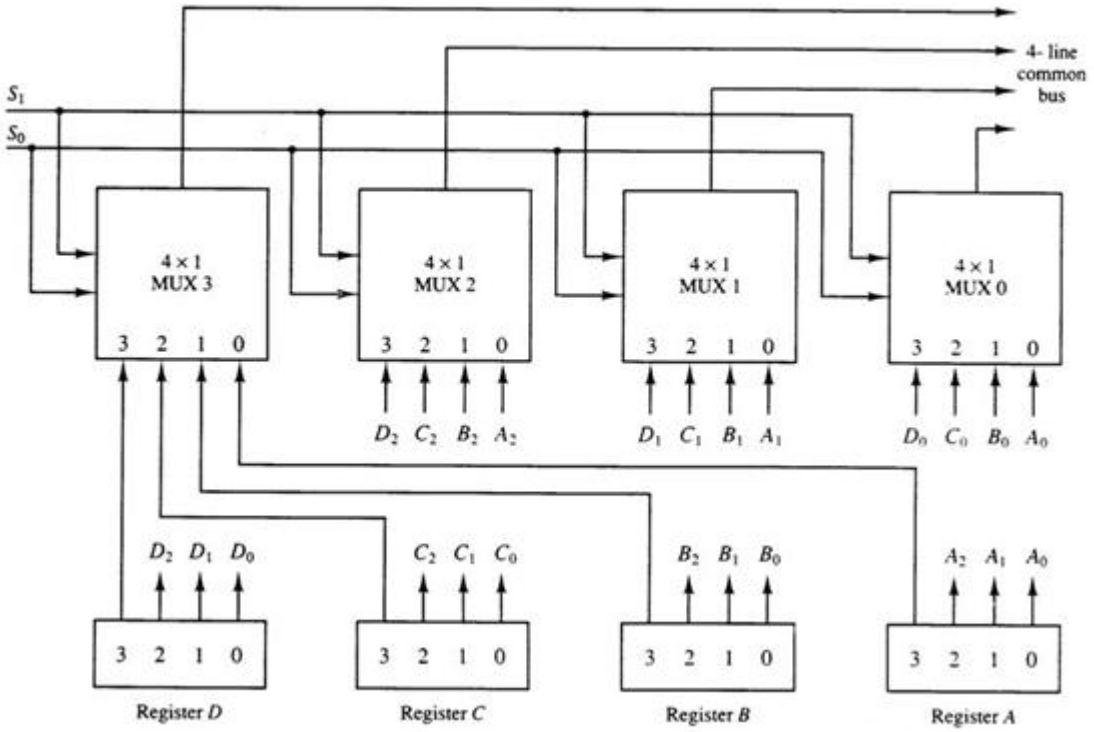
Basic Symbols for Register Transfers

Symbol	Description	Examples
Letters (and numerals)	Denotes a register	*MAR*, *R2*
Parentheses ( )	Denotes a part of a register	*R2*(0–7), *R2*(*L*)
Arrow ←	Denotes transfer of information	*R2* ← *R1*
Comma ,	Separates two microoperations	*R2* ← *R1*, *R1* ← *R2*

Bus and Memory Transfers

- Rather than connecting wires between all registers, a common bus is used
- A bus structure consists of a set of common lines, one for each bit of a register
- Control signals determine which register is selected by the bus during each transfer
- Multiplexers can be used to construct a common bus
- Multiplexers select the source register whose binary information is then placed on the bus
- The select lines are connected to the selection inputs of the multiplexers and choose the bits of one register

Bus system for four registers.

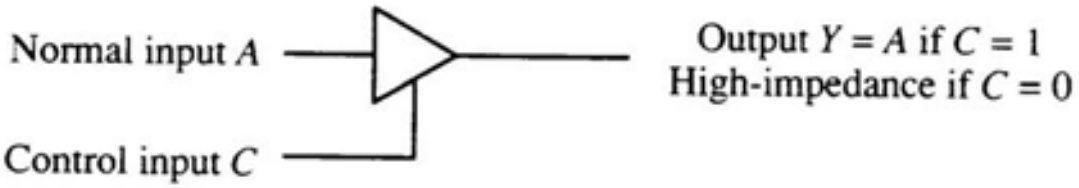


- In general, a bys system will multiplex *k* registers of *n* bits each to produce an *n*-line common bus
- This requires *n* multiplexers – one for each bit
- The size of each multiplexer must be *k* x 1
- The number of select lines required is *log k*
- To transfer information from the bus to a register, the bus lines are connected to the inputs of all destination registers and the corresponding load control line must be activated
- Rather than listing each step as

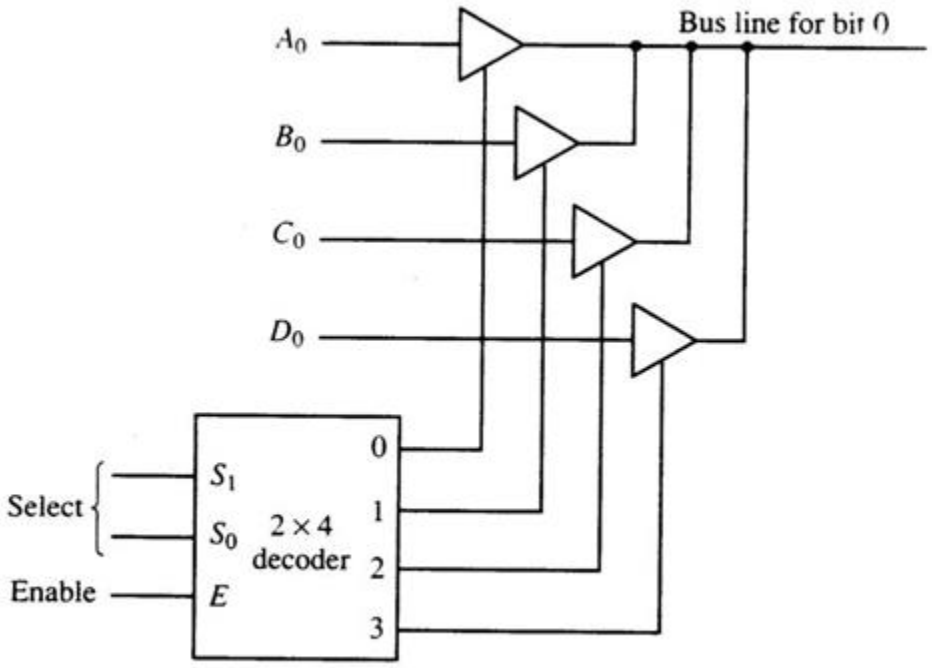
BUS □ C, R1 □ BUS,
use R1 □ C, since the bus is implied

- Instead of using multiplexers, *three-state gates* can be used to construct the bus system
- A three-state gate is a digital circuit that exhibits three states
- Two of the states are signals equivalent to logic 1 and 0
- The third state is a *high-impedance* state – this behaves like an open circuit, which means the output is disconnected and does not have a logic significance

Graphic symbols for three-state buffer.



- The three-state buffer gate has a normal input and a control input which determines the output state
- With control 1, the output equals the normal input
- With control 0, the gate goes to a high-impedance state
- This enables a large number of three-state gate outputs to be connected with wires to form a common bus line without endangering loading effects



Bus line with three state-buffers.

- Decoders are used to ensure that no more than one control input is active at any given time
- This circuit can replace the multiplexer in Figure 4.3
- To construct a common bus for four registers of *n* bits each using three-state buffers, we need *n* circuits with four buffers in each
- Only one decoder is necessary to select between the four registers

- Designate a memory word by the letter M
- It is necessary to specify the address of M when writing memory transfer operations
- Designate the address register by AR and the data register by DR
- The read operation can be stated as:
  Read: DR □ M[AR]
- The write operation can be stated as:
  Write: M[AR] □ R1

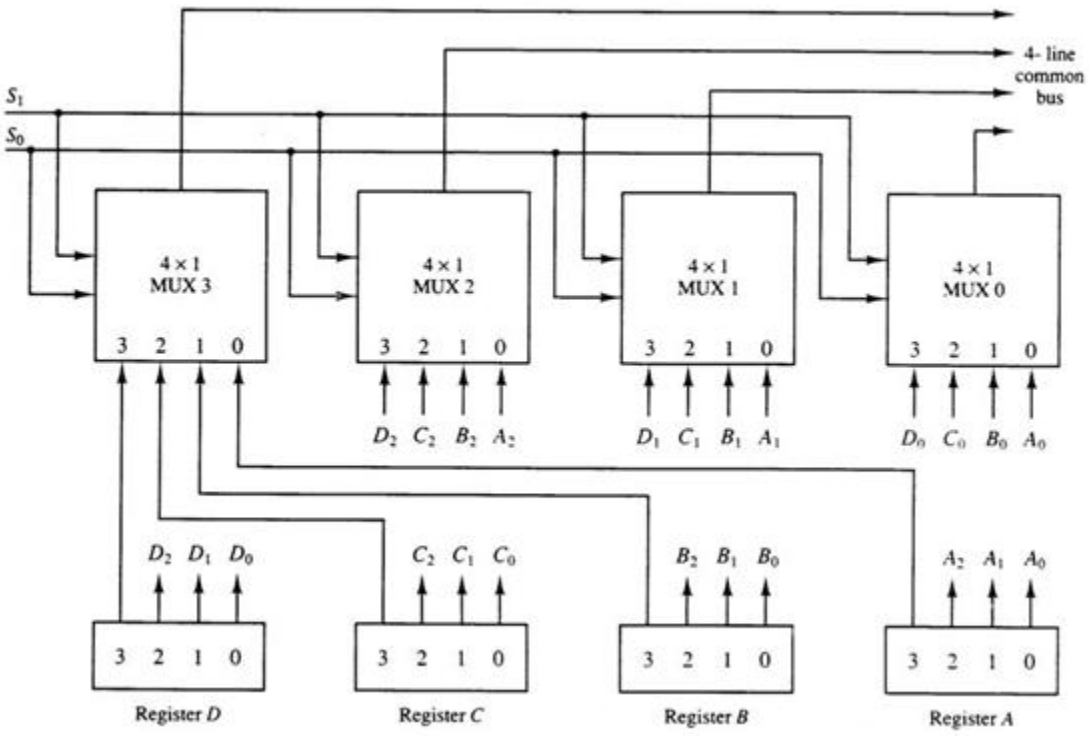
Arithmetic Microoperations

- There are four categories of the most common microoperations:
  - Register transfer: transfer binary information from one register to another
  - Arithmetic: perform arithmetic operations on numeric data stored in registers

- Logic: perform bit manipulation operations on non-numeric data stored in registers
- Shift: perform shift operations on data stored in registers

- The basic arithmetic microoperations are addition, subtraction, increment, decrement, and shift
- Example of addition: R3 □ R1 +R2
- Subtraction is most often implemented through complementation and addition
- Example of subtraction: R3 □ R1 + ~~R2~~ + 1 (strikethrough denotes bar on top – 1's complement of R2)
- Adding 1 to the 1's complement produces the 2's complement
- Adding the contents of R1 to the 2's complement of R2 is equivalent to subtracting

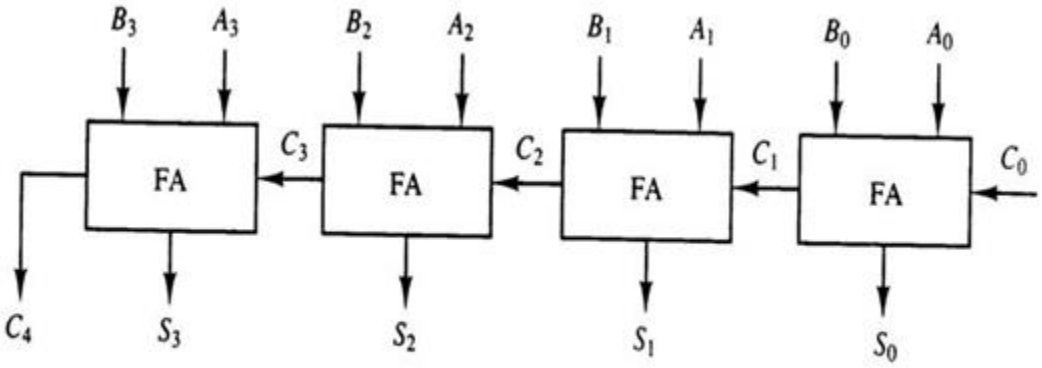
Bus system for four registers.



- Multiply and divide are not included as microoperations
- A microoperation is one that can be executed by one clock pulse
- Multiply (divide) is implemented by a sequence of add and shift microoperations (subtract and shift)

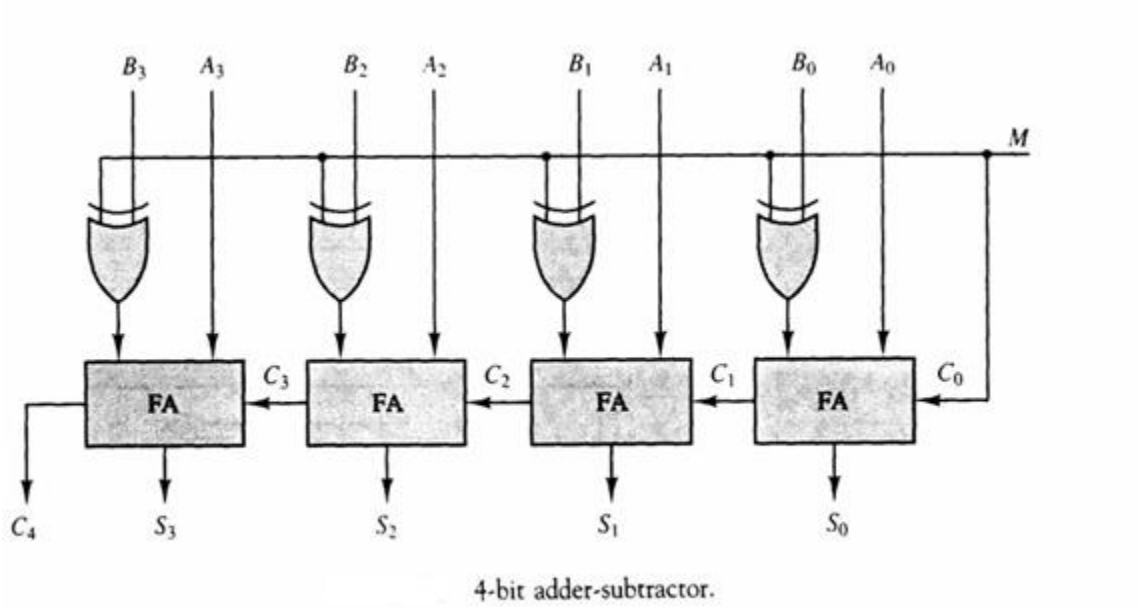
- To implement the add microoperation with hardware, we need the registers that hold the data and the digital component that performs the addition
- A full-adder adds two bits and a previous carry

- A *binary adder* is a digital circuit that generates the arithmetic sum of two binary numbers of any length
- A binary added is constructed with full-adder circuits connected in cascade
- An *n*-bit binary adder requires *n* full-adders



4-bit binary adder.

- The subtraction A-B can be carried out by the following steps
  - Take the 1's complement of B (invert each bit)
  - Get the 2's complement by adding 1
  - Add the result to A
- The addition and subtraction operations can be combined into one common circuit by including an XOR gate with each full-adder

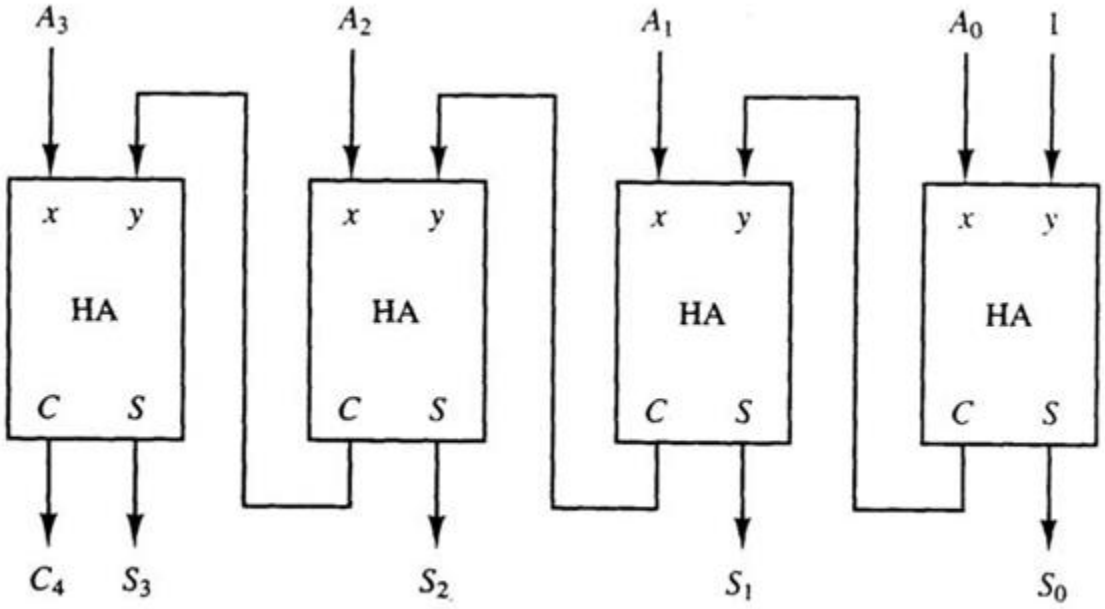


4-bit adder-subtractor.

- The increment microoperation adds one to a number in a register
- This can be implemented by using a binary counter – every time the count enable is active, the count is incremented by one
- If the increment is to be performed independent of a particular register, then use

half-adders connected in cascade

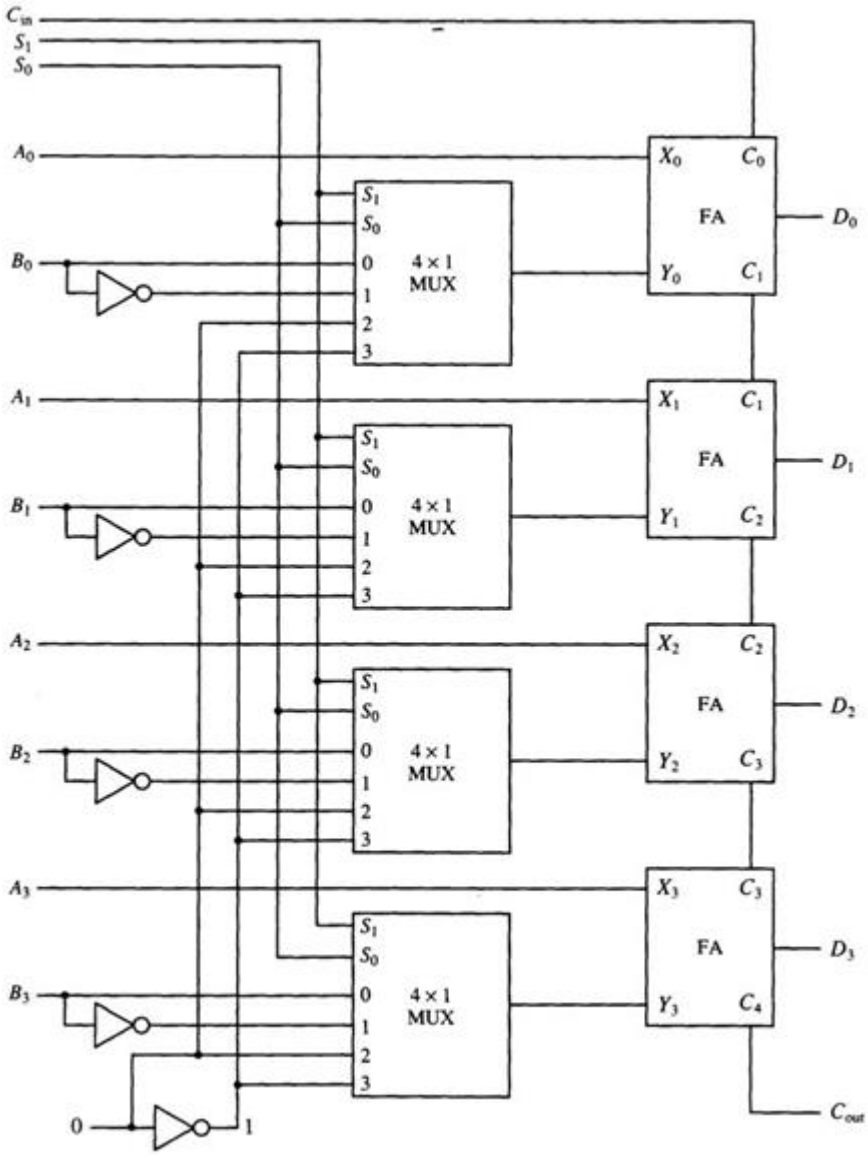
- An *n*-bit binary incrementer requires *n* half-adders



4-bit binary incrementer.

- Each of the arithmetic microoperations can be implemented in one composite arithmetic circuit
- The basic component is the parallel adder
- Multiplexers are used to choose between the different operations
- The output of the binary adder is calculated from the following sum:

$$D = A + Y + C_{in}$$



4-bit arithmetic circuit.

Arithmetic Circuit Function Table

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

Logic Microoperations

- Logic operations specify binary operations for strings of bits stored in registers and treat each bit separately
- Example: the XOR of R1 and R2 is symbolized by

  P: R1 □ R1 ⊕ R2
- Example: R1 = 1010 and R2 = 1100

  1010 Content of R1
  1100 Content of R2
  0110 Content of R1 after P = 1

- Symbols used for logical microoperations:
  - OR: □
  - AND: □
  - XOR: ⊕

- The + sign has two different meanings: logical OR and summation
- When + is in a microoperation, then summation
- When + is in a control function, then OR
- Example:

  P + Q: R1 □ R2 + R3, R4 □ R5 □ R6
- There are 16 different logic operations that can be performed with two binary variables

Truth Tables for 16 Functions of Two Variables

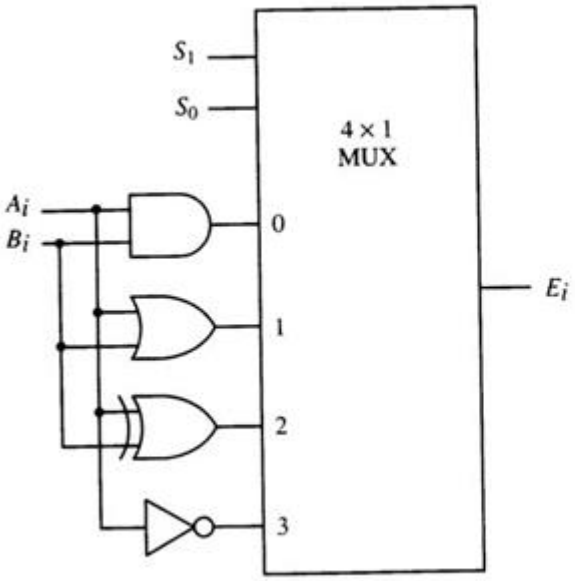
x	y	F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}	F_{11}	F_{12}	F_{13}	F_{14}	F_{15}
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

Sixteen Logic Microoperations

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

- The hardware implementation of logic microoperations requires that logic gates be inserted for each bit or pair of bits in the registers
- All 16 microoperations can be derived from using four logic gates

One stage of logic circuit.



(a) Logic diagram

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

(b) Function table

- Logic microoperations can be used to change bit values, delete a group of bits, or insert new bit values into a register
- The *selective-set* operation sets to 1 the bits in A where there are corresponding 1's in B

  1010 A before
  <u>1100</u> B (logic operand)
  1110 A after

  A □ A □ B

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

  1010 A before
  <u>1100</u> B (logic operand)
  0110 A after

  A □ A ⊕ B

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

  1010 A before
  <u>1100</u> B (logic operand)
  0010 A after

A □ A □ ~~B~~

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

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

  A □ A □ B

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

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

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

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

  1010 A
  1010 B

  0000 A □ A ⊕ B

Shift Microoperations

- Shift microoperations are used for serial transfer of data
- They are also used in conjunction with arithmetic, logic, and other data-processing operations
- There are three types of shifts: logical, circular, and arithmetic
- A *logical shift* is one that transfers 0 through the serial input
- The symbols *shl* and *shr* are for logical shift-left and shift-right by one position

  R1 □ shl R1

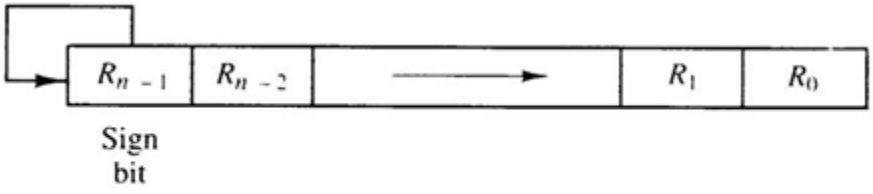
- The *circular shift* (aka rotate) circulates the bits of the register around the two ends without loss of information
- The symbols *cil* and *cir* are for circular shift left and right

Shift Microoperations

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

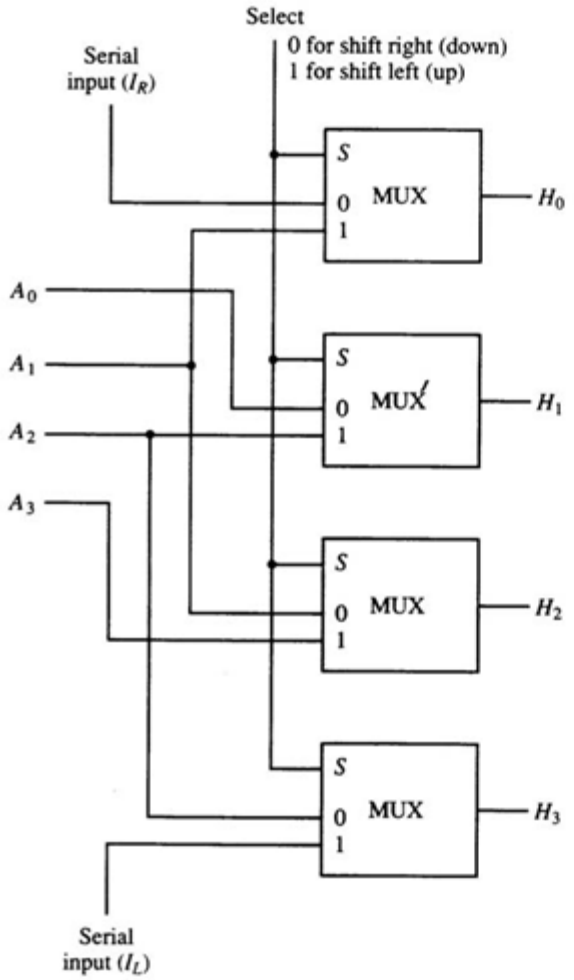
- The *arithmetic shift* shifts a signed binary number to the left or right
- To the left is multiplying by 2, to the right is dividing by 2
- Arithmetic shifts must leave the sign bit unchanged
- A sign reversal occurs if the bit in R_{n-1} changes in value after the shift
- This happens if the multiplication causes an overflow
- An overflow flip-flop V_s can be used to detect the overflow

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



Arithmetic shift right.

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



Function table

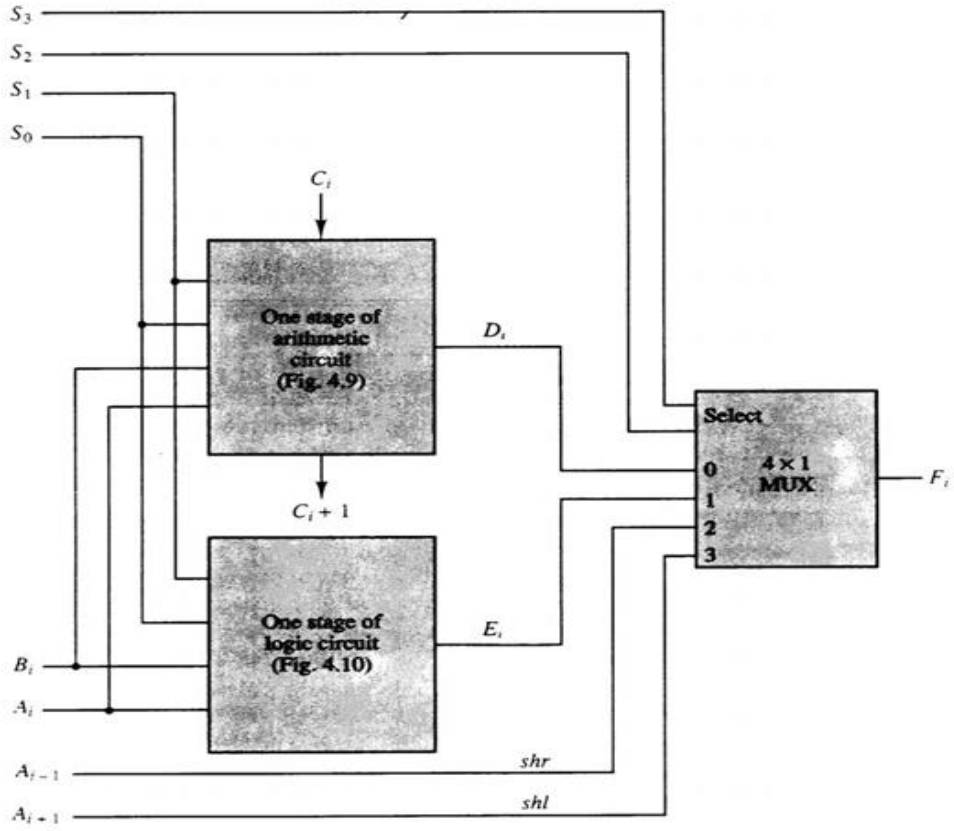
Select	Output			
S	H_0	H_1	H_2	H_3
0	I_R	A_0	A_1	A_2
1	A_1	A_2	A_3	I_L

4-bit combinational circuit shifter.

Arithmetic Logic Shift Unit

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

One stage of arithmetic logic shift unit.



Function Table for Arithmetic Logic Shift Unit

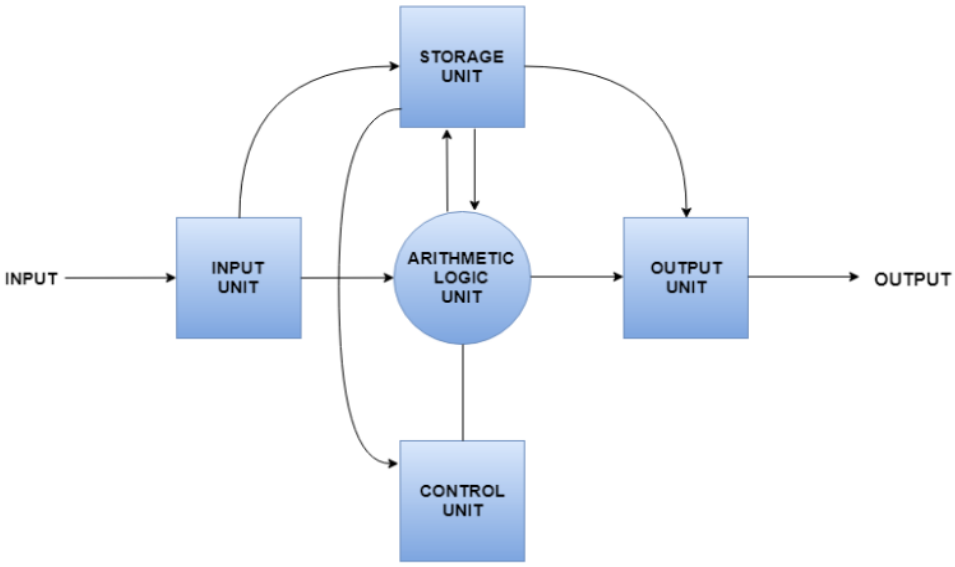
Operation select						
S_3	S_2	S_1	S_0	C_{in}	Operation	Function
0	0	0	0	0	$F = A$	Transfer A
0	0	0	0	1	$F = A + 1$	Increment A
0	0	0	1	0	$F = A + B$	Addition
0	0	0	1	1	$F = A + B + 1$	Add with carry
0	0	1	0	0	$F = A + \overline{B}$	Subtract with borrow
0	0	1	0	1	$F = A + \overline{B} + 1$	Subtraction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

2.7 Basic Computer Organization and Design

2.7.1 Basic Computer Organization: A computer system is basically a machine that simplifies complicated tasks. It should maximize performance and reduce costs as well as power consumption. The different components in the Computer System Architecture are Input Unit, Output Unit, Storage Unit, Arithmetic Logic Unit, Control Unit etc.

Architecture and function of general computer system:

A diagram that shows the flow of data between these units is as follows:



The input data travels from input unit to ALU. Similarly, the computed data travels from ALU to output unit. The data constantly moves from storage unit to ALU and back again. This is because stored data is computed on before being stored again. The control unit controls all the other units as well as their data.

Details about all the computer units are:

1. **Input Unit**

The input unit provides data to the computer system from the outside. So, basically it links the external environment with the computer. It takes data from the input devices, converts it into machine language and then loads it into the computer system. Keyboard, mouse etc. are the most commonly used input devices.

2. **Output Unit**

The output unit provides the results of computer process to the users i.e it links the computer with the external environment. Most of the output data is the form of audio or video. The different output devices are monitors, printers, speakers, headphones etc.

3. **Storage Unit**

Storage unit contains many computer components that are used to store data. It is traditionally divided into primary storage and secondary storage. Primary storage is also known as the main memory and is the memory directly accessible by the CPU. Secondary or external storage is not directly accessible by the CPU. The data from secondary storage needs to be brought into the primary storage before the CPU can use it. Secondary storage contains a large amount of data permanently.

4. **Arithmetic Logic Unit**

All the calculations related to the computer system are performed by the arithmetic logic unit. It can perform operations like addition, subtraction, multiplication, division etc. The control unit transfers data from storage unit to arithmetic logic unit when calculations need to be performed. The arithmetic logic unit and the control unit together form the central processing unit.

5. **Control Unit**

This unit controls all the other units of the computer system and so is known as its central nervous system. It transfers data throughout the computer as required including from storage unit to central processing unit and vice versa. The control unit also dictates how the memory, input output devices, arithmetic logic unit etc. should behave.

2.7.2 Design: Flynn's classification divides computers into four major groups as follows:

Single instruction stream, single data stream (SISD)
Single instruction stream, multiple data stream (SIMD) Multiple instruction streams, single data stream (MISD)
Multiple instruction streams, multiple data stream (MIMD)

SISD represents the organizations of a single computer containing a control unit, a processor unit, and a memory unit. Instructions are executed sequentially and the system may or may not have internal parallel processing capabilities. Parallel processing in this case may be achieved by means of multiple functional units or by pipeline processing.

SIMD represents an organization that includes many processing units under the supervision of a common control unit. All processors receive the same instruction from the control unit but operate on different items of data. The shared memory unit must contain multiple modules so that it can communicate with all the processors simultaneously.

MISD structure is only of theoretical interest since no practical system has been constructed using this organization.

MIMD organization refers to a computer system capable of processing several programs at the same time. Most multiprocessor and multi-computer systems can be classified in this category.

Flynn's classification depends on the distinction between the performance of the control unit and the data processing unit. It emphasizes the behavioral characteristics of the computer system rather than its operational and structural interconnections.

Function of General Computer System

The four basic functions of a computer system are as follows:

- input
- output
- processing
- storage

Let's look at each individually:

Input: Transferring of information into the system. This may be through a user input device - i.e. keyboard, mouse, scanner etc.. Or though previously loaded software/program, cd etc.

Output: Output is the exact opposite of input. Output is the function that allows a computer to display information, from the system, for the user. This can be accomplished through the monitor (or other graphical display), printer, speakers etc.

Processing: This is where the computer actually does the 'work' - manipulating and controlling data over the entire system.

Storage: Most computers are able to store data both temporarily (in order to process), but also long-term (i.e., permanently). Storage takes place on hard-drives or external storage devices.

2.7.3 Instruction Codes

While a **Program**, as we all know, is, A set of instructions that specify the operations, operands, and the sequence by which processing has to occur. An **instruction code** is a group of bits that tells the computer to perform a specific operation part.

Instruction Code: Operation Code

The operation code of an instruction is a group of bits that define operations such as add, subtract, multiply, shift and compliment. The number of bits required for the operation code depends upon the total number of operations available on the computer. The operation code must consist of at least **n bits** for a given **2^n** operations. The operation part of an instruction code specifies the operation to be performed.

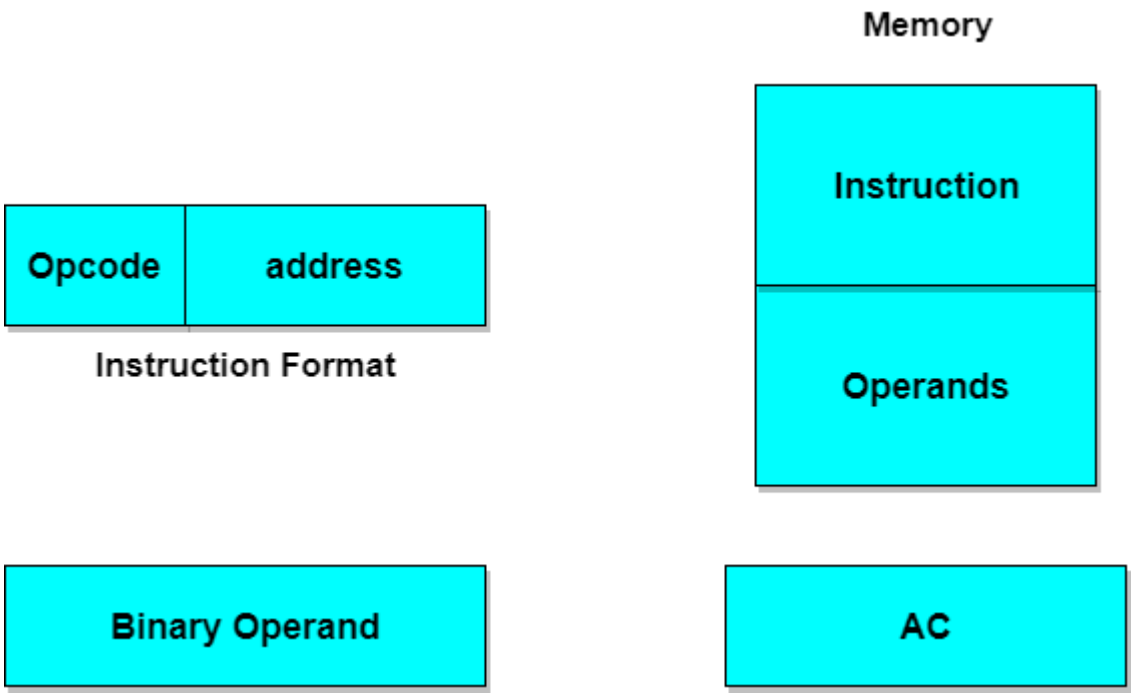
Instruction Code: Register Part

The operation must be performed on the data stored in registers. An instruction code therefore specifies not only operations to be performed but also the registers where the operands(data) will be found as well as the registers where the result has to be stored.

Stored Program Organization

The simplest way to organize a computer is to have **Processor Register** and instruction code with two parts. The first part specifies the operation to be performed and second specifies an address. The memory address tells where the operand in memory will be found.

Instructions are stored in one section of memory and data in another.



Computer with a single processor register is known as **Accumulator (AC)**. The operation is performed with the memory operand and the content of AC.

2.7.4 Computer Registers

Registers are a type of computer memory used to quickly accept, store, and transfer data and instructions that are being used immediately by the CPU. The registers used by the CPU are often termed as Processor registers.

A processor register may hold an instruction, a storage address, or any data (such as bit sequence or individual characters).

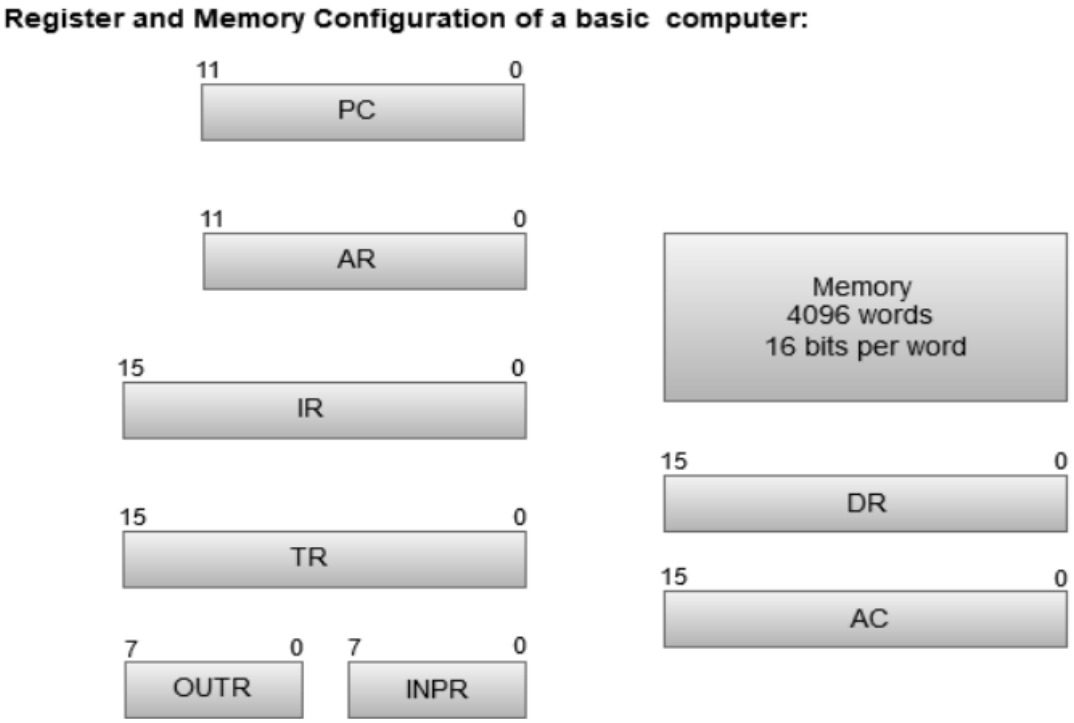
The computer needs processor registers for manipulating data and a register for holding a memory address. The register holding the memory location is used to calculate the address of the next instruction after the execution of the current instruction is completed.

Following is the list of some of the most common registers used in a basic computer:

Register	Symbol	Number of bits	Function
Data register	DR	16	Holds memory operand
Address register	AR	12	Holds address for the memory
Accumulator	AC	16	Processor register
Instruction register	IR	16	Holds instruction code
Program counter	PC	12	Holds address of the instruction
Temporary register	TR	16	Holds temporary data
Input register	INPR	8	Carries input character

Output register	OUTR	8	Carries output character

The following image shows the register and memory configuration for a basic computer.



- The Memory unit has a capacity of 4096 words, and each word contains 16 bits.
- The Data Register (DR) contains 16 bits which hold the operand read from the memory location.
- The Memory Address Register (MAR) contains 12 bits which hold the address for the memory location.
- The Program Counter (PC) also contains 12 bits which hold the address of the next instruction to be read from memory after the current instruction is executed.
- The Accumulator (AC) register is a general purpose processing register.
- The instruction read from memory is placed in the Instruction register (IR).
- The Temporary Register (TR) is used for holding the temporary data during the processing.
- The Input Registers (IR) holds the input characters given by the user.
- The Output Registers (OR) holds the output after processing the input data.

2.7.5 Computer Instructions

The basic computer has three instruction code formats. The **Operation code** (opcode) part of the instruction contains 3 bits and remaining 13 bits depends upon the operation code encountered.

There are three types of formats:

1. Memory Reference Instruction

It uses 12 bits to specify the address and 1 bit to specify the addressing mode (**I**). **I** is equal to 0 for *direct address* and 1 for *indirect address*.

2. Register Reference Instruction

These instructions are recognized by the opcode 111 with a 0 in the left most bit of instruction. The other 12 bits specify the operation to be executed.

3. Input-Output Instruction

These instructions are recognized by the operation code 111 with a 1 in the left most bit of instruction. The remaining 12 bits are used to specify the input-output operation.

Format of Instruction

The format of an instruction is depicted in a rectangular box symbolizing the bits of an instruction. Basic fields of an instruction format are given below:

1. An operation code field that specifies the operation to be performed.
2. An address field that designates the memory address or register.
3. A mode field that specifies the way the operand of effective address is determined.

Computers may have instructions of different lengths containing varying number of addresses. The number of address field in the instruction format depends upon the internal organization of its registers.

Timing and Control

The timing for all registers in the basic computer is controlled by a master clock generator. The clock pulses are applied to all flip-flops and registers in the system, including the flip-flops and registers in the control unit.

The clock pulses do not change the state of a register unless the register is enabled by a control signal. The control signals are generated in the control unit and provide control inputs for the multiplexers in the common bus, control inputs in processor registers, and microoperations for the accumulator.

There are two major types of control organization: hardwired control and microprogrammed control. In the hardwired organization, the control logic is implemented with gates, flip-flops, decoders, and other digital circuits. It has the advantage that it can be optimized to produce a fast mode of operation. In the microprogrammed organization, the control information is stored in a control memory.

The control memory is programmed to initiate the required sequence of microoperations. A hardwired control, as the name implies, requires changes in the wiring among the various components if the design has to be modified or changed. In the microprogrammed control, any required changes or modifications can be done by updating the microprogram in control memory. A hardwired control for the basic computer is presented in this section..

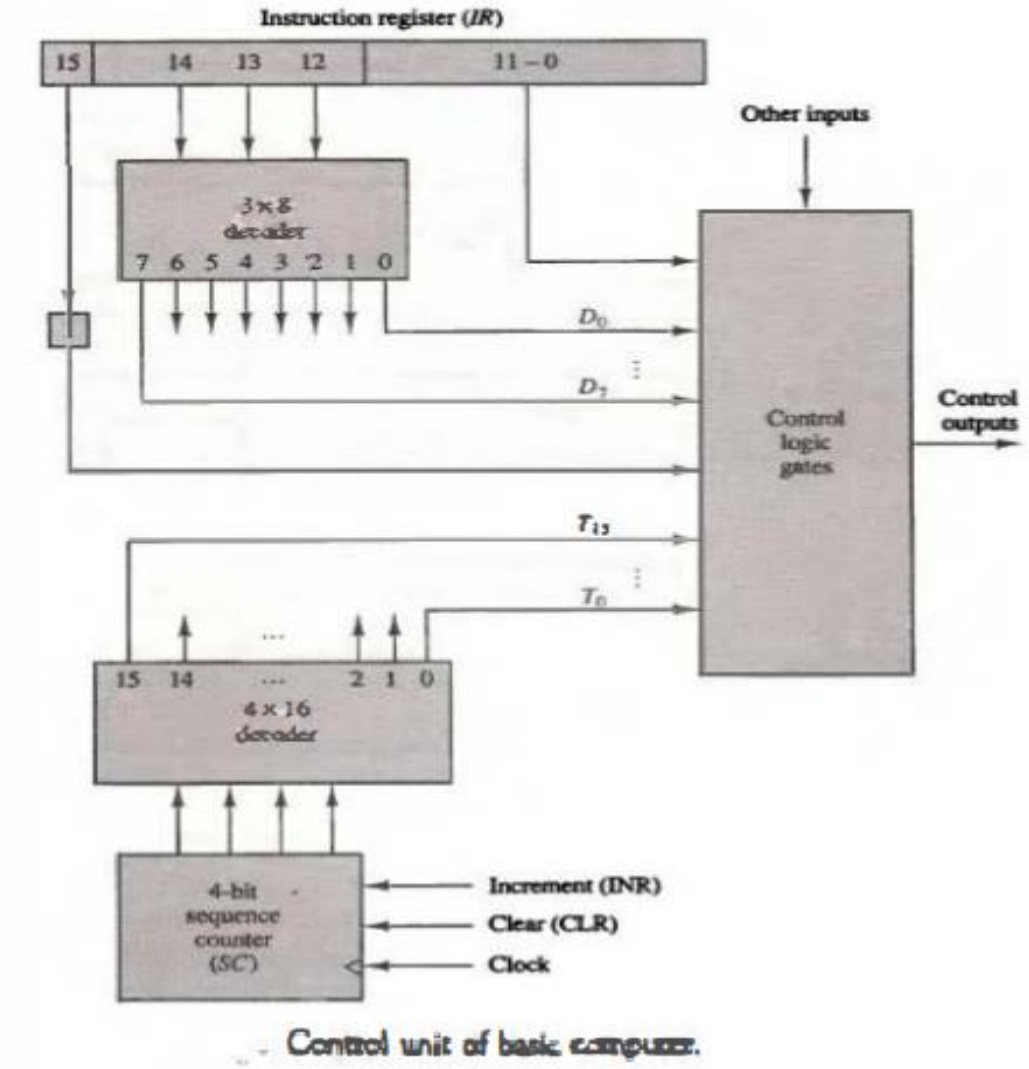
The block diagram of the control unit is shown in Fig. It consists of two decoders, a sequence counter, and a number of control logic gates. An instruction read from memory is placed in the instruction register (IR).

The position of this register in the common bus system is indicated in Fig. 5-4. The instruction register is shown again in Fig., where it is divided into three parts: the I bit, the operation code, and bits 0 through 11. The operation code in bits 12 through 14 are decoded with a 3 x 8 decoder. The eight outputs of the decoder are designated by the symbols D0 through D7

The subscripted decimal number is equivalent to the binary value of the corresponding operation code. Bit 15 of the instruction is transferred to a flip-flop designated by the symbol I. Bits 0 through 11 are applied to the control logic gates. The 4-bit sequence counter can count in binary from 0 through 15. The outputs of the counter are decoded into 16 timing signals T0 through T15

The internal logic of the control gates will be derived later when we consider the design of the computer in detail.

The sequence counter SC can be incremented or cleared synchronously. Most of the time, the counter is incremented to provide the sequence of timing signals out of the 4 x 16 decoder. Once in awhile, the counter is cleared to 0, causing the next active timing signal to be T0.



Control unit of basic computer.

Instruction Cycle

A program residing in the memory unit of a computer consists of a sequence of instructions. These instructions are executed by the processor by going through a cycle for each instruction.

Instruction Cycle: An instruction cycle, also known as fetch-decode-execute cycle is the basic operational process of a computer. This process is repeated continuously by CPU from boot up to shut down of computer.

In a basic computer, each instruction cycle consists of the following phases:

1. Fetch the Instruction

The instruction is fetched from memory address that is stored in PC(Program Counter) and stored in the instruction register IR. At the end of the fetch operation, PC is incremented by 1 and it then points to the next instruction to be executed.

2. Decode the Instruction

The instruction in the IR is executed by the decoder.

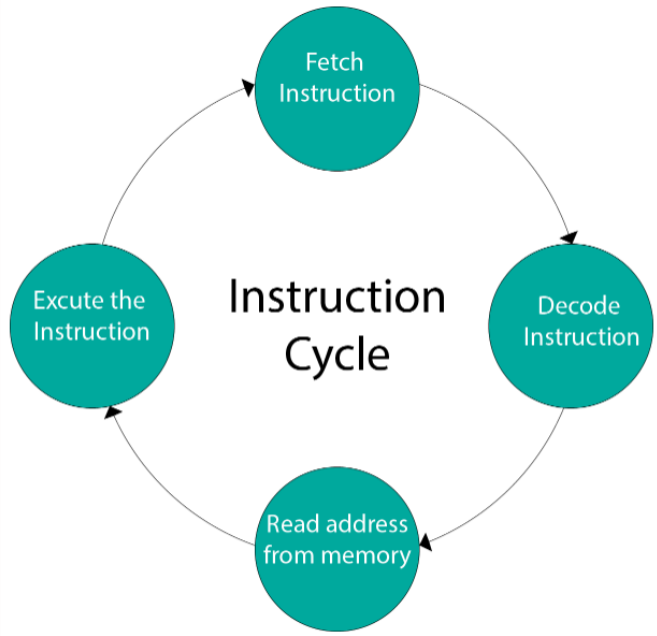
3. Read the Effective Address

If the instruction has an indirect address, the effective address is read from the memory. Otherwise operands are directly read in case of immediate operand instruction.

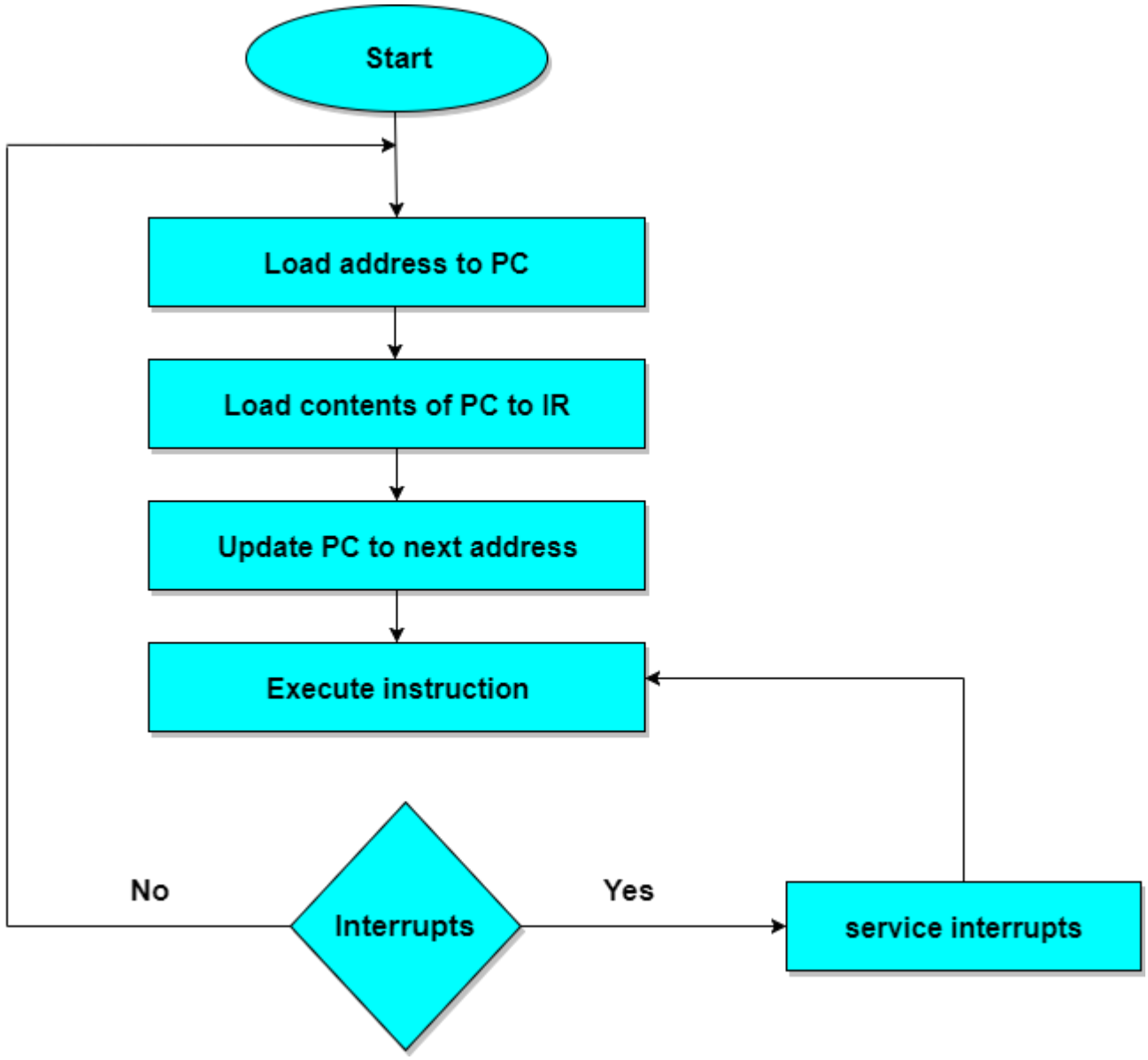
4. Execute the Instruction

The Control Unit passes the information in the form of control signals to the functional unit of CPU. The result generated is stored in main memory or sent to an output device

The cycle is then repeated by fetching the next instruction. Thus in this way the instruction cycle is repeated continuously.



FLOWCHART OF INSTRUCTION CYCLE



Registers Involved In Each Instruction Cycle:

- **Memory address registers(MAR)** : It is connected to the address lines of the system bus. It specifies the address in memory for a read or write operation.
- **Memory Buffer Register(MBR)** : It is connected to the data lines of the system bus. It contains the value to be stored in memory or the last value read from the memory.
- **Program Counter(PC)** : Holds the address of the next instruction to be fetched.
- **Instruction Register(IR)** : Holds the last instruction fetched.

Each computer's CPU can have different cycles based on different instruction sets, but will be similar to the following cycle:

1. **Fetch Stage:** The next instruction is fetched from the memory address that is currently stored in the program counter and stored into the instruction register. At the end of the fetch operation, the PC points to the next instruction that will be read at the next cycle.
2. **Decode Stage:** During this stage, the encoded instruction present in the instruction register is interpreted by the decoder.

   Read the effective address: In the case of a memory instruction (direct or indirect), the execution phase will be during the next clock pulse. If the instruction has an indirect address, the effective address is read from main memory, and any required data is fetched from main memory to be processed and then placed into data registers (clock pulse: T3). If the instruction is direct, nothing is done during this clock pulse. If this is an I/O instruction or a register instruction, the operation is performed during the clock pulse.

3. **Execute Stage**: The control unit of the CPU passes the decoded information as a sequence of control signals to the relevant function units of the CPU to perform the actions required by the instruction, such as reading values from registers, passing them to the ALU to perform mathematical or logic functions on them, and writing the result back to a register. If the ALU is involved, it sends a condition signal back to the CU. The result generated by the operation is stored in the main memory or sent to an output device. Based on the feedback from the ALU, the PC may be updated to a different address from which the next instruction will be fetched.
4. **Repeat Cycle**

2.9.1 Memory-Reference Instructions

Memory-Reference Instructions: In order to specify the microoperations needed for the execution of each instruction, it is necessary that the function that they are intended to perform be defined precisely. Some instructions have an ambiguous description. This is because the explanation of an instruction in words is usually lengthy, and not enough space is available in the table for such a lengthy explanation.

We will now show that the function of the memory-reference instructions can be defined precisely by means of register transfer notation.

The decoded D; for i = 0, 1, 2, 3, 4, 5, and 6 from the operation decoder that belongs to each instruction is included in the table. The effective address of the instruction is in the address register AR and was placed there during timing signal T2 when I = 0, or during timing signal T3 when I = 1. The execution of the memory-reference

instructions starts with timing signal T4• The symbolic description of each instruction is specified in the table in terms of register transfer notation.

The actual execution of the instruction in the bus system will require a sequence of microoperations. This is because data stored in memory cannot be processed directly. The data must be read from memory to a register where they can be operated on with logic circuits. We now explain the operation of each instruction and list the control functions and microoperations needed for their execution.

Memory-Reference Instructions

Symbol	Operation decoder	Symbolic description
AND	D_0	$AC \leftarrow AC \wedge M[AR]$
ADD	D_1	$AC \leftarrow AC + M[AR]$, $E \leftarrow C_{out}$
LDA	D_2	$AC \leftarrow M[AR]$
STA	D_3	$M[AR] \leftarrow AC$
BUN	D_4	$PC \leftarrow AR$
BSA	D_5	$M[AR] \leftarrow PC$, $PC \leftarrow AR + 1$
ISZ	D_6	$M[AR] \leftarrow M[AR] + 1$, If $M[AR] + 1 = 0$ then $PC \leftarrow PC + 1$

AND to AC

This is an instaruction that perform the AND logic operation on pairs of bits in AC and the memory word specified by the effective address. The result of

the operation is transferred to AC . The microoperations that execute this instruction are:

D0T4: DR <- M [AR]

D0T5: AC <- AC /\ DR, SC <--- 0

The control function for this instruction uses the operation decoder D0 since this output of the decoder is active when the instruction has an AND operation whose binary code value 000. Two timing signals are needed to execute the instaruction. The clock transition associared with timing signal T4 transfers the operand from memory into DR . The clock transition associated with the next timing signal T5 transfers to AC the result of the AND logic operation between the contents of DR and AC. The same clock transition clears SC to 0, transferring control to timing signal T0 to start a new instruction cycle.

ADD to AC

This instruction adds the content of the memory word specified by the effective address to the value of AC . The sum is transferred into AC and the output carry C_{out} is transferred to the E (extended accumulator) flip-flop. The rnicrooperations needed to execute this instruction are

D_1T_4: DR← M[AR]

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

Same Two timing signals, T, and T5, are used again but with operation decoder D1 instead of D0, which was used for the AND instruction. After the instruction is fetched from memory and decoded, only one output of the operation decoder will be active, and that output determines the sequence of microoperations that the control follows during the execution of a memory-reference instruction.

LDA: Load to AC

This instruction transfers the memory word specified by the effective address to AC . The microoperations needed to execute this instruction are

D_2T_4: DR← M [AR]

D_2T_5: AC← DR , ← 0

Looking back at the bus system shown in Fig. 5-4 we note that there is no direct path from the bus into AC . The adder and logic circuit receive information from DR which can be transferred into AC . Therefore, it is necessary to read the memory word into DR first and then transfer the content of DR into AC . The reason for not connecting the bus to the inputs of AC is the delay encountered in the adder and logic circuit. It is assumed that the time it takes to read from memory and transfer the word through the bus as well as the adder and logic circuit is more than the time of one clock cycle. By not connecting the bus to the inputs of AC we can maintain one clock cycle per microoperation.

STA: Store AC

This instruction stores the content of AC into the memory word specified by the effective address. Since the output of AC is applied to the bus and the data input of memory is connected to the bus, we can execute this instruction with one microoperation:

D3T4: M [AR] ← AC, SC ← 0

BUN: Branch Unconditionally

This instruction transfers the program to the instruction specified by the effective address. Remember that PC holds the address of the instruction to be read from memory in the next instruction cycle. PC is incremented at

time T1 to prepare it for the address of the next instruction in the program sequence. The BUN instruction allows the programmer to specify an instruction out of sequence and we say that the program branches (or jumps) unconditionally. The instruction is executed with one microoperation:

D4T4: PC ← AR, SC ← 0

The effective address frpom AR is transferred through the common bus to PC .Resetting SC to 0 transfers control to T0• The next instruction is then fetched and executed from the memory address given by the new value in PC .

BSA: Branch and Save Return Address

This instruction is useful for branching to a portion of the program called a subroutine or procedure. When executed, the BSA instruction stores the address of the next instruction in sequence (which is available in PC) into a memory location specified by the effective address. The effective address plus one is then transferred to PC to serve as the address of the first instruction in the subroutine. This operation was specified in Table 5-4 with the following register transfer:

M[AR] <-- PC, PC <-- AR + I

A numerical example that demonstrates how this instruction is used with a subroutine is shown in Fig. 5-10. The BSA instruction is assumed to be in memory at address 20. The I bit is 0 and the address part of the instruction has the binary equivalent of 135. After the fetch and decode phases, PC contains 21, which is the address of the next instruction in the program (referred to as the return address). AR holds the effective address 135. This is shown in part (a) of the figure. The BSA instruction performs the following numerical operation:

M[135] <-- 21, PC <-- 135 + 1 = 136

The result of this operation is shown in part (b) of the figure. The return address 21 is stored in memory location 135 and control continues with the subroutine program starting from address 136. The return to the original program (at address 21) is accomplished by means of an indirect BUN instruction placed at the end of the subroutine. When this instruction is executed, control goes to the indirect phase to read the effective address at location 135, where it finds the previously saved address 21. When the BUN instruction is executed, the effective address 21 is transferred to PC . The next instruction cycle finds PC with the value 21, so control continues to execute the instruction at the return address.

ISZ: Increment and Skip if Zero

This instruction increments the word specified by the effective address, and if the incremented value is equal to 0, PC is incremented by 1. The programmer usually stores a negative number (in 2's complement) in the memory word. As this negative number is repeatedly incremented by one, it eventually reaches the value of zero. At that time PC is incremented by one in order to skip the next instruction in the program.

Since it is not possible to increment a word inside the memory, it is necessary to read the word into DR, increment DR, and store the word back into memory. This is done with the following sequence of microoperations:

D6T4: DR <-- M [AR]

D6T5: DR <-- DR + 1

D,T,: M [AR] <-- DR,

if (DR = 0) then (PC ← PC + 1), SC ← 0

Input-Output and Interrupt

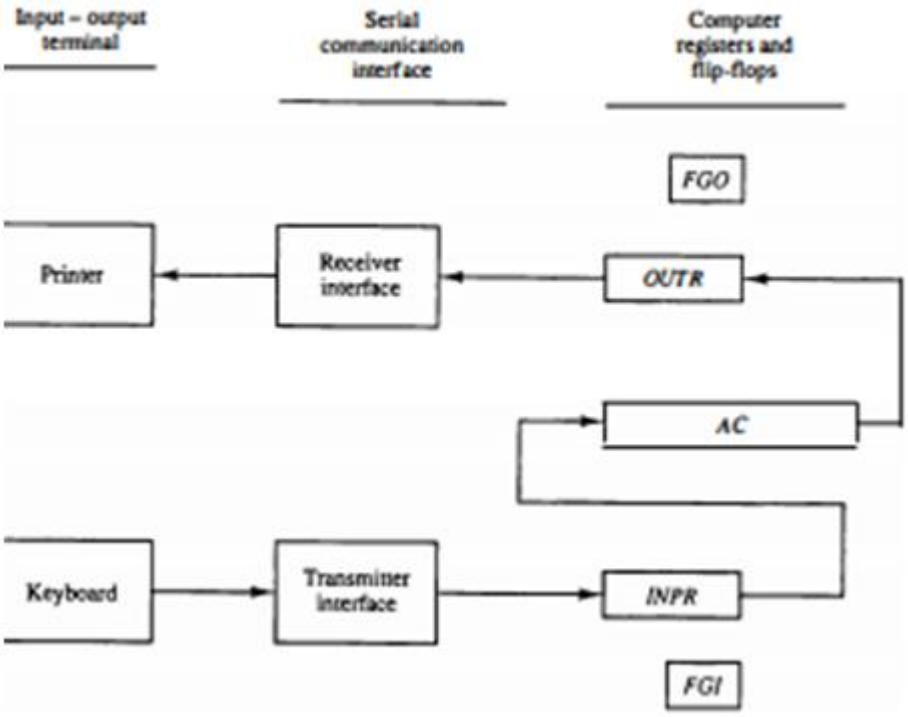
A computer can serve no useful purpose unless it communicates with the external environment. Instructions and data stored in memory must come from some input device. Computational results must be transmitted to the user through some output device. Commercial computers include many types of input and output devices. To demonstrate the most basic reqttirements for input and output communication, we will use as an illustration a terminal unit with a keyboard and printer. Input-output organization is dicsussed further in Chap. 11.

Input-Output Configuration

The terminal sends and receives serial information. Each quantity of information has eight bits of an alphanumeric code. The serial information from the keyboard is shifted into the input register INPR. The serial information for the printer is stored in the output register OUTR. These two registers communicate with a communication interface serially and with the AC in parallel. The input-output configuration is shown in Fig. 5-12. The transmitter interface receives serial information from the keyboard and transmits it to INPR. The receiver interface receives information from OUTR and sends it to the printer serially. The operation of the serial communication interface is explained in Sec. 11-3.

Input Register: The input register INPR consists of eight bits and holds an alphanumeric input information. The 1-bit input flag FGI is a control flip-flop. The flag bit is set to 1 when new information is available in the input device and is cleared to 0 when the information is accepted by the computer. The flag is needed to synchronize the timing rate difference between the input device and the computer. The process of information transfer is as follows. Initially, the input flag FGI is cleared to 0. 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 cannot be changed by striking another key. The computer checks the flag bit; if it is 1, the information from INPR is transferred in parallel into AC and FGI is cleared to 0. Once the flag is cleared, new information can be shifted into INPR by striking another key.

Input-output configuration.



Unit – IV

General Register Organization:

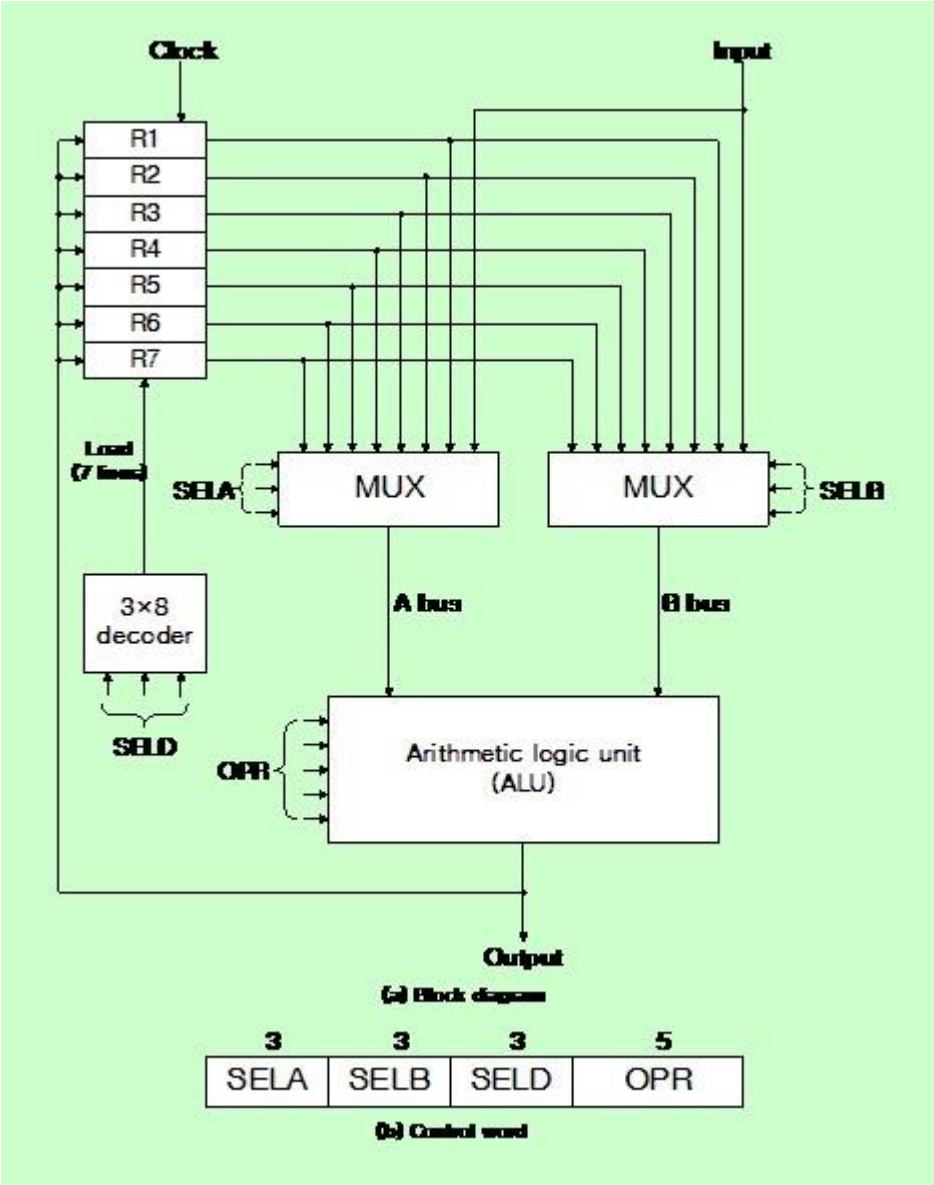
The number of registers in a processor unit may vary from just one processor register to as many as 64 registers or more.

1. One of the CPU registers is called as an accumulator AC or 'A' register. It is the main operand register of the ALU.
2. The data register (DR) acts as a buffer between the CPU and main memory. It is used as an input operand register with the accumulator.
3. The instruction register (IR) holds the opcode of the current instruction.
4. The address register (AR) holds the address of the memory in which the operand resides.
5. The program counter (PC) holds the address of the next instruction to be fetched for execution.

Additional addressable registers can be provided for storing operands and address. This can be viewed as replacing the single accumulator by a set of registers. If the registers are used for many purposes, the resulting computer is said to have general register organization. In the case of processor registers, a registers is selected by the multiplexers that form the buses.

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

The output of each register is connected to true multiplexer (MUX) to form the two buses A & B. The selection lines in each multiplexer select one register or the input data for the particular bus. The A and B buses forms the input to a common ALU. The operation selected in the ALU determines the arithmetic or logic micro-operation that is to be performed. The result of the micro-operation is available for output and also goes into the inputs of the registers. The register that receives the information from the output bus is selected by a decoder. The decoder activates one of the register load inputs, thus providing a transfer both between the data in the output bus and the inputs of the selected destination register.



A Bus organization for seven CPU registers:

The control unit that operates the CPU bus system directs the information flow through the registers and ALU by selecting the various components in the systems.

R1 ® R2 + R3

(1) MUX A selection (SEC A): to place the content of R2 into bus A

(2) MUX B selection (sec B): to place the content of R3 into bus B

(3) ALU operation selection (OPR): to provide the arithmetic addition (A + B)

(4) Decoder destination selection (SEC D): to transfer the content of the output bus into R1

These form the control selection variables are generated in the control unit and must be available at the beginning of a clock cycle. The data from the two source registers propagate through the gates in the multiplexer and the ALU, to the output bus, and into the destination registers, all during the clock cycle intervals.

Stack Organization

The CPU of most computers comprises of a stack or last-in-first-out (LIFO) list wherein information is stored in such a manner that the item stored last is the first to be retrieved. The operation of a stack can be compared to a stack of trays. The last tray placed on top of the stack is the first to be taken off.

The stack in digital computers is essentially a memory unit with an address register that can count only (after an initial value is loaded into it). A Stack Pointer (SP) is the register where the address for the stack is held because its value always points at the top item in the stack. The physical registers of a stack are always available for reading or writing unlike a stack of trays where the tray itself may be taken out or inserted because it is the content of the word that is inserted or deleted.

A stack has only two operations i.e. the insertion and deletion of items. The operation insertion is called push (or push-down) because it can be thought of as the result of pushing a new item on top. The deletion operation is called pop (or pop-up) because it can be thought of as the result of removing one item so that the stack pops up. In actual, nothing is exactly pushed or popped in a computer stack. These operations are simulated by incrementing or decrementing the stack pointer register.

Register Stack

There are two ways to place a stack. Either it can be placed in a portion of a large memory or it can be organized as a collection of a finite number of memory words or registers.

In a 64-word stack, the stack pointer contains 6 bits because 26 = 64. Since SP has only six bits, it cannot exceed a number greater than 63 (111111 in binary). When 63 is incremented by l, the result is 0 since 111111 + 1 = 1000000 in binary, but SP can accommodate only the six least significant bits. Similarly, when 000000 is decremented by 1, the result is 111111. The 1-bit register FULL is set to 1 when the stack is full, and the one-bit register EMTY is set to 1 when the stack is empty of items. DR is the data register that holds the binary data to be written into or read out of the stack.

Initially, SP is cleared to 0, EMTY is set to 1, and FULL is cleared to 0, so that SP points to the word at address 0 and the stack is marked empty and not full. If the stack is not full (if FULL = 0), a new item is inserted with a push operation. The push operation is

implemented with the following sequence of micro-operations:

SP□← SP + 1 Increment stack pointer

M[SP]□← DR Write item on top of the stack

If (SP= 0) then (FULL□← l) Check if stack is full

The stack pointer is incremented so that it points to the address of the next-higher word. The word from DR is inserted into the top of the stack by the memory write operation. The M[SP] denotes the memory word specified by the address presently available in SP whereas the SP holds the address the top of the stack. The storage of the first item is done at address 1 whereas as the last item is store at address 0. If SP reaches 0, the stack is full of items, so FULL is set to 1. This condition is reached if the top item prior to the last push was in location 63 and after incrementing SP, the last item is stored in location 0. Once an item is stored in location 0, there are no more empty registers in the stack. If an item is written in the stack, obviously the stack cannot be empty, so EMTY is cleared to 0.

A new item is deleted from the stack if the stack is not empty (if EMTY <> 0). The pop

operation consists of the following sequence of micro-operations:

DR□← M[SP] Read item from the top of stack

SP□← SP - 1 Decrement stack pointer

If (SP == 0) then (FULL□← 1)

Check if stack is empty EMTY□← 0

Mark the stack not full

DR. reads the top item from the stack. Then the stack pointer is decremented. If its value attains zero, the stack is empty, so EMTY is set to 1. This condition is reached if the item read was in location 1. Once this item is read out, SP is decremented and it attain reaches the value 0, which is the initial value of SP. Note that if a pop operation reads the item from location 0 and then SP is decremented, SP changes to 111111, which is equivalent to decimal 63. In this configuration, the word in address 0 receives the last item in the stack. Note also that an erroneous operation will result if the stack is pushed when FULL = 1 or popped when EMPTY = 1.

Memory Stack

As shown in Fig. 5.3, stack can exist as a stand-alone unit or can be executed in a random-access memory attached to a CPU. The implementation of a stack in the CPU is done by assigning a portion of memory. A portion of memory is assigned to a stack operation and a processor register is used as a stack pointer to execute stack in the CPU. Figure 5.4 shows a portion of computer memory partitioned into three segments - program, data, and stack. The address of the next instruction in the program is located by the program counter PC while an array of data is pointed by address register AR. The top of the stack is located by the stack pointer SP. The three registers are connected to a common address bus, which connects the three registers and either one can provide an address for memory. PC is used during the fetch phase to read an instruction. AR is used during the execute phase to read an operand. SP is used to push or pop items into or from the stack.

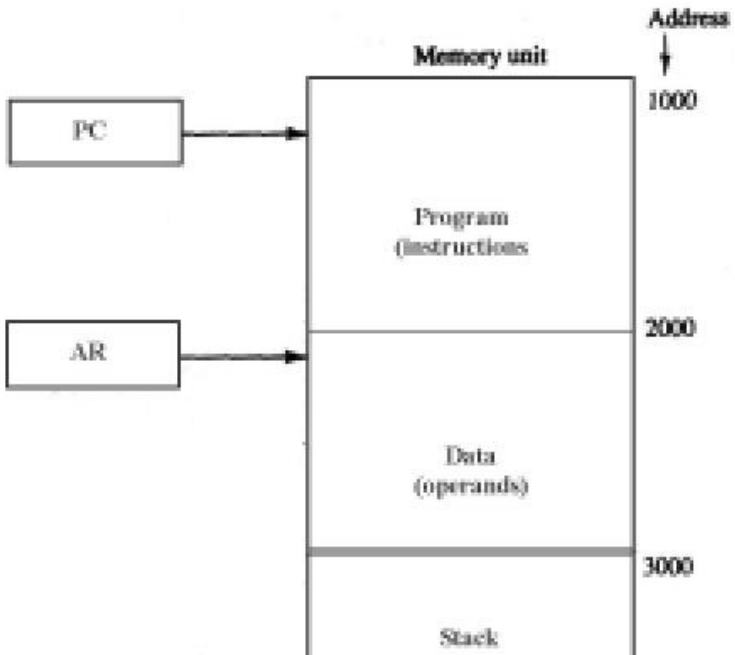


Fig.:Computer memory with program, data, and stack segments.

Fig displays the initial value of SP at 4001 and the growing of stack with decreasing addresses. Thus the first item stored in the stack is at address 4000, the second item is stored at address 3999, and the last address that can be used for the stack is 3000. No checks are provided for checking stack limits.

We assume that the items in the stack communicate with a data register DR. A new

item is inserted with the push operation as follows:

SP□← SP - 1 M[SP]□← DR

The stack pointer is decremented so that it points at the address of the next word. A memory write operation inserts the word form DR into the top of the stack. A new item

is deleted with a pop operation as follows:

DR□← M[SP]

SP□← SP + 1

The top item is read from the stack into DR. The stack pointer is then incremented to point at the next item in the stack.

Most computers are not equipped with hardware to check for stack overflow (full stack) or underflow (empty stack). The stack limits can be checked by using two processor registers: one to hold the upper limit (3000 in this case), and the other to hold the lower limit (40001 in this case). After a push operation, SP is compared with the upper-limit register and after a pop operation, SP is compared with the lower-limit register.

Instruction Formats

- It is the function of the control unit within the CPU to interpret each instruction code
- The bits of the instruction are divided into groups called fields
- The most common fields are:
  - Operation code
  - Address field - memory address or a processor register
  - Mode field - specifies the way the operand or effective address is determined

- A register address is a binary number of k bits that defines one of 2k registers in the CPU
- The instructions may have several different lengths containing varying number of addresses
- The number of address fields in the instruction format of a computer depends on the internal organization of its registers
- Most computers fall into one of the three following organizations:
  o Single accumulator organization

  o General register organization

  o Stack organization

- Single accumulator org. uses one address field

ADD X : AC□← AC + M[X]

- The general register org. uses three address fields

ADD R1, R2, R3: R1□← R2 + R3

- Can use two rather than three fields if the destination is assumed to be one of

the source registers

- Stack org. would require one address field for PUSH/POP operations and none

for operation-type instructions

PUSH X

ADD

- Some computers combine features from more than one organizational structure

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

Three-address instructions:

ADD R1, A, B R1□← M[A] + M[B]

ADD R2, C, D R2□← M[C] + M[D]

M UL X, R1, R2 M[X]□← R1 * R2

Two-address instructions:

MOV R1, A / R 1□← M [A]

ADD R1, B /R1□← R1 + M[B]

MOV R2, C / R2□← M[C]

One-address instructions:

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]

M UL T AC□← AC * M[T]

STORE X M[X]□← AC

Zero-address instructions:

PUSH TOS□← AC

POP M[X]□← TOS

Arithmetic instructions:

Increment INC

Divide DIV

Decrement DEC

Add w/carry ADDC

Add ADD

Sub. w/borrow SUBB

Subtract SUB

Negate (2's comp) N EG

Multiply M UL

- **Some computers have different instructions depending upon the data type**

ADDI Add two binary integer numbers

ADDF Add two floating point numbers

ADDD Add two decimal numbers in BCD

- **Logical and bit manipulation instructions:**

Clear	CLR
Complement	COM
AND	AND
OR	OR
Exclusive-OR	XOR
Clear carry	CLRC
Set carry	SETC
Comp. carry	COMC
Enable inter.	EI
Disable inter.	DI
Clear selected bits -	AND instruction
Set selected bits -	OR instruction
Complement selected bits -	XOR instruction

- **Shift instructions:**

Logical shift right	SHR
Rotate right	ROR
Logical shift left	SHL
Rotate left	ROL
Arithmetic shift right	SHRA
ROR thru carry	RORC
Arithmetic shift left	SHLA
ROL thru carry	ROLC

Program Control

- Program control instructions: provide decision-making capabilities and change the program path
- Typically, the program counter is incremented during the fetch phase to the location of the next instruction
- A program control type of instruction may change the address value in the program counter and cause the flow of control to be altered

• This provides control over the flow of program execution and a capability for branching to different program segments

Branch	BR
Return	RET
Jump	JM P
Compare	CMP
Skip	SKP
Test	TST
Call	CALL

TST and CMP cause branches based upon four status bits: C, S, Z, and V

RISC and CISC

CISC characteristics

CISC, which stands for Complex Instruction Set Computer, is a philosophy for designing chips that are easy to program and which make efficient use of memory. Each instruction in a CISC instruction set might perform a series of operations inside the processor. This reduces the number of instructions required to implement a given program, and allows the programmer to learn a small but flexible set of instructions.

Most common microprocessor designs --- including the Intel(R) 80x86 and Motorola 68K series --- also follow the CISC philosophy.

As we shall see, recent changes in software and hardware technology have forced a re- examination of CISC. But first, let's take a closer look at the decisions which led to CISC.

The disadvantages of CISC

Still, designers soon realized that the CISC philosophy had its own problems, including:

Earlier generations of a processor family generally were contained as a subset in every new version --- so instruction set & chip hardware become more complex with each generation of computers.

So that as many instructions as possible could be stored in memory with the least possible wasted space, individual instructions could be of almost any length---this means that different instructions will take different amounts of clock time to execute, slowing down the overall performance of the machine.

Many specialized instructions aren't used frequently enough to justify their existence --- approximately 20% of the available instructions are used in a typical program.

CISC instructions typically set the condition codes as a side effect of the instruction. Not only does setting the condition codes take time, but programmers have to remember to examine the condition code bits before a subsequent instruction changes them.

RISC

The design of the instruction set for the processor is very important in terms of computer architecture. It's the instruction set of a particular computer that determines the way that machine language programs are constructed. Computer hardware is improvised by various factors, such as upgrading existing models to provide more customer applications adding instructions that facilitate the translation from high-level language into machine language programs and striving to develop machines that move functions from software implementation into hardware implementation. A computer with a large number of instructions is classified as a complex instruction set computer, abbreviated as CISC.

An important aspect of computer architecture is the design of the instruction set for the processor

The instruction set determines the way that machine language programs are constructed

- Many computers have instructions sets of about 100 - 250 instructions

- These computers employ a variety of data types and a large number of addressing modes - complex instruction set computer (CISC)

- A RISC uses fewer instructions with simple constructs so they can be executed much faster within the CPU without having to use memory as often

- The essential goal of CISC architecture is to attempt to provide a single machine instruction for each statement that is written in a high-level language

The major characteristics of CISC architecture are:

- Large number of instructions: Some instructions that perform specialized tasks and are used infrequently
- Large variety of addressing modes
- Variable length instruction formats
- Instructions that manipulate operands in memory
- The goal of RISC architecture is to reduce execution time by simplifying the instructions set

RISC Characteristics

The essential goal of RISC architecture involves an attempt to reduce execution time by simplifying the instruction set of the computer.

The major characteristics of a RISC processor are:

- Relatively few instructions.
- Relatively few addressing modes.
- Memory access limited to load and store instructions.
- All operations done within the registers of the CPU. Fixed length easily decoded instruction format. Single-cycle instruction execution.
- Hardwired rather than micro-programmed control.

Parallel Processing

Parallelism: Performing multiple operations at the same time

Flynn's Taxonomy

SISD: One control unit, one instruction per instruction cycle on one piece of data. May include pipelining (later).

SIMD: Same instruction operating on multiple streams of data at the same time.

MISD: Not used

MIMD: Multiple processors that can execute different instructions at the same time. Multi-core PCs, clusters.

Flynn's classification divides computers into four major groups as follows:

Single instruction stream, single data stream (SISD)

Single instruction stream, multiple data stream (SIMD)

Multiple instruction streams, single data stream (MISD)

Multiple instruction streams, multiple data stream (MIMD)

SISD represents the organizations of a single computer containing a control unit, a processor unit, and a memory unit. Instructions are executed sequentially and the system may or may not have internal parallel processing capabilities. Parallel processing in this case may be achieved by means of multiple functional units or by pipeline processing.

SIMD represents an organization that includes many processing units under the supervision of a common control unit. All processors receive the same instruction from the control unit but operate on different items of data. The shared memory unit must contain multiple modules so that it can communicate with all the processors simultaneously.

MISD structure is only of theoretical interest since no practical system has been constructed using this organization.

MIMD organization refers to a computer system capable of processing several programs at the same time. Most multiprocessor and multi-computer systems can be classified in this category.

Flynn's classification depends on the distinction between the performance of the control unit and the data processing unit. It emphasizes the behavioral characteristics of the computer system rather than its operational and structural interconnections. One type of parallel processing that does not fit Flynn's classification is pipelining.

Parallel Processing

PARALLEL PROCESSING is a term used to denote a large class of techniques that are used to provide simultaneous data-processing tasks for the purpose of increasing the computational speed of a computer system. Instead of processing each instruction sequentially as in a conventional computer, a parallel processing system is able to perform concurrent data processing to achieve faster execution time. For example, while an instruction is being executed in the ALU, the next instruction can be read from memory. The system may have two or more ALUs and be able to execute two or more instructions at the same time. Furthermore, the system may have two or

more processors operating concurrently. The purpose of parallel processing is to speed up the computer processing capability and increase its throughput, that is, the amount of processing that can be accomplished during a given interval of time. With the increase in parallel processing, the cost of the system increases. However, technological developments have reduced hardware costs to the point where parallel processing techniques are economically feasible.

Here we are considering parallel processing under the following main topics:

1. Pipeline processing
2. Vector processing
3. Array processors

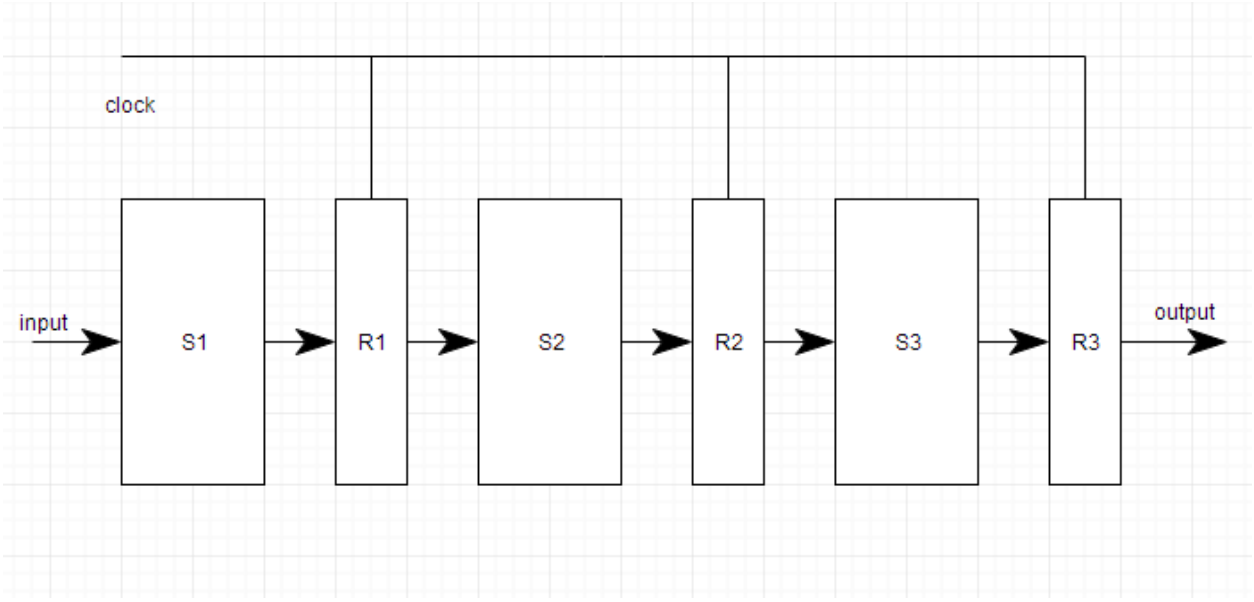
Pipeline processing is an implementation technique where arithmetic sub operations or the phases of a computer instruction cycle overlap in execute vector-processing deals with computations involving large vectors and matrices. Array processors compute on large arrays of data.

Pipelining: Pipelining is the process of accumulating instruction from the processor through a pipeline. It allows storing and executing instructions in an orderly process. It is also known as **pipeline processing**.

Pipelining is a technique where multiple instructions are overlapped during execution. Pipeline is divided into stages and these stages are connected with one another to form a pipe like structure. Instructions enter from one end and exit from another end.

Pipelining increases the overall instruction throughput.

In pipeline system, each segment consists of an input register followed by a combinational circuit. The register is used to hold data and combinational circuit performs operations on it. The output of combinational circuit is applied to the input register of the next segment.



Types of Pipeline

It is divided into 2 categories:

1. Arithmetic Pipeline
2. Instruction Pipeline

Arithmetic Pipeline

Arithmetic pipelines are usually found in most of the computers. They are used for floating point operations, multiplication of fixed point numbers etc. For example: The input to the Floating Point Adder pipeline is:

*X = A*2^a*

*Y = B*2^b*

Here A and B are mantissas (significant digit of floating point numbers), while **a** and **b** are exponents.

The floating point addition and subtraction is done in 4 parts:

1. Compare the exponents.
2. Align the mantissas.
3. Add or subtract mantissas
4. Produce the result.

Registers are used for storing the intermediate results between the above operations.

Instruction Pipeline

In this a stream of instructions can be executed by overlapping *fetch, decode* and *execute* phases of an instruction cycle. This type of technique is used to increase the throughput of the computer system.

An instruction pipeline reads instruction from the memory while previous instructions are being executed in other segments of the pipeline. Thus we can execute multiple instructions simultaneously. The pipeline will be more efficient if the instruction cycle is divided into segments of equal duration.

1. A typical instruction cycle can be divided into many sub cycles like Fetch instruction, Decode instruction, Execute and Store. The instruction cycle and the corresponding sub cycles are performed for each instruction. These sub cycles for different instructions can thus be interleaved or in other words these sub cycles of many instructions can be carried out simultaneously, resulting in reduced overall execution time. This is called instruction pipelining.
2. As mentioned above, to effectively apply pipelining to the process of instruction execution, the instruction cycle must be divided into following phases or sub cycles:

   i) Fetch instruction (F): In this phase, the CPU reads the next instruction from the memory.

ii) Decode instruction (D): The instruction fetched in the previous phase is decoded and interpreted any data operand(s) if needed can also be fetched by the CPU at this time.

iii) Execute (E): The decoded instructions are finally executed by the CPU.

iv) Store(S): The result obtained as a result can then be stored back to the memory. This marks the end of the current instruction cycle.

3. Here instruction processing is divided into four stages. Hence it is also called four stage instruction pipelining. If the instruction cycle is divided into more phases, more pipelining can be achieved. Thus more efficient execution is also possible.
4. When there is no pipelining, a typical processor would take 12 clock cycles to execute three instructions (Assuming that each sub cycle takes one clock cycle to complete).
5. However when a pipelining is used, three instructions would be executed in 6 clock cycles as shown in Figure 5

Clock	1	2	3	4	5	6
I_1	F_1	D_1	E_1	S_1		
I_2		F_2	D_2	E_2	S_2	
I_3			F_3	D_3	E_3	S_3

Figure 5

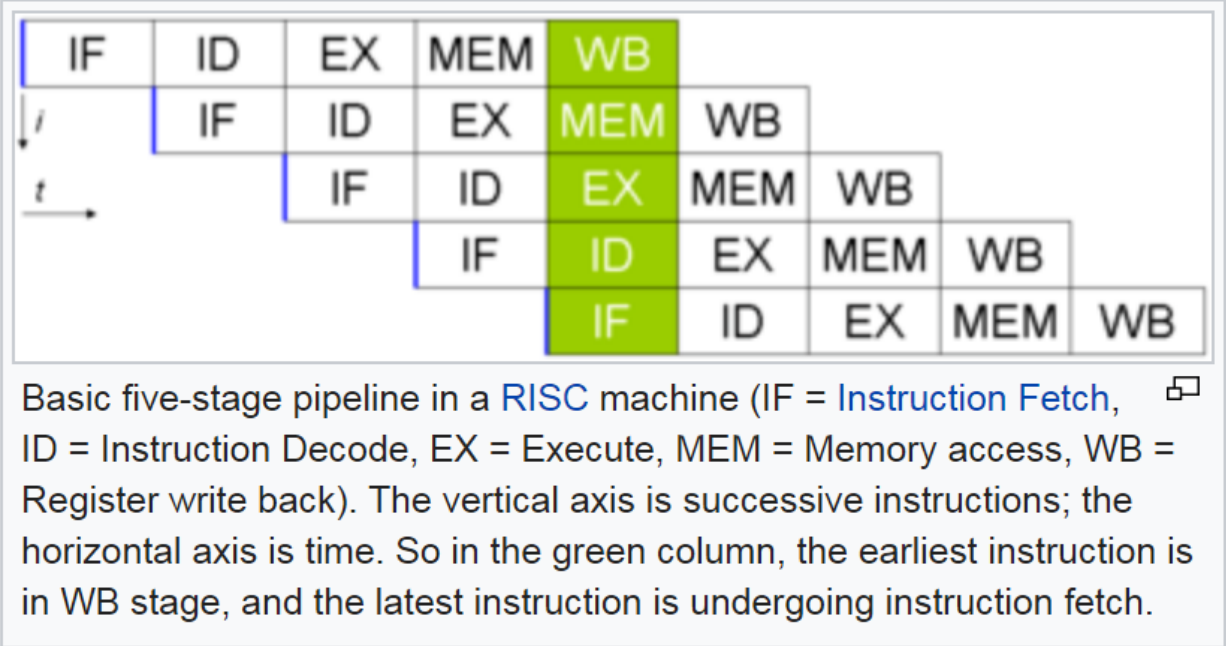
1. In the third clock cycle, the decoding phase of the second instruction is done simultaneously with the fetching of the third instruction and execution of the first instruction. Similar interleaving of sub cycles occurs at all clock cycles except the first and the last one. This is responsible for speeding up the entire process.
2. In this case, the processor hardware needs to be divided into four independent functional units so that the fetch, decode, execute and store phases could be done simultaneously. The need for separate hardware units is sometimes considered.
3. The pipeline works normally only if there are no branch instructions and no interrupts occur. In case of branch instructions or interrupts, the pipeline is flushed. Thus making the pipelining useless.

RISC pipeline

In the history of computer hardware, some early reduced instruction set computer central processing units (RISC CPUs) used a very similar architectural solution, now called a **classic RISC pipeline**. Those CPUs were: MIPS, SPARC, Motorola 88000, and later the notional CPU DLX invented for education.

Each of these classic scalar RISC designs fetched and tried to execute one instruction per cycle. The main common concept of each design was a five-stage execution instruction pipeline. During operation, each pipeline stage worked on one instruction at a time. Each of these stages consisted of an initial set of flip-flops and combinational logic that operated on the outputs of those flip-flops.

Five stage RISC pipeline



Basic five-stage pipeline in a RISC machine (IF = Instruction Fetch, ID = Instruction Decode, EX = Execute, MEM = Memory access, WB = Register write back). The vertical axis is successive instructions; the horizontal axis is time. So in the green column, the earliest instruction is in WB stage, and the latest instruction is undergoing instruction fetch.

Instruction fetch

The Instruction Cache on these machines had a latency of one cycle, meaning that if the instruction was in the cache, it would be ready on the next clock cycle. During the Instruction Fetch stage, a 32-bit instruction was fetched from the cache.

The Program Counter, or PC, is a register that holds the address of the current instruction. It feeds into the PC predictor, which then sends the Program Counter (PC) to the Instruction Cache to read the current instruction. At the same time, the PC predictor predicts the address of the next instruction by incrementing the PC by 4 (all instructions were 4 bytes long). This prediction was always wrong in the case of a taken branch, jump, or exception (see **delayed branches**, below). Later machines would use more complicated and accurate algorithms (branch prediction and branch target prediction) to guess the next instruction address.

Instruction decode

Unlike earlier microcoded machines, the first RISCmachines had no microcode. Once fetched from the instruction cache, the instruction bits were shifted down the pipeline, so that simple combinational logic in each pipeline stage could produce the control signals for the datapath directly from the instruction bits. As a result, very little decoding is done in the stage

traditionally called the decode stage. A consequence of this lack of decoding meant however that more instruction bits had to be used specifying what the instruction should do (and also, what it should not), and that leaves fewer bits for things like register indices.

All MIPS, SPARC, and DLX instructions have at most two register inputs. During the decode stage, these two register names are identified within the instruction, and the two registers named are read from the register file. In the MIPS design, the register file had 32 entries.

At the same time the register file was read, instruction issue logic in this stage determined if the pipeline was ready to execute the instruction in this stage. If not, the issue logic would cause both the Instruction Fetch stage and the Decode stage to stall. On a stall cycle, the stages would prevent their initial flip-flops from accepting new bits.

If the instruction decoded was a branch or jump, the target address of the branch or jump was computed in parallel with reading the register file. The branch condition is computed after the register file is read, and if the branch is taken or if the instruction is a jump, the PC predictor in the first stage is assigned the branch target, rather than the incremented PC that has been computed. Some architectures made use of the ALU in the Execute stage, at the cost of slightly decreased instruction throughput.

The decode stage ended up with quite a lot of hardware: MIPS had the possibility of branching if two registers were equal, so a 32-bit-wide AND tree ran in series after the register file read, making a very long critical path through this stage. Also, the branch target computation generally required a 16 bit add and a 14 bit incrementer. Resolving the branch in the decode stage made it possible to have just a single-cycle branch mispredict penalty. Since branches were very often taken (and thus mispredicted), it was very important to keep this penalty low.

Execute

The Execute stage is where the actual computation occurs. Typically this stage consists of an Arithmetic and Logic Unit, and also a bit shifter. It may also include a multiple cycle multiplier and divider.

The Arithmetic and Logic Unit is responsible for performing boolean operations (and, or, not, nand, nor, xor, xnor) and also for performing integer addition and subtraction. Besides the result, the ALU typically provides status bits such as whether or not the result was 0, or if an overflow occurred.

The bit shifter is responsible for shift and rotations.

Instructions on these simple RISC machines can be divided into three latency classes according to the type of the operation:

- Register-Register Operation (Single-cycle latency): Add, subtract, compare, and logical operations. During the execute stage, the two arguments were fed to a simple ALU, which generated the result by the end of the execute stage.
- Memory Reference (Two-cycle latency). All loads from memory. During the execute stage, the ALU added the two arguments (a register and a constant offset) to produce a virtual address by the end of the cycle.
- Multi-cycle Instructions (Many cycle latency). Integer multiply and divide and all floating-point operations. During the execute stage, the operands to these operations were fed to the multi-cycle multiply/divide unit. The rest of the pipeline was free to continue execution while the multiply/divide unit did its work. To avoid complicating the writeback stage and issue logic, multicycle instruction wrote their results to a separate set of registers.

Memory access

If data memory needs to be accessed, it is done so in this stage.

During this stage, single cycle latency instructions simply have their results forwarded to the next stage. This forwarding ensures that both one and two cycle instructions always write their results in the same stage of the pipeline so that just one write port to the register file can be used, and it is always available.

For direct mapped and virtually tagged data caching, the simplest by far of the numerous data cache organizations, two SRAMs are used, one storing data and the other storing tags.

Writeback

During this stage, both single cycle and two cycle instructions write their results into the register file.

Vector Processing

Vector processing performs the arithmetic operation on the large array of integers or floating-point number. Vector processing operates on all the elements of the array in parallel providing each pass is independent of the other.

Vector processing avoids the **overhead** of the loop control mechanism that occurs in general-purpose computers.

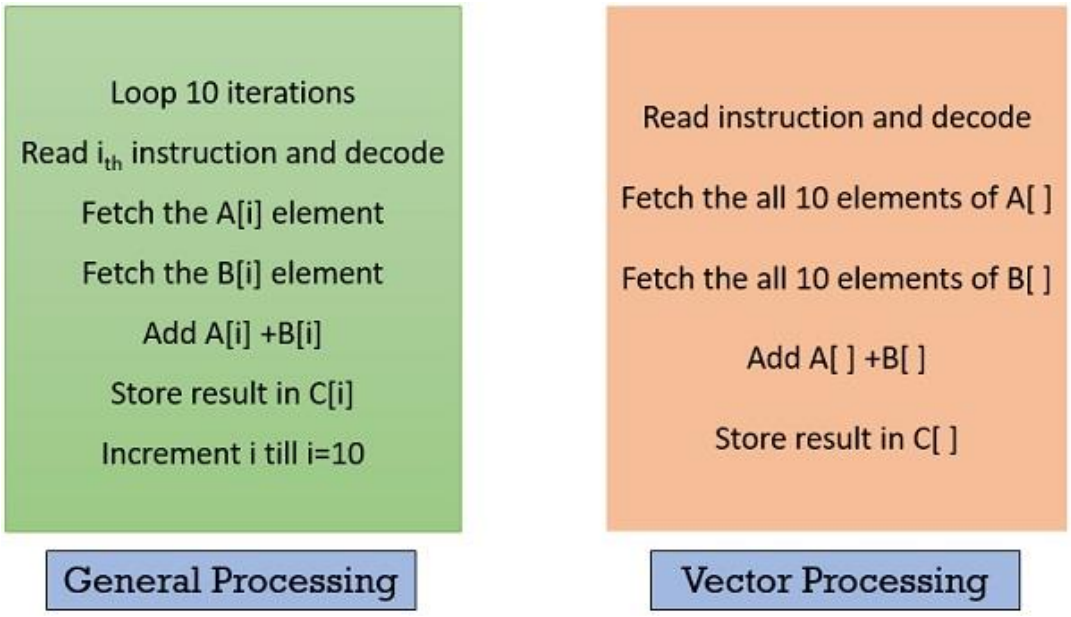
In this section, we will have a brief introduction on vector processing, its characteristics, about vector instructions and how the performance of the vector processing can be enhanced? So lets us start.

Introduction

We need computers that can solve mathematical problems for us which include, arithmetic operations on the large arrays of integers or floating-point numbers quickly. The general-purpose computer would use loops to operate on an array of integers or floating-point numbers. But, for large array using loop would cause overhead to the processor.

To avoid the overhead of processing loops and fasten the computation, some kind of parallelism must be introduced. **Vector processing** operates on the entire array in just one operation i.e. it operates on elements of the array in **parallel**. But, vector processing is possible only if the operations performed in parallel are **independent**.

Look at the figure below, and compare the vector processing with the general computer processing, you will notice the difference. Below, instructions in both the blocks are set to add two arrays and store the result in the third array. Vector processing adds both the array in parallel by avoiding the use of the loop.



Operating on multiple data in just one instruction is also called **Single Instruction Multiple Data** (SIMD) or they are also termed as **Vector instructions**. Now, the data for vector instruction are stored in **vector registers**.

Each vector register is capable of storing several data elements at a time. These several data elements in a vector register is termed as a **vector operand**. So, if there are n number of elements in a vector operand then n is the **length of the vector**.

Supercomputers were evolved to deal with billions of floating-point operations/second. Supercomputer optimizes numerical computations (vector computations).

But, along with vector processing supercomputers are also capable of doing scalar processing. Later, **Array processor** was introduced which particularly deals with vector processing; they do not indulge in scalar processing.

The pipelined vector processors can be classified into two types based on from where the operand is being **fetched** for vector processing. The two architectural classifications are Memory-to-Memory and Register-to-Register.

In **Memory-to-Memory** vector processor the operands for instruction, the intermediate result and the final result all these are retrieved from the **main memory**. TI-ASC, CDC STAR-100, and Cyber-205 use memory-to-memory format for vector instructions.

In **Register-to-Register** vector processor the source operands for instruction, the intermediate result, and the final result all are retrieved from **vector or scalar registers**. Cray-1 and Fujitsu VP-200 use register-to-register format for vector instructions.

Array Processors

The classical structure of an SIMD array architecture is conceptually simple, and is illustrated in Figure 1. In such architectures a program consists of a mixture of scalar and array instructions. The scalar instructions are sent to the scalar processor and the array instructions are *broadcast* to all array elements in parallel. Array elements are incapable of operating autonomously, and must be driven by the control unit.

There are two important control mechanisms: a *local control* mechanism by which array elements use local state information to determine whether they should execute a broadcast instruction or ignore it, and a *global control* mechanism by which the control unit extracts global information from the array elements to determine the outcome of a conditional control transfer within the user's program. Global information can be extracted in one of two ways. Either the control unit reads state information from one, or a group, of array elements, or it senses a boolean control line representing the logical OR (or possibly the logical AND) of a particular local state variable from *every* array element.

The three major components of an array structure are the array units, the memory they access, and the connections between the two. There are two ways in which these components can be organised. Figure 2 shows the basic structure of an array processor in which memory is shared between the array elements and Figure 3 illustrates the basic structure of an array processor in which all memory is distributed amongst the array elements.

If all memory is shared then the switch network connecting the array units to the memory must be capable of sustaining a high rate of data transfer, since *every* instruction will require massive movement of data between these two components. Alternatively, if the memory is distributed then the majority of operands will hopefully reside within the local memory of each processing element (where processing element = arithmetic unit + memory module), and a much lower performance from the switch network can be tolerated. The design of the switch network is of central importance, a topic is covered in the section on **Networks**.

Early examples of these two styles of array processor architecture were the highly influential ILLIAC IV machine, which had a fully distributed memory, and the ill-fated Burroughs Scientific Processor (BSP), which had a shared memory.

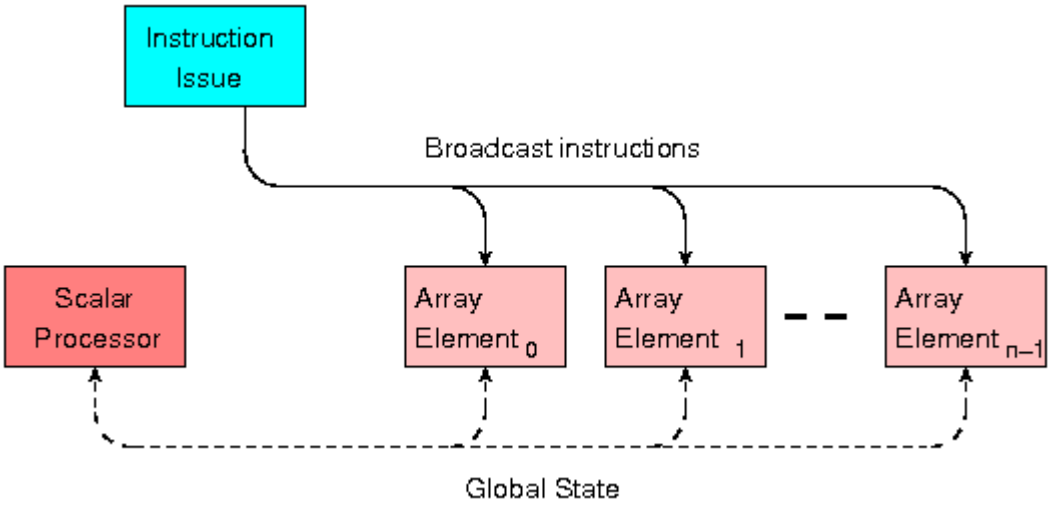


Figure 1. Classical SIMD Array Architecture

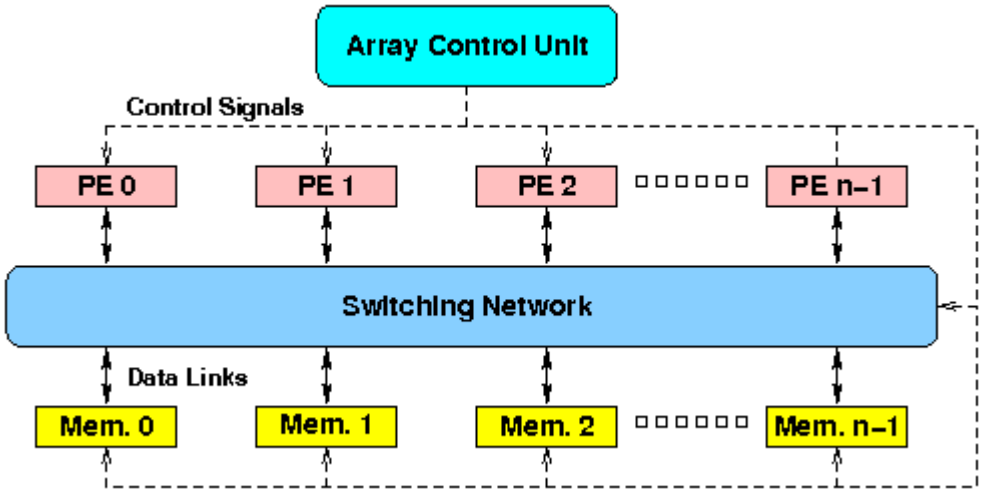


Figure 2. Array Processor with Shared Memory

UNIT 5

COMPUTER ARITHMETIC

Introduction:

Data is manipulated by using the arithmetic instructions in digital computers. Data is manipulated to produce results necessary to give solution for the computation problems. The Addition, subtraction, multiplication and division are the four basic arithmetic operations. If we want then we can derive other operations by using these four operations.

To execute arithmetic operations there is a separate section called arithmetic processing unit in central processing unit. The arithmetic instructions are performed generally on binary or decimal data. Fixed-point numbers are used to represent integers or fractions. We can have signed or unsigned negative numbers. Fixed-point addition is the simplest arithmetic operation.

If we want to solve a problem then we use a sequence of well-defined steps. These steps are collectively called algorithm. To solve various problems we give algorithms.

In order to solve the computational problems, arithmetic instructions are used in digital computers that manipulate data. These instructions perform arithmetic calculations.

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

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

Addition and Subtraction :

Addition and Subtraction with Signed –Magnitude Data

We designate the magnitude of the two numbers by A and B. Where the signed numbers are added or subtracted, we find that there are eight different conditions to consider, depending on the sign of the numbers and the operation performed. These conditions are listed in the first column of Table 4.1. The other columns in the table show the actual operation to be performed with the magnitude of the numbers. The last column is needed to present a negative zero. In other words, when two equal numbers are subtracted, the result should be +0 not -0.

The algorithms for addition and subtraction are derived from the table and can be stated as follows (the words parentheses should be used for the subtraction algorithm)

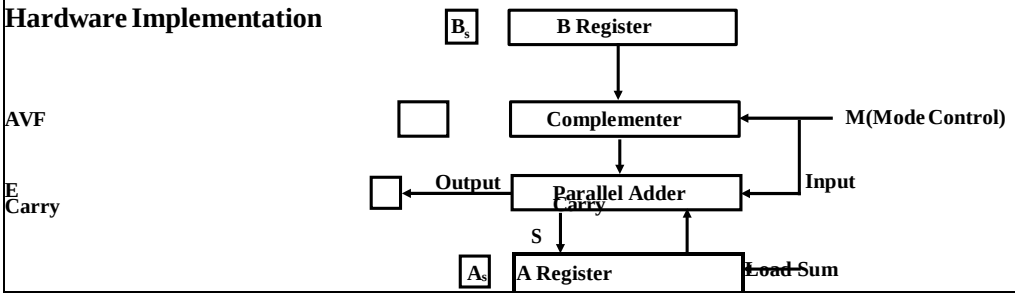
Addition and Subtraction of Signed-Magnitude Numbers

SIGNED MAGNITUDEADDITION AND SUBTRACTION

Addition: A + B ; A: Augend; B: Addend
Subtraction: A - B: A: Minuend; B: Subtrahend

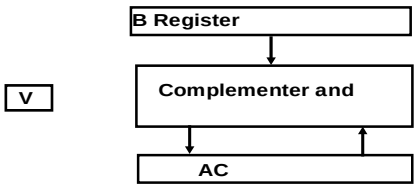
Operation	Add Magnitude	Subtract Magnitude		
		When A>B	When A<B	When A=B
(+A) + (+B)	+(A + B)			
(+A) + (- B)		+(A - B)	- (B - A)	+(A - B)
(- A) + (+B)		- (A - B)	+(B - A)	+(A - B)
(- A) + (- B)	- (A + B)			
(+A) - (+B)		+(A - B)	- (B - A)	+(A - B)
(+A) - (- B)	+(A + B)			
(- A) - (+B)	- (A + B)			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

Hardware Implementation

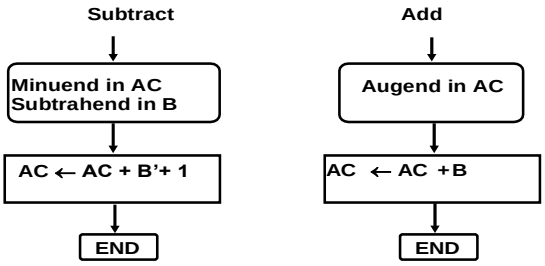


SIGNED 2'S COMPLEMENT ADDITION AND SUBTRACTION

Hardware



Algorithm



Algorithm:

The flowchart is shown in Figure 7.1. The two signs A, and B, are compared by an exclusive-OR gate.

If the output of the gate is 0 the signs are identical; If it is 1, the signs are different.

For an add operation, identical signs dictate that the magnitudes be added. For a subtract operation, different signs dictate that the magnitudes be added.

The magnitudes are added with a microoperation EA A + B, where EA is a register that combines E and A. The carry in E after the addition constitutes an overflow if it is equal to 1. The value of E is transferred into the add-overflow flip-flop AVF.

The two magnitudes are subtracted if the signs are different for an add operation or identical for a subtract operation. The magnitudes are subtracted by adding A to the 2's complemented B. No overflow can occur if the numbers are subtracted so AVF is cleared to 0.

1 in E indicates that A >= B and the number in A is the correct result. If this numbs is zero, the sign A must be made positive to avoid a negative zero.

0 in E indicates that A < B. For this case it is necessary to take the 2's complement of the value in A. The operation can be done with one microoperation A A' +1.

However, we assume that the A register has circuits for microoperations complement and increment, so the 2's complement is obtained from these two microoperations.

In other paths of the flowchart, the sign of the result is the same as the sign of A. so no change in A is required. However, when A < B, the sign of the result is the complement of the original sign of A. It is then necessary to complement A, to obtain the correct sign.

The final result is found in register A and its sign in As. The value in AVF provides an overflow indication. The final value of E is immaterial.

Figure 7.2 shows a block diagram of the hardware for implementing the addition and subtraction operations.

It consists of registers A and B and sign flip-flops As and Bs. Subtraction is done by adding A to the 2's complement of B.

The output carry is transferred to flip-flop E , where it can be checked to determine the relative magnitudes of two numbers.

The add-overflow flip-flop *AVF* holds the overflow bit when A and B are added.

The A register provides other microoperations that may be needed when we specify the sequence of steps in the algorithm.

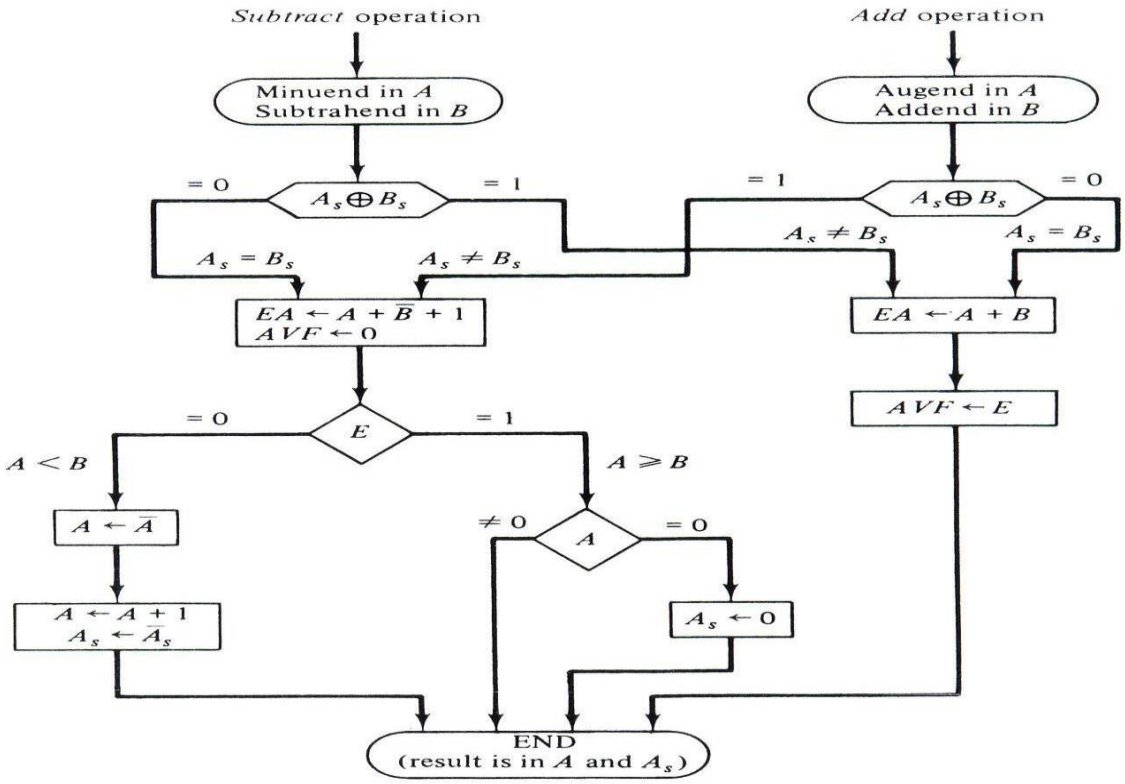
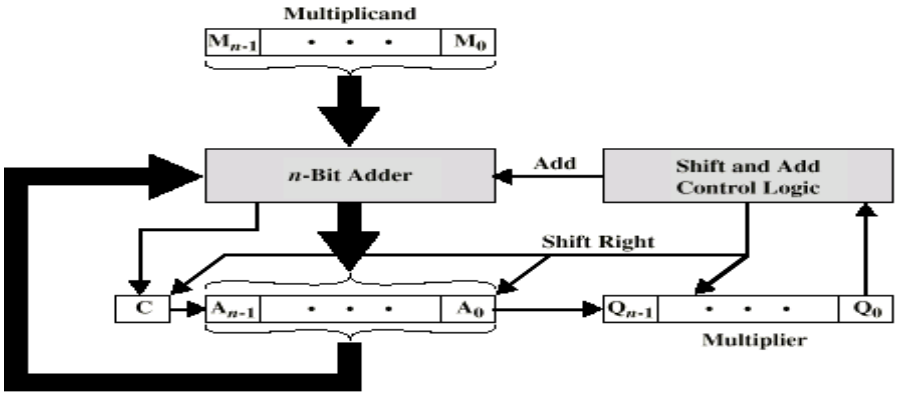


Figure 10-2 Flowchart for add and subtract operations.

Multiplication Algorithm:

In the beginning, the multiplicand is in B and the multiplier in Q. Their corresponding signs are in Bs and Qs respectively. We compare the signs of both A and Q and set to corresponding sign of the product since a double-length product will be stored in registers A and Q. Registers A and E are cleared and the sequence counter SC is set to the number of bits of the multiplier. Since an operand must be stored with its sign, one bit of the word will be occupied by the sign and the magnitude will consist of n-1 bits.

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



C	A	Q	M		
0	0000	1101	1011	Initial Values	
0	1011	1101	1011	Add	First Cycle
0	0101	1110	1011	Shift	
0	0010	1111	1011	Shift	Second Cycle
0	1101	1111	1011	Add	Third Cycle
0	0110	1111	1011	Shift	
1	0001	1111	1011	Add	Fourth Cycle
0	1000	1111	1011	Shift	

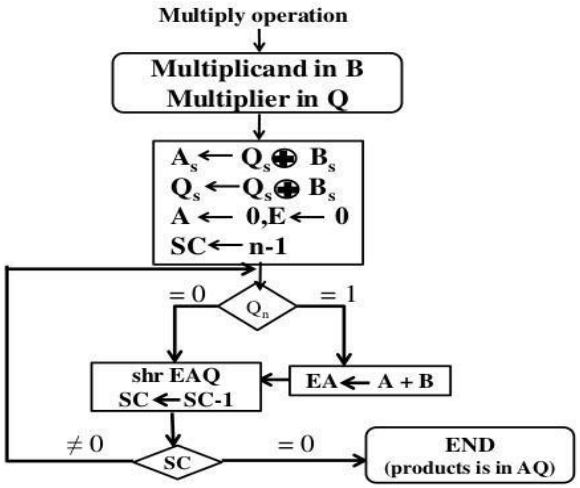


Figure: Flowchart for multiply operation.

Booth's algorithm :

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

It operates on the fact that strings of 0's in the multiplier require no addition but just

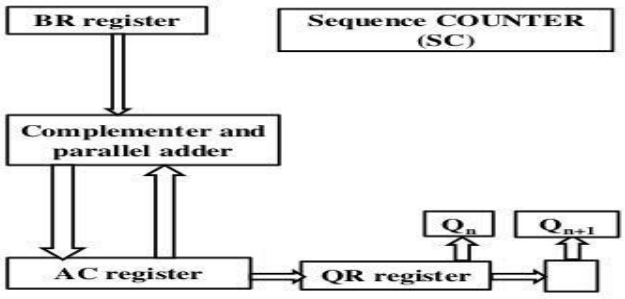
shifting, and a string of 1's in the multiplier from bit weight 2^k to weight 2^m can be treated as $2^{k+1} - 2^m$.

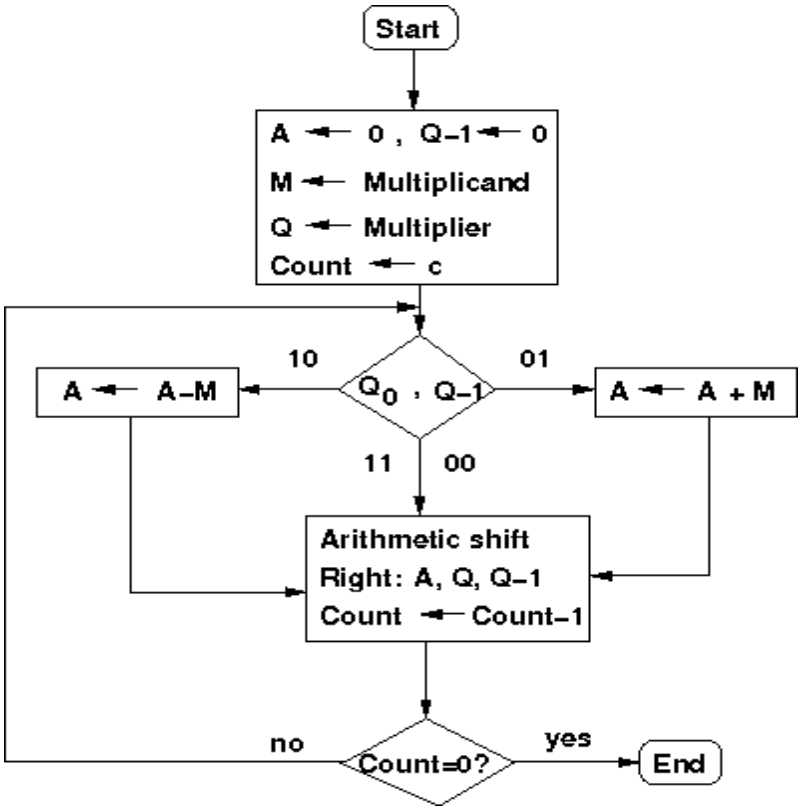
For example, the binary number 001110 (+14) has a string 1's from 2^3 to 2^1 (k=3, m=1). The number can be represented as $2^{k+1} - 2^m$. = $2^4 - 2^1 = 16 - 2 = 14$. Therefore, the multiplication M X 14, where M is the multiplicand and 14 the multiplier, can be done as M X 2^4 – M X 2^1.

Thus the product can be obtained by shifting the binary multiplicand M four times to the left and subtracting M shifted left once.

Hardware for Booth Algorithm

- Sign bits are not separated from the rest of the registers
- rename registers A,B, and Q as AC,BR and QR respectively
- Q_n designates the least significant bit of the multiplier in register QR
- Flip-flop Qn+1 is appended to QR to facilitate a double bit inspection of the multiplier





As in all multiplication schemes, booth algorithm requires examination of the multiplier bits and shifting of partial product.

Prior to the shifting, the multiplicand may be added to the partial product, subtracted from the partial, or left unchanged according to the following rules:

1. The multiplicand is subtracted from the partial product upon encountering the first least significant 1 in a string of 1's in the multiplier.

2. The multiplicand is added to the partial product upon encountering the first 0 in a string of 0's in the multiplier.

3. The partial product does not change when multiplier bit is identical to the previous multiplier bit.

- The algorithm works for positive or negative multipliers in 2's complement representation.

- This is because a negative multiplier ends with a string of 1's and the last operation will be a subtraction of the appropriate weight.

- The two bits of the multiplier in Qn and Qn+1 are inspected.

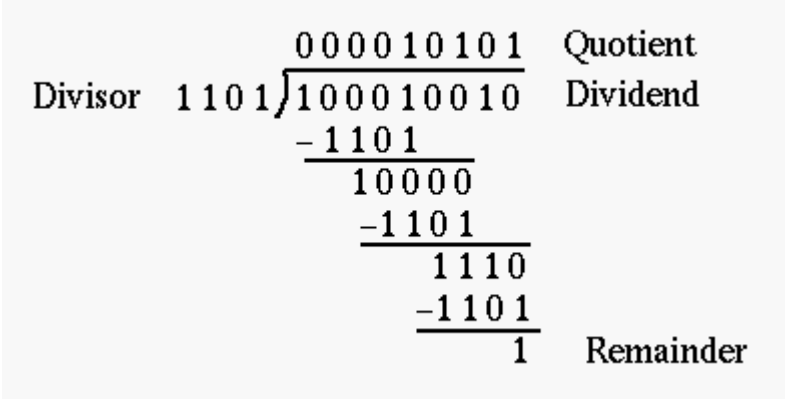
- If the two bits are equal to 10, it means that the first 1 in a string of 1 's has been encountered. This requires a subtraction of the multiplicand from the partial product in AC.

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

- When the two bits are equal, the partial product does not change.

Division Algorithms

Division of two fixed-point binary numbers in signed magnitude representation is performed with paper and pencil by a process of successive compare, shift and subtract operations. Binary division is much simpler than decimal division because here the quotient digits are either 0 or 1 and there is no need to estimate how many times the dividend or partial remainder fits into the divisor. The division process is described in Figure



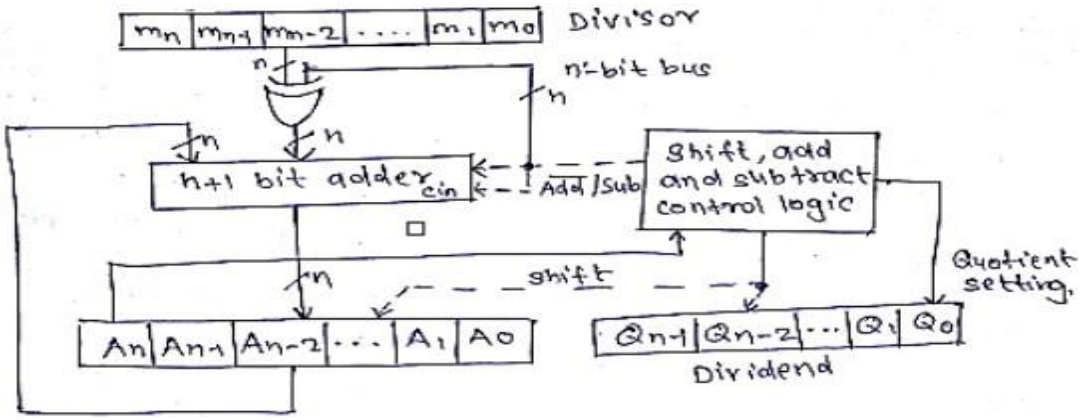
The devisor is compared with the five most significant bits of the dividend. Since the 5-bit number is smaller than B, we again repeat the same process. Now the 6-bit number is greater than B, so we place a 1 for the quotient bit in the sixth position above the dividend. Now we shift the divisor once to the right and subtract it from the dividend. The difference is known as a partial remainder because the division could have stopped here to obtain a quotient of 1 and a remainder equal to the partial

remainder. Comparing a partial remainder with the divisor continues the process. If the partial remainder is greater than or equal to the divisor, the quotient bit is equal to
1. The divisor is then shifted right and subtracted from the partial remainder. If the partial remainder is smaller than the divisor, the quotient bit is 0 and no subtraction is needed. The divisor is shifted once to the right in any case. Obviously the result gives both a quotient and a remainder.

Hardware Implementation for Signed-Magnitude Data

In hardware implementation for signed-magnitude data in a digital computer, it is convenient to change the process slightly. Instead of shifting the divisor to the right, two dividends, or partial remainders, are shifted to the left, thus leaving the two numbers in the required relative position. Subtraction is achieved by adding A to the 2's complement of B. End carry gives the information about the relative magnitudes.

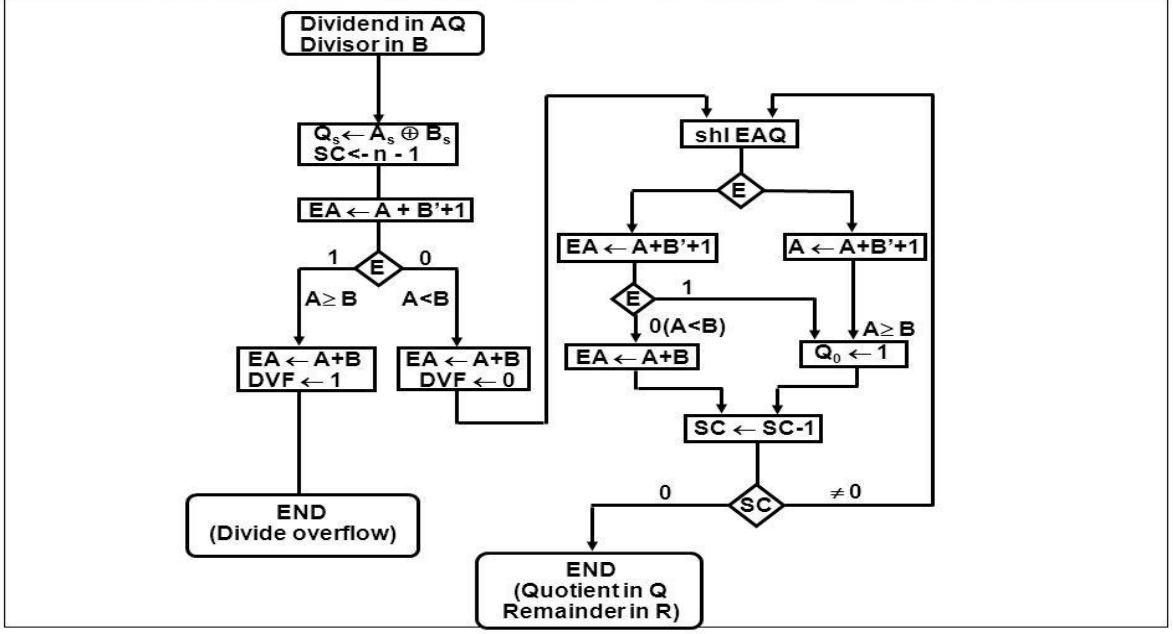
The hardware required is identical to that of multiplication. Register EAQ is now shifted to the left with 0 inserted into Qn and the previous value of E is lost. The example is given in Figure 4.10 to clear the proposed division process. The divisor is stored in the B register and the double-length dividend is stored in registers A and Q. The dividend is shifted to the left and the divisor is subtracted by adding its 2's complement value. E



Hardware Implementation for Signed-Magnitude Data

Algorithm:

FLOWCHART OF DIVIDE OPERATION



Example of Binary Division with Digital Hardware

Divisor $B = 10001$ $\quad \overline{B} + 1 = 01111$

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

Floating-point Arithmetic operations :

In many high-level programming languages we have a facility for specifying floating-point numbers. The most common way is by a real declaration statement. High level programming languages must have a provision for handling floating-point arithmetic operations. The operations are generally built in the internal hardware. If no hardware is available, the compiler must be designed with a package of floating-point software subroutine. Although the hardware method is more expensive, it is much more efficient than the software method. Therefore, floating- point hardware is included in most computers and is omitted only in very small ones.

Basic Considerations :

There are two part of a floating-point number in a computer - a mantissa m and an exponent e. The two parts represent a number generated from multiplying m times a radix r raised to the value of e. Thus

$m \times r^e$

The mantissa may be a fraction or an integer. The position of the radix point and the value of the radix r are not included in the registers. For example, assume a fraction representation and a radix
10. The decimal number 537.25 is represented in a register with m = 53725 and e = 3 and is interpreted to represent the floating-point number

$.53725 \times 10^3$

A floating-point number is said to be normalized if the most significant digit of the mantissa in nonzero. So the mantissa contains the maximum possible number of significant digits. We cannot normalize a zero because it does not have a nonzero digit. It is represented in floating-point by all 0's in the mantissa and exponent.

Floating-point representation increases the range of numbers for a given register. Consider a computer with 48-bit words. Since one bit must be reserved for the sign, the range of fixed-point integer numbers will be $+ (24^7 – 1)$, which is approximately $+ 101^4$. The 48 bits can be used to represent a floating-point number with 36 bits for the mantissa and 12 bits for the exponent. Assuming fraction representation for the mantissa and taking the two sign bits into consideration, the range of numbers that can be represented is

$+ (1 – 2^{-35}) \times 2^{2047}$

This number is derived from a fraction that contains 35 1's, an exponent of 11 bits (excluding its sign), and because $2^{11}–1 = 2047$. The largest number that can be accommodated is approximately 10^{615}. The mantissa that can accommodated is 35 bits (excluding the sign) and if considered as an integer it can store a number as large as $(2^{35} –1)$. This is approximately equal to 10^{10}, which is equivalent to a decimal number of 10 digits.

Computers with shorter word lengths use two or more words to represent a floating-point number. An 8-bit microcomputer uses four words to represent one floating-point number. One word of 8 bits are reserved for the exponent and the 24 bits of the other three words are used in the mantissa.

Arithmetic operations with floating-point numbers are more complicated than with fixed-point numbers. Their execution also takes longer time and requires more complex hardware. Adding or subtracting two numbers requires first an alignment of the radix point since the exponent parts must be made equal before adding or subtracting the mantissas. We do this alignment by shifting one mantissa while its exponent is adjusted until it becomes equal to the other exponent. Consider the sum of the following floating-point numbers:

$$.5372400 \times 10^{2}$$

$$+ .1580000 \times 10^{-1}$$

Floating-point multiplication and division need not do an alignment of the mantissas. Multiplying the two mantissas and adding the exponents can form the product. Dividing the mantissas and subtracting the exponents perform division.

The operations done with the mantissas are the same as in fixed-point numbers, so the two can share the same registers and circuits. The operations performed with the exponents are compared and incremented (for aligning the mantissas), added and subtracted (for multiplication) and division), and decremented (to normalize the result). We can represent the exponent in any one of the three representations - signed-magnitude, signed 2's complement or signed 1's complement.

Biased exponents have the advantage that they contain only positive numbers. Now it becomes simpler to compare their relative magnitude without bothering about their signs. Another advantage is that the smallest possible biased exponent contains all zeros. The floating-point representation of zero is then a zero mantissa and the smallest possible exponent.

Register Configuration

The register configuration for floating-point operations is shown in figure 4.13. As a rule, the same registers and adder used for fixed-point arithmetic are used for processing the mantissas. The difference lies in the way the exponents are handled.

The register organization for floating-point operations is shown in Fig. 4.13. Three registers are there, BR, AC, and QR. Each register is subdivided into two parts. The mantissa part has the same uppercase letter symbols as in fixed-point representation. The exponent part may use corresponding lower-case letter symbol.

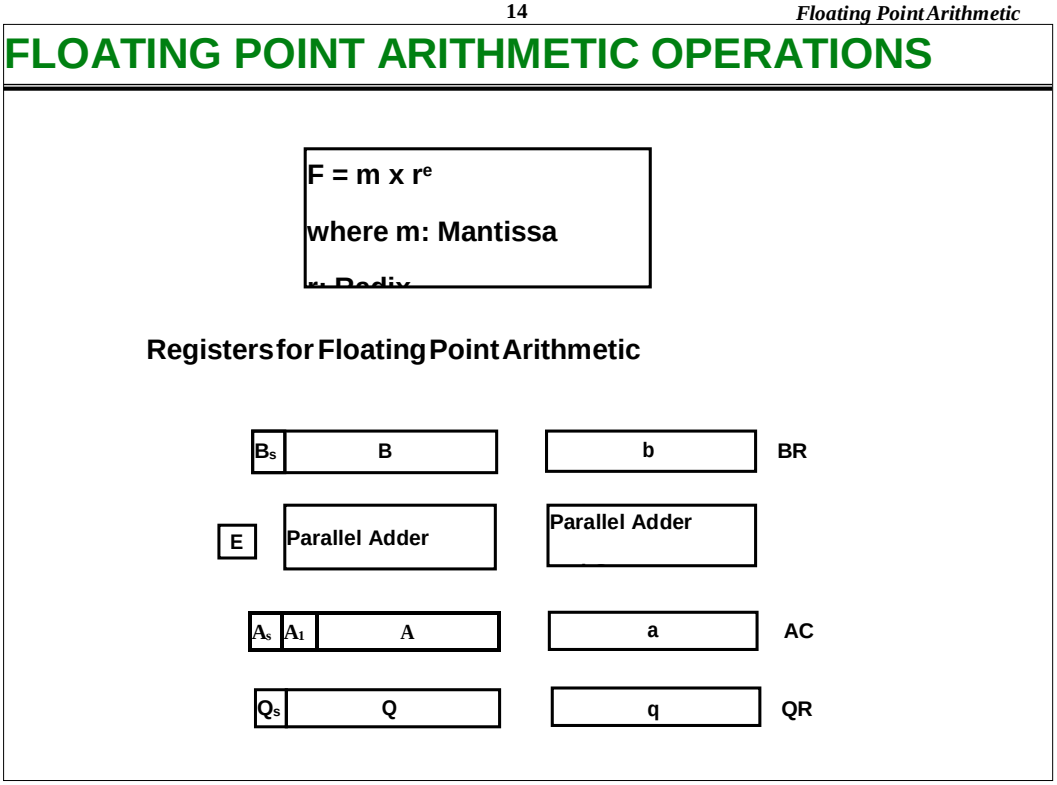


Figure 4.13: Registers for Floating Point arithmetic operations

Assuming that each floating-point number has a mantissa in signed-magnitude representation and a biased exponent. Thus the AC has a mantissa whose sign is in As, and a magnitude that is in A. The diagram shows the most significant bit of A, labeled by A1. The bit in his position must be a 1 to normalize the number. Note that the symbol AC represents the entire register, that is, the concatenation of As, A and a.

In the similar way, register BR is subdivided into Bs, B, and b and QR into Qs, Q and q. A parallel-adder adds the two mantissas and loads the sum into A and the carry into E. A separate parallel adder can be used for the exponents. The exponents do not have a district sign bit because they are biased but are represented as a biased positive quantity. It is assumed that the floating- point number are so large that the chance of an exponent overflow is very remote and so the exponent overflow will be neglected. The exponents are also connected to a magnitude comparator that provides three binary outputs to indicate their relative magnitude.

The number in the mantissa will be taken as a fraction, so they binary point is assumed to reside to the left of the magnitude part. Integer representation for floating point causes certain scaling problems during multiplication and division. To avoid these problems, we adopt a fraction representation.

The numbers in the registers should initially be normalized. After each arithmetic operation, the result will be normalized. Thus all floating-point operands are always normalized.

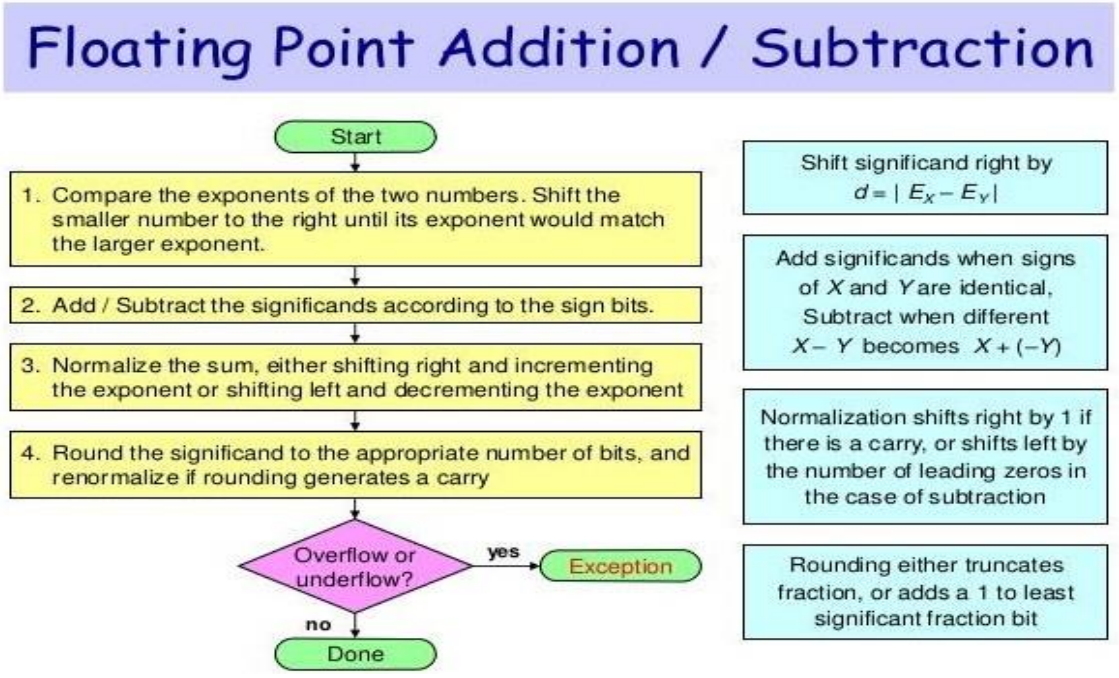
Addition and Subtraction of Floating Point Numbers

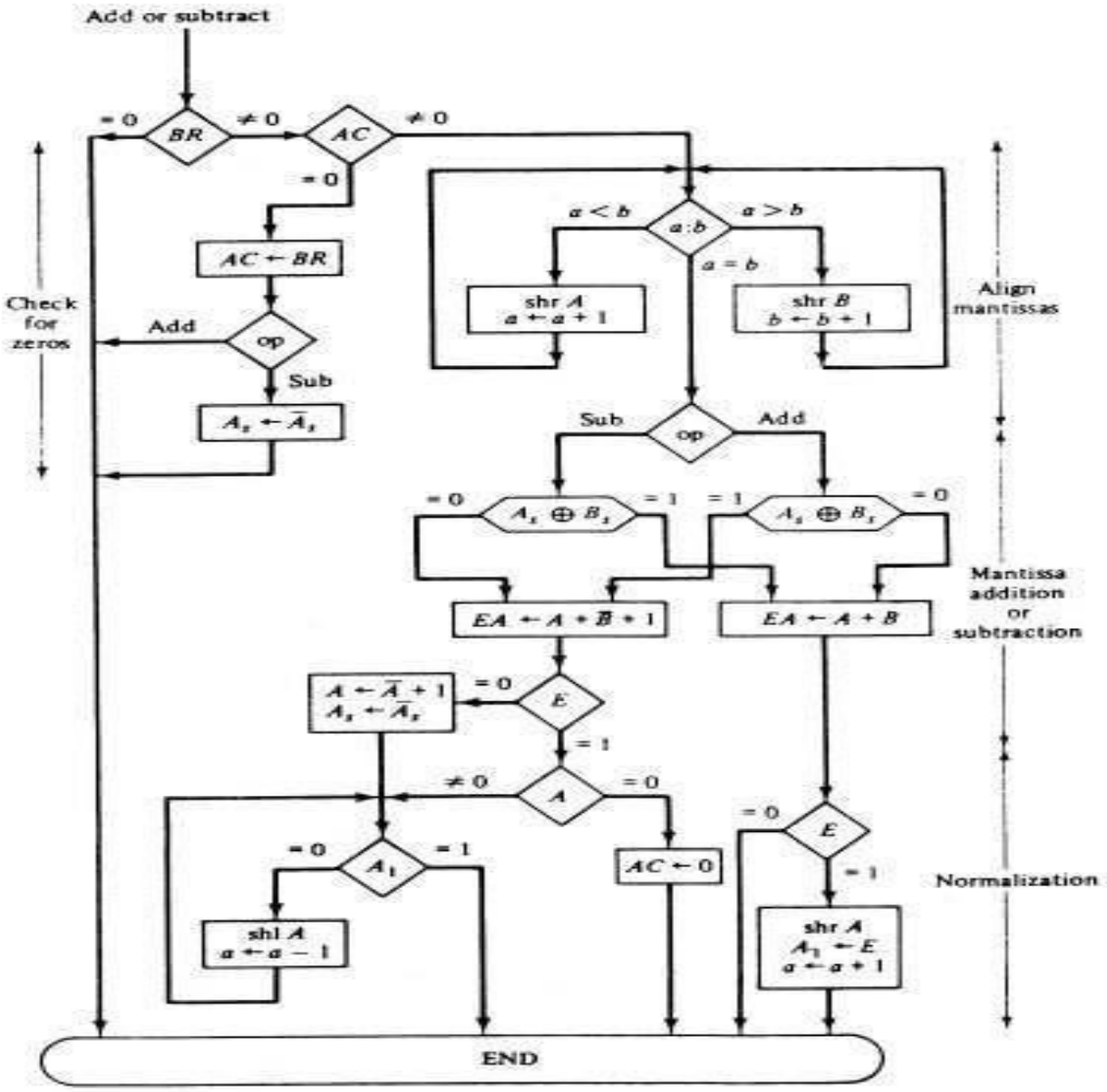
During addition or subtraction, the two floating-point operands are kept in AC and BR. The sum or difference is formed in the AC. The algorithm can be divided into four consecutive parts:

1. Check for zeros.
2. Align the mantissas.
3. Add or subtract the mantissas
4. Normalize the result

A floating-point number cannot be normalized, if it is 0. If this number is used for computation, the result may also be zero. Instead of checking for zeros during the normalization process we check for zeros at the beginning and terminate the process if necessary. The alignment of the mantissas must be carried out prior to their operation. After the mantissas are added or subtracted, the result may be un-normalized. The normalization procedure ensures that the result is normalized before it is transferred to memory.

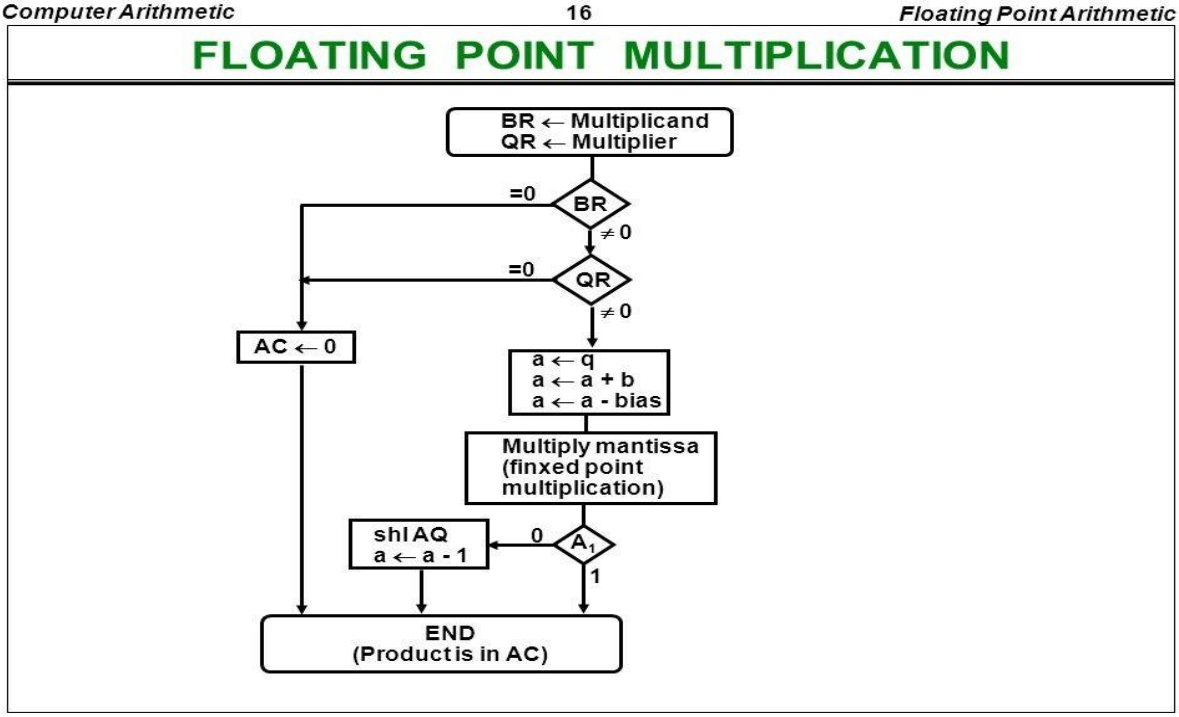
If the magnitudes were subtracted, there may be zero or may have an underflow in the result. If the mantissa is equal to zero the entire floating-point number in the AC is cleared to zero. Otherwise, the mantissa must have at least one bit that is equal to 1. The mantissa has an underflow if the most significant bit in position A1, is 0. In that case, the mantissa is shifted left and the exponent decremented. The bit in A1 is checked again and the process is repeated until A1 = 1. When A1 = 1, the mantissa is normalized and the operation is completed.

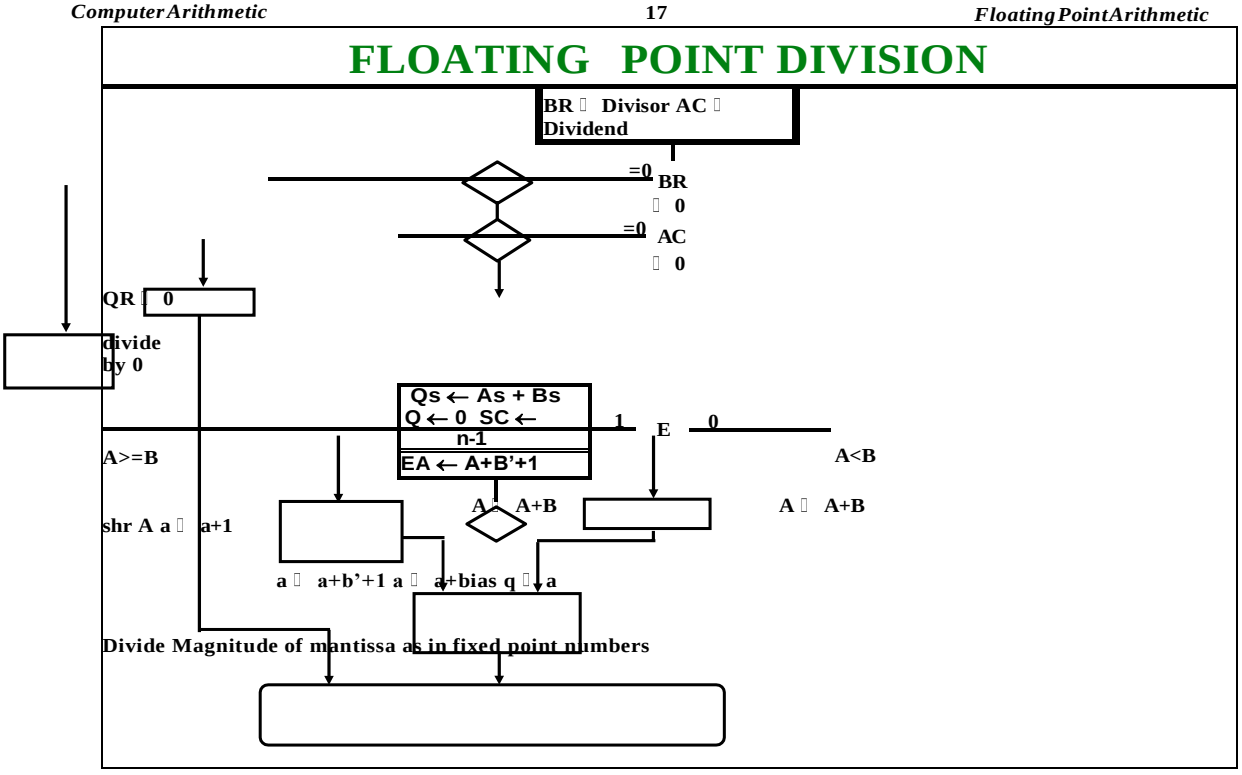




Algorithm for Floating Point Addition and Subtraction

Multiplication:





INPUT-OUTPUT ORGANIZATION

MODES OF DATA TRANSFER WITH I/O DEVICES

1. Programmed I/O
2. Interrupt Driven I/O
3. Direct Memory Access

Programmed I/O

The simplest strategy for handling communication between the CPU and an I/O module is programmed I/O. Using this strategy, the CPU is responsible for all communication with I/O modules, by executing instructions which control the attached devices, or transfer data.

For example, if the CPU wanted to send data to a device using programmed I/O, it would first issue an instruction to the appropriate I/O module to tell it to expect data. The CPU must then wait until the module responds before sending the data. If the module is slower than the CPU, then the CPU may also have to wait until the transfer is complete. This can be very inefficient.

Another problem exists if the CPU must read data from a device such as a keyboard. Every so often the CPU must issue an instruction to the appropriate I/O module to see if any keys have been pressed. This is also extremely inefficient. Consequently this strategy is only used in very small microprocessor controlled devices.

Interrupt Driven I/O

Virtually all computers provide a mechanism y which other modules (I/O, memory) may interrupt the normal processing of the CPU. Table 3.1 lists the most common classes of interrupts. The specific nature of these interrupts is examined later in this book, especially in chapters 6 and 11. However, we need to introduce the concept now in order to understand more clearly the nature of the instruction cycle and the implications of interrupts on the interconnection structure. The reader need not be concerned at this stage about the details of the generation and processing of interrupts, but only focus on the communication between modules those results from interrupts. Interrupts are provided primarily as a way to improve processing efficiency.

Interrupts are generated by:

- Generated by some condition that occurs as a result of an instruction execution, such as arithmetic overflow, division by zero, attempt to execute an illegal machine instruction, and reference outside a allowed memory space.
- Generated by a timer within the processor, This allows the operating system to perform certain functions on a regular basis.
- Generated by an I/O controller, to signal normal completion of an operation or to signal a variety of error conditions.

An I/O module interrupts the CPU simply by activating a control line in the control bus. The sequence of events is as follows.

1. The I/O module interrupts the CPU.

2. The CPU finishes executing the current instruction.
3. The CPU acknowledges the interrupt.
4. The CPU saves its current state.
5. The CPU jumps to a sequence of instructions which will handle the interrupt.

Interrupts and the Instruction Cycle

With interrupts, the processor can be engaged in executing other instructions while an I/O operation is in progress. Consider the flow of control in Figure 3.7b.As before , the user program reaches a point at which it makes a system call in the form of a WRITE call . The I/O program that is invoked in this case consists only of the preparation code and the actual I/O command. After these few instructions have been executed, control returns to the user program. Meanwhile, the external device is busy accepting data from computer memory and printing it. This I/O operation is conducted concurrently with the execution of instructions in the user program.

When the external device becomes ready to be serviced, that is , when it is ready to accept more data from the processor , the I/O module for that external device sends an interrupt request signal to the processor. The processor responds by suspending operation of the current program , branching off to a program to service that particular I/O device, known as an interrupt handler, and resuming the original execution after the device is serviced. The points at which such interrupts occur are indicated by an asterisk (*) in Figure.

Priority Interrupt

A priority interrupt establishes a priority to decide which condition is to be serviced first when two or more requests arrive simultaneously. The system may also determine which conditions are permitted to interrupt the computer while another interrupt is being serviced. Higher-priority interrupt levels are assigned to requests, which if delayed or interrupted, could have serious consequences. Devices with high-speed transfers are given high priority, and slow devices receive low priority. When two devices interrupt the computer at the same time, the computer services the device, with the higher priority first. Establishing the priority of simultaneous interrupts can be done by software or hardware. We can use a polling procedure to identify the highest-priority. There is one common branch address for all interrupts. The program that takes care of interrupts begins at the branch address and polls the interrupt sources in sequence. The order in which they are tested determines the priority of each interrupt. We test the highest-priority source first, and if its interrupt signal is on, control branches to a service routine for this source. Otherwise, the next-lower-priority source is tested, and so on. Thus the initial service routine interrupts consists of a program that tests the interrupt sources in sequence and branches to one of many possible service routines. The particular service routine reached belongs to the highest-priority device among all devices that interrupted the computer.

Interrupt Cycle

The interrupt makes flip-flop IEN so that can be set or cleared by program instructions. When IEN is cleared, the interrupt request coming from 1ST is neglected by the CPU. The program-controlled IEN bit allows the programmer to choose whether to use the interrupt facility. If an instruction to clear IEN has been inserted in the program, it means that the user does not want his program to be interrupted. An

instruction to set IEN indicates that the interrupt facility will be used while the current program is running. Most computers include internal hardware that clears IEN to 0 every time an interrupt is acknowledged by the processor.

CPU checks IEN and the interrupt signal from IST at the end of each instruction cycle the. If either 0, control continues with the next instruction. If both IEN and IST are equal to 1, the CPU goes to an interrupt cycle. During the interrupt cycle the CPU performs the following sequence of micro-operations:

SP□←SP-1		Decrement stack pointer
M [SP]	←PC	Push PC into stack
INTACK□←1		Enable interrupt
acknowledge PC□←VAD		Transfer vector address to PC
IEN□←0		Disable further interrupts
Go to fetch next instruction		

Direct Memory Access

Although interrupt driven I/O is much more efficient than program controlled I/O, all data is still transferred through the CPU. This will be inefficient if large quantities of data are being transferred between the peripheral and memory. The transfer will be slower than necessary, and the CPU will be unable to perform any other actions while it is taking place.

DMA Controller

Many systems therefore use an additional strategy, known as direct memory access (DMA). DMA uses an additional piece of hardware - a DMA controller. The DMA controller can take over the system bus and transfer data between an I/O module and main memory without the intervention of the CPU. Whenever the CPU wants to transfer data, it tells the DMA controller the direction of the transfer, the I/O module involved, the location of the data in memory, and the size of the block of data to be transferred. It can then continue with other instructions and the DMA controller will interrupt it when the transfer is complete.

The CPU and the DMA controller cannot use the system bus at the same time, so some way must be found to share the bus between them. One of two methods is normally used.

Burst mode

The DMA controller transfers blocks of data by halting the CPU and controlling the system bus for the duration of the transfer. The transfer will be as quick as the weakest link in the I/O module/bus/memory chain, as data does not pass through the CPU, but the CPU must still be halted while the transfer takes place.

Cycle stealing

The DMA controller transfers data one word at a time, by using the bus during a part of an instruction cycle when the CPU is not using it, or by pausing the CPU for a single clock cycle on each instruction. This may slow the CPU down slightly overall, but will still be very efficient.

Channel I/O

This is a system traditionally used on mainframe computers, but is becoming more common on smaller systems. It is an extension of the DMA concept, where the DMA controller becomes a full-scale computer system itself which handles all communication with the I/O modules.

I/O Interfaces

The interface of an I/O module is the connection to the peripheral(s) attached to it. The
interface handles synchronisation and control of the peripheral, and the actual transfer of data. For example, to send data to a peripheral, the sequence of events would be as follows.

a) The I/O module sends a control signal to the peripheral requesting permission to send data.

b) The peripheral acknowledges the request.

c) The I/O module sends the data (this may be either a word at a time or a block at a time depending on the peripheral).

d) The peripheral acknowledges receipt of the data.

This process of synchronisation is known as handshaking.

The internal buffer allows the I/O module to compensate for some of the difference in the speed at which the interface can communicate with the peripheral, and the speed of the system bus.

I/O interfaces can be divided into two main types.

I/O Function

This section introduces the concepts of input/output devices, modules and interfaces. It considers the various strategies used for communication between the CPU and I/O modules, and the interface between an I/O module and the device(s) connected to it. Some common I/O devices are considered in the last section.

DMA Transfer

There are three independent channels for DMA transfers. Each channel receives its trigger for the transfer through a large multiplexer that chooses from among a large number of signals. When these signals activate, the transfer occurs.

Input-output Processor (IOP)

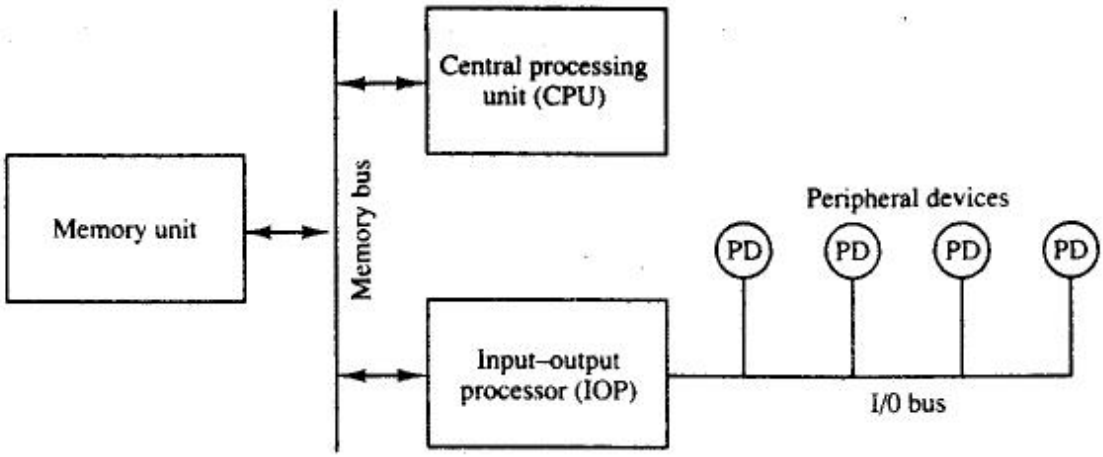
The CPU or processor is the part that makes the computer smart. It is a single integrated circuit referred to as a microprocessor. The earlier microprocessors were Intel 8080 or 8086, they were very slow. Then came faster models from Intel such as 80286, 80386, 80486 and now Pentium processors. Each of these vary in speed of their operation. The AT compatibles - 80286 onwards, run in one of the two modes:

- Real mode
- Protected mode

The processor complex is the name of the circuit board that contains the main system

processor and any other circuitry directly related to it, such as clock control, cache, and so forth. The processor complex design allows the user to easily upgrade the system later to a new processor type by changing one card. In effect, it amounts to a modular motherboard with a replaceable processor section.

The block diagram of a computer with two processors is shown in Figure 6.39. The memory unit occupies a central position and can communicate with each processor by means of direct memory access. The CPU is responsible for processing data needed in the solution of computational tasks. The IOP provides a path for transfer of data between various peripheral devices and the memory unit.



CPU-IOP Communication

There are many form of the communication between CPU and IOP. These are depending on the particular computer considered. In most cases the memory unit acts as a message center where each processor leaves information for the other. To appreciate the operation of a typical IOP, we will illustrate by a specific example the method by which the CPU and IOP communicate. This is a simplified example that omits many operating details in order to provide an overview of basic concepts.

The sequence of operations may be carried out as shown in the flowchart of Fig. 6.40. The CPU sends an instruction to test the IOP path. The IOP responds by inserting a status word in memory for the CPU to check. The bits of the status word indicate the condition of the IOP and I/O device, such as IOP overload condition, device busy with another transfer, or device ready for I/O transfer. The CPU refers to the status word in memory to decide what to do next. If all is in order, the CPU sends the instruction to start I/O transfer. The memory address received with this instruction tells the IOP where to find its program.

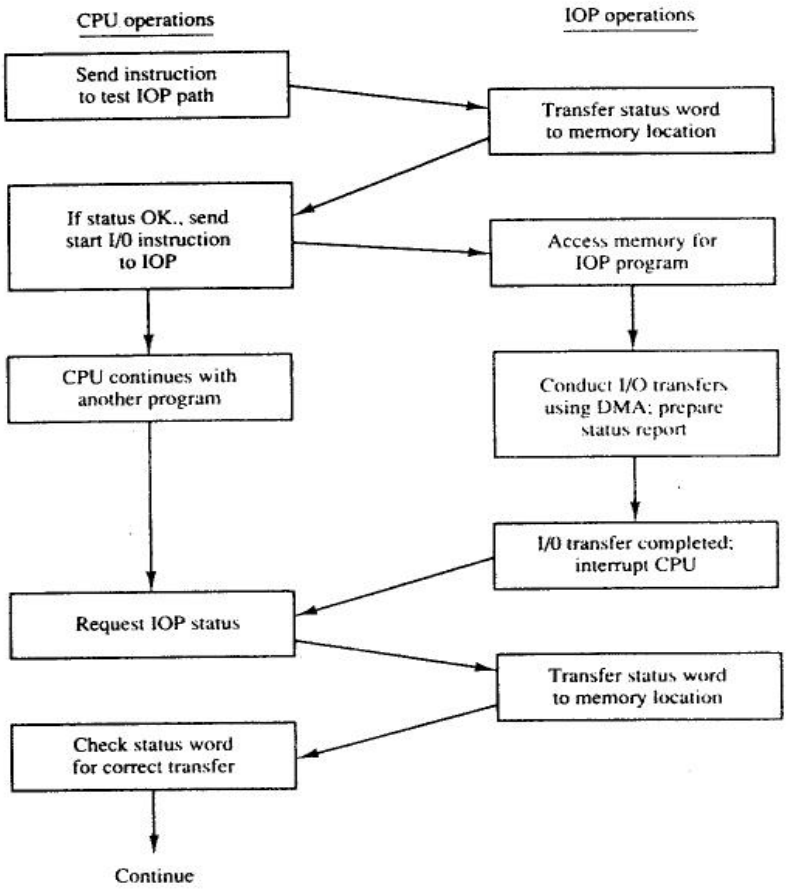


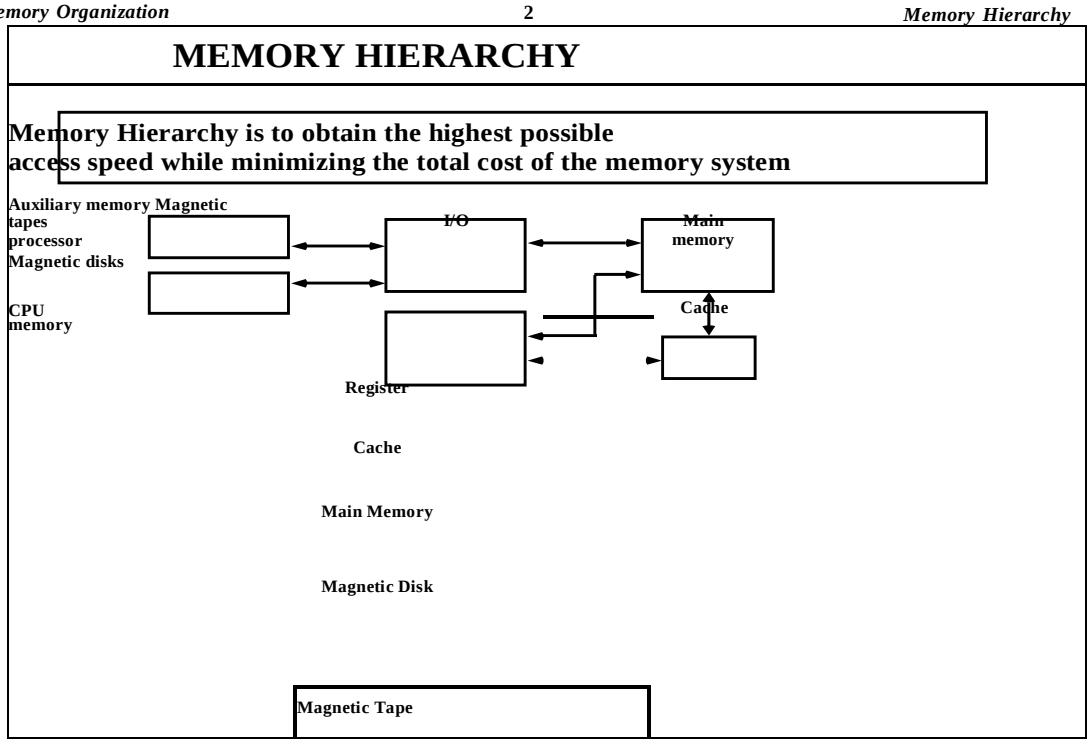
Figure: CPU IOP Communication

Unit 6

MEMORY ORGANIZATION:

Memory Hierarchy, Main Memory, Auxiliary memory, Associative Memory, Cache Memory, Virtual Memory

Memory Hierarchy :



memory address map of RAM and ROM.

Main Memory

- The main memory is the central storage unit in a computer system.

- Primary memory holds only those data and instructions on which computer is currently working.

It has limited capacity and data is lost when power is switched off. It is generally made up of semiconductor device.
These memories are not as fast as registers.
The data and instruction required to be processed reside in main memory. It is divided into two subcategories RAM and ROM.

Memory address map of RAM and ROM

- The designer of a computer system must calculate the amount of memory required for the particular application and assign it to either RAM or ROM.

- The interconnection between memory and processor is then established from knowledge of the size of memory needed and the type of RAM and ROM chips available.

- The addressing of memory can be established by means of a table that specifies the memory address assigned to each chip.

- The table, called a **memory address map**, is a pictorial representation of assigned address space for each chip in the system, shown in table 9.1.

 To demonstrate with a particular example, assume that a computer system needs 512 bytes of RAM and 512 bytes of ROM.

 The RAM and ROM chips to be used are specified in figure 9.1 and figure 9.2.

Memory address map of RAM and ROM

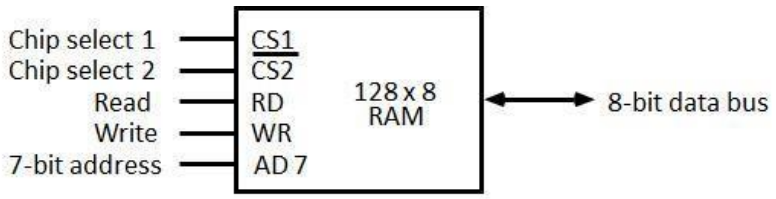


Figure 9.1: Typical RAM chip

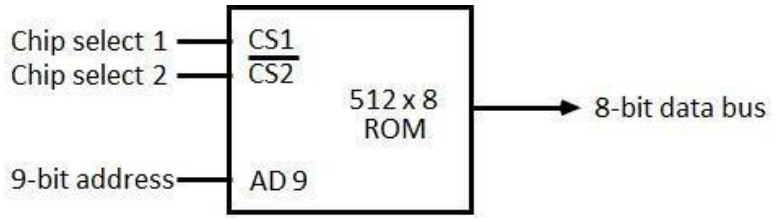


Figure 9.2: Typical ROM chip

Component	Hexa 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

 The component column specifies whether a RAM or a ROM chip is used.

 The hexadecimal address column assigns a range of hexadecimal equivalent addresses for each chip.

 The address bus lines are listed in the third column.

 Although there are 16 lines in the address bus, the table shows only 10 lines because the other 6 are not used in this example and are assumed to be zero.

 The small x's under the address bus lines designate those lines that must be connected to the address inputs in each chip.

 The RAM chips have 128 bytes and need seven address lines. The ROM chip has 512 bytes and needs 9 address lines.

 The x's are always assigned to the low-order bus lines: lines 1 through 7 for the RAM and lines 1 through 9 for the ROM.

 It is now necessary to distinguish between four RAM chips by assigning to each a different address. For this particular example we choose bus lines 8 and 9 to represent four distinct binary combinations.

 The table clearly shows that the nine low-order bus lines constitute a memory space for RAM equal to $2^9 =$ 512 bytes.

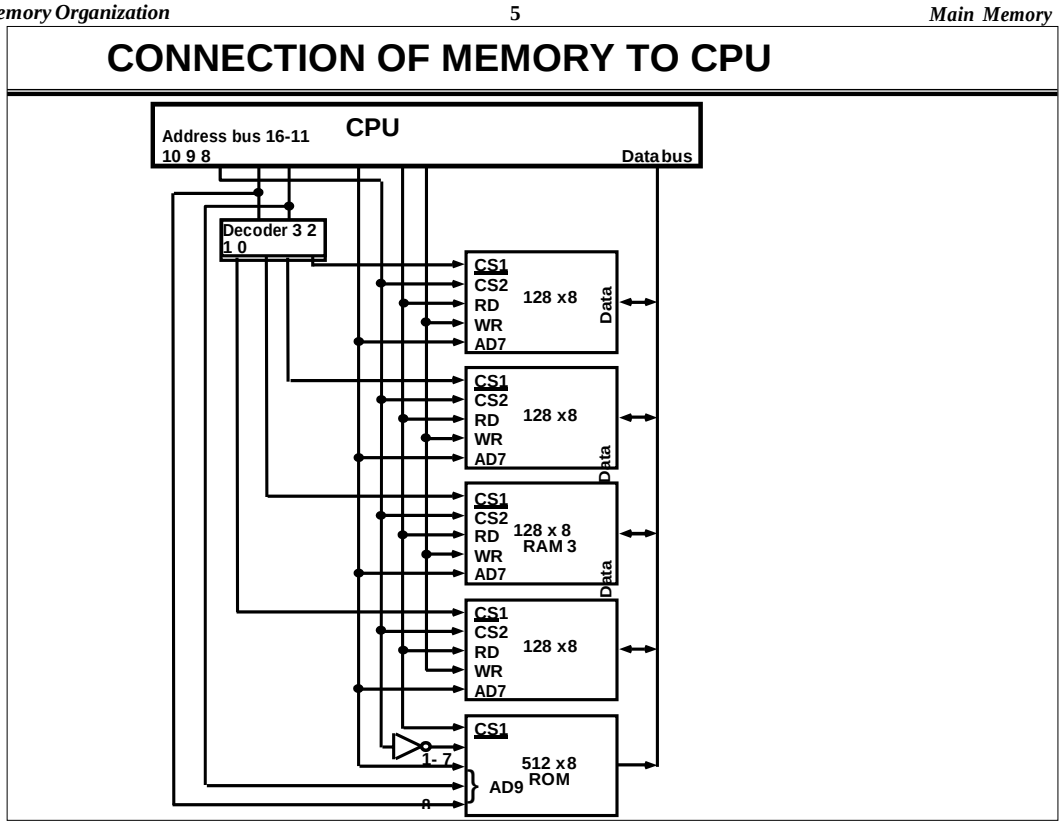
 The distinction between a RAM and ROM address is done with another bus line. Here we choose line 10 for this purpose.

When line 10 is 0, the CPU selects a RAM, and when this line is equal to 1, it selects the ROM

Memory connections to CPU :

- RAM and ROM chips are connected to a CPU through the data and address buses

- The low-order lines in the address bus select the byte within the chips and other lines in the address bus select a particular chip through its chip select inputs.



Auxiliary Memory :

- **Magnetic Tape:** Magnetic tapes are used for large computers like mainframe computers where large volume of data is stored for a longer time. In PC also you can use tapes in the form of cassettes. The cost of storing data in tapes is inexpensive. Tapes consist of magnetic materials that store data permanently. It can be 12.5 mm to 25 mm wide plastic film-type and 500 meter to 1200 meter long which is coated with magnetic material. The deck is connected to the central processor and information is fed into or read from the tape through the processor. It's similar to cassette tape recorder.

Magnetic tape is an information storage medium consisting of a magnetisable coating on a thin plastic strip. Nearly all recording tape is of this type, whether used for video with a video cassette recorder, audio storage (reel-to-reel tape, compact audio cassette, digital audio tape (DAT), digital linear tape (DLT) and other formats including 8-track cartridges) or general purpose digital data storage using a computer (specialized tape formats, as well as the above-mentioned compact audio cassette, used with home computers of the 1980s, and DAT, used for backup in workstation installations of the 1990s).

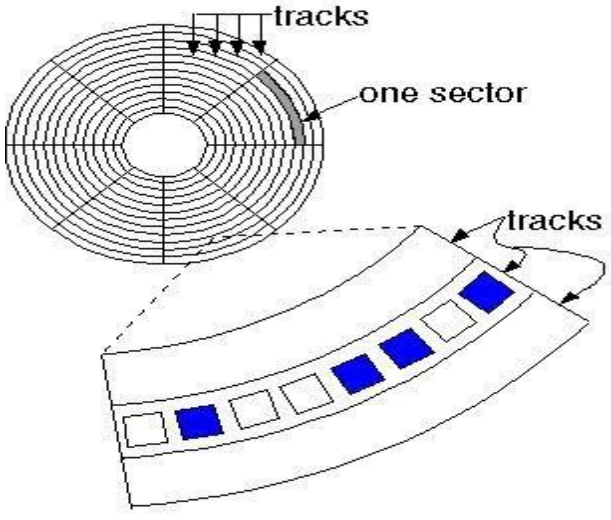
- Magneto-optical and optical tape storage products have been developed using many of the same concepts as magnetic storage, but have achieved little commercial success.

- **Magnetic Disk:** You might have seen the gramophone record, which is circular like a disk and coated with magnetic material. Magnetic disks used in computer are made on the same principle. It rotates with very high speed inside the computer drive. Data is stored on both the surface of the disk. Magnetic disks are most popular for direct access storage device. Each disk consists of a number of invisible concentric circles called tracks. Information is recorded on tracks of a disk surface in the form of tiny magnetic spots. The presence of a magnetic spot represents one bit and its absence represents zero bit. The information stored in a disk can be read many times without affecting the stored data. So the reading operation is non-destructive. But if you want to write a new data, then the existing data is erased from the disk and new data is recorded. For Example-Floppy Disk.

The primary computer storage device. Like tape, it is magnetically recorded and can be re-recorded over and over. Disks are rotating platters with a mechanical arm that moves a read/write head between the outer and inner edges of the platter's surface. It can take as long as one second to find a location on a floppy disk to as little as a couple of milliseconds on a fast hard disk. See hard disk for more details.

The disk surface is divided into concentric tracks (circles within circles). The thinner the tracks, the more storage. The data bits are recorded as tiny magnetic spots on the tracks. The smaller the spot, the more bits per inch and the greater the storage.

Sectors
Tracks are further divided into sectors, which hold a block of data that is read or written at one time; for example, READ SECTOR 782, WRITE SECTOR 5448. In order to update the disk, one or more sectors are read into the computer, changed and written back to disk. The operating system figures out how to fit data into these fixed spaces. Modern disks have more sectors in the outer tracks than the inner ones because the outer radius of the platter is greater than the inner radius



Block diagram of Magnetic Disk

Optical Disk: With every new application and software there is greater demand for memory capacity. It is the necessity to store large volume of data that has led to the development of optical disk storage medium. Optical disks can be divided into the following categories:

1. **Compact Disk/ Read Only Memory (CD-ROM**
2. **Write Once, Read Many (WORM)**
3. **Erasable Optical Disk**

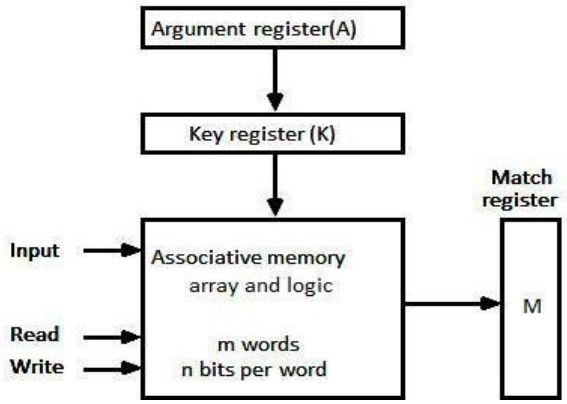
Associative Memory :Content Addressable Memory (CAM).

 The time required to find an item stored in memory can be reduced considerably if stored data can be identified for access by the content of the data itself rather than by an address.

 A memory unit accessed by content is called an associative memory or content addressable memory (CAM).

 This type of memory is accessed simultaneously and in parallel on the basis of data content rather than by specific address or location.

 The block diagram of an associative memory is shown in figure 9.3.



It consists of a memory array and logic form words with n bits per word.
The argument register A and key register K each have n bits, one for each bit of a word. The match register M has m bits, one for each memory word.
Each word in memory is compared in parallel with the content of the argument register.

 The words that match the bits of the argument register set a corresponding bit in the match register.

 After the matching process, those bits in the match register that have been set indicate the fact that their corresponding words have been matched.

 Reading is accomplished by a sequential access to memory for those words whose corresponding bits in the match register have been set.

Hardware Organization

 The key register provides a mask for choosing a particular field or key in the argument word. The entire

argument is compared with each memory word if the key register contains all 1's.

 Otherwise, only those bits in the argument that have 1st in their corresponding position of the key register are compared.

 Thus the key provides a mask or identifying piece of information which specifies how the reference to memory is made.

 To illustrate with a numerical example, suppose that the argument register A and the key register K have the bit configuration shown below.

 Only the three leftmost bits of A are compared with memory words because K has 1's in these position.

A	101 111100	
K	111 000000	
Word1	100 111100	no match
Word2	101 000001	match

 Word 2 matches the unmasked argument field because the three leftmost bits of the argument and the word are equal.

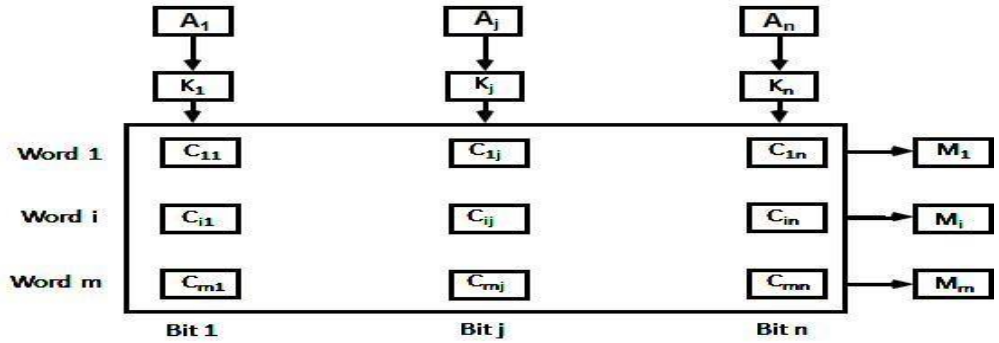


Figure 9.4: Associative memory of *m* word, n cells per word.

 The relation between the memory array and external registers in an associative memory is shown in figure 9.4.

 The cells in the array are marked by the letter C with two subscripts.

 The first subscript gives the word number and the second specifies the bit position in the word. Thus cell Cij is the cell for bit j in words i.

 A bit Aj in the argument register is compared with all the bits in column j of the array provided that Kj =1.

 This is done for all columns j = 1, 2... n.

 If a match occurs between all the unmasked bits of the argument and the bits in word i, the corresponding bit Mi in the match register is set to 1.

 If one or more unmasked bits of the argument and the word do not match, Mi is cleared to 0.

Cache Memory :

Cache is a fast small capacity memory that should hold those information which are most likely to be accessed.

 The basic operation of the cache is, when the CPU needs to access memory, the cache is examined.

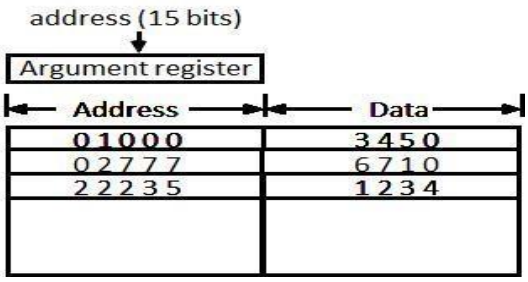
 If the word is found in the cache, it is read from the fast memory. If the word addressed by the CPU is not found in the cache, the main memory is accessed to read the word.

 The transformation of data from main memory to cache memory is referred to as a **mapping process**.

Associative mapping

Consider the main memory can store 32K words of 12 bits each.
The cache is capable of storing 512 of these words at any given time.
For every word stored in cache, there is a duplicate copy in main memory. The CPU communicates with both memories.
It first sends a 15-bit address to cache. If there is a hit, the CPU accepts the 12-bit data from cache, if there is miss, the CPU reads the word from main memory and the word is then transferred to cache.

Figure 9.5: Associative mapping cache (all numbers in octal)



 The associative memory stores both the address and content (data) of the memory word.

 This permits any location in cache to store any word from main memory.

 The figure 9.5 shows three words presently stored in the cache. The address value of 15 bits is shown as a five-digit octal number and its corresponding 12-bit word is shown as a four-digit octal number.

 A CPU address of 15 bits is placed in the argument register and the associative memory is searched for a matching address.

If the address is found the corresponding 12-bit data is read and sent to CPU. If no match occurs, the main memory is accessed for the word.
The address data pairs then transferred to the associative cache memory.

 If the cache is full, an address data pair must be displaced to make room for a pair that is needed and not presently in the cache.

 This constitutes a first-in first-one (FIFO) replacement policy.

direct mapping in organization of cache memory:

- The CPU address of 15 bits is divided into two fields.

- The nine least significant bits constitute the index field and the remaining six bits from the tag field.

The figure 9.6 shows that main memory needs an address that includes both the tag and the index.

Figure 9.6: Addressing relationships between main and cache memories

- The number of bits in the index field is equal to the number of address bits required to access the cache memory.
- The internal

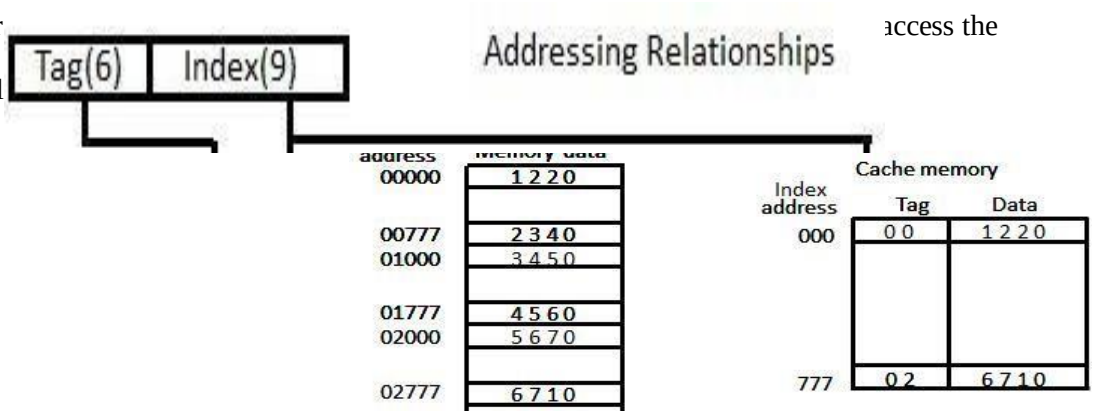


Figure 9.7: Direct mapping cache organization

- Each word in cache consists of the data word and its associated tag.

- When a new word is first brought into the cache, the tag bits are stored alongside the data bits.

- When the CPU generates a memory request the index field is used for the address to access the cache.

- The tag field of the CPU address is compared with the tag in the word read from the cache.

If the two tags match, there is a hit and the desired data word is in cache.
If there is no match, there is a miss and the required word is read from main memory. It is then stored in the cache together with the new tag, replacing the previous value.

- The word at address zero is presently stored in the cache (index = 000, tag = 00, data = 1220). Suppose that the CPU now wants to access the word at address 02000.

The index address is 000, so it is used to access the cache. The two tags are then compared. The

cache tag is 00 but the address tag is 02, which does not produce a match.

Therefore, the main memory is accessed and the data word 5670 is transferred to the CPU. The

cache word at index address 000 is then replaced with a tag of 02 and data of 5670.

The **disadvantage** of direct mapping is that two words with the same index in their address but with different tag values cannot reside in cache memory at the same time.

The comparison logic is done by an associative search of the tags in the set similar to an associative memory search: thus the name "set-associative".

When a miss occurs in a set-associative cache and the set is full, it is necessary to replace one of the tag-data items with a new value.

The most common replacement algorithms used are: random replacement, first-in first-out (FIFO), and least recently used (LRU).

Write-through and Write-back cache write method.
Write Through

- The simplest and most commonly used procedure is to update main memory with every memory write operation.

- The cache memory being updated in parallel if it contains the word at the specified address. This is called the *write-through* method.

- This method has the advantage that main memory always contains the same data as the cache.

This characteristic is important in systems with direct memory access transfers.

- It ensures that the data residing in main memory are valid at all times so that an I/O device communicating through DMA would receive the most recent updated data.

Write-Back (Copy-Back)

The second procedure is called the write-back method.
In this method only the cache location is updated during a write operation.

- The location is then marked by a flag so that later when the word is removed from the cache it is copied into main memory.

- The reason for the write-back method is that during the time a word resides in the cache, it may be updated several times.

- However, as long as the word remains in the cache, it does not matter whether the copy in main memory is out of date, since requests from the word are filled from the cache.

- It is only when the word is displaced from the cache that an accurate copy need be rewritten into main memory.

Virtual Memory

- Virtual memory is used to give programmers the illusion that they have a very large memory at their disposal, even though the computer actually has a relatively small main memory.

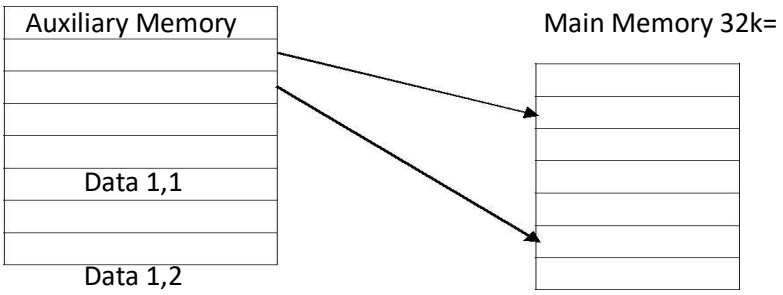
- A virtual memory system provides a mechanism for translating program-generated addresses into correct main memory locations.

Address space

An address used by a programmer will be called a virtual address, and the set of such addresses is known as address space.

Memory space

An address in main memory is called a location or physical address. The set of such locations is called the memory space.



Program 1

Program 2

Data 2,1

Address space 1024k=2^{10}

 As an illustration, consider a computer with a main-memory capacity of 32K words (K = 1024). Fifteen bits are needed to specify a physical address in memory since 32K = 2^{15}.

 Suppose that the computer has available auxiliary memory for storing 2^{20} = 1024K words.

 Thus auxiliary memory has a capacity for storing information equivalent to the capacity of 32 main memories.

 Denoting the address space by N and the memory space by M, we then have for this example N = 1024K and M = 32K.

 In a multiprogramming computer system, programs and data are transferred to and from auxiliary memory and main memory based on demands imposed by the CPU.

 Suppose that program 1 is currently being executed in the CPU. Program 1 and a portion of its associated data are moved from auxiliary memory into main memory as shown in figure 9.9.

 Portions of programs and data need not be in contiguous locations in memory since information is being moved in and out, and empty spaces may be available in scattered locations in memory.

 In our example, the address field of an instruction code will consist of 20 bits but physical memory addresses must be specified with only 15 bits.

 Thus CPU will reference instructions and data with a 20-bit address, but the information at this address must be taken from physical memory because access to auxiliary storage for individual words will be too long.

Address mapping using pages.

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

 The physical memory is broken down into groups of equal size called blocks, which may range from 64 to 4096 words each.

The term page refers to groups of address space of the same size.
Consider a computer with an address space of 8K and a memory space of 4K.

 If we split each into groups of 1K words we obtain eight pages and four blocks as shown in figure 9.9

 At any given time, up to four pages of address space may reside in main memory in any one of the four blocks.

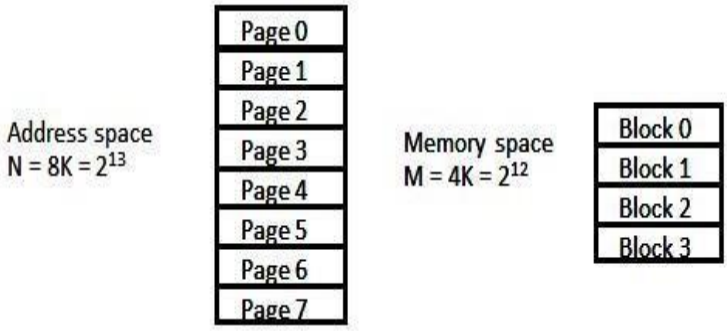


Figure 9.10 Address and Memory space split into group of 1K words

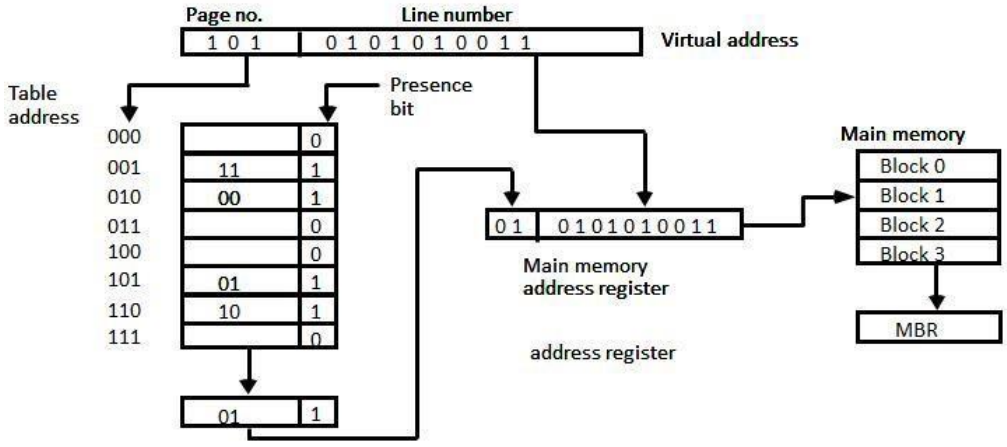


Figure 9.11: Memory table in paged system

- The organization of the memory mapping table in a paged system is shown in figure 9.10.
- The memory-page table consists of eight words, one for each page.
- The address in the page table denotes the page number and the content of the word give the block number where that page is stored in main memory.
- The table shows that pages 1, 2, 5, and 6 are now available in main memory in blocks 3, 0, 1, and 2, respectively.
- A presence bit in each location indicates whether the page has been transferred from auxiliary memory into main memory.
- A 0 in the presence bit indicates that this page is not available in main memory.

The CPU references a word in memory with a virtual address of 13 bits.

The three high-order bits of the virtual address specify a page number and also an address for the memory-page table.

The content of the word in the memory page table at the page number address is read out into the memory table buffer register.

If the presence bit is a 1, the block number thus read is transferred to the two high- order bits of the main memory address register.

The line number from the virtual address is transferred into the 10 low-order bits of the memory address register.

A read signal to main memory transfers the content of the word to the main memory buffer register ready to be used by the CPU.

If the presence bit in the word read from the page table is 0, it signifies that the content of the word referenced by the virtual address does not reside in main memory.

Segment

A segment is a set of logically related instructions or data elements associated with a given name.

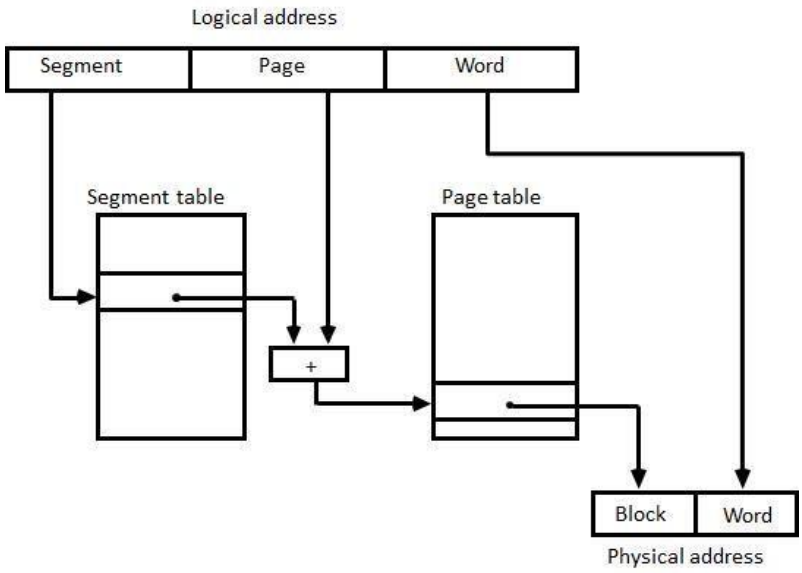
Logical address

The address generated by segmented program is called a logical address.

Segmented page mapping

 The length of each segment is allowed to grow and contract according to the needs of the program being executed. Consider logical address shown in figure 9.12.

Figure 9.12: Logical to physical address mapping



 The logical address is partitioned into three fields.

 The segment field specifies a segment number.

 The page field specifies the page within the segment and word field gives specific word within the page.

A page field of k bits can specify up to 2k pages.
A segment number may be associated with just one page or with as many as 2k pages.

 Thus the length of a segment would vary according to the number of pages that are assigned to it.

 The mapping of the logical address into a physical address is done by means of two tables, as shown in figure 9.12.

 The segment number of the logical address specifies the address for the segment table.
The entry in the segment table is a pointer address for a page table base.
The page table base is added to the page number given in the logical address. The sum produces

a pointer address to an entry in the page table.

- The concatenation of the block field with the word field produces the final physical mapped address.

- The two mapping tables may be stored in two separate small memories or in main memory.

- In either case, memory reference from the CPU will require three accesses to memory: one from the segment table, one from the page table and the third from main memory.

- This would slow the system significantly when compared to a conventional system that requires only one reference to memory.

Multiprocessors

Characteristics of multiprocessors

- A multiprocessor system is an interconnection of two or more CPUs with memory and input-output equipment.
- The term "processor" in multiprocessor can mean either a central processing unit (CPU) or an input-output processor (IOP).
- Multiprocessors are classified as *multiple instruction stream, multiple data stream* (MIMD) systems
- The similarity and distinction between multiprocessor and multicomputer are
  - Similarity
    - Both support concurrent operations
  - Distinction
    - The network consists of several autonomous computers that may or may not communicate with each other.
    - A multiprocessor system is controlled by one operating system that provides interaction between processors and all the components of the system cooperate in the solution of a problem.
- Multiprocessing improves the reliability of the system.
- The benefit derived from a multiprocessor organization is an improved system performance.
  - Multiple independent jobs can be made to operate in parallel.
  - A single job can be partitioned into multiple parallel tasks.
- Multiprocessing can improve performance by decomposing a program into parallel executable tasks.
  - The user can explicitly declare that certain tasks of the program be executed in parallel.
    - This must be done prior to loading the program by specifying the parallel executable segments.
  - The other is to provide a compiler with multiprocessor software that can automatically detect parallelism in a user's program.
- Multiprocessor are classified by the way their memory is organized.
  - A multiprocessor system with *common shared memory* is classified as a *shared-memory* or *tightly coupled multiprocessor*.
    - Tolerate a *higher degree* of interaction between tasks.
  - Each processor element with its own *private local memory* is classified as a *distributed-memory* or *loosely coupled system.*
    - Are most efficient when the interaction between tasks is *minimal*

Interconnection Structures

- The components that form a multiprocessor system are CPUs, IOPs connected to input-output devices, and a memory unit.
- The interconnection between the components can have different physical configurations, depending on the number of transfer paths that are available
    - Between the processors and memory in a shared memory system
    - Among the processing elements in a loosely coupled system
- There are several physical forms available for establishing an interconnection network.
    - Time-shared common bus
    - Multiport memory
    - Crossbar switch
    - Multistage switching network
    - Hypercube system

Time Shared Common Bus

- A common-bus multiprocessor system consists of a number of processors connected through a common path to a memory unit.
- *Disadv.*:
    - Only one processor can communicate with the memory or another processor at any given time.
    - As a consequence, the total overall transfer rate within the system is limited by the speed of the single path
- A more economical implementation of a dual bus structure is depicted in Fig. below.
- Part of the local memory may be designed as a *cache memory* attached to the CPU.

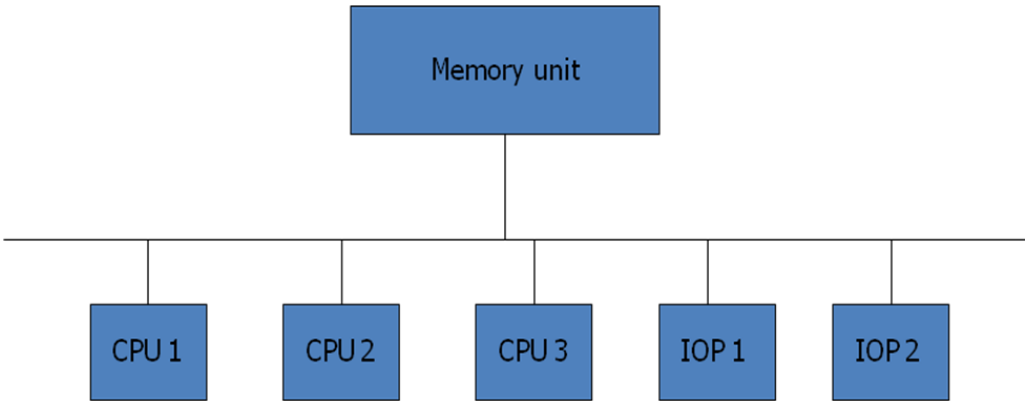


Fig: Time shared common bus organization

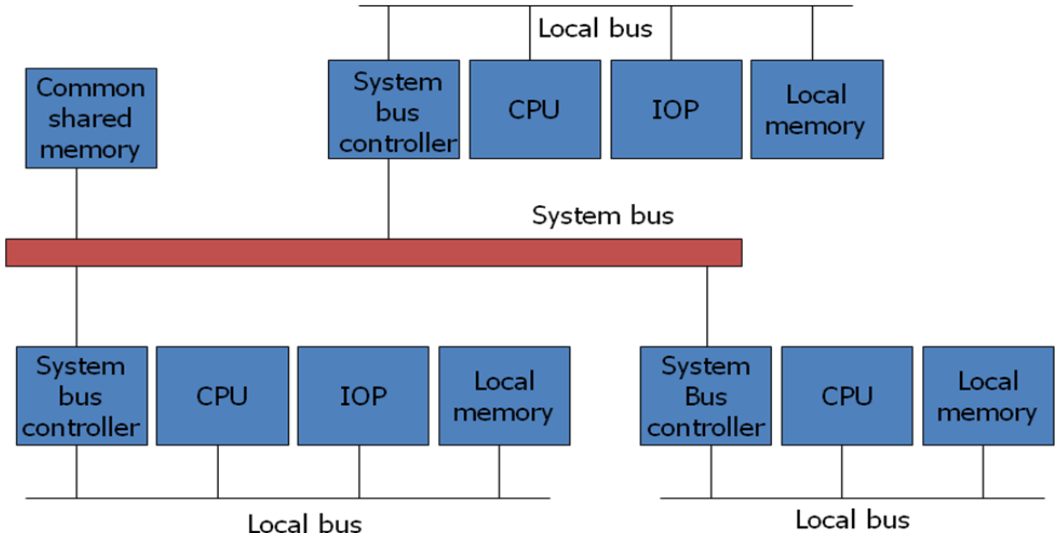


Fig: System bus structure for multiprocessorsa

Multiport Memory

- A multiport memory system employs separate buses between each memory module and each CPU.
- The module must have internal control logic to determine which port will have access to memory at any given time.
- Memory access conflicts are resolved by assigning fixed priorities to each memory port.
- *Adv.*:
  - The high transfer rate can be achieved because of the multiple paths.
- *Disadv.*:
  - It requires expensive memory control logic and a large number of cables and connections

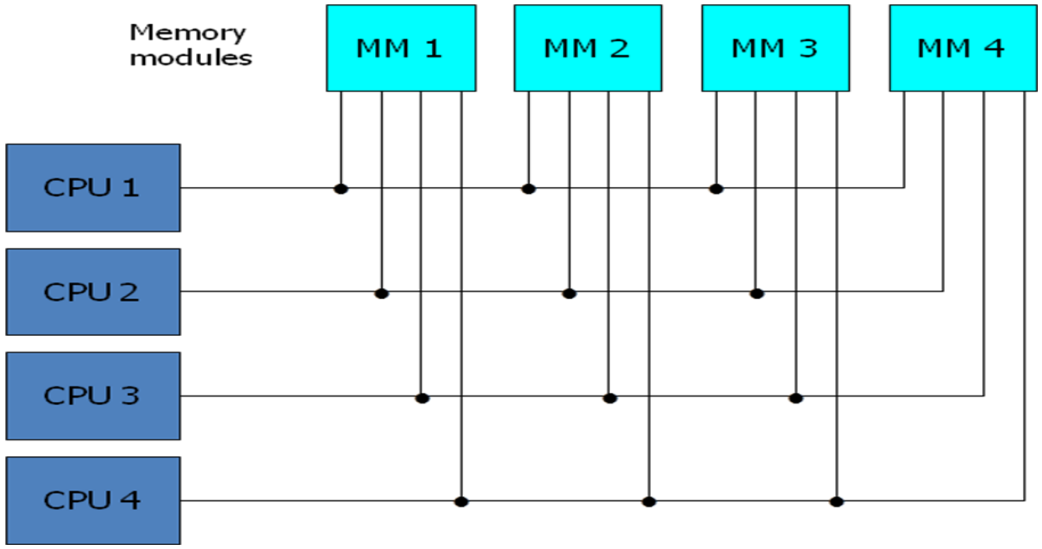


Fig: Multiport memory organization

Crossbar Switch

- Consists of a number of *crosspoints* that are placed at intersections between processor buses and memory module paths.
- The small square in each crosspoint is a *switch* that determines the path from a processor to a memory module.
- Adv.:
  - Supports simultaneous transfers from all memory modules
- Disadv.:
  - The hardware required to implement the switch can become quite large and complex.
- Below fig. shows the functional design of a crossbar switch connected to one memory module.

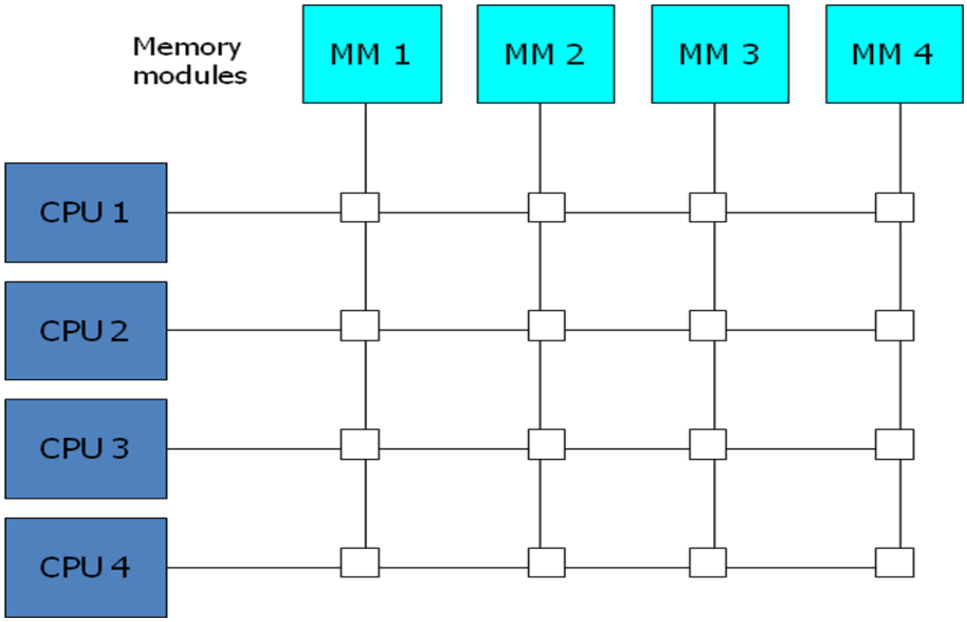


Fig: Crossbar switch

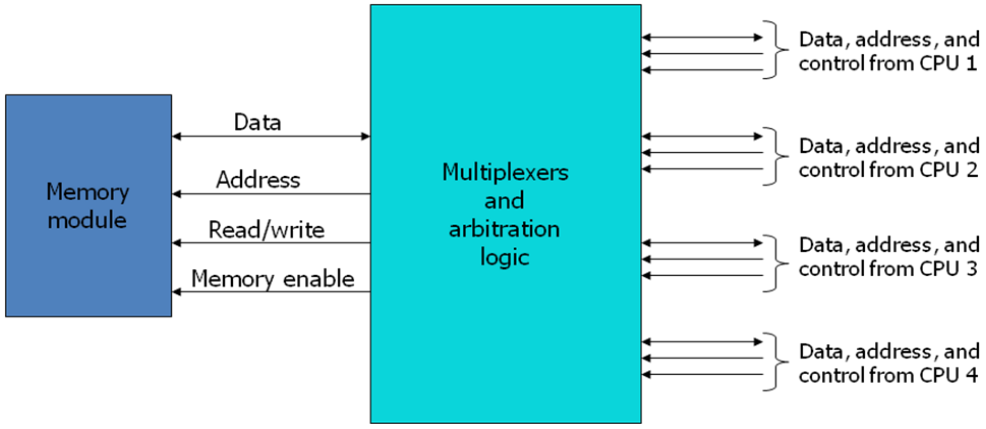
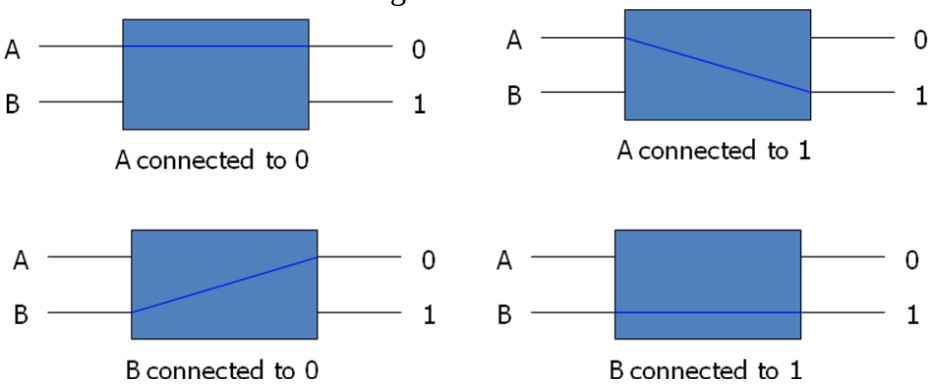


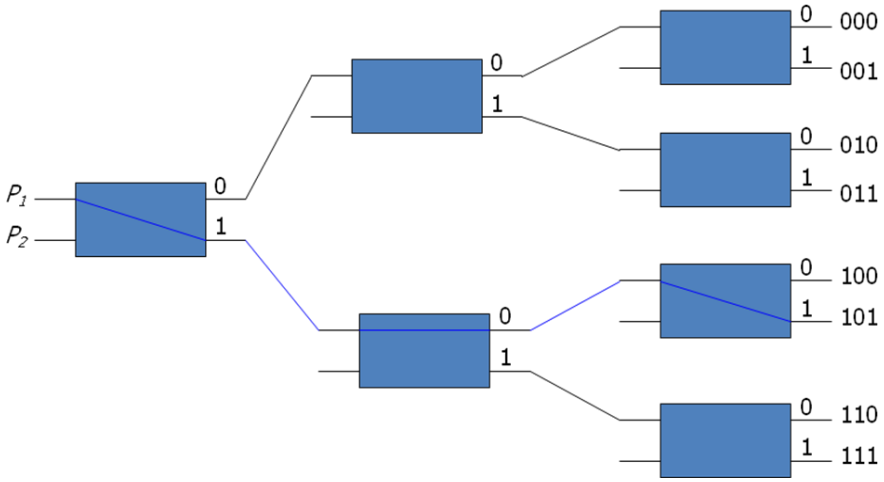
Fig: Block diagram of crossbar switch

Multistage Switching Network

- The basic component of a multistage network is a two-input, two-output interchange switch as shown in Fig. below.



- Using the 2x2 switch as a building block, it is possible to build a multistage network to control the communication between a number of sources and destinations.
  - To see how this is done, consider the binary tree shown in Fig. below.
  - Certain request patterns cannot be satisfied simultaneously. i.e., if P_1 □ 000~011, then P_2 □ 100~111



- One such topology is the omega switching network shown in Fig. below

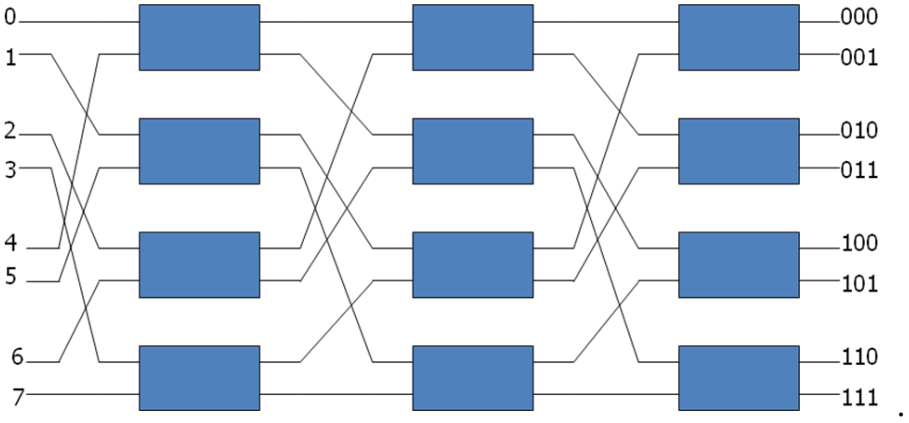


Fig: 8 x 8 Omega Switching Network

- Some request patterns cannot be connected simultaneously. i.e., any two sources cannot be connected simultaneously to destination 000 and 001
- In a tightly coupled multiprocessor system, the source is a processor and the destination is a memory module.
- Set up the path transfer the address into memory transfer the data
- In a loosely coupled multiprocessor system, both the source and destination are processing elements.

Hypercube System

- The hypercube or binary n-cube multiprocessor structure is a loosely coupled system composed of $N=2^n$ processors interconnected in an n-dimensional binary cube.
    - Each processor forms a node of the cube, in effect it contains not only a CPU but also local memory and I/O interface.
    - Each processor address differs from that of each of its n neighbors by exactly one bit position.
- Fig. below shows the hypercube structure for n=1, 2, and 3.
- Routing messages through an *n*-cube structure may take from one to *n* links from a source node to a destination node.
    - A routing procedure can be developed by computing the exclusive-OR of the source node address with the destination node address.
    - The message is then sent along any one of the axes that the resulting binary value will have 1 bits corresponding to the axes on which the two nodes differ.
- A representative of the hypercube architecture is the Intel iPSC computer complex.
    - It consists of 128(*n*=7) microcomputers, each node consists of a CPU, a floating-point processor, local memory, and serial communication interface units.

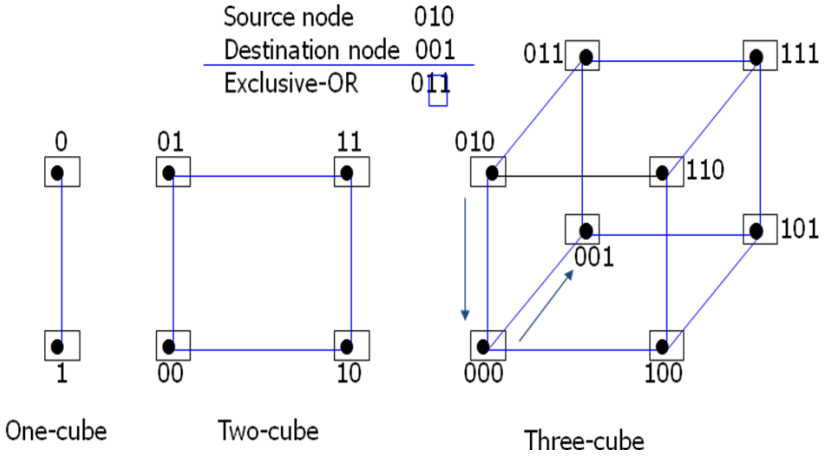


Fig: Hypercube structures for n=1,2,3

Inter processor Communication and Synchronization

- The various processors in a multiprocessor system must be provided with a facility for *communicating* with each other.
    - A communication path can be established through *a portion of memory* or *a common input-output channels*.
- The sending processor structures a request, a message, or a procedure, and places it in the memory mailbox.
    - *Status bits* residing in common memory
    - The receiving processor can check the mailbox *periodically*.
    - The response time of this procedure can be time consuming.
- A more efficient procedure is for the sending processor to alert the receiving processor directly by means of an *interrupt signal*.
- In addition to shared memory, a multiprocessor system may have other shared resources. e.g., a magnetic disk storage unit.
- To prevent conflicting use of shared resources by several processors there must be a provision for assigning resources to processors. i.e., operating system.
- There are three organizations that have been used in the design of operating system for multiprocessors: *master-slave configuration*, *separate operating system*, and *distributed operating system*.
- In a master-slave mode, one processor, master, always executes the operating system functions.
- In the separate operating system organization, each processor can execute the operating system routines it needs. This organization is more suitable for *loosely coupled systems*.
- In the distributed operating system organization, the operating system routines are distributed among the available processors. However, each particular operating system function is assigned to only one processor at a time. It is also referred to as a *floating operating system*.

Loosely Coupled System

- There is *no shared memory* for passing information.
- The communication between processors is by means of message passing through *I/O channels*.
- The communication is initiated by one processor calling a *procedure* that resides in the memory of the processor with which it wishes to communicate.
- The communication efficiency of the interprocessor network depends on the *communication routing protocol, processor speed, data link speed, and the topology of the network.*

Interprocess Synchronization

- The instruction set of a multiprocessor contains basic instructions that are used to implement communication and synchronization between cooperating processes.
    - Communication refers to the exchange of data between different processes.
    - Synchronization refers to the special case where the data used to communicate between processors is control information.

- Synchronization is needed to enforce the *correct sequence of processes* and to ensure

mutually exclusive access to shared writable data.

- Multiprocessor systems usually include various mechanisms to deal with the synchronization of resources.
    - Low-level primitives are implemented directly by the hardware.
    - These primitives are the basic mechanisms that enforce mutual exclusion for more complex mechanisms implemented in software.
    - A number of hardware mechanisms for mutual exclusion have been developed.
        - A binary semaphore

Mutual Exclusion with Semaphore

- A properly functioning multiprocessor system must provide a mechanism that will guarantee orderly access to shared memory and other shared resources.
    - Mutual exclusion: This is necessary to protect data from being changed simultaneously by two or more processors.
    - Critical section: is a program sequence that must complete execution before another processor accesses the same shared resource.
- A *binary variable* called a *semaphore* is often used to indicate whether or not a processor is executing a critical section.
- Testing and setting the semaphore is itself a critical operation and must be performed as a single indivisible operation.
- A semaphore can be initialized by means of a *test and set instruction* in conjunction with a hardware *lock* mechanism.
- The instruction TSL SEM will be executed in two memory cycles (the first to read and the second to write) as follows: R □ M[*SEM*], M[*SEM*] □ 1
- Note that the lock signal must be active during the execution of the test-and-set instruction.