13. Subject notes/PPTs/self study material

UNIT 1:

Digital Computers: Introduction, Block diagram of Digital Computer, Definition of Computer Organization, Computer Design and Computer Architecture.
Register Transfer Language and Micro operations: Register Transfer language, Register Transfer, Bus and memory transfers, Arithmetic Micro operations, logic micro operations, shift micro operations, Arithmetic logic 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.

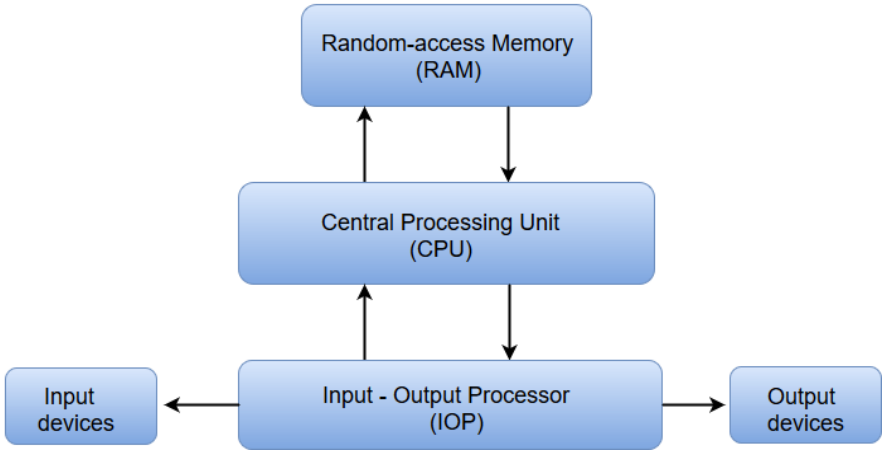
DIGITAL COMPUTERS

A Digital computer can be considered as a digital system that performs various computational tasks.

The first electronic digital computer was developed in the late 1940s and was used primarily for numerical computations. By convention, the digital computers use the binary number system, which has two digits: 0 and 1. A binary digit is called a bit. A computer system is subdivided into two functional entities: Hardware and Software.

The hardware consists of all the electronic components and electromechanical devices that comprise the physical entity of the device. The software of the computer consists of the instructions and data that the computer manipulates to perform various data-processing tasks.

Block diagram of a digital computer:



- The Central Processing Unit (CPU) contains an arithmetic and logic unit for manipulating data, a number of registers for storing data, and a control circuit for fetching and executing instructions.

- The memory unit of a digital computer contains storage for instructions and data.
- The Random Access Memory (RAM) for real-time processing of the data.
- The Input-Output devices for generating inputs from the user and displaying the final results to the user.
- The Input-Output devices connected to the computer include the keyboard, mouse, terminals, magnetic disk drives, and other communication devices.

BASIC COMPUTER ORGANIZATION:

Most of the computer systems found in automobiles and consumer appliances to personal computers and main frames have some basic organization. The basic computer organization has three main components:

- CPU
- Memory subsystem
- I/O subsystem.

The generic organization of these components is shown in the figure below.

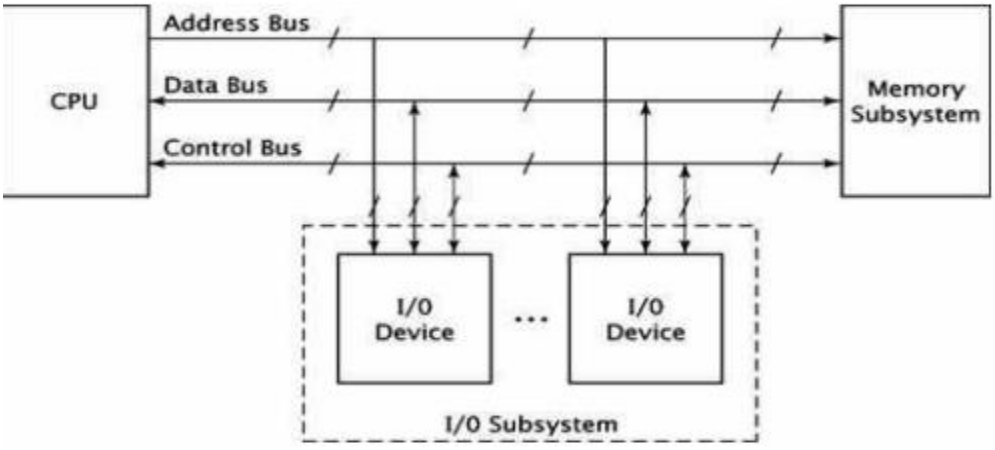


Fig 1.1 Generic computer Organization

Computer organization: Computer organization is the knowing, what the functional components of a computer are, how they work and how their performance is measured and optimized. Computer

Organization refers to the level of abstraction above the digital logic level, but below the operating system level.

Computer design and architecture:

Computer design is concerned with the determination of what hardware should be used and how the parts should be connected. This aspect of computer hardware is sometimes referred to as computer implementation. Computer architecture is concerned with the structure and behavior of the computer as seen by the user.

Register Transfer Language and Micro Operations:

Register Transfer language:

 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 micro operations

 A micro operation 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 micro operations

 The internal hardware organization of a digital computer is best defined by specifying

o The set of registers it contains and their functions

o The sequence of micro operations performed on the binary information stored

o The control that initiates the sequence of micro operations

 Use symbols, rather than words, to specify the sequence of micro operations

 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 micro operations and describe associated hardware that can implement the micro operations

Register Transfer

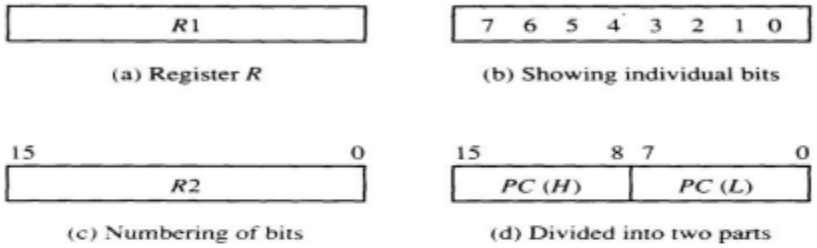
 Designate computer registers by capital letters to denote its function

 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

Figure 4-1 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)

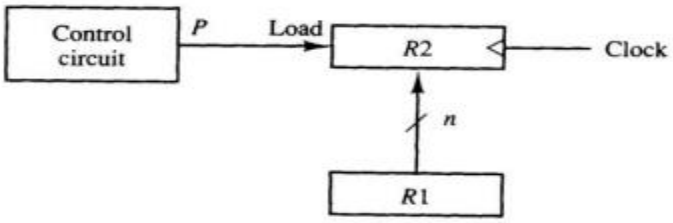
  or,

  P: R2 R1,

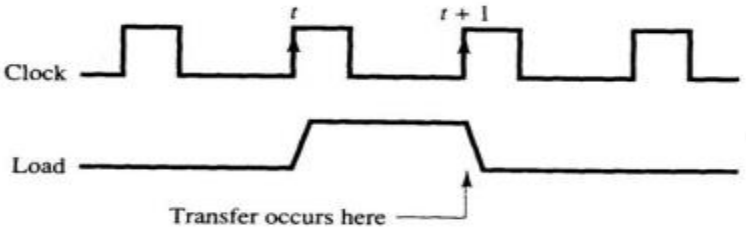
where P is a control function that can be either 0 or 1

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

Figure 4-2 Transfer from $R1$ to $R2$ when $P = 1$.



(a) Block diagram



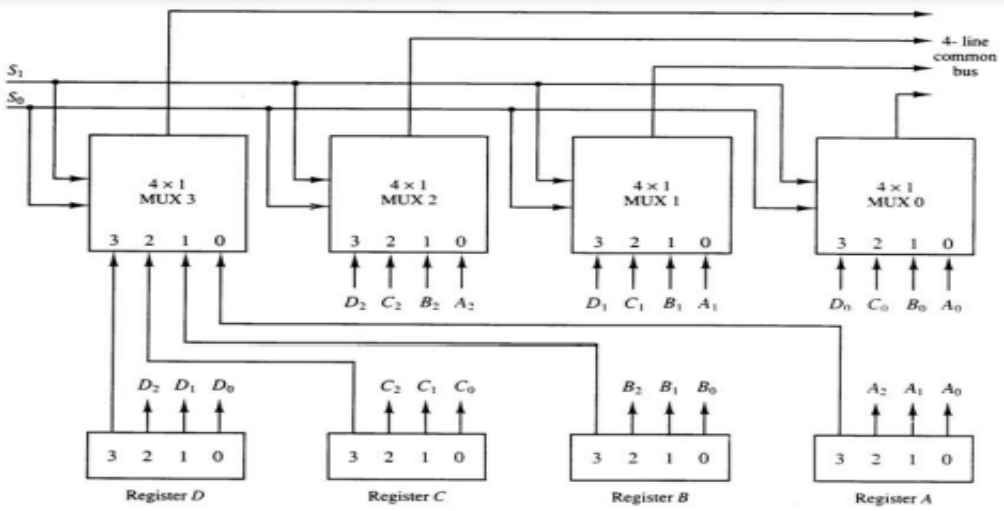
- It is assumed that all transfers occur during a clock edge transition
- All microoperations written on a single line are to be executed at the same time T: R2 □ R1, R1 □R2

TABLE 4-1 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

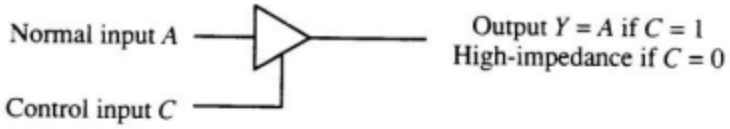


- In general, a bys system will multiplex k registers of n bits each to produce an n- line common bus
- This requires n multiplexers – one for each bit
- The size of each multiplexer must be k x 1
- The number of select lines required is log k
- To transfer information from the bus to a register, the 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 isimplied

Figure 4-4 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

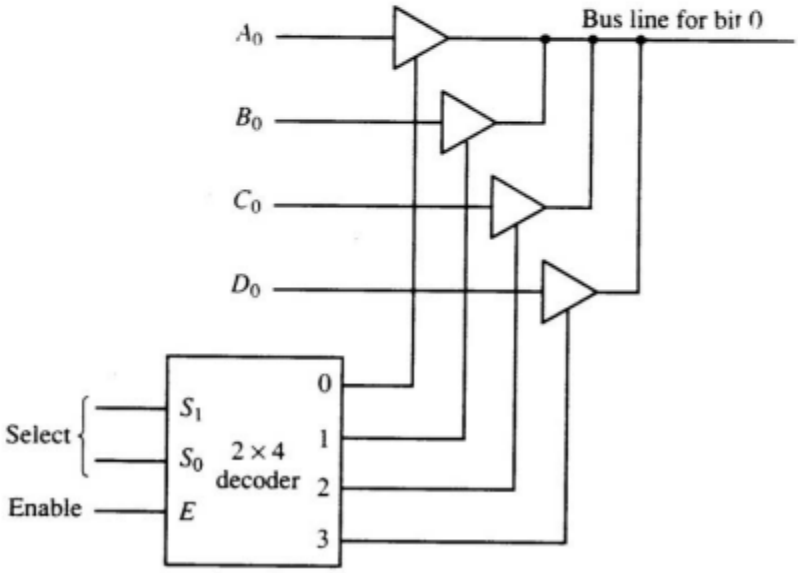


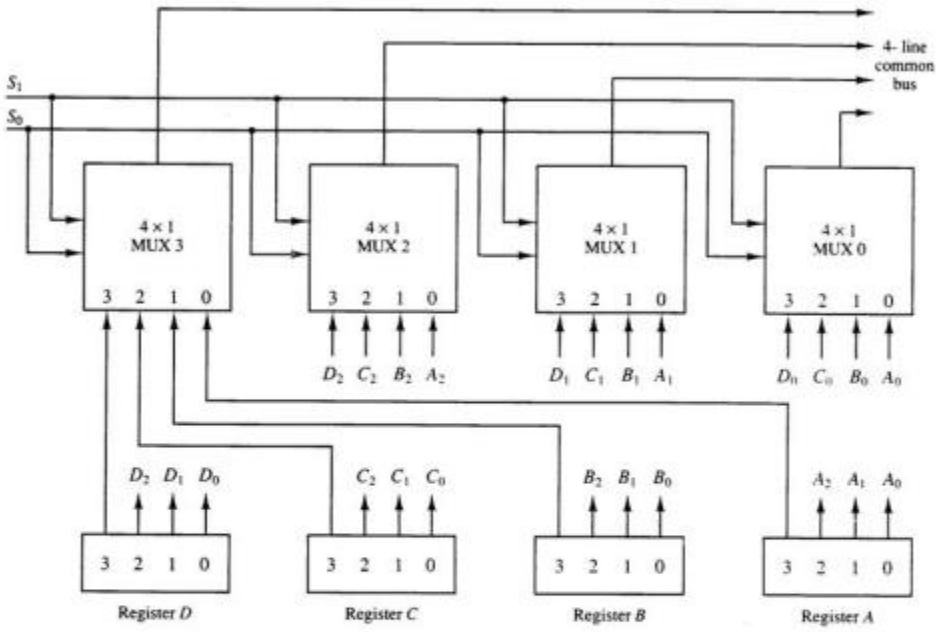
Figure 4-5 Bus line with three-state buffers.

- 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

Figure 4-3 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

- An n-bit binary adder requires n full-adders

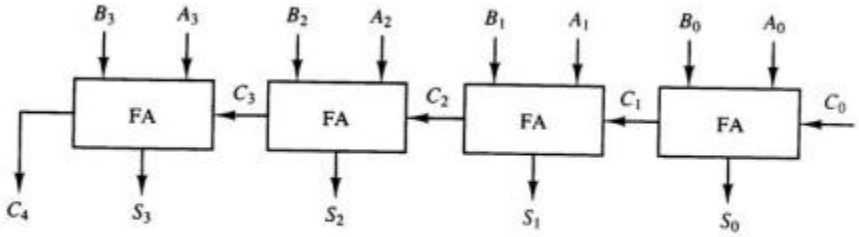


Figure 4-6 4-bit binary adder.

- The subtraction A-B can be carried out by the following steps
  - Take the 1's complement of B (invert each bit)

NRCM

- 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

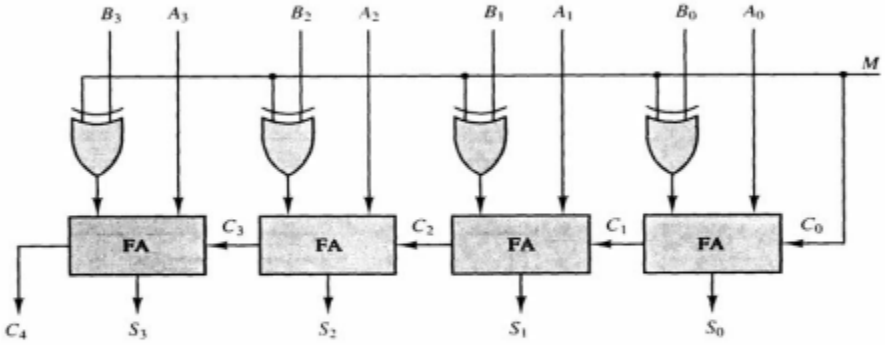


Figure 4-7 4-bit adder-subtractor.

- 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

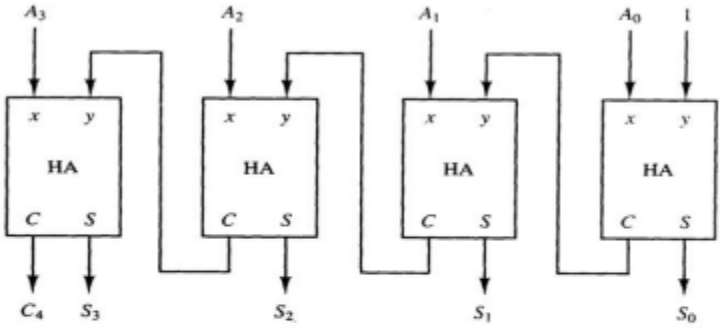


Figure 4-8 4-bit binary incrementer.

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

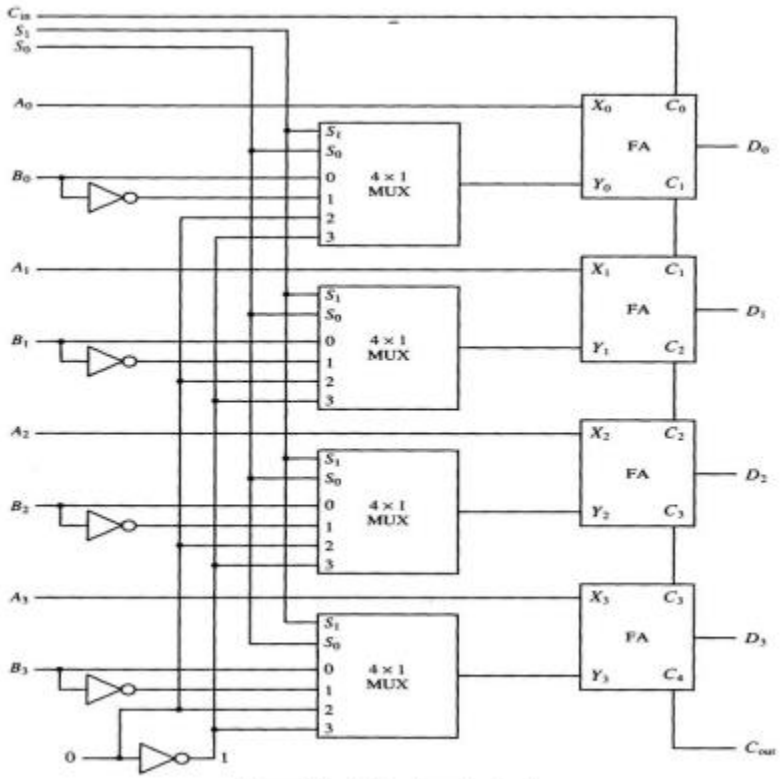


Figure 4-9 4-bit arithmetic circuit.

TABLE 4-4 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

Logic Micro operations

□ 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

TABLE 4-5 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

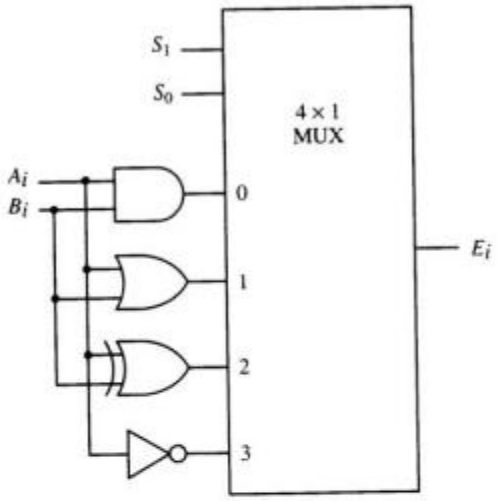
TABLE 4-6 Sixteen Logic Microoperations

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

Figure 4-10 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
1100 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
1100 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
1100 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 □ shlR1

- 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

TABLE 4-7 Shift Microoperations

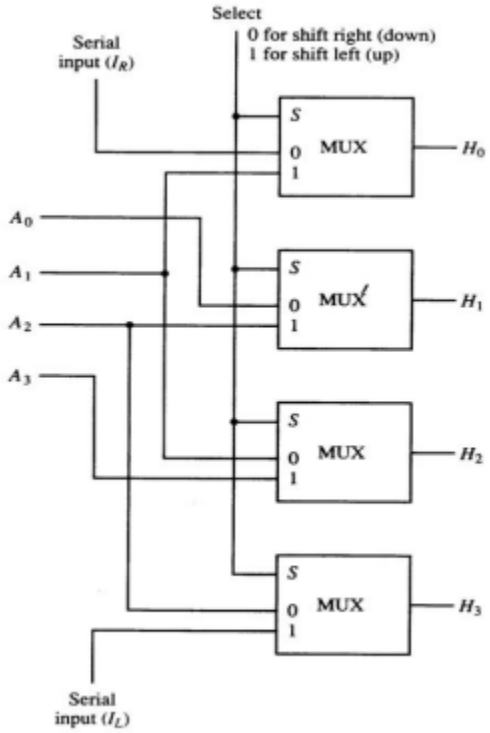
Symbolic designation	Description
$R \leftarrow \text{shl } R$	Shift-left register R
$R \leftarrow \text{shr } R$	Shift-right register R
$R \leftarrow \text{cil } R$	Circular shift-left register R
$R \leftarrow \text{cir } R$	Circular shift-right register R
$R \leftarrow \text{ashl } R$	Arithmetic shift-left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

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

Figure 4-11 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



Function table

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

Figure 4-12 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.

Figure 4-13 One stage of arithmetic logic shift unit.

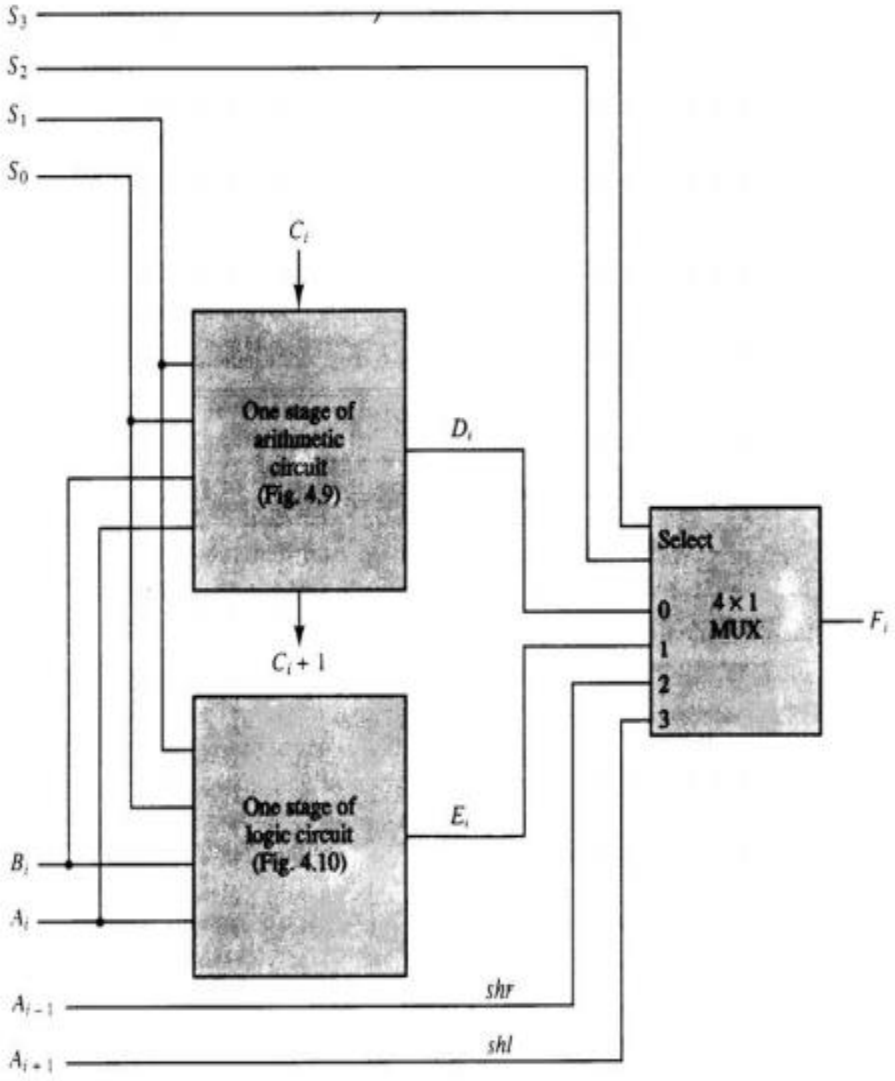


TABLE 4-8 Function Table for Arithmetic Logic Shift Unit

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

Instruction Formats:

A computer will usually have a variety of instruction code formats. It is the function of the control unit within the CPU to interpret each instruction code and provide the necessary control functions needed to process the instruction.

The format of an instruction is usually depicted in a rectangular box symbolizing the bits of the instruction as they appear in memory words or in a control register. The bits of the instruction are divided into groups called fields. The most common fields found in instruction formats are:

1 An operation code field that specifies the operation to be performed.

2. An address field that designates a memory address or a processor registers.

3. A mode field that specifies the way the operand or the effective address is determined.

Other special fields are sometimes employed under certain circumstances, as for example a field that gives the number of shifts in a shift-type instruction.

The operation code field of an instruction is a group of bits that define various processor operations, such as add, subtract, complement, and shift.

The bits that define the mode field of an instruction code specify a variety of alternatives for choosing the operands from the given address.

Operations specified by computer instructions are executed on some data stored in memory or processor registers, Operands residing in processor registers are specified with a register address. A register address is a binary number of k bits that defines one of 2k registers in the CPU. Thus a CPU with 16 processor registers R0 through R15 will have a register address field of four bits. The binary number 0101, for example, will designate register R5. Computers may have instructions of 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 three types of CPU organizations: 1 Single accumulator organization. 2 General register organization. 3 Stack organization. All operations are performed with an implied accumulator register. The instruction format in this type of computer uses one address field. For example, the instruction that specifies an arithmetic addition is defined by an assembly language instruction as ADD. Where X is the address of the operand. The ADD instruction in this case results in the operation AC ← AC + M[X]. AC is the accumulator register and M[X] symbolizes the memory word located at address X. An example of a general register type of organization was presented in Fig. 7.1. The instruction format in this type of computer needs three register address fields. Thus the instruction for an arithmetic addition may be written in an assembly language as ADD R1, R2, R3 To denote the operation R1 ← R2 + R3. The number of address fields in the instruction can be reduced from three to two if the destination register is the same as one of the source registers. Thus the instruction ADD R1, R2 Would denote the operation R1 ← R1 + R2. Only register addresses for R1 and R2 need be specified in this instruction. Computers with multiple processor registers use the move instruction with a mnemonic MOV to symbolize a transfer instruction. Thus the instruction MOV R1, R2 Denotes the transfer R1 ← R2 (or R2 ← R1, depending on the particular computer). Thus transfer-type instructions need two address fields to specify the source and the destination.

General register-type computers employ two or three address fields in their instruction format. Each address field may specify a processor register or a memory word. An instruction symbolized by ADD R1, X Would specify the operation R1 ← R + M [X]. It has two address fields, one for register R1 and the other for the memory address X. The stack-organized CPU was presented in Fig. 8-4. Computers with stack organization would have PUSH and POP instructions which require an address field. Thus the instruction PUSH X Will push the word at address X to the top of the stack. The stack pointer is updated automatically. Operation-type instructions do not need an address field in stack-organized computers. This is because the operation is performed on the two items that are on top of the stack. The instruction ADD in a stack computer consists of an operation code only with no address field. This operation has the effect of popping the two top numbers from the stack, adding the numbers, and pushing the sum into the stack. There is no need to specify operands with an address field since all operands are implied to be in the stack. To illustrate the influence of the number of addresses on computer programs, we will evaluate the arithmetic statement X = (A + B) ∗ (C + D). Using zero, one, two, or three address instruction. We will use the symbols ADD, SUB,

MUL, and DIV for the four arithmetic operations; MOV for the transfer-type operation; and LOAD and STORE for transfers to and from memory and AC register. We will assume that the operands are in memory addresses A, B, C, and D, and the result must be stored in memory at address X. Three-Address Instructions Computers with three-address instruction formats can use each address field to specify either a processor register or a memory operand. The program in assembly language that evaluates X = (A + B) ∗ (C + D) is shown below, together with comments that explain the register transfer.

operation of each instruction.

ADD R1, A, B R1 ←

M [A] + M [B]

ADD R2, C, D R2 ←

M [C] + M [D]

MUL X, R1, R2 M [X]

← R1 ∗R2

It is assumed that the computer has two processor registers, R1 and R2. The symbol M [A] denotes the operand at memory address symbolized by A.

The advantage of the three-address format is that it results in short programs when evaluating arithmetic expressions. The disadvantage is that the binarycoded instructions require too many bits to specify three addresses. An example of a commercial computer that uses three-address instructions is the Cyber 170. The instruction formats in the Cyber computer are restricted to either three register address fields or two register address fields and one memory address field

Two-Address Instructions

Two address instructions are the most common in commercial computers. Here again each address field can specify either a processor register or a memory word. The program to evaluate X = (A + B) ∗ (C + D) is as

follows:

MOV R1, A R1 ← M [A]

ADD R1, B R1 ← R1 + M [B]

MOV R2, C R2 ← M [C]

ADD R2, D R2 ← R2 + M [D]

MUL R1, R2 R1 ← R1∗R2

MOV X, R1 M [X] ← R1

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

One-Address Instructions

One-address instructions use an implied accumulator (AC) register for all data manipulation. For multiplication and division there is a need for a second register. However, here we will neglect the second and assume that the AC contains the result of tall operations. The program to evaluate X =

(A + B) ∗ (C + D) is

LOAD A AC ← M [A]

ADD B AC ← A [C] + M [B]

STORE T M [T] ← AC

LOAD C AC ← M [C]

ADD D AC ← AC + M [D]

MUL T AC ← AC ∗ M [T]

STORE X M [X] ← AC

All operations are done between the AC register and a memory operand. T is the address of a temporary memory location required for storing the intermediate result.

Zero-Address Instructions

A stack-organized computer does not use an address field for the instructions ADD and MUL. The PUSH and POP instructions, however, need an address field to specify the operand that communicates with the stack. The following program shows how X = (A + B) ∗ (C + D) will be

written for a stack organized computer. (TOS stands for top of stack)

PUSH A TOS ← A

PUSH B TOS ← B

ADD

PUSH C

TOS ← (A + B)

TOS ← C

PUSH D TOS ← D

ADD TOS ← (C + D)

MUL

POP X

TOS ← (C + D) ∗ (A + B)

M [X] ← TOS

To evaluate arithmetic expressions in a stack computer, it is necessary

to convert the expression into reverse Polish notation. The name "zeroaddress" is given to this type of computer because of the absence of an address field in the computational instructions.

Instruction Codes

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.

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.

Addressing Modes

To understand the various addressing modes to be presented in this section, it is imperative that we understand the basic operation cycle of the computer.

The control unit of a computer is designed to go through an instruction cycle that is divided into three major phases:

1. Fetch the instruction from memory

2. Decode the instruction.

3. Execute the instruction.

There is one register in the computer called the program counter of PC that keeps track of the instructions in the program stored in memory. PC holds the address of the instruction to be executed next and is incremented each time an instruction is fetched from memory. The decoding done in step 2 determines the operation to be performed, the addressing mode of the instruction and the location of the operands. The computer then executes the instruction and returns to step 1 to fetch the next instruction in sequence. In some computers the addressing mode of the instruction is specified with a distinct binary code, just like the operation code is specified. Other computers use a single binary code that designates both the operation and the mode of the instruction. Instructions may be defined with a variety of addressing modes, and sometimes, two or more addressing modes are combined in one instruction.

1. The operation code specified the operation to be performed. The mode field is sued to locate the operands needed for the operation. There may or may not be an address field in the instruction. If there is an address field, it may designate a memory address or a processor register. Moreover, as discussed in the preceding section, the instruction may have more than one address field, and each address field may be associated with its own particular addressing mode.

Although most addressing modes modify the address field of the instruction, there are two modes that need no address field at all. These are the implied and immediate modes.

1 Implied Mode: In this mode the operands are specified implicitly in the definition of the instruction. For example, the instruction “complement accumulator” is an implied-mode instruction because the operand in the accumulator register is implied in the definition of the instruction. In fact, all register reference instructions that sue an accumulator are implied-mode instructions.

Op code Mode Address

Figure 1: Instruction format with mode field

Zero-address instructions in a stack-organized computer are impliedmode instructions since the operands are implied to be on top of the stack.

2 Immediate Mode: In this mode the operand is specified in the instruction itself. Inother words, an immediate- mode instruction has an operand field rather than an address field. The operand field contains the actual operand to be used in conjunction with the operation specified in the instruction. Immediate-mode instructions are useful for initializing registers to a constant value.

It was mentioned previously that the address field of an instruction may specify either a memory word or a processor register. When the address field specifies a processor register, the instruction is said to be in the register mode.

3 Register Mode: In this mode the operands are in registers that reside within the CPU.The particular register is selected from a register field in the instruction. A kbit field can specify any one of 2k registers.

4 Register Indirect Mode: In this mode the instruction specifies a register in the CPUwhose contents give the address of the operand in memory. In other words, the selected register contains the address of the operand rather than the operand itself. Before using a register indirect mode instruction, the programmer must ensure that the memory address fo the operand is placed in the processor register with a previous instruction. A reference to the register is then equivalent to specifying a memory address. The advantage of a register indirect mode instruction is that the address field of the instruction sues fewer bits to select a register than would have been required to specify a memory address directly.

5.Auto increment or Auto decrement Mode: This is similar to the register indirect modeexcept that the register is incremented or decremented after (or before) its value is used to access memory. When the address stored in the register refers to a table of data in memory, it is necessary to increment or decrement the register after every access to the table. This can be achieved by using the increment or decrement instruction.

However, because it is such a common requirement, some computers incorporate a special mode that automatically increments or decrements the content of the register after data access.

The address field of an instruction is used by the control unit in the CPU to obtain the operand from memory. Sometimes the value given in the address field is the address of the operand, but sometimes it is just an address from which the address of the operand is calculated. To differentiate among the various addressing modes it is necessary to distinguish between the address part of the instruction

and the effective address used by the control when executing the instruction. The effective address is defined to be the memory address obtained from the computation dictated by the given addressing mode. The effective address is the address of the operand in a computational-type instruction. It is the address where control branches in response to a branch-type instruction. We have already defined two addressing modes in previous chapter.

6 Direct Address Mode: In this mode the effective address is equal to the address part ofthe instruction. The operand resides in memory and its address is given directly by the address field of the instruction. In a branch-type instruction the address field specifies the actual branch address.

7 Indirect Address Mode: In this mode the address field of the instruction gives theaddress where the effective address is stored in memory. Control fetches the instruction from memory and uses its address part to access memory again to read the effective address.

8 Relative Address Mode: In this mode the content of the program counter is added to the address part of the instruction in order to obtain the effective address. The address part of the instruction is usually a signed number (in 2"s complement representation) which can be either positive or negative. When this number is added to the content of the program counter, the result produces an effective address whose position in memory is relative to the address of the next instruction. To clarify with an example, assume that the program counter contains the number 825 and the address part of the instruction contains the number 24. The instruction at location 825 is read from memory during the fetch phase and the program counter is then incremented by one to 826 + 24 = 850. This is 24 memory locations forward from the address of the next instruction. Relative addressing is often used with branch-type instructions when the branch address is in the area surrounding the instruction word itself. It results in a shorter address field in the instruction format since the relative address can be specified with a smaller number of bits compared to the number of bits required to designate the entire memory address.

9 Indexed Addressing Mode: In this mode the content of an index register is added to theaddress part of the instruction to obtain the effective address. The index register is a special CPU register that contains an index value. The address field of the instruction defines the beginning address of a data array in memory. Each operand in the array is stored in memory relative to the beginning address. The distance between the beginning address and the address of the operand is the index value stores in the index register. Any operand in the array can be accessed with the same instruction provided that the index register contains the correct index value. The index register can be incremented to facilitate access to consecutive operands. Note that if an index-type instruction does not include an address field in its format, the instructionconverts to the register indirect mode of operation. Some computers dedicate one CPU register to function solely as an index register. This register is involved implicitly when the index-mode instruction is used. In computers with many processor registers, any

one of the CPU registers can contain the index number. In such a case the register must be specified explicitly in a register field within the instruction format.

10 Base Register Addressing Mode: In this mode the content of a base register is added tothe address part of the instruction to obtain the effective address. This is similar to the indexed addressing mode except that the register is now called a base register instead of an index register. The difference between the two modes is in the way they are used rather than in the way that they are computed. An index register is assumed to hold an index number that is relative to the address part of the instruction. A base register is assumed to hold a base address and the address field of the instruction gives a displacement relative to this base address. The base register addressing mode is used in computers to facilitate the relocation of programs in memory. When programs and data are moved from one segment of memory to another, as required in multiprogramming systems, the address values of the base register requires updating to reflect the beginning of a new memory segment.

Numerical Example

Numerical Example

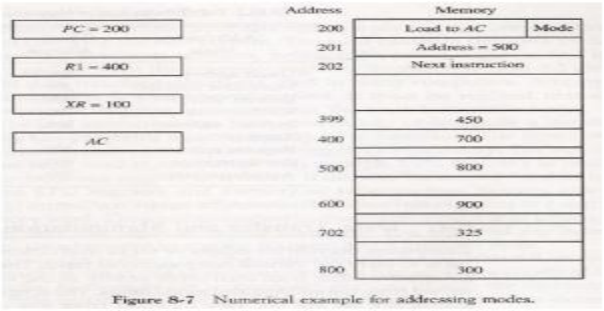


Figure 8-7 Numerical example for addressing modes.

TABLE 8-4 Tabular List of Numerical Example

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

Computer Registers

Register symbol	Number of bits	Register name	Register Function----------
DR	16	Data register	Holds memory operands
AR	12	Address register	Holds address for memory
AC	16	Accumulator	Processor register
IR	16	Instruction register	Holds instruction code
PC	12	Program counter	Holds address of instruction
TR	16	Temporary register	Holds temporary data
INPR	8	Input register	Holds input character
OUTR	8	Output register	Holds output character

Data Register(DR) : hold the operand(Data) read from memory

Accumulator Register(AC) : general purpose processing register

Instruction Register(IR) : hold the instruction read frommemory

Temporary Register(TR) : hold a temporary data during processing

Address Register(AR) : hold a memory address, 12 bit width

Program Counter(PC) :

»hold the address of the next instruction to be read frommemory after the current instruction is executed

»Instruction words are read and executed in sequence unless a branch instruction is encountered

»A branch instruction calls for a transfer to a nonconsecutive instruction in the program

»The address part of a branch instruction is transferred to PCto become the address of the next instruction

Input Register(INPR) : receive an 8-bit character from an input device

Output Register(OUTR) : hold an 8-bit character for an

output device

The following registers are used in Mano‟s example computer.

Register Number Register Register

symbol of bits name Function----------

DR 16 Data register Holds memory operands

AR 12 Address register Holds address for memory

AC 16 Accumulator Processor register

IR 16 Instruction register Holds instruction code

PC 12 Program counter Holds address of instruction

TR 16 Temporary register Holds temporary data

INPR 8 Input register Holds input character

OUTR 8 Output register Holds output character

Computer Instructions:

The basic computer has 16 bit instruction register (IR) which can denote either memory reference or register reference or input-output instruction.

1. Memory Reference – These instructions refer to memory address as an operand. The other operand is always accumulator. Specifies 12 bit address, 3 bit opcode (other than 111) and 1 bit addressing mode for direct and indirect addressing.

Example –

IR register contains = 0001XXXXXXXXXXX, i.e. ADD after fetching and decoding of instruction we find out that it is a memory reference instruction for ADD operation.

Hence, DR <- M[AR]

AC <- AC+ DR, SC <- 0

2. Register Reference – These instructions perform operations on registers rather than memory addresses. The IR(14-12) is 111 (differentiates it from memory reference) and IR(15) is 0 (differentiates it from input/output instructions). The rest 12 bits specify register operation.

Example –

IR register contains = 0111001000000000, i.e. CMA after fetch and decode cycle we find out that it is a register reference instruction for complement accumulator.

Hence, AC <- ~AC

3. Input/Output – These instructions are for communication between computer and outside environment. The IR(14-12) is 111 (differentiates it from memory reference) and IR(15) is 1 (differentiates it from register reference instructions). The rest 12 bits specify I/O operation.

Example –

IR register contains = 1111100000000000, i.e. INP after fetch and decode cycle we find out that it is an input/output instruction for inputing character. Hence, INPUT character from peripheral device.

Timing and Control

All sequential circuits in the Basic Computer CPU are driven by a master clock, with the exception of the INPR register. At each clock pulse, the control unit sends control signals to control inputs of the bus, the registers, and the ALU.

Control unit design and implementation can be done by two general methods:

 A hardwired control unit is designed from scratch using traditional digital logic design techniques to produce a minimal, optimized circuit. In other words, the control unit is like an ASIC (application-specific integrated circuit).

 A microprogrammed control unit is built from some sort of ROM. The desired control signals are simply stored in the ROM, and retrieved in sequence to drive the microoperations needed by a particular instruction.

Instruction Cycle

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

1. Fetch instruction

2. Decode

3. Fetch operand

4. Execute

Program execution can be represented as a top-down design:

1. Program execution

a. Instruction 1

i. Fetch instruction

ii. Decode

iii. Fetch operand

iv. Execute

b. Instruction 2

i. Fetch instruction

ii. Decode

iii. Fetch operand

iv. Execute

c. Instruction 3 ...

Program execution begins with:

PC ← address of first instruction, SC ← 0

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

Instruction Fetch and Decode

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

Everything that happens in this phase is driven entirely by timing variables T0, T1 and T2. Hence, all control inputs in the CPU during fetch and decode are functions of these three variables alone.

T0: AR ← PC

T1: IR ← M[AR], PC ← PC + 1

T2: D0-7 ← decoded IR(12-14), AR ← IR(0-11), I ← IR(15)

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

UNIT 2:

Microprogrammed Control: Control memory, Address sequencing, micro program example, design of control unit.

Central Processing Unit: General Register Organization, Instruction Formats, Addressing modes, Data Transfer and Manipulation, Program Control.

Micro Programmed Control:

Control Memory

 The control unit in a digital computer initiates sequences of microoperations

 The complexity of the digital system is derived form the number of sequences that are performed

 When the control signals are generated by hardware, it is hardwired

 In a bus-oriented system, the control signals that specify microoperations are groups of bits that select the paths in multiplexers, decoders, and ALUs.

 The control unit initiates a series of sequential steps of microoperations

 The control variables can be represented by a string of 1"s and 0"s called a control word

 A microprogrammed control unit is a control unit whose binary control variables are stored in memory

 A sequence of microinstructions constitutes a microprogram

 The control memory can be a read-only memory

 Dynamic microprogramming permits a microprogram to be loaded and uses a writable control memory

 A computer with a microprogrammed control unit will have two separate memories: a main memory and a control memory

 The microprogram consists of microinstructions that specify various internal control signals for execution of register microoperations

 These microinstructions generate the microoperations to: o fetch the instruction from main memory o evaluate the effective address o execute the operation o return control to the fetch phase for the next instruction

 The control memory address register specifies the address of the microinstruction

 The control data register holds the microinstruction read from memory

 The microinstruction contains a control word that specifies one or more microoperations for the data processor

 The location for the next microinstruction may, or may not be the next in sequence

 Some bits of the present microinstruction control the generation of the address of the next microinstruction

 The next address may also be a function of external input conditions

 While the microoperations are being executed, the next address is computed in the next address generator circuit (sequencer) and then transferred into the CAR to read the next microinstructions

 Typical functions of a sequencer are: o incrementing the CAR by one o loading into the CAR and address from control memory o transferring an external address o loading an initial address to start the control operations

 A clock is applied to the CAR and the control word and next-address information are taken directly from the control memory

 The address value is the input for the ROM and the control work is the output

 No read signal is required for the ROM as in a RAM

 The main advantage of the microprogrammed control is that once the hardware configuration is established, there should be no need for h/w or wiring changes

 To establish a different control sequence, specify a different set of microinstructions for control memory

Address Sequencing

 Microinstructions are stored in control memory in groups, with each group specifying a routine

 Each computer instruction has its own microprogram routine to generate the microoperations

 The hardware that controls the address sequencing of the control memory must be capable of sequencing the microinstructions within a routine and be able to branch from one routine to another

 Steps the control must undergo during the execution of a single computer instruction: o Load an initial address into the CAR when power is turned on in the computer. This address is usually the address of the first microinstruction that activates the instruction fetch routine – IR holds instruction

o The control memory then goes through the routine to determine the effective address of the operand – AR holds operand address o The next step is to generate the microoperations that

execute the instruction by considering the opcode and applying a mapping o After execution, control must return to the fetch routine by executing an unconditional branch

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

 Conditional branching is obtained by using part of the microinstruction to select a specific status bit in order to determine its condition

 The status conditions are special bits in the system that provide parameter information such as the carry-out of an adder, the sign bit of a number, the mode bits of an instruction, and i/o status conditions

 The status bits, together with the field in the microinstruction that specifies a branch address, control the branch logic

 The branch logic tests the condition, if met then branches, otherwise, increments the CAR

 If there are 8 status bit conditions, then 3 bits in the microinstruction are used to specify the condition and provide the selection variables for the multiplexer

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

 A special type of branch exists when a microinstruction specifies a branch to the first word in control memory where a microprogram routine is located

 The status bits for this type of branch are the bits in the opcode

 Assume an opcode of four bits and a control memory of 128 locations

 The mapping process converts the 4-bit opcode to a 7-bit address for control memory

 This provides for each computer instruction a microprogram routine with a capacity of four microinstructions

 Subroutines are programs that are used by other routines to accomplish a particular task and can be called from any point within the main body ofthe microprogram

 Frequently many microprograms contain identical section of code

 Microinstructions can be saved by employing subroutines that use common sections of microcode

 Microprograms that use subroutines must have a provisions for storing the return address during a subroutine call and restoring the address during a subroutine return

 A subroutine register is used as the source and destination for the addresses

Microprogram Example

•In the block diagram four registers and ALU areassociated with the processor unit. –DR, AR, PC, AC and–ALU•DR can receive information from AC, PC or memory (selected byMUX)•AR can receive information from PC or DR (selected by MUX)•PC can receive information only from AR.

➢The process of code generation for the control memory is called microprogramming. ➢Transfer of information among registers in the processor is through MUXs rather than a bus.

Design of Control Unit

The Control Unit is classified into two major categories:

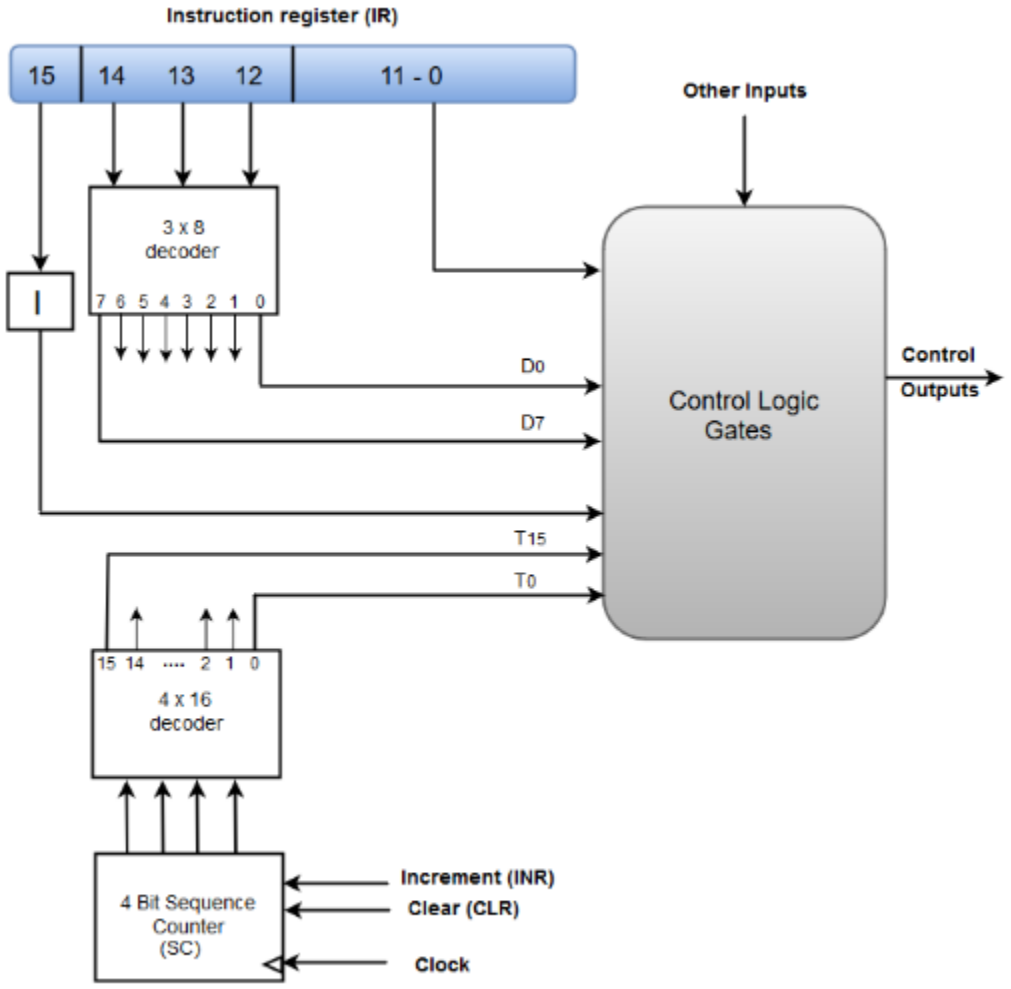
Hardwired Control

Microprogrammed Control

Hardwired Control

The Hardwired Control organization involves the control logic to be implemented with gates, flip-flops, decoders, and other digital circuits.

The following image shows the block diagram of a Hardwired Control organization.



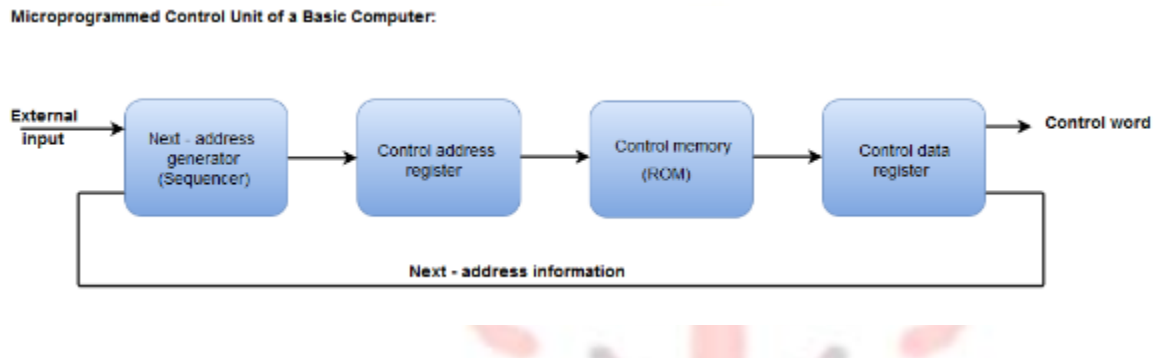
- A Hard-wired Control consists of two decoders, a sequence counter, and a number of logic gates.
- An instruction fetched from the memory unit is placed in the instruction register (IR).
- The component of an instruction register includes; I bit, the operation code, and bits 0 through 11.
- The operation code in bits 12 through 14 are coded with a 3 x 8 decoder.
- The outputs of the decoder are designated by the symbols D0 through D7.
- The operation code at bit 15 is transferred to a flip-flop designated by the symbol I.
- The operation codes from Bits 0 through 11 are applied to the control logic gates.

- The Sequence counter (SC) can count in binary from 0 through 15.

The Microprogrammed Control organization is implemented by using the programming approach.

In Microprogrammed Control, the micro-operations are performed by executing a program consisting of micro-instructions.

The following image shows the block diagram of a Microprogrammed Control organization.



- The Control memory address register specifies the address of the micro-instruction.
- The Control memory is assumed to be a ROM, within which all control information is permanently stored.
- The control register holds the microinstruction fetched from the memory.
- The micro-instruction contains a control word that specifies one or more micro-operations for the data processor.
- While the micro-operations are being executed, the next address is computed in the next address generator circuit and then transferred into the control address register to read the next microinstruction.
- The next address generator is often referred to as a micro-program sequencer, as it determines the address sequence that is read from control memory.

Central Processing Unit:

The operation or task that must perform by CPU is:

• Fetch Instruction: The CPU reads an instruction from memory.

• Interpret Instruction: The instruction is decoded to determine what action is required.

• Fetch Data: The execution of an instruction may require reading data from memory or I/O module.

• Process data: The execution of an instruction may require performing some arithmetic or logical operation on data.

• Write data: The result of an execution may require writing data to memory or an I/O module.

To do these tasks, it should be clear that the CPU needs to store some data temporarily. It must remember the location of the last instruction so that it can know where to get the next instruction. It needs to store instructions and data temporarily while an instruction is being executed. In other words, the CPU needs a small internal memory. These storage locations are generally referred as registers. The major components of the CPU are an arithmetic and logic unit (ALU) and a control unit (CU). The ALU does the actual computation or processing of data. The CU controls the movement of data and instruction into and out of the CPU and controls the operation of the ALU. The CPU is connected to the rest of the system through system bus. Through system bus, data or information gets transferred between the CPU and the other component of the system. The system bus may have three components: Data Bus: Data bus is used to transfer the data between main memory and CPU. Address Bus: Address bus is used to access a particular memory location by putting the address of the memory location. Control Bus: Control bus is used to provide the different control signal generated by CPU to different part of the system. As for example, memory read is a signal generated by CPU to indicate that a memory read operation has to be performed. Through control bus this signal is transferred to memory module to indicate the required operation.

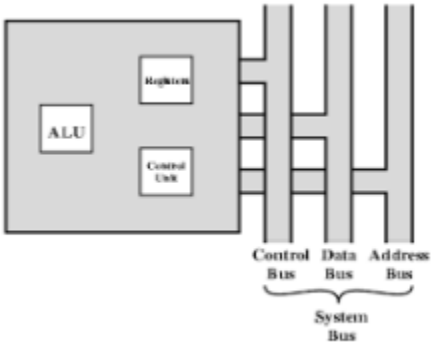
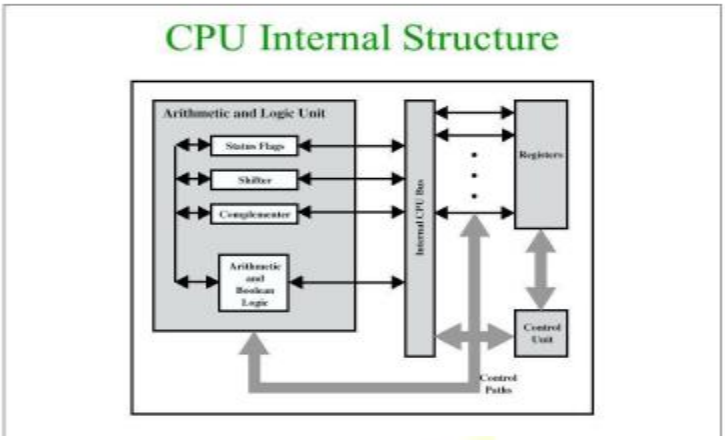


Figure 1: CPU with the system bus.



Stack Organization:

A useful feature that is included in the CPU of most computers is a stack or last in, first out (LIFO) list. A stack is a storage device that stores information in such a manner that the item stored last is the first item 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 only(after an initial value is loaded in to it).The register that hold the address for the stack is called a stack pointer (SP) because its value always points at the top item in stack. Contrary to a stack of trays where the tray it self may be taken out or inserted, the physical registers of a stack are always available for reading or writing. The two operation of stack are the insertion and deletion of items. The operation of insertion is called PUSH because it can be thought of as the result of pushing a new item on top. The operation of deletion is called POP because it can be thought of as the result of removing one item so that the stack pops up. However, nothing is pushed or popped in a computer stack. These operations are simulated by incrementing or decrementing the stack pointer register.

INSTRUCTION FORMATS:

We know that a machine instruction has an opcode and zero or more operands. Encoding an instruction set can be done in a variety of ways. Architectures are differentiated from one another by the number of bits allowed per instruction (16, 32, and 64 are the most common), by the number of operands allowed per instruction, and by the types of instructions and data each can process. More specifically, instruction sets are differentiated by the following features: 1. Operand storage in the CPU (data can be stored in a stack structure or in registers) 2. Number of explicit operands per instruction (zero, one, two, and three being the most common) 3. Operand location (instructions can be classified as register-to-register, register-tomemory or memory-to-memory, which simply refer to

the combinations of operands allowed per instruction) 4. Operations (including not only types of operations but also which instructions can access memory and which cannot) 5. Type and size of operands (operands can be addresses, numbers, or even characters) Number of Addresses: One of the characteristics of the ISA(Industrial Standard Architecture) that shapes the architecture is the number of addresses used in an instruction. Most operations can be divided into binary or unary operations. Binary operations such as addition and multiplication require two input operands whereas the unary operations such as the logical NOT need only a single operand. Most operations produce a single result. There are exceptions, however. For example, the division operation produces two outputs: a quotient and a remainder. Since most operations are binary, we need a total of three addresses: two addresses to specify the two input operands and one to specify where the result should go.

Three-Address Machines:

In three-address machines, instructions carry all three addresses explicitly. The RISC processors use three addresses. Table X1 gives some sample instructions of a threeaddress machine.

In these machines, the C statement

A = B + C * D - E + F + A

is converted to the following code:

mult T,C,D ; T = C*D

add T,T,B ; T = B + C*D

sub T,T,E ; T = B + C*D - E

add T,T,F ; T = B + C*D - E + F

add A,T,A ; A = B + C*D - E + F + A

Table :T1 Sample three-address machine instructions

Instruction	Semantics
add dest,src1,src2	Adds the two values at src1 and src2 and stores the result in dest M(dest) = [src1] + [src2]
sub dest,src1,src2	Subtracts the second source operand at src2 from the first at src1 and stores the result in dest M(dest) = [src1] - [src2]
mult dest,src1,src2	Multiplies the two values at src1 and src2 and stores the result in dest M(dest) = [src1] * [src2]

We use the notation that each variable represents a memory address that stores the value associated with that variable. This translation from symbol name to the memory address is done by using a symbol table.

As you can see from this code, there is one instruction for each arithmetic operation. Also notice that all instructions, barring the first one, use an address twice. In the middle three instructions, it is the temporary T and in the last one, it is A. This is the motivation for using two addresses, as we show next.

Two-Address Machines : In two-address machines, one address doubles as a source and destination. Usually, we use dest to indicate that the address is used for destination. But you should note that this address also supplies one of the source operands. The Pentium is an example processor that uses two addresses. Sample instructions of a two-address machine On these machines, the C statement A = B + C * D - E + F + A is converted to the following code: load T,C ; T = C mult T,D ; T = C*D add T,B ; T = B + C*D sub T,E ; T = B + C*D - E add T,F ; T = B + C*D - E + F add A,T ; A = B + C*D - E + F + A Table :T2 Sample Two-address machine instructions:

Instruction	Semantics
load dest,src	Copies the value at src to dest M(dest) = [src]
add dest,src	Adds the two values at src and dest and stores the result in dest M(dest) = [dest] + [src]
sub dest,src	Subtracts the second source operand at src from the first at dest and stores the result in dest M(dest) = [dest] - [src]
mult dest,src	Multiplies the two values at src and dest and stores the result in dest M(dest) = [dest] * [src]

One-Address Machines : In the early machines, when memory was expensive and slow, a special set of registers was used to provide an input operand as well as to receive the result from the ALU. Because of this, these registers are called the accumulators. In most machines, there is just a single accumulator register. This kind of design, called accumulator machines, makes sense if memory is expensive. In accumulator machines, most operations are performed on the contents of the accumulator and the operand supplied by the instruction. Thus, instructions for these machines need to specify only the address of a single operand. There is no need to

store the result in memory: this reduces the need for larger memory as well as speeds up the computation by reducing the number of memory accesses. A few sample accumulator machine instructions are shown in Table X3. In these machines, the C statement A = B + C * D - E + F + A is converted to the following code: load C ; load C into the accumulator mult D ; accumulator = C*D add B ; accumulator = C*D+B sub E ; accumulator = C*D+B-E add F ; accumulator = C*D+B-E+F.

add A ; accumulator = C*D+B-E+F+A store A ; store the accumulator contents in A

Table :T3 Sample ONE-address machine instructions

Instruction		Semantics
load	addr	Copies the value at address addr into the accumulator accumulator = [addr]
store	addr	Stores the value in the accumulator at the memory address addr M(addr) = accumulator
add	addr	Adds the contents of the accumulator and value at address addr accumulator = accumulator + [addr]
sub	addr	Subtracts the value at memory address addr from the contents of the accumulator accumulator = accumulator - [addr]
mult	addr	Multiplies the contents of the accumulator and value at address addr accumulator = accumulator * [addr]

Zero-Address Machines : In zero-address machines, locations of both operands are assumed to be at a default location. These machines use the stack as the source of the input operands and the result goes back into the stack. Stack is a LIFO (last-in-first-out) data structure that all processors support, whether or not they are zero-address machines. As the name implies, the last item placed on the stack is the first item to be taken out of the stack. A good analogy is the stack of trays you find in a cafeteria. All operations on this type of machine assume that the required input operands are the top two values on the stack. The result of the operation is placed on top of the stack. Table X4 gives some sample instructions for the stack machines.

Table :T4 Sample Zero-address machine instructions

Instruction	Semantics
push addr	Places the value at address addr on top of the stack push([addr])
pop addr	Stores the top value on the stack at memory address addr M(addr) = pop
add	Adds the top two values on the stack and pushes the result onto the stack push(pop + pop)
sub	Subtracts the second top value from the top value of the stack and pushes the result onto the stack push(pop – pop)
mult	Multiplies the top two values in the stack and pushes the result onto the stack push(pop * pop)

two are special instructions that use a single address and are used to move data between memory and stack.

All other instructions use the zero-address format. Let‟s see how the stack machine translates the arithmetic expression we have seen in the previous subsections. In these machines, the C statement

A = B + C * D - E + F + A

is converted to the following code:

push E ; <E>

push C ; <C, E>

push D ; <D, C, E>

mult ; <C*D, E>

push B ; <B, C*D, E>

add ; <B+C*D, E>

sub ; <B+C*D-E>

push F ; <F, B+D*C-E>

add ; <F+B+D*C-E>

push A ; <A, F+B+D*C-E>

add ; <A+F+B+D*C-E>

pop A ; < >

On the right, we show the state of the stack after executing each instruction.

The top element of the stack is shown on the left. Notice that we pushed E early because we need to subtract it from (B+C*D).

Stack machines are implemented by making the top portion of the stack internal to the processor. This is referred to as the stack depth. The rest of the stack is placed in memory. Thus, to access the top values that are within the stack depth, we do not have to access the memory. Obviously, we get better performance by increasing the stack depth.

Addressing Modes

We have examined the types of operands and operations that may be specified by machine instructions. Now we have to see how is the address of an operand specified, and how are the bits of an instruction organized to define the operand addresses and operation of that instruction

Addressing Modes: The most common addressing techniques are

• Immediate

• Direct

• Indirect

• Register

• Register Indirect

• Displacement

• Stack

All computer architectures provide more than one of these addressing modes.

The question arises as to how the control unit can determine which addressing mode is being used in a particular instruction. Several approaches are used. Often, different opcodes will use different addressing modes. Also, one or more bits in the instruction format can be used as a mode field. The value of the mode field determines which addressing mode is to be used.

What is the interpretation of effective address. In a system without virtual memory, the effective address will be either a main memory address or a register. In a virtual memory system, the effective address is a virtual address or a register. The actual mapping to a physical address is a function of the paging mechanism and is invisible to the programmer. To explain the addressing modes, we use the following notation:

A	=	contents of an address field in the instruction that refers to a memory
R	=	contents of an address field in the instruction that refers to a register
EA	=	actual (effective) address of the location containing the referenced operand
(X)	=	contents of location X

Immediate Addressing:

The simplest form of addressing is immediate addressing, in which the operand is actually present in the instruction: OPERAND = A This mode can be used to define and use constants or set initial values of variables. The advantage of immediate addressing is that no memory reference other than the instruction fetch is required to obtain the operand. The disadvantage is that the size of the

number is restricted to the size of the address field, which, in most instruction sets, is small compared with the world length.

Direct Addressing:

A very simple form of addressing is direct addressing, in which the address field contains the effective address of the operand:

EA = A

It requires only one memory reference and no special calculation.

Indirect Addressing:

With direct addressing, the length of the address field is usually less than the word length, thus limiting the address range. One solution is to have the address field refer to the address of a word in memory, which in turn contains a full-length address of the operand.

Register Addressing: Register addressing is similar to direct addressing. The only difference is that the address field refers to a register rather than a main memory address: EA = R

The advantages of register addressing are that only a small address field is needed in the instruction and no memory reference is required. The disadvantage of register addressing is that the address space is very limited.

The exact register location of the operand in case of Register Addressing Mode is shown in the Figure 34.4. Here, 'R' indicates a register where the operand is present.

Register Indirect Addressing:

Register indirect addressing is similar to indirect addressing, except that the address field refers to a register instead of a memory location. It requires only one memory reference and no special calculation.

EA = (R)

Register indirect addressing uses one less memory reference than indirect addressing. Because, the first information is available in a register which is nothing but a memory address. From that memory location, we use to get the data or information. In general, register access is much more faster than the memory access.

Diaplacement Addressing: A very powerful mode of addressing combines the capabilities of direct addressing and register indirect addressing, which is broadly categorized as displacement addressing: EA = A + (R) Displacement addressing requires that the instruction have two address fields, at least one of which is explicit. The value contained in one address field (value = A) is used directly. The other address field, or an implicit reference based on opcode, refers to a register whose contents are

added to A to produce the effective address. The general format of Displacement Addressing is shown in the Figure 4.6. Three of the most common use of displacement addressing are: • Relative addressing • Base-register addressing • Indexing

Relative Addressing: For relative addressing, the implicitly referenced register is the program counter (PC). That is, the current instruction address is added to the address field to produce the EA. Thus, the effective address is a displacement relative to the address of the instruction. Base-Register Addressing: The reference register contains a memory address, and the address field contains a displacement from that address. The register reference may be explicit or implicit. In some implementation, a single segment/base register is employed and is used implicitly. In others, the programmer may choose a register to hold the base address of a segment, and the instruction must reference it explicitly. Indexing: The address field references a main memory address, and the reference register contains a positive displacement from that address. In this case also the register reference is sometimes explicit and sometimes implicit.

UNIT 3:

Data Representation: Data types, Complements, Fixed Point Representation, Floating Point Representation.

Computer Arithmetic: Addition and subtraction, multiplication Algorithms, Division Algorithms, Floating–point Arithmetic operations. Decimal Arithmetic unit, Decimal Arithmetic operations.

BASIC COMPUTER DATA TYPES:

• Binary information in digital computers is stored in memory or processor registers.

• The data types found in the registers of digital computers may be classified as being one of the following categories: (1) numbers used in arithmetic computations, (2) letters of the alphabet used in data processing, and (3) other discrete symbols used for specific purposes. All types of data, except binary numbers, are represented in computer registers in binary-coded form.

• A number system of base or radix r is a system of that uses distinct symbols for r digits

• The decimal number system in everyday use employs radix 10 system. The 10 symbols are 0, 1, 2, 3, 4, 5, 6,7, 8, and 9; highest number being r-1

The binary number system uses the radix 2. The two digit symbols used are 0 and 1.

• The string of digits 101101 is interpreted to represent 1 x 25 + 0 x 24 + 1 x 23 + 1 x 22 + 0 x 21 + 1 x 20 = 45

• Besides the decimal and binary number systems,

• Octal (radix 8)- 0,1,2,3,4,5,6,7 and

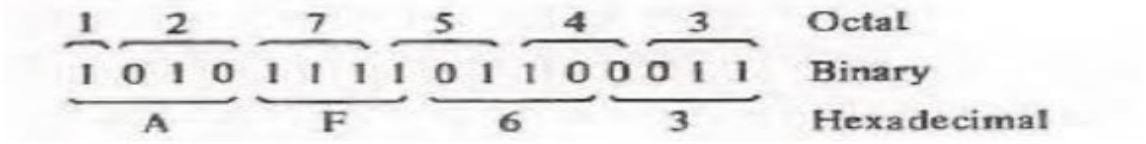
• Hexadecimal (radix 16) – 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F • Octal can be converted to decimal as follows (736.4)8 = 7 X 82 + 3 X 81 + 6 X 80 + 4X8-1 = 7X64 + 3X8 + 6X1 + 4/8 = (478.5) 10

Basic computer data types

- Hexadeciamal can be converted to decimal

$$(F3)_{16} = F \times 16 + 3$$
$$= 15 \times 16 + 3$$
$$= (243)_{10}$$

Octal and Hex can be obtained from Binary as shown below:



Complements:

• A binary code is a group of n bits that assume upto 2^n distinct combinations of 0s and 1s.

• A BCD code is a binary coded decimal i.e. binary coding decimal numbers.

• ASCII (American Standard Code for Information Interchange),

• which uses seven bits to code 128 characters is standard alphanumeric character code.

• Complements are used in digital computers for simplifying the subtraction operation and for logical manipulation. There are two types of complements

• For each base r system: the r's complement and the (r - 1)'s Complement

For binary base 2 system: the 2's complement and the 1's complement • For decimal base 10 system: the 10's complement and the 9's complement • The 9's complement of 546700 is 999999 - 546700 = 453299 • The 9's complement of 12389 is 99999 - 12389 = 87610 • The 1's complement of a binary number is formed by Changing 1's into 0's and 0's into 1's. • For example, the 1's complement of 1011001 is 0100110 and the 1's complement of 0001111 is 1110000.

The 10's complement of the decimal 2389 is 7610 + 1 = 7611 and is obtained by adding 1 to the 9's complement value.

• The 2's complement of binary 101100 is 010011 + 1 = 010100 and is obtained by adding 1 to the 1's complement value.

• For example, the 1's complement of 1011001 is 0100110 and the 1's complement of 0001111 is 1110000.

Solve: Find the 9's and 10's complement of 246700. Find the 1's and 2's complement of 1101100.

Fixed-Point Representation

• In computer binary systems it is customary to represent the sign with a bit placed in the leftmost position of the number.

• sign bit is equal to 0 for positive and to 1 for negative.

• a number may have a binary (or decimal) point.

• There are two ways of specifying the position of the binary point: by giving it a fixed position or by employing a floatingpoint representation.

• The fixed-point method assumes that the binary point is always. fixed in one position.

• The two positions most widely used are (1) a binary point in the extreme left of the register to make the stored number a fraction, and (2) a binary point in the extreme right of the register to make the stored number an integer. In either case, the binary point is not actually present.

Integer Representation for signed numbers

• signed-magnitude representation 1 0001110

• signed-1's complement representation 1 1110001

• signed-2's complement representation 1 1110010

Floating Point Representation:

The floating-point representation uses a second register to store a number that designates the position of the decimal point in the first register. • The floating-point representation of a number has two parts. The first part represents a signed, fixed-point number called the mantissa. The second part designates the position of the decimal (or binary) point and is called the exponent The fixedpoint mantissa may be a fraction or an integer. For example, • the decimal number +6132.789 is represented in floating-point with a fraction and an exponent as follows:

Fraction	**Exponent**
+0.6132789	**+04**

A floating-point binary number is represented in a similar manner except that it uses base 2 for the exponent.

• For example, the binary number +1001.11 is represented with an 8-bit fraction and 6-bit exponent as follows:

Fraction Exponent

01001110 00010

Sign-Magnitude

used in every day arithmetic calculations .

Left most bit is sign bit- **0 – positive 1 – negative**

• +18 = **00010010**

• -18 = **10010010**

• Problems

—**Need to consider both sign and magnitude in**

arithmetic

—**Two representations of zero (+0 and -0)**

Addition and Subtraction:

Normal binary addition

• Monitor sign bit for overflow

• So we only need addition and complement

Circuits

Assume:-

-magnitudes of two numbers as A and B.

when those sign numbers are added we have eight different conditions depending on the sign bits and operations performed.

SIGNED MAGNITUDE ADDITION AND SUBTRACTION

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

Case 1: addition

- **If signs are same:**
- 3+5 both numbers positive
- (-3) + (-5) both numbers negative -(3+5)
- **If signs differ:**
- (+3) + (-5) numbers differ in sign so, 3-5 **(results in subtraction operation)**
- (-3) + (+5) entin signs so, -(3-5)
- **(results in subtraction operation)**

Case 2: Subtraction

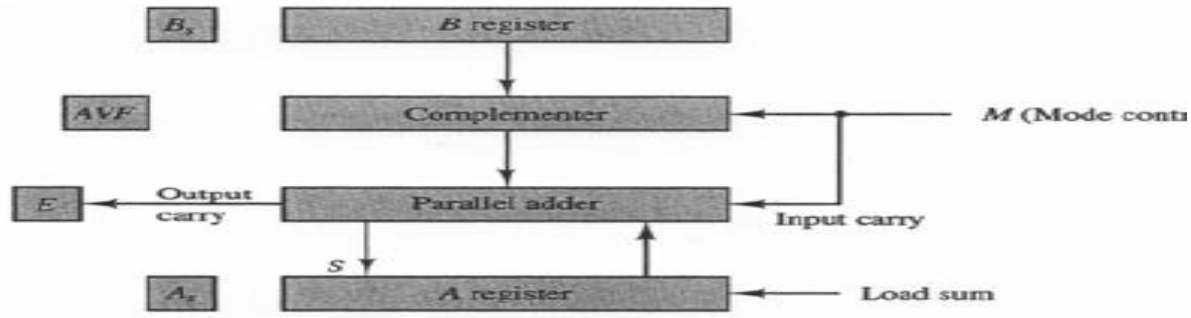
- **If signs are same:**
- 3-5 both numbers positive
- (-3) - (-5) both numbers negative -(3-5)
- **If signs differ:**
- (+3) - (-5) numbers differ in sign so, 3+5 **(results in addition operation)**
- (-3) -(+5) entin signs so, -(3+5)
- **(results in addition operation)**

Add / Subtract Signed-Magnitude

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

Forces zero to be positive

Hardware



- A_S Sign of A
- B_S Sign of B
- A_S & A Accumulator
- AVF Overflow bit for $A + B$
- E Output carry for parallel adder

ALGORITHM:

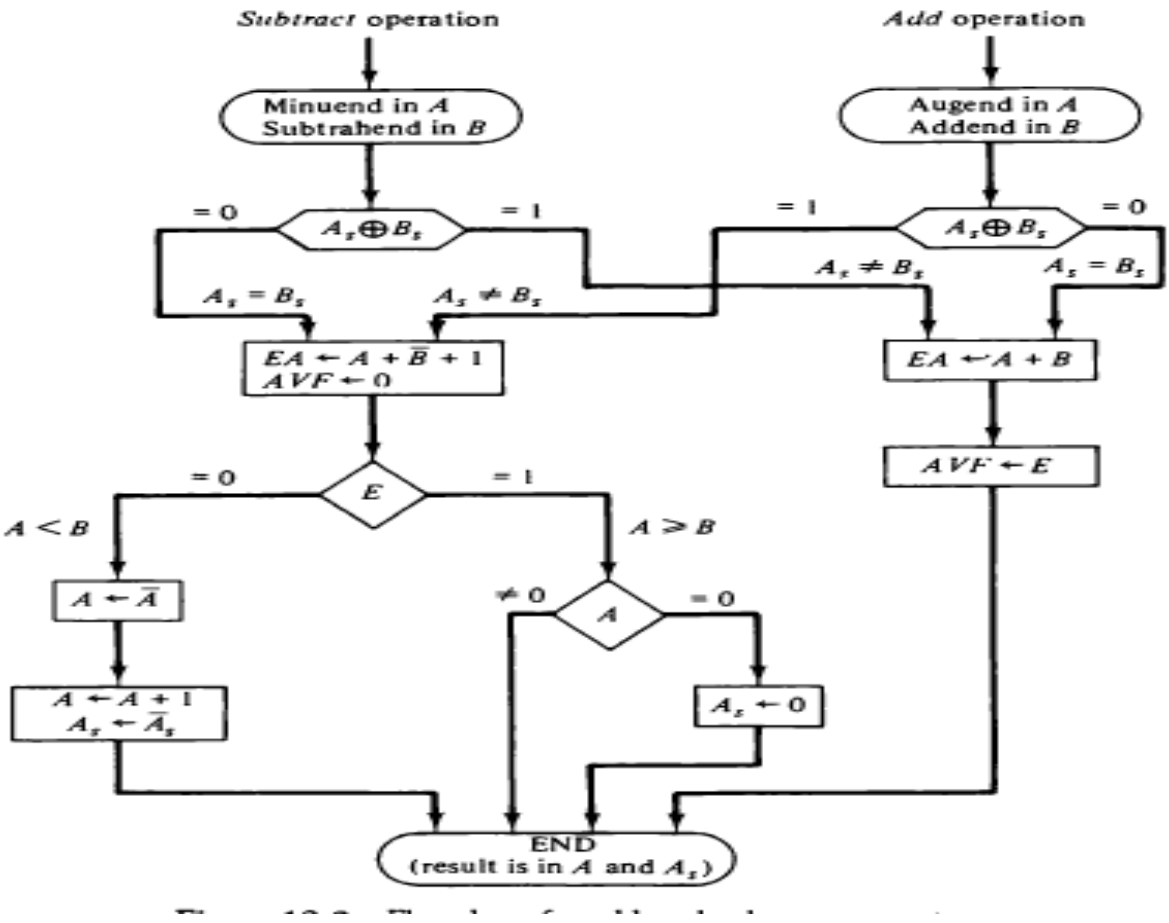
The two signs As and Bs are compared by EX-OR them. If result is 0 then As = Bs and if result is 1 the As ≠ Bs.

o For add operations if have same sign bits the magnitude must be added. For subtract operations different sign bits means magnitudes be added as well.

o E bit is carry bit after addition and moves to AVE overflow bit only at this state.

o If sign bits are different in add operations or the same in subtract operations the two magnitudes will be subtracted A – B. No overflow can occur here.

o After subtract if E=1 this means A>B and if E=0 then A<B. then here it is necessary to get 2's complement of A (by invert A then add 1) and sign of A is inverted only in this case.



SIGNED 2'S COMPLEMENT ADDITION AND SUBTRACTION:

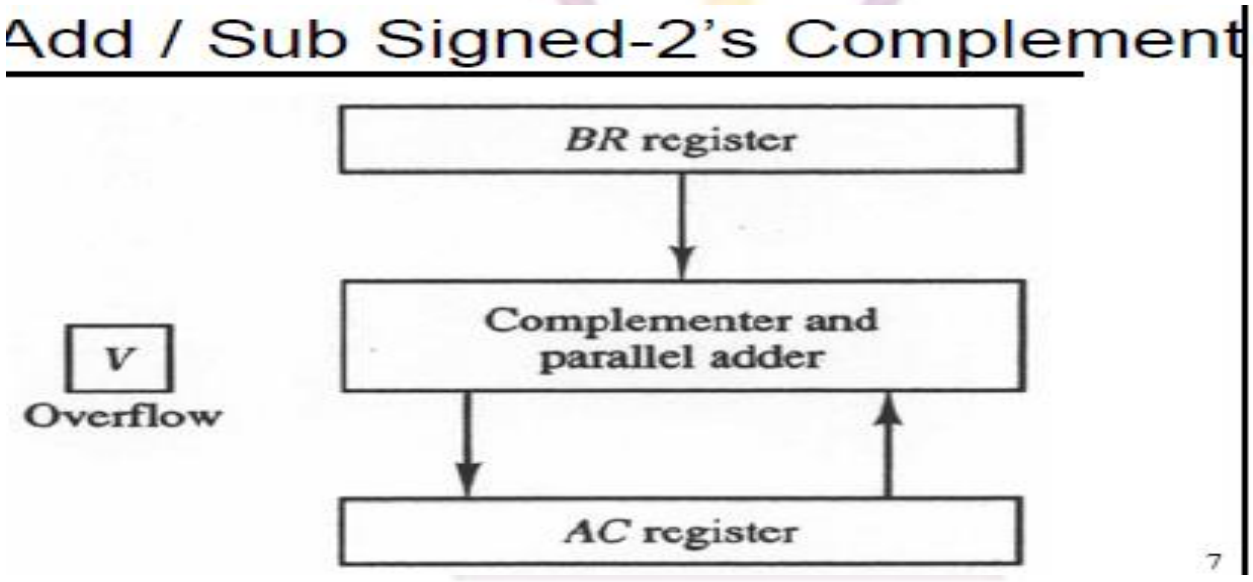
The left most bit in 2's complement represented binary number is the sign bit. If 0 the number is positive and if 1 then number is negative. If sign bit is 1 the entire number is represented in 2's complement.

The addition of two numbers represented in 2's complement is carried out by normal binary addition with carry discarded.

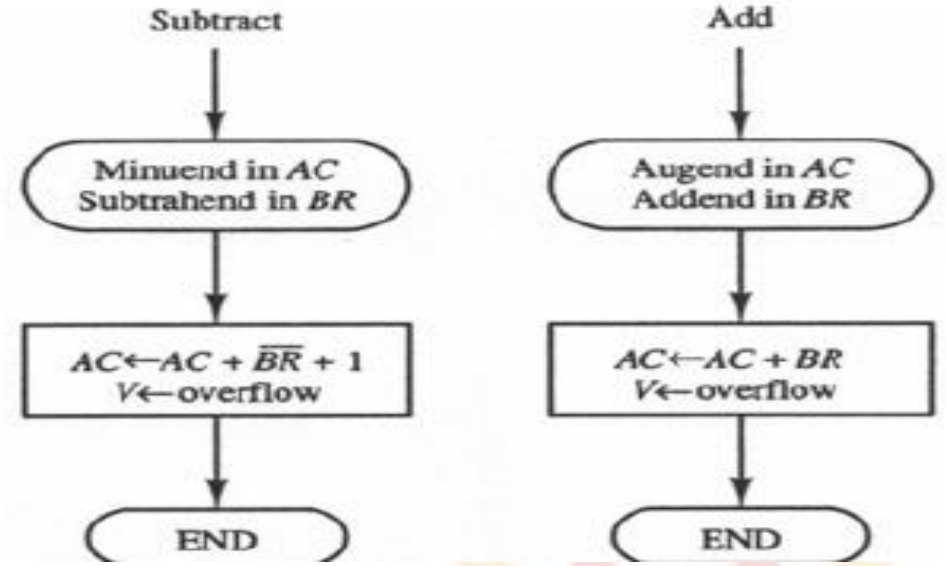
The subtraction is carried out by taking 2's complement (B) of subtrahend and adding it to minuend (A).

Overflow can be detected by inspecting last 2 carries out of addition by EX-OR them. If different then overflow is detected.

For addition simply implement add then see overflow. For subtract add 2's complement of B to A and watch overflow since the A and –B could be of same sign.



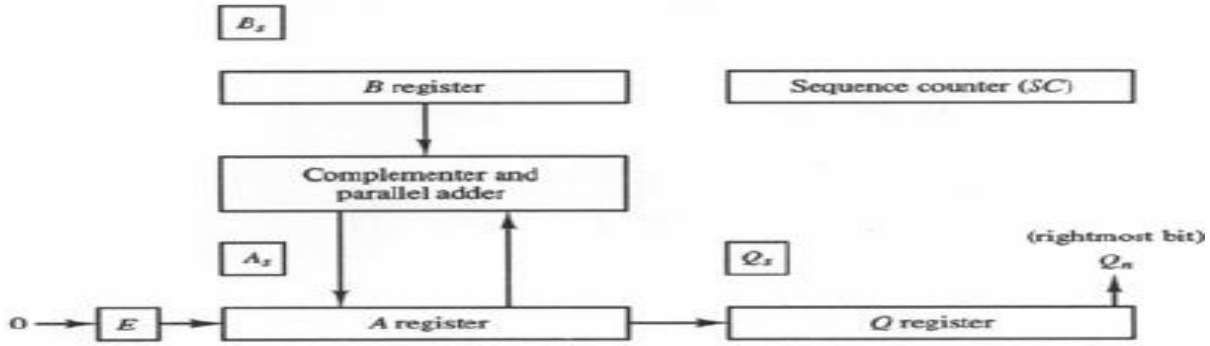
Algorithm



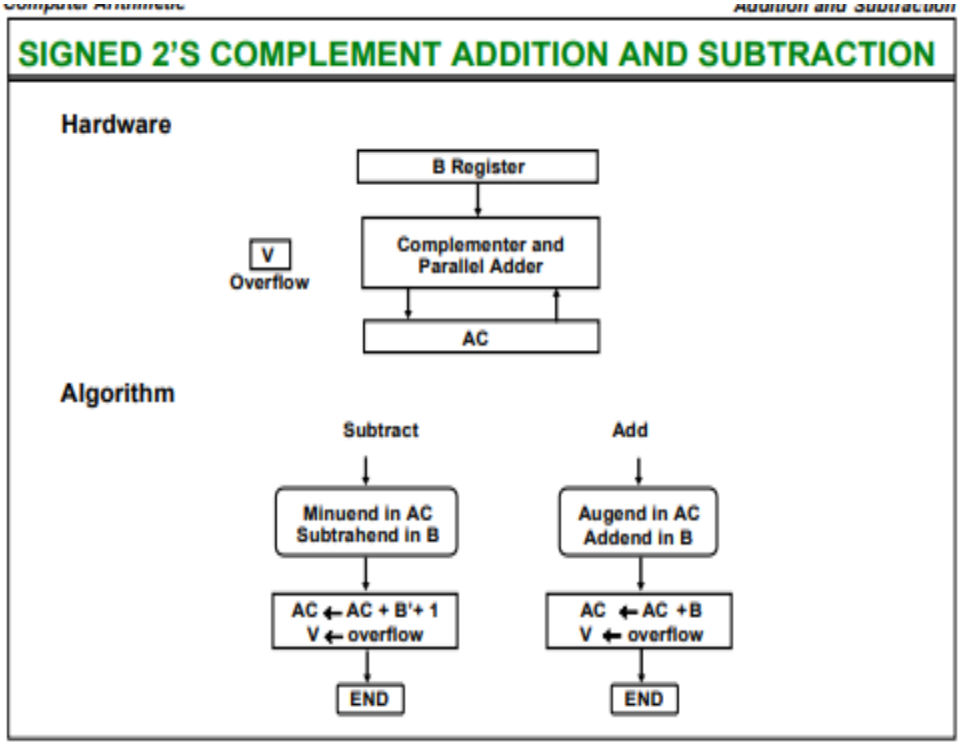
Multiply Signed-Magnitude

- Series of successive shift and add operations
- 23 10111 Multiplicand
  19 x 10011 Multiplier
  10111
  10111
  00000
  00000
  10111 +
  437 110110101 Product

Hardware



- Q multiplier
- B multiplicand
- A 0
- SC number of bits in multiplier
- E overflow bit for A
- Do SC times
  - If low-order bit of Q is 1
    - $A \leftarrow A + B$
  - Shift right EAQ
- Product is in AQ


SIGNED 2'S COMPLEMENT ADDITION AND SUBTRACTION
Hardware
B Register
Complementer and Parallel Adder
V
Overflow
AC
Algorithm
Subtract
Minuend in AC
Subtrahend in B
AC ← AC + B'+ 1
V ← overflow
END
Add
Augend in AC
Addend in B
AC ← AC +B
V ← overflow
END

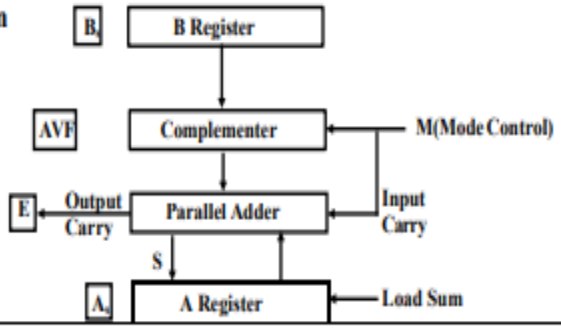
SIGNED MAGNITUDEADDITION AND SUBTRACTION

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



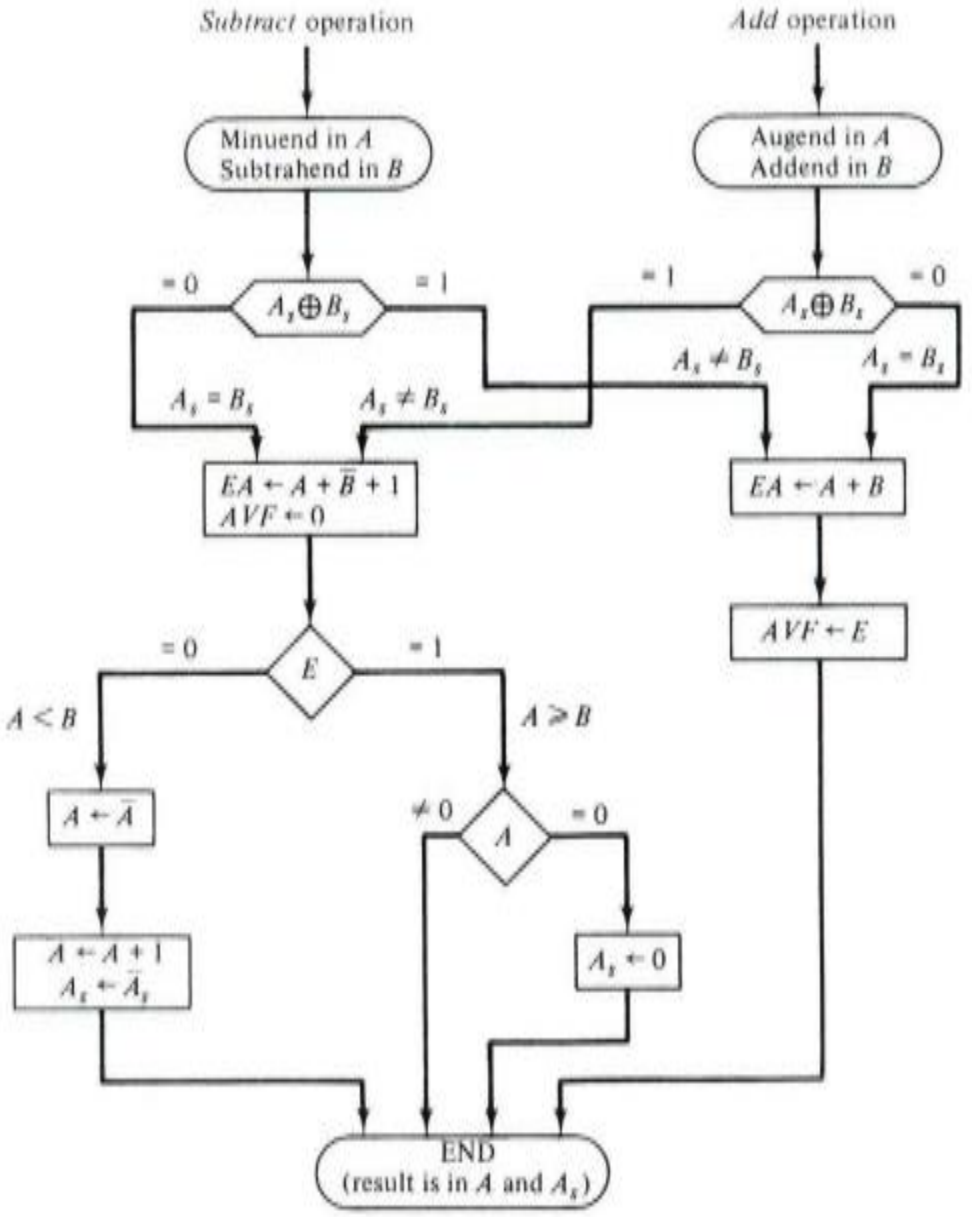
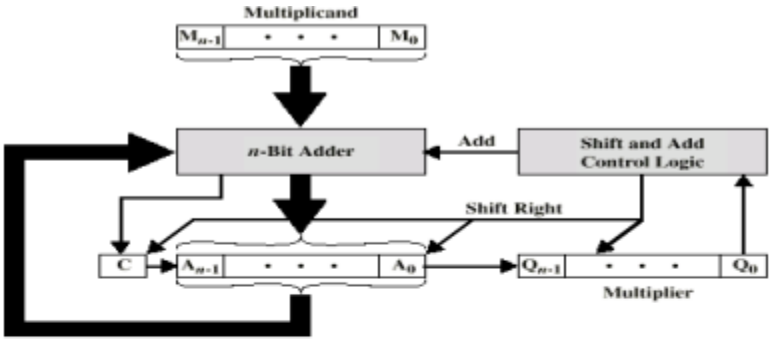


Figure 10-2 Flowchart for add and subtract operations.

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	

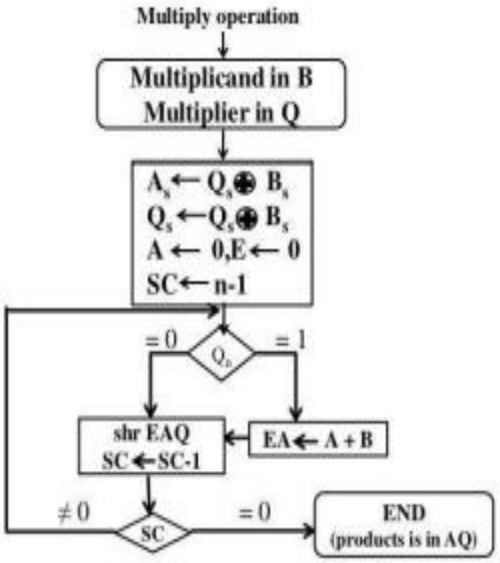


Figure: Flowchart for multiply operation.

Booth's algorithm :

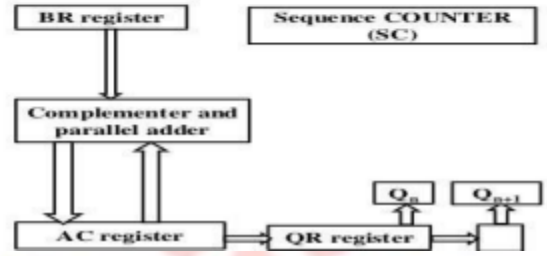
- Booth algorithm gives a procedure for multiplying binary integers in signed- 2's complement representation.

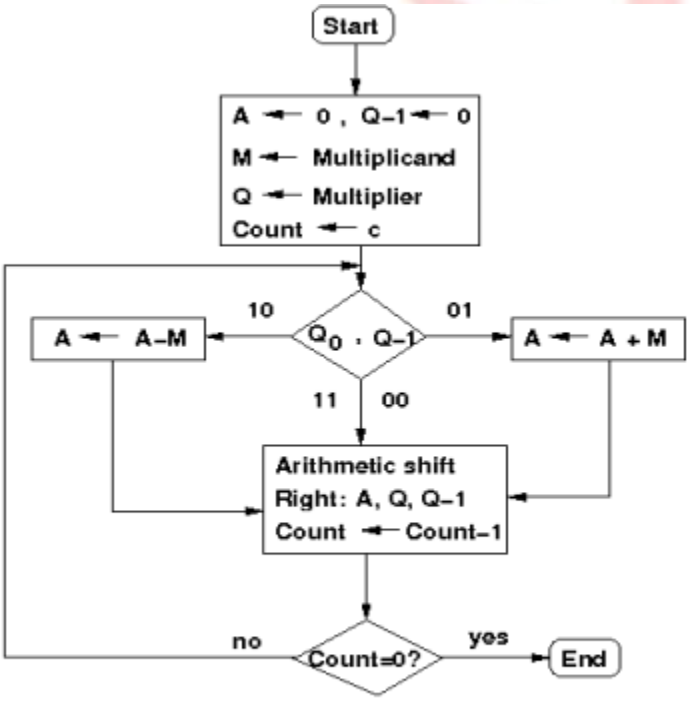
shifting, and a string of 1's in the multiplier from bit weight 2^k to weight 2^m can be treated as $2^{k+1} - 2^m$.

- 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

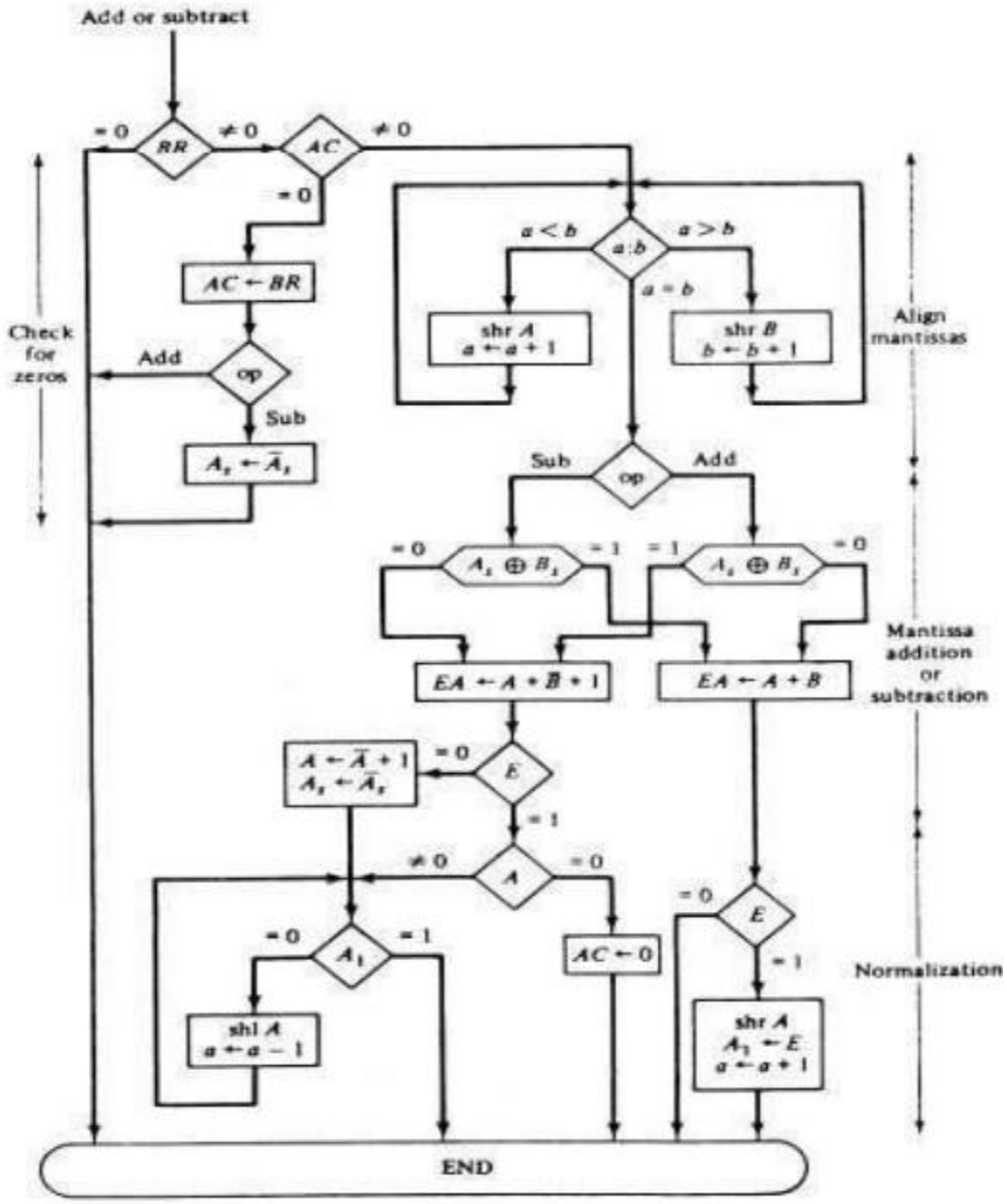




- As in all multiplication schemes, booth algorithm requires examination of the multiplier bits and shifting of partial product.

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.

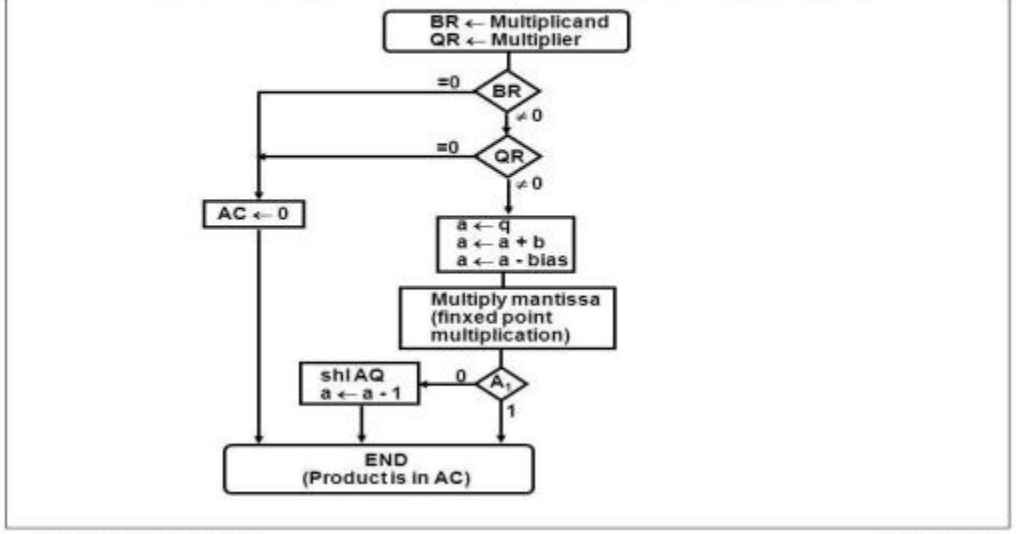
The devisor is compared with the five most significant bits of the dividend. Since the 5-bit number is smaller than B, we again repeat the same process. Now the 6-bit number is greater than B, so we place a 1 for the quotient bit in the sixth position above the dividend. Now we shift the divisor once 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.


Add or subtract
BR
AC
= 0
≠ 0
AC ← BR
Add
op
Sub
A_s ← Ā_s
Check for zeros
a < b
a : b
a > b
a = b
shr A
a ← a + 1
shr B
b ← b + 1
Align mantissas
Sub
op
Add
A_i ⊕ B_i
EA ← A + B + 1
EA ← A + B
A ← Ā + 1
A_s ← Ā_s
E
A
A_1
shl A
a ← a − 1
AC ← 0
shr A
A_1 ← E
a ← a + 1
Mantissa addition or subtraction
Normalization
END

Multiplication:

Computer Arithmetic 16 Floating Point Arithmetic

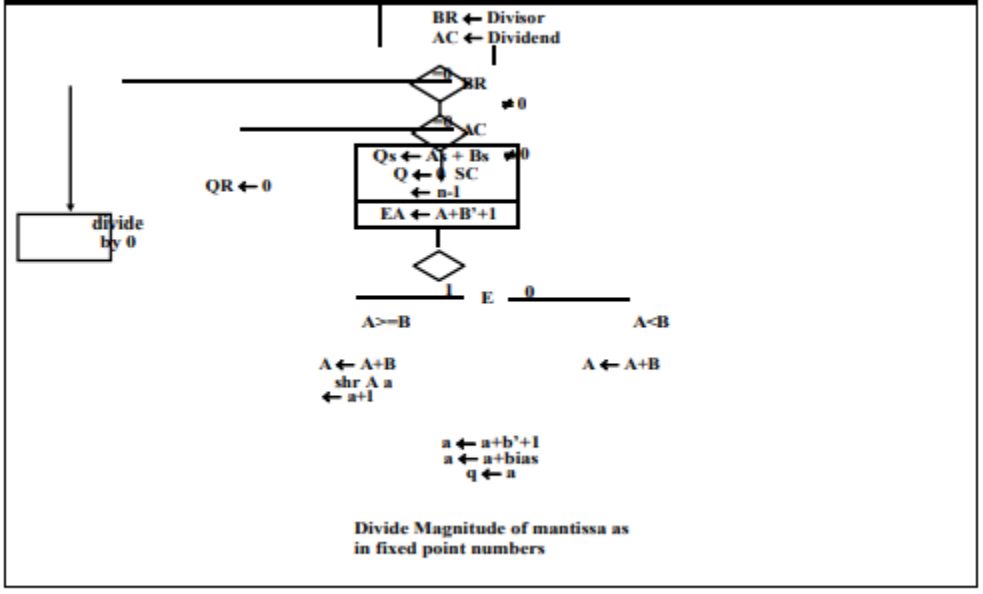
FLOATING POINT MULTIPLICATION



Computer Organization Prof. H. Yoon

Computer Arithmetic 17 Floating Point Arithmetic

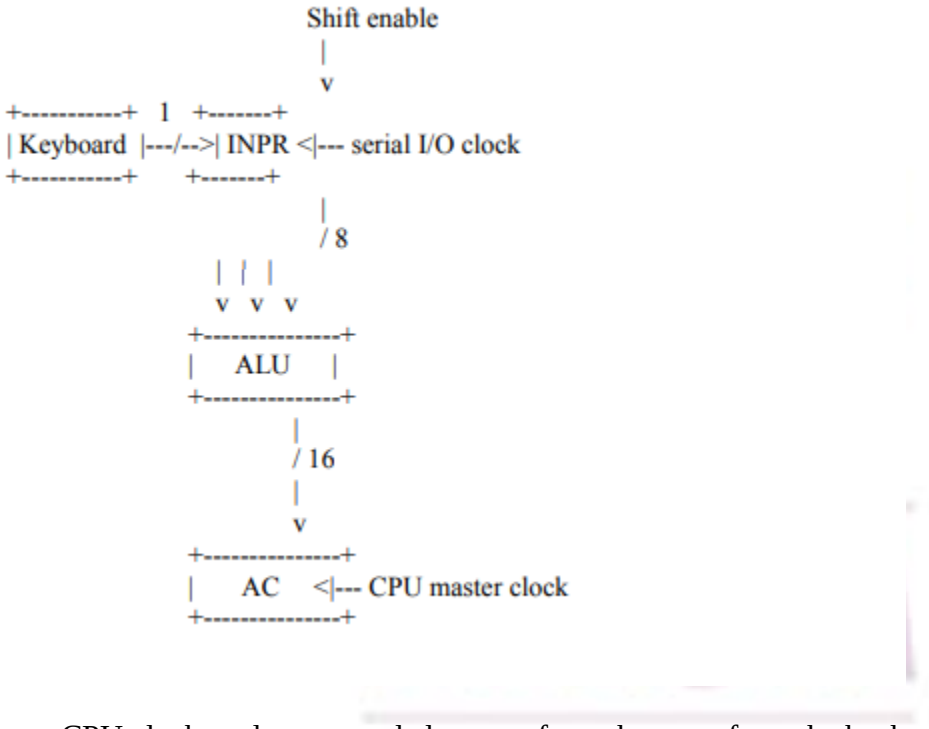
FLOATING POINT DIVISION



UNIT 4:

Input-Output Organization: Input-Output Interface, Asynchronous data transfer, Modes of Transfer, Priority Interrupt Direct memory Access.
Memory Organization: Memory Hierarchy, Main Memory, Auxiliary memory, Associate Memory, Cache Memory.

The Basic Computer I/O consists of a simple terminal with a keyboard and a printer/monitor. The keyboard is connected serially (1 data wire) to the INPR register. INPR is a shift register capable of shifting in external data from the keyboard one bit at a time. INPR outputs are connected in parallel to the ALU.



How many CPU clock cycles are needed to transfer a character from the keyboard to the INPR register? (tricky)

Are the clock pulses provided by the CPU master clock?

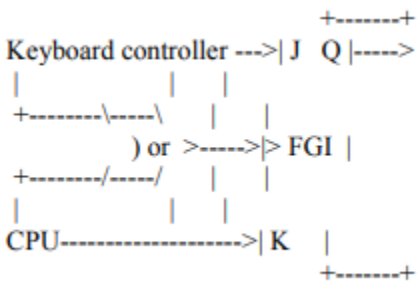
RS232, USB, Firewire are serial interfaces with their own clock independent of the CPU. (USB speed is independent of processor speed.)

- RS232: 115,200 kbps (some faster)
- USB: 11 mbps

- USB2: 480 mbps
- FW400: 400 mbps
- FW800: 800 mbps
- USB3: 4.8 gbps

OUTR inputs are connected to the bus in parallel, and the output is connected serially to the terminal. OUTR is another shift register, and the printer/monitor receives an end-bit during each clock pulse.

I/O Operations Since input and output devices are not under the full control of the CPU (I/O events are asynchronous), the CPU must somehow be told when an input device has new input ready to send, and an output device is ready to receive more output. The FGI flip-flop is set to 1 after a new character is shifted into INPR. This is done by the I/O interface, not by the control unit. This is an example of an asynchronous input event (not synchronized with or controlled by the CPU). The FGI flip-flop must be cleared after transferring the INPR to AC. This must be done as a microoperation controlled by the CU, so we must include it in the CU design. The FGO flip-flop is set to 1 by the I/O interface after the terminal has finished displaying the last character sent. It must be cleared by the CPU after transferring a character into OUTR. Since the keyboard controller only sets FGI and the CPU only clears it, a JK flip-flop is convenient:



How do we control the CK input on the FGI flip-flop? (Assume leading-edge triggering.) There are two common methods for detecting when I/O devices are ready, namely software polling and interrupts. These two methods are discussed in the following sections. Table 5-5 outlines the Basic Computer input-output instructions.

Interrupts To alleviate the problems of software polling, a hardware solution is needed. 108 Analogies to software polling in daily life tend to look rather silly. For example, imagine a teacher is analogous to a CPU, and the students are I/O devices. The students are working asynchronously, as the teacher walks around the room constantly asking each individual student "are you done yet?". What would be a better approach? With interrupts, the running program is not responsible for

checking the status of I/O devices. Instead, it simply does its own work, and assumes that I/O will take care of itself! When a device becomes ready, the CPU hardware initiates a branch to an I/O subprogram called an interrupt service routine (ISR), which handles the I/O transaction with the device. An interrupt can occur during any instruction cycle as long as interrupts are enabled. When the current instruction completes, the CPU interrupts the flow of the program, executes the ISR, and then resumes the program. The program itself is not involved and is in fact unaware that it has been interrupted. Figure 5-13 outlines the Basic Computer interrupt process. Interrupts can be globally enabled or disabled via the IEN flag (flip-flop). Some architectures have a separate ISR for each device. The Basic Computer has a single ISR that services both the input and output devices. If interrupts are enabled, then when either FGI or FGO gets set, the R flag also gets set. (R = FGI v FGO) This allows the system to easily check whether any I/O device needs service. Determining which one needs service can be done by the ISR. If R = 0, the CPU goes through a normal instruction cycle. If R = 1, the CPU branches to the ISR to process an I/O transaction. How much time does checking for interrupts add to the instruction cycle? Interrupts are usually disabled while the ISR is running, since it is difficult to make an ISR reentrant. (Callable while it is already in progress, such as a recursive function.) Hence, IEN and R are cleared as part of the interrupt cycle. IEN should be re-enabled by the ISR when it is finished. (In many architectures this is done by a special return instruction to ensure that interrupts are not enabled before the return is actually executed.) The Basic Computer interrupt cycle is shown in figure 5-13 (above). The Basic Computer interrupt cycle in detail: T0'T1'T2'(IEN)(FGI v FGO): R ← 1 109 RT0: AR ← 0, TR ← PC RT1: M[AR] ← TR, PC ← 0 RT2: PC ← PC + 1, IEN ← 0, R ← 0, SC ← 0 To enable the use of interrupts requires several steps: 1. Write an ISR 2. Install the ISR in memory at some arbitrary address X 3. Install the instruction "BUN X" at address 1 4. Enable interrupts with the ION instruction The sequence of events utilizing an interrupt to process keyboard input is as follows: 1. A character is typed 2. FGI ← 1 (same as with polling) 3. R ← 1, IEN ← 0 4. M[0] ← PC (store return address) 5. PC ← 1 (branch to interrupt vector) 6. BUN X (branch to ISR) 7. ISR checks FGI (found to be 1) 8. INP (AC ← INPR) 9. Character in AC is placed in a queue 10. ISR checks FGO (found to be 0) 11. ION 12. BUN 0 I Programs then read their input from a queue rather than directly from the input device. The ISR adds input to the queue as soon as it is typed, regardless of what code is running, and then returns to the running program.

Input-Output Interface Peripherals connected to a computer need special communication links for interfacing with CPU. In computer system, there are special hardware components between the CPU and peripherals to control or manage the input-output transfers. These components are called input-output interface units because they provide communication links between processor bus and peripherals. They provide a method for transferring information between internal system and input-output devices. Asynchronous Data Transfer We know that, the internal operations in individual unit of digital system are synchronized by means of clock pulse, means clock pulse is given to all registers within a unit, and all data transfer among internal registers occur simultaneously during

occurrence of clock pulse.Now, suppose any two units of digital system are designed independently such as CPU and I/O interface. And if the registers in the interface(I/O interface) share a common clock with CPU registers, then transfer between the two units is said to be synchronous.But in most cases, the internal timing in each unit is independent from each other in such a way that each uses its own private clock for its internal registers.In that case, the two units are said to be asynchronous to each other, and if data transfer occur between them this data transfer is said to be Asynchronous Data Transfer. But, the Asynchronous Data Transfer between two independent units requires that control signals be transmitted between the communicating units so that the time can be indicated at which they send data.

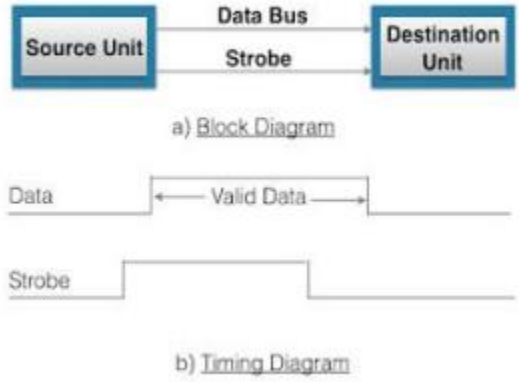
This asynchronous way of data transfer can be achieved by two methods: 1. One way is by means of strobe pulse which is supplied by one of the units to other unit.When transfer has to occur.This method is known as "Strobe Control". 2. Another method commonly used is to accompany each data item being transferred with a control signal that indicates the presence of data in the bus.The unit receiving the data item responds with another signal to acknowledge receipt of the data.This method of data transfer between two independent units is said to be "Handshaking". The strobe pulse and handshaking method of asynchronous data transfer are not restricted to I/O transfer.In fact, they are used extensively on numerous occasion requiring transfer of data between two independent units.So, here we consider the transmitting unit as source and receiving unit as destination. As an example: The CPU, is the source during an output or write transfer and is the destination unit during input or read transfer. And thus, the sequence of control during an asynchronous transfer depends on whether the transfer is initiated by the source or by the destination. So, while discussing each way of data transfer asynchronously we see the sequence of control in both terms when it is initiated by source or when it is initiated by destination.In this way, each way of data transfer, can be further divided into parts, source initiated and destination initiated. We can also specify, asynchronous transfer between two independent units by means of a timing diagram that shows the timing relationship that exists between the control and the data buses.

Now, we will discuss each method of asynchronous data transfer in detail one by one.

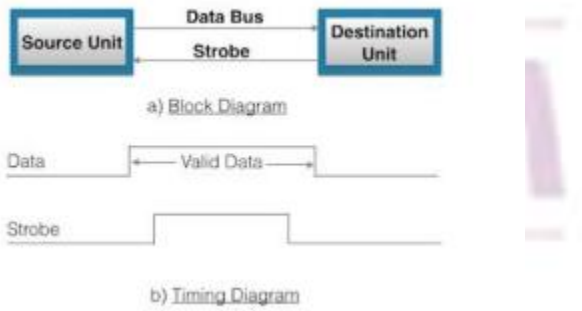
1. Strobe Control:

The Strobe Control method of asynchronous data transfer employs a single control line to time each transfer .This control line is also known as strobe and it may be achieved either by source or destination, depending on which initiate transfer. Source initiated strobe for data transfer:

The block diagram and timing diagram of strobe initiated by source unit is shown in figure below:



In block diagram we see that strobe is initiated by source, and as shown in timing diagram, the source unit first places the data on the data bus.After a brief delay to ensure that the data settle to a steady value, the source activates a strobe pulse.The information on data bus and strobe control signal remain in the active state for a sufficient period of time to allow the destination unit to receive the data.Actually, the destination unit, uses a falling edge of strobe control to transfer the contents of data bus to one of its internal registers.The source removes the data from the data bus after it disables its strobe pulse.New valid data will be available only after the strobe is enabled again. Destination-initiated strobe for data transfer: The block diagram and timing diagram of strobe initiated by destination is shown in figure below:



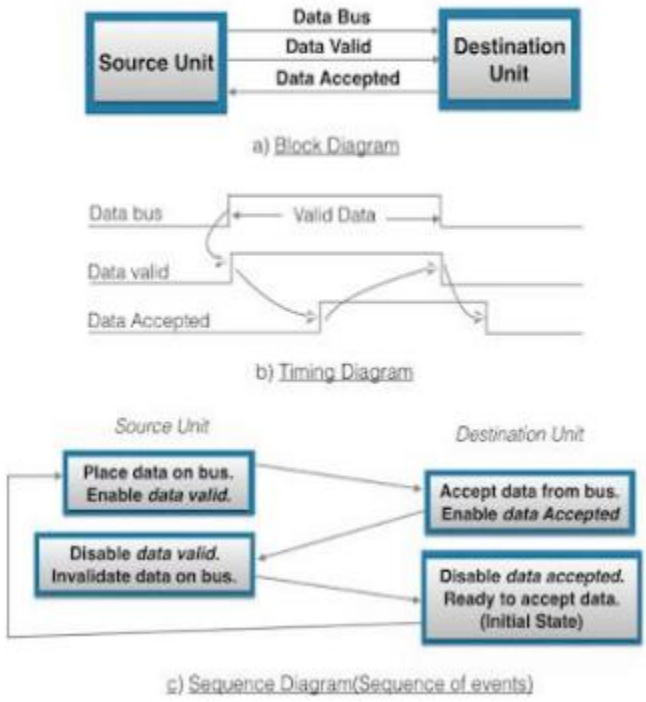
In block diagram, we see that, the strobe initiated by destination, and as shown in timing diagram, the destination unit first activates the strobe pulse, informing the source to provide the data.The source unit responds by placing the requested binary information on the data bus.The data must be valid and remain in the bus long enough for the destination unit to accept it.The falling edge of strobe pulse can be used again to trigger a destination register.The destination unit then disables the strobe.And source removes the data from data bus after a per determine time interval. Now, actually in computer, in the first case means in strobe initiated by source - the strobe may be a memory-write control signal from the CPU to a memory unit.The source, CPU, places the word on the data bus and

informs the memory unit, which is the destination, that this is a write operation. And in the second case i.e, in the strobe initiated by destination - the strobe may be a memory read control from the CPU to a memory unit.The destination, the CPU, initiates the read operation to inform the memory, which is a source unit, to place selected word into the data bus.

2. Handshaking: The disadvantage of strobe method is that source unit that initiates the transfer has no way of knowing whether the destination has actually received the data that was placed in the bus.Similarly, a destination unit that initiates the transfer has no way of knowing whether the source unit, has actually placed data on the bus. This problem can be solved by handshaking method. Hand shaking method introduce a second control signal line that provides a replay to the unit that initiates the transfer. In it, one control line is in the same direction as the data flow in the bus from the source to destination.It is used by source unit to inform the destination unit whether there are valid data in the bus.The other control line is in the other direction from destination to the source.It is used by the destination unit to inform the source whether it can accept data.And in it also, sequence of control depends on unit that initiate transfer.Means sequence of control depends whether transfer is initiated by source and destination.Sequence of control in both of them are described below:

Source initiated Handshaking:

The source initiated transfer using handshaking lines is shown in figure below:



In its block diagram, we se that two handshaking lines are "data valid", which is generated by the source unit, and "data accepted", generated by the destination unit. The timing diagram shows the timing relationship of exchange of signals between the two units.Means as shown in its timing diagram, the source initiates a transfer by placing data on the bus and enabling its data valid signal.The data accepted signal is then activated by destination unit after it accepts the data from the bus.The source unit then disable its data valid signal which invalidates the data on the bus.After this, the destination unit disables its data accepted signal and the system goes into initial state.The source unit does not send the next data item until after the destination unit shows its readiness to accept new data by disabling the data accepted signal. This sequence of events described in its sequence diagram, which shows the above sequence in which the system is present, at any given time.

Modes of I/O Data Transfer

Data transfer between the central unit and I/O devices can be handled in generally three types of modes which are given below:

1. Programmed I/O

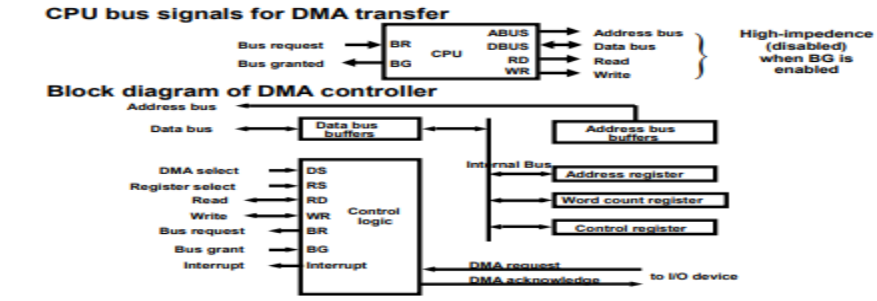
2. Interrupt Initiated I/O

3. Direct Memory Access

Programmed I/O Programmed I/O instructions are the result of I/O instructions written in computer program. Each data item transfer is initiated by the instruction in the program. Usually the program controls data transfer to and from CPU and peripheral. Transferring data under programmed I/O requires constant monitoring of the peripherals by the CPU. Interrupt Initiated I/O In the programmed I/O method the CPU stays in the program loop until the I/O unit indicates that it is ready for data transfer. This is time consuming process because it keeps the processor busy needlessly. This problem can be overcome by using interrupt initiated I/O. In this when the interface determines that the peripheral is ready for data transfer, it generates an interrupt. After receiving the interrupt signal, the CPU stops the task which it is processing and service the I/O transfer and then returns back to its previous processing task. Direct Memory Access Removing the CPU from the path and letting the peripheral device manage the memory buses directly would improve the speed of transfer. This technique is known as DMA. In this, the interface transfer data to and from the memory through memory bus. A DMA controller manages to transfer data between peripherals and memory unit. Many hardware systems use DMA such as disk drive controllers, graphic cards, network cards and sound cards etc. It is also used for intra chip data transfer in multicore processors. In DMA, CPU would initiate the transfer, do other operations while the transfer is in progress and receive an interrupt from the DMA controller when the transfer has been completed. Priority Interrupt A priority interrupt is a system which decides the priority at which various devices, which generates the interrupt signal at the same time, will be serviced by the CPU. The system has authority to decide which conditions are allowed to interrupt the CPU, while some other interrupt is being serviced. Generally, devices with high speed transfer such as magnetic disks are given high priority and slow devices such as keyboards are given low priority. When two or more devices interrupt the computer simultaneously, the computer services the device with the higher priority first.

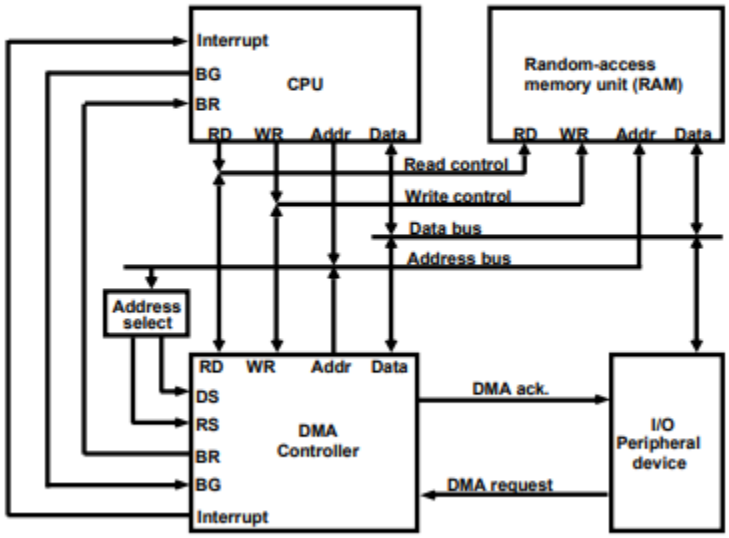
DIRECT MEMORY ACCESS

Block of data transfer from high speed devices, Drum, Disk, Tape



* DMA controller - Interface which allows I/O transfer directly between Memory and Device, freeing CPU for other tasks
* CPU initializes DMA Controller by sending memory address and the block size(number of words)

DMA TRANSFER



MEMORY ORGANIZATION

• RAM composed of a large number of (2M) of addressable locations, each of which stores a w-bit word.

• RAM operates as follows: first the address of the target location to be accessed is transferred via the address bus to the RAM's address buffer.

• The address is then processed by the address decoder, which selects the required location in the storage cell unit.

• If a read operation is requested, the contents of the addressed location are transferred from the storage cell unit to the data buffer and from there to the data bus.

• If a write operation is requested, the word to be stored is transferred from the data bus to the selected location in the stored unitThe storage unit is made up of many identical 1-bit memory cells and their Interconnections. In each line connected to the storage cell unit, we can expect to

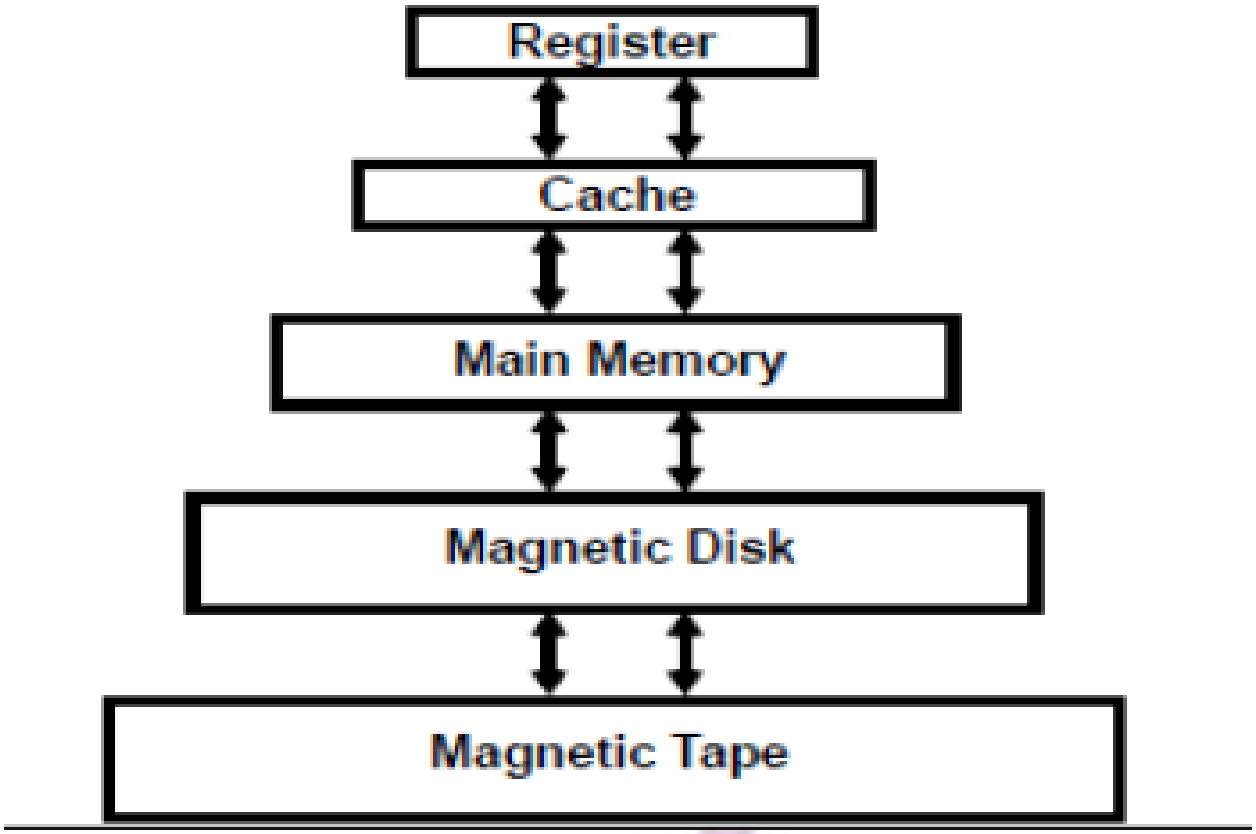
• find a driver that acts as either an amplifier or a transducer of physical signals. Organization

• assume that each word is stored in a single track and that each access results In the transfer of a block of words.

• The address of the data to be accessed is applied to the address decoder, whose output determines the track to be used and the location of the desired block of Information within the track.

• the track address determines the particular read-write head to be selected.The selected head is moved into position to transfer data to of from the target track. A track position indicator generates the address of the block that isCurrently passing the read-write head.

• The generated address is compared with the block address produced by the address decoder.The selected head is enabled and the data transfer between the storage track and the memory data buffer register begins.



• The read-write head is disabled when a complete block information has been transferred

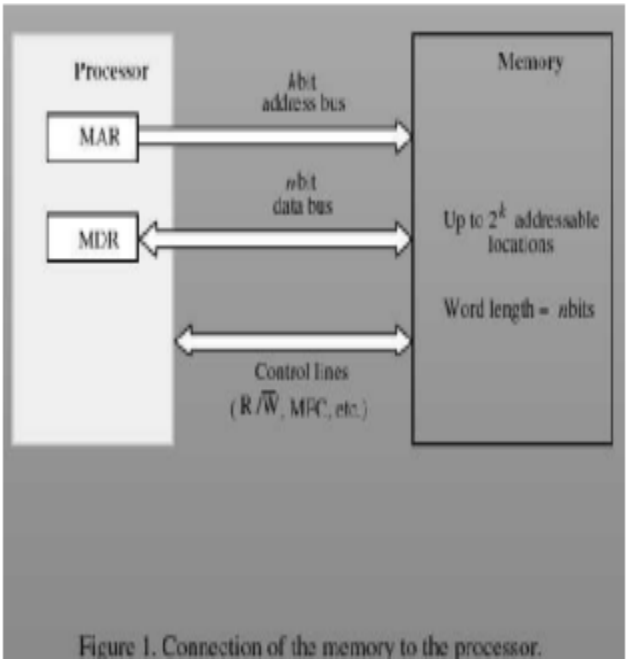


Figure 1. Connection of the memory to the processor.

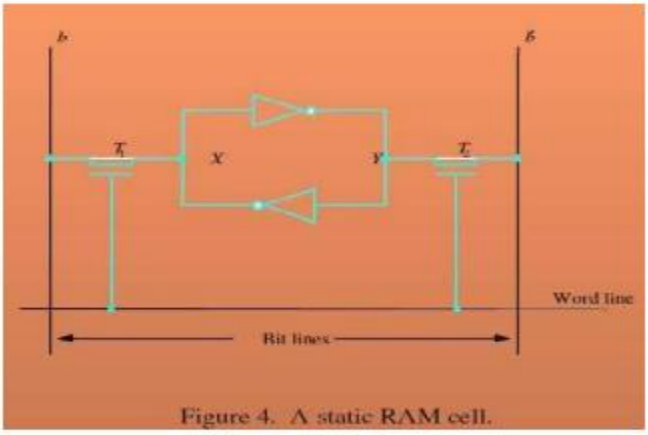


Figure 4. A static RAM cell.

Static memories (RAM)

Circuits capable of retaining their state as long as power is applied

Static RAM(SRAM)

volatile

DRAMS:

Charge on a capacitor

Needs —Refreshing

Synchronous DRAMs

Synchronized with a clock signal

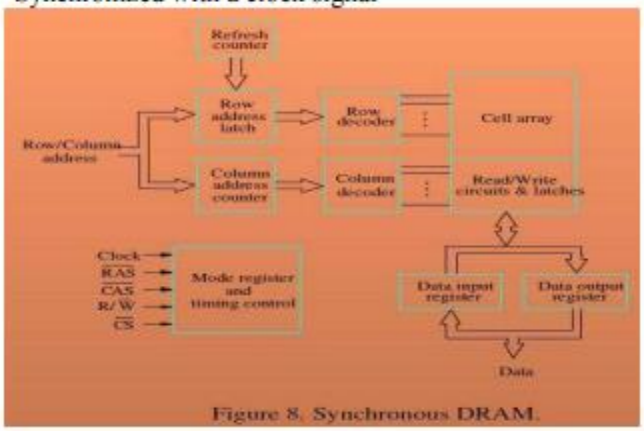


Figure 8. Synchronous DRAM.

Memory system considerations

– Cost
– Speed
– Power dissipation
– Size of chip

Principle of locality:

Temporal locality (locality in time): If an item is referenced, it will tend to be referenced again soon.

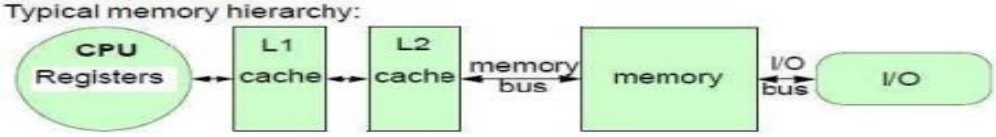
Spatial locality (locality in space): If an item is referenced, items whose addresses are close by will tend to be referenced soon.

Sequentiality (subset of spatial locality).

The principle of locality can be exploited implementing the memory of computer as a *memory hierarchy*, taking advantage of all types of memories.

Method: The level closer to processor (the fastest) is a subset of any level further away, and all the data is stored at the lowest level (the slowest).

Typical memory hierarchy:



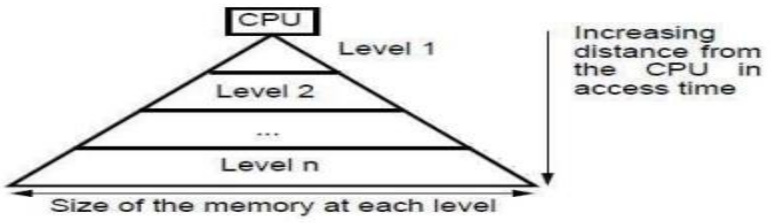
Level	1	2	3	4
Named as	Registers	Cache	memory	disk storage
Typical size	<1 KB	< 4 MB	<2 GB	>2GB
Access time (ns)	2 - 5	3 - 10	80 - 400	5'000'000
Bandwidth(MB/sec)	4000 - 32'000	000 - 5000	400 - 2000	4 - 32
Managed by	Compiler	I lardware	Operating system	Operating system / user

Cache Memories

– Speed of the main memory is very low in comparison with the speed of processor – For good performance, the processor cannot spend much time of its time waiting to access instructions and data in main memory. – Important to device a scheme that reduces the time to ace the information – An efficient solution is to use fast cache memory When a cache is full and a memory word 101 that is not in the cache is referenced, the cache control hardware must decide which block should be removed to create space for the new block that contain the referenced word.

The basics of Caches

" The caches are organized on basis of *blocks*, the smallest amount of data which can be copied between two adjacent levels at a time.
" If data requested by the processor is present in some block in the upper level,
it is called a *hit*.
" If data is not found in the upper level, the request is called a *miss* and the data is retrieved from the lower level in the hierarchy.
" The fraction of memory accesses found in the upper level is called a *hit ratio*.
" The storage, which takes advantage of locality of accesses is called a *cache*



Amdahl's Law about overall speedup:

$$\text{Speedup} = \frac{1}{(1 - \text{fraction of time cache can be used}) + \frac{\text{fraction of time cache can be used}}{\text{Speedup using cache}}}$$

Alternatively, CPU stalls can be considered[1]:

CPU execution time = (CPU clock cycles + Memory stall cycles) × Clock cycle

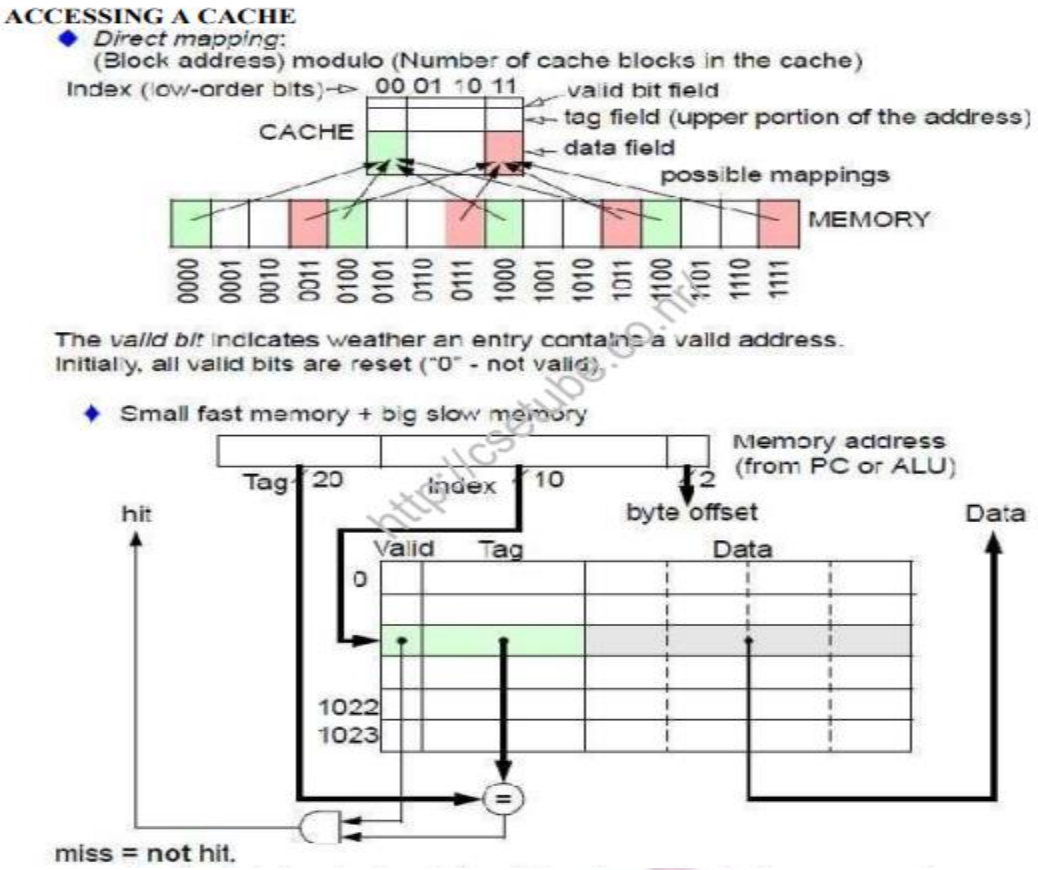
The number of memory stall cycles depends:

Memory stall cycles = IC × Memory references per instruction × Miss rate × Miss penalty

Size	Instruction cache	Data cache	
1 KB	3.06%	24.61%	13.34%
4 KB	1.76%	15.94%	7.24%
16 KB	0.64%	6.47%	2.87%
64 KB	0.15%	3.77%	1.35%

Table: Direct mapped cache, 32-byte blocks, SPEC92, DECstation 5000.

1. Here is assumed that CPU clock cycles include the time to handle a cache hit, and the CPU is stalled during a cache miss.



Virtual memory It a computer system technique which gives an application program the impression that it has contiguous working memory (an address space), while in fact it may be physically fragmented and may even overflow on to disk storage. Virtual memory provides two primary functions: 1. Each process has its own address space, thereby not required to be relocated nor required to use relative addressing mode. 2. Each process sees one contiguous block of free memory upon launch. Fragmentation is hidden.

Auxiliary Memory

Devices that provide backup storage are called auxiliary memory. For example: Magnetic

disks and tapes are commonly used auxiliary devices. Other devices used as auxiliary memory are

magnetic drums, magnetic bubble memory and optical disks.

It is not directly accessible to the CPU, and is accessed using the Input/Output channels.

Cache Memory

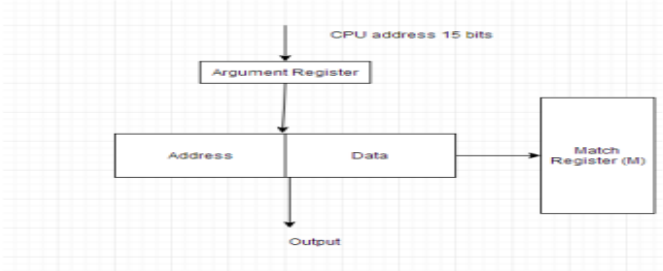
The data or contents of the main memory that are used again and again by CPU, are stored in

the cache memory so that we can easily access that data in shorter time.

Whenever the CPU needs to access memory, it first checks the cache memory. If the data is not found in cache memory then the CPU moves onto the main memory. It also transfers block of recent data into the cache and keeps on deleting the old data in cache to accomodate the new one.

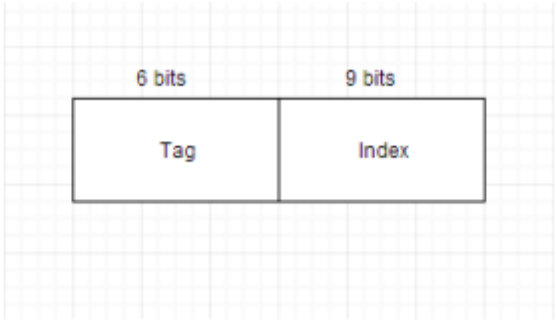
Memory Mapping and Concept of Virtual Memory:

The transformation of data from main memory to cache memory is called mapping. There are 3 main types of mapping: Associative Mapping• Direct Mapping• Set Associative Mapping• Associative Mapping The associative memory stores both address and data. The address value of 15 bits is 5 digit octal numbers and data is of 12 bits word in 4 digit octal number. A CPU address of 15 bits is placed in argument register and the associative memory is searched for matching address.



Direct Mapping

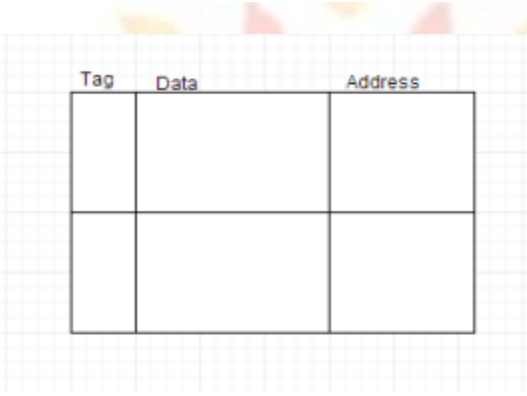
The CPU address of 15 bits is divided into 2 fields. In this the 9 least significant bits constitute the **index** field and the remaining 6 bits constitute the **tag** field. The number of bits in index field is equal to the number of address bits required to access cache memory.



Set Associative Mapping

The disadvantage of direct mapping is that two words with same index address can't reside in cache memory at the same time. This problem can be overcome by set associative mapping.

In this we can store two or more words of memory under the same index address. Each data word is stored together with its tag and this forms a set.



Replacement Algorithms

Data is continuously replaced with new data in the cache memory using replacement algorithms. Following are the 2 replacement algorithms used:

- FIFO - First in First out. Oldest item is replaced with the latest item.
- LRU - Least Recently Used. Item which is least recently used by CPU is removed.

Writing in to cache and cache Initialization:

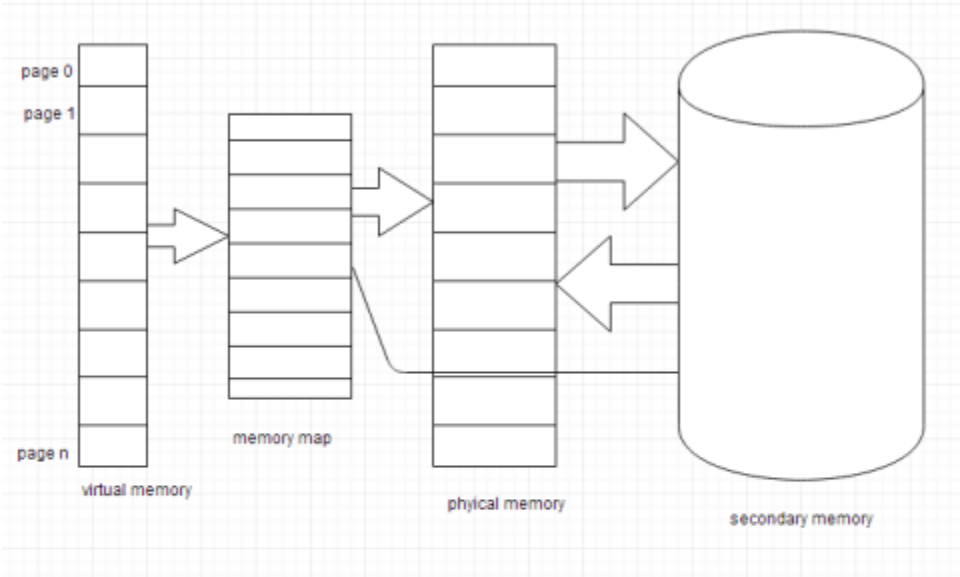
The benefit of write-through to main memory is that it simplifies the design of the computer system. With write-through, the main memory always has an up-to-date copy of the line. So when a read is done, main memory can always reply with the requested data.

If write-back is used, sometimes the up-to-date data is in a processor cache, and sometimes it is in main memory. If the data is in a processor cache, then that processor must stop main memory from replying to the read request, because the main memory might have a stale copy of the data. This is more complicated than write-through.

Virtual Memory:

Virtual memory is the separation of logical memory from physical memory. This separation provides large virtual memory for programmers when only small physical memory is available.

Virtual memory is used to give programmers the illusion that they have a very large memory even though the computer has a small main memory. It makes the task of programming easier because the programmer no longer needs to worry about the amount of physical memory available.



Address mapping using pages:

- The table implementation of the address mapping is simplified if the information in the address space. And the memory space is each divided into groups of fixed size.
- Moreover, 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.
- Also, 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 the figure.

UNIT 5:

Reduced Instruction Set Computer: CISC Characteristics, RISC Characteristics.

Pipeline and Vector Processing: Parallel Processing, Pipelining, Arithmetic Pipeline, Instruction Pipeline, RISC Pipeline, Vector Processing, Array Processor.

MultiProcessors: Characteristics of Multiprocessors, Interconnection Structures, Interprocessor arbitration, Interprocessor communication and synchronization, cache Coherence.

Reduced Instruction Set Computer:

Reduced Instruction Set Architecture (RISC) –
The main idea behind this is to make hardware simpler by using an instruction set composed of a few basic steps for loading, evaluating, and storing operations just like a load command will load data, a store command will store the data.
Complex Instruction Set Architecture (CISC) –
The main idea is that a single instruction will do all loading, evaluating, and storing operations just like a multiplication command will do stuff like loading data, evaluating, and storing it, hence it's complex.
Earlier when programming was done using assembly language, a need was felt to make instruction do more tasks because programming in assembly was tedious and error-prone due to which CISC architecture evolved but with the uprise of high-level language dependency on assembly reduced RISC architecture prevailed.

Characteristic of RISC –
1. Simpler instruction, hence simple instruction decoding.
2. Instruction comes undersize of one word.
3. Instruction takes a single clock cycle to get executed.
4. More general-purpose registers.
5. Simple Addressing Modes.
6. Fewer Data types.
7. A pipeline can be achieved.

Characteristic of CISC –
1. Complex instruction, hence complex instruction decoding.
2. Instructions are larger than one-word size.
3. Instruction may take more than a single clock cycle to get executed.
4. Less number of general-purpose registers as operations get performed in memory itself.
5. Complex Addressing Modes.
6. More Data types.

Example – Suppose we have to add two 8-bit numbers:
- **CISC approach:** There will be a single command or instruction for this like ADD which will perform the task.
- **RISC approach:** Here programmer will write the first load command to load data in registers then it will use a suitable operator and then it will store the result in the desired location.

So, add operation is divided into parts i.e. load, operate, store due to which RISC programs are longer and require more memory to get stored but require fewer transistors due to less complex command.

Difference – RISC and CISC

RISC	**CISC**
Focus on software	Focus on hardware

RISC	**CISC**
Uses only Hardwired control unit	Uses both hardwired and microprogrammed control unit
Transistors are used for more registers	Transistors are used for storing complex Instructions
Fixed sized instructions	Variable sized instructions
Can perform only Register to Register Arithmetic operations	Can perform REG to REG or REG to MEM or MEM to MEM
Requires more number of registers	Requires less number of registers
Code size is large	Code size is small
An instruction executed in a single clock cycle	Instruction takes more than one clock cycle
An instruction fit in one word	Instructions are larger than the size of one word

Both approaches try to increase the CPU performance

- **RISC:** Reduce the cycles per instruction at the cost of the number of instructions per program.
- **CISC:** The CISC approach attempts to minimize the number of instructions per program but at the cost of an increase in the number of cycles per instruction.

Parallel processing

Execution of Concurrent Events in the computing process to achieve faster Computational Speed

Levels of Parallel Processing

- Job or Program level

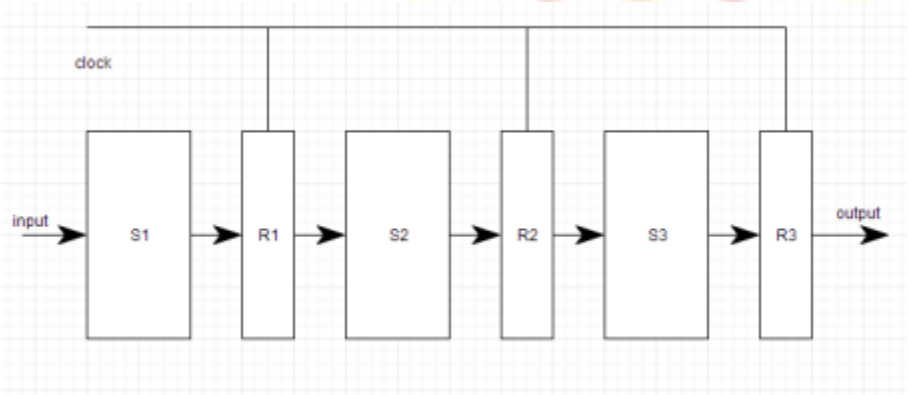
- Task or Procedure level

- Inter-Instruction level

- Intra-Instruction level

PARALLEL COMPUTERS Architectural Classification Flynn's classification » Based on the multiplicity of Instruction Streams and Data Streams » Instruction Stream Sequence of Instructions read from memory » Data Stream Operations performed on the data in the processor.

What is 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



Pipeline system is like the modern day assembly line setup in factories. For example in a car manufacturing industry, huge assembly lines are setup and at each point, there are robotic arms to perform a certain task, and then the car moves on ahead to the next arm. 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. Advantages of Pipelining 1. The cycle time of the processor is reduced. 2. It increases the throughput of the system 3. It makes the system reliable. Disadvantages of Pipelining 1. The design of pipelined processor is complex and costly to manufacture. 2. The instruction latency is more.

Vector(Array) Processing There is a class of computational problems that are beyond the capabilities of a conventional computer. These problems require vast number of computations on multiple data items, that will take a conventional computer(with scalar processor) days or even weeks to complete. Such complex instructions, which operates on multiple data at the same time, requires a better way of instruction execution, which was achieved by Vector processors. Scalar CPUs can manipulate one or two data items at a time, which is not very efficient. Also, simple instructions like ADD A to B, and store into C are not practically efficient. Addresses are used to point to the memory location where the data to be operated will be found, which leads to added overhead of data lookup. So until the data is found, the CPU would be sitting ideal, which is a big performance issue. Hence, the concept of Instruction Pipeline comes into picture, in which the instruction passes through several sub-units in turn.

These sub-units perform various independent functions, for example: the first one decodes the instruction, the second sub-unit fetches the data and the thirdsub-unit performs the math itself. Therefore, while the data is fetched for one instruction, CPU does not sit idle, it rather works on decoding the next instruction set, ending up working like an assembly line. Vector processor, not only use Instruction pipeline, but it also pipelines the data, working on multiple data at the same time. A normal scalar processor instruction would be ADD A, B, which leads to addition of two operands, but what if we can instruct the processor to ADD a group of numbers(from 0 to n memory location) to another group of numbers(lets say, n to k memory location). This can be achieved by vector processors. In vector processor a single instruction, can ask for multiple data operations, which saves time, as instruction is decoded once, and then it keeps on operating on different data items.

Applications of Vector Processors

Computer with vector processing capabilities are in demand in specialized applications. The following are some areas where vector processing is used:

1. Petroleum exploration.

2. Medical diagnosis.

3. Data analysis.

4. Weather forecasting.

5. Aerodynamics and space flight simulations.

6. Image processing.

A parallel processing system is able to perform concurrent data processing to achieve faster execution time

• The system may have two or more ALUs and be able to execute two or more instructions at the same time

• Also, the system may have two or more processors operating concurrently

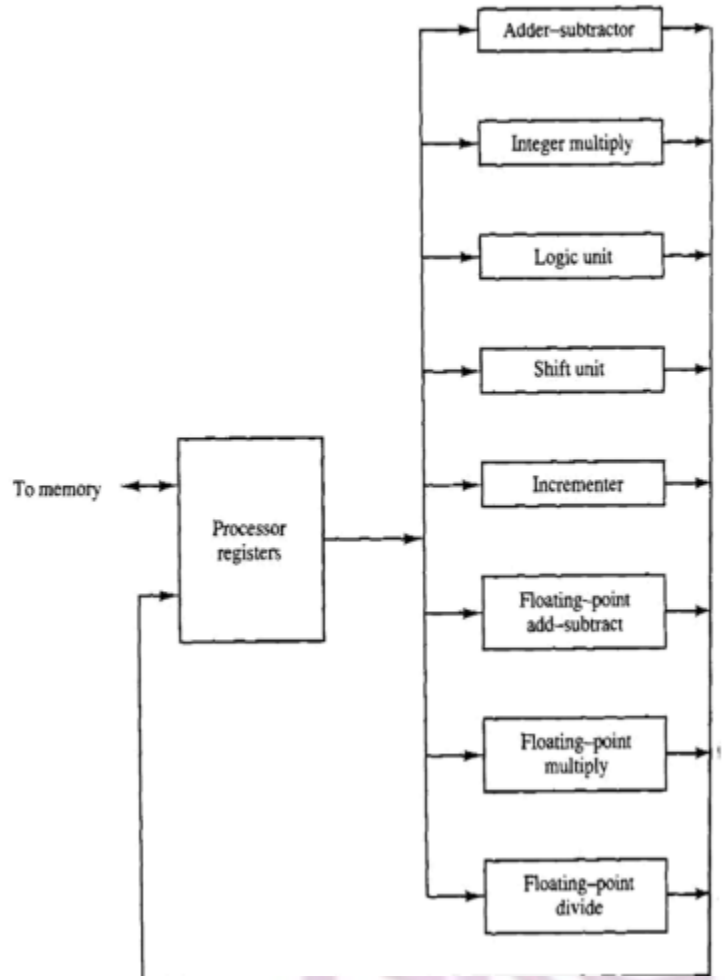
• Goal is to increase the throughput – the amount of processing that can be accomplished during a given interval of time

• Parallel processing increases the amount of hardware required

• Example: the ALU can be separated into three units and the operands diverted to each unit under the supervision of a control unit

• All units are independent of each other • A multifunctional organization is usually associated with a complex control unit to coordinate all the activities among the various components

Figure 9-1 Processor with multiple functional units.



• Parallel processing can be classified from:

oThe internal organization of the processors

o The interconnection structure between processors

o The flow of information through the system

o The number of instructions and data items that are manipulated simultaneously

• The sequence of instructions read from memory is the instruction stream

• The operations performed on the data in the processor is the data stream

• Parallel processing may occur in the instruction stream, the data stream, or both Computer classification:

o Single instruction stream, single data stream – SISD

o Single instruction stream, multiple data stream – SIMD

o Multiple instruction stream, single data stream – MISD

o Multiple instruction stream, multiple data stream – MIMD

• SISD – Instructions are executed sequentially. Parallel processing may be achieved by means of multiple functional units or by pipeline processing

• SIMD – Includes multiple processing units with a single control unit. All processors receive the same instruction, but operate on different data.

• MIMD – A computer system capable of processing several programs at the same time.

• We will consider parallel processing under the following main topics:

PIPELINING

• Pipelining is a technique of decomposing a sequential process into sub operations, with each sub process being executed in a special dedicated segment that operates concurrently with all other segments

• Each segment performs partial processing dictated by the way the task is partitioned

• The result obtained from the computation in each segment is transferred to the next segment in the pipeline

• The final result is obtained after the data have passed through all segments

• Can imagine that each segment consists of an input register followed by an combinational circuit

• A clock is applied to all registers after enough time has elapsed to perform all segment activity

• The information flows through the pipeline one step at a time

• Example: Ai * Bi + Ci

for i = 1, 2, 3, …, 7

The suboperations performed in each segment are:

R1 ← Ai

, R2 ← Bi

R3 ← R1 * R2, R4 ← Ci

R5 ← R3 + R4

Figure 9-2 Example of pipeline processing.

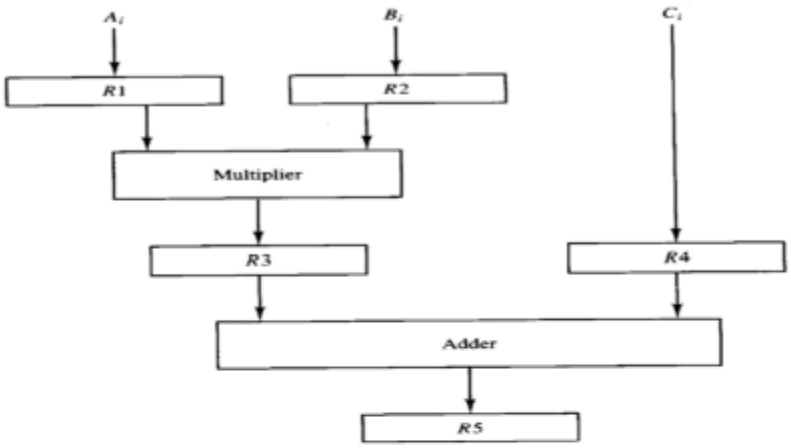


TABLE 9-1 Content of Registers in Pipeline Example

Clock Pulse Number	Segment 1		Segment 2		Segment 3
	R1	R2	R3	R4	R5
1	A_1	B_1	—	—	—
2	A_2	B_2	$A_1 * B_1$	C_1	—
3	A_3	B_3	$A_2 * B_2$	C_2	$A_1 * B_1 + C_1$
4	A_4	B_4	$A_3 * B_3$	C_3	$A_2 * B_2 + C_2$
5	A_5	B_5	$A_4 * B_4$	C_4	$A_3 * B_3 + C_3$
6	A_6	B_6	$A_5 * B_5$	C_5	$A_4 * B_4 + C_4$
7	A_7	B_7	$A_6 * B_6$	C_6	$A_5 * B_5 + C_5$
8	—	—	$A_7 * B_7$	C_7	$A_6 * B_6 + C_6$
9	—	—	—	—	$A_7 * B_7 + C_7$

Any operation that can be decomposed into a sequence of suboperations of about the same complexity can be implemented by a pipeline processor

• The technique is efficient for those applications that need to repeat the same task many time with different sets of data

• A task is the total operation performed going through all segments of a pipeline

• The behavior of a pipeline can be illustrated with a space-time diagram

• This shows the segment utilization as a function of time

• Once the pipeline is full, it takes only one clock period to obtain an output

Figure 9-4 Space-time diagram for pipeline.

Segment:	1	2	3	4	5	6	7	8	9	→ Clock cycles
1	T_1	T_2	T_3	T_4	T_5	T_6				
2		T_1	T_2	T_3	T_4	T_5	T_6			
3			T_1	T_2	T_3	T_4	T_5	T_6		
4				T_1	T_2	T_3	T_4	T_5	T_6	

Figure 9-2 Example of pipeline processing.

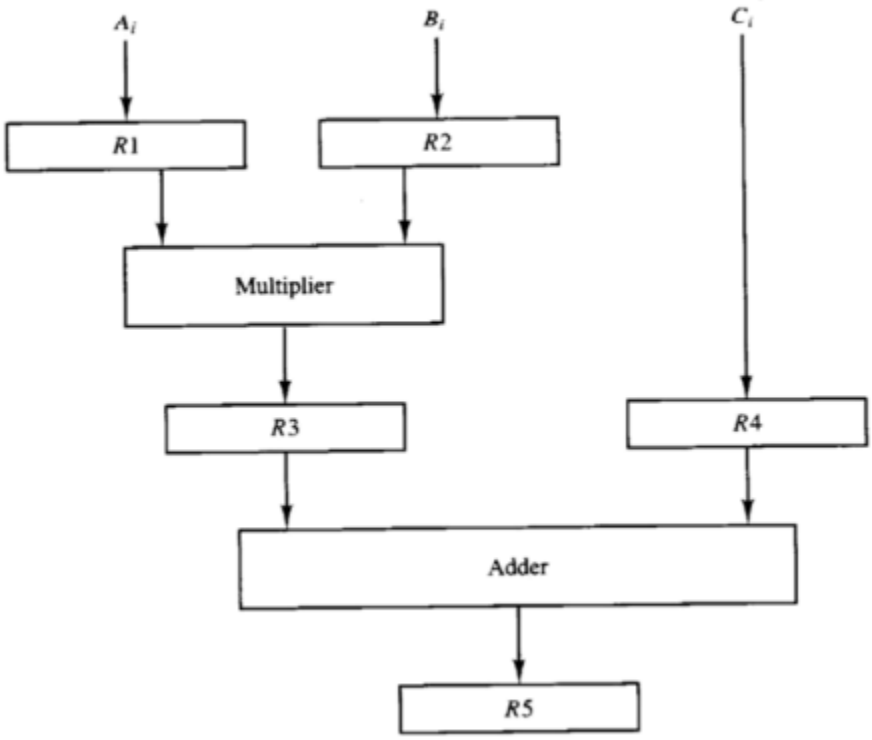


TABLE 9-1 Content of Registers in Pipeline Example

Clock Pulse Number	Segment 1		Segment 2		Segment 3
	R1	R2	R3	R4	R5
1	A_1	B_1	—	—	—
2	A_2	B_2	$A_1 * B_1$	C_1	—
3	A_3	B_3	$A_2 * B_2$	C_2	$A_1 * B_1 + C_1$
4	A_4	B_4	$A_3 * B_3$	C_3	$A_2 * B_2 + C_2$
5	A_5	B_5	$A_4 * B_4$	C_4	$A_3 * B_3 + C_3$
6	A_6	B_6	$A_5 * B_5$	C_5	$A_4 * B_4 + C_4$
7	A_7	B_7	$A_6 * B_6$	C_6	$A_5 * B_5 + C_5$
8	—	—	$A_7 * B_7$	C_7	$A_6 * B_6 + C_6$
9	—	—	—	—	$A_7 * B_7 + C_7$

Arithmetic Pipeline

• Pipeline arithmetic units are usually found in very high speed computers

• They are used to implement floating-point operations, multiplication of fixed-point numbers, and similar computations encountered in scientific problems

• Example for floating-point addition and subtraction

• Inputs are two normalized floating-point binary numbers

X = A x 2a

Y = B x 2b

• A and B are two fractions that represent the mantissas

• a and b are the exponents

Instruction Pipeline

• An instruction pipeline reads consecutive instructions from memory while previous instructions are being executed in other segments

• This causes the instruction fetch and execute phases to overlap and perform simultaneous operations

• If a branch out of sequence occurs, the pipeline must be emptied and all the instructions that have been read from memory after the branch instruction mustbe discarded

• Consider a computer with an instruction fetch unit and an instruction execution unit forming a two segment pipeline

• A FIFO buffer can be used for the fetch segment

• Thus, an instruction stream can be placed in a queue, waiting for decoding and processing by the execution segment

• This reduces the average access time to memory for reading instructions

• Whenever there is space in the buffer, the control unit initiates the next instruction fetch phase

• The following steps are needed to process each instruction:

o Fetch the instruction from memory

o Decode the instruction

o Calculate the effective address

o Fetch the operands from memory

o Execute the instruction

o Store the result in the proper place

Up to four suboperations in the instruction cycle can overlap and up to four different instructions can be in progress of being processed at the same time

• It is assumed that the processor has separate instruction and data memories

• Reasons for the pipeline to deviate from its normal operation are:

o Resource conflicts caused by access to memory by two segments at the same time.

o Data dependency conflicts arise when an instruction depends on the result of a previous instruction, but his result is not yet available.

Types of Multiprocessors

There are mainly two types of multiprocessors i.e. symmetric and asymmetric multiprocessors. Details about them are as follows −

Symmetric Multiprocessors

In these types of systems, each processor contains a similar copy of the operating system and they all communicate with each other. All the processors are in a peer to peer relationship i.e. no master - slave relationship exists between them.

An example of the symmetric multiprocessing system is the Encore version of Unix for the Multimax Computer.

Asymmetric Multiprocessors

In asymmetric systems, each processor is given a predefined task. There is a master processor that gives instruction to all the other processors. Asymmetric multiprocessor system contains a master slave relationship.

Asymmetric multiprocessor was the only type of multiprocessor available before symmetric multiprocessors were created. Now also, this is the cheaper option.

Advantages of Multiprocessor Systems

There are multiple advantages to multiprocessor systems. Some of these are −

More reliable Systems

In a multiprocessor system, even if one processor fails, the system will not halt. This ability to continue working despite hardware failure is known as graceful degradation. For example: If there are 5 processors in a multiprocessor system and one of them fails, then also 4 processors are still working. So the system only becomes slower and does not ground to a halt.

Enhanced Throughput

If multiple processors are working in tandem, then the throughput of the system increases i.e. number of processes getting executed per unit of time increase. If there are N processors then the throughput increases by an amount just under N.

More Economic Systems

Multiprocessor systems are cheaper than single processor systems in the long run because they share the data storage, peripheral devices, power supplies etc. If there are multiple processes that share data, it is better to schedule them on multiprocessor systems with shared data than have different computer systems with multiple copies of the data.

Disadvantages of Multiprocessor Systems

There are some disadvantages as well to multiprocessor systems. Some of these are:

Increased Expense

Even though multiprocessor systems are cheaper in the long run than using multiple computer systems, still they are quite expensive. It is much cheaper to buy a simple single processor system than a multiprocessor system.

Complicated Operating System Required

There are multiple processors in a multiprocessor system that share peripherals, memory etc

Characteristics of Multiprocessor

There are the major characteristics of multiprocessors are as follows −

- **Parallel Computing** − This involves the simultaneous application of multiple processors. These processors are developed using a single architecture to execute a common task. In general, processors are identical and they work together in such a way that the users are under the impression that they are the only users of the system. In reality, however, many users are accessing the system at a given time.
- **Distributed Computing** − This involves the usage of a network of processors. Each processor in this network can be considered as a computer in its own right and have the capability to solve a problem. These processors are heterogeneous, and generally, one task is allocated to a single processor.
- **Supercomputing** − This involves the usage of the fastest machines to resolve big and computationally complex problems. In the past, supercomputing machines were vector computers but at present, vector or parallel computing is accepted by most people.
- **Pipelining** − This is a method wherein a specific task is divided into several subtasks that must be performed in a sequence. The functional units help in performing each subtask. The units are attached serially and all the units work simultaneously.
- **Vector Computing** − It involves the usage of vector processors, wherein operations such as 'multiplication' are divided into many steps and are then applied to a stream of operands ("vectors").
- **Systolic** − This is similar to pipelining, but units are not arranged in a linear order. The steps in systolic are normally small and more in number and performed in a lockstep manner. This is more frequently applied in special-purpose hardware such as image or signal processors.

Interconnection structures :

The processors must be able to share a set of main memory modules & I/O devices in a multiprocessor system. This sharing capability can be provided through interconnection structures. The interconnection structure that are commonly used can be given as follows –

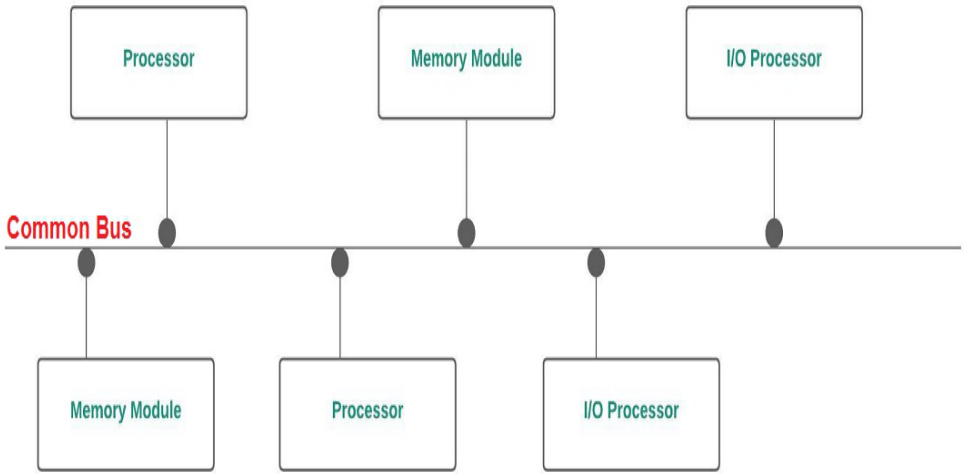
1. Time-shared / Common Bus
2. Cross bar Switch
3. Multiport Memory
4. Multistage Switching Network (Covered in 2nd part)
5. Hypercube System

In this article, we will cover Time shared / Common Bus in detail.

1. Time-shared / Common Bus (Interconnection structure in Multiprocessor System) :

In a multiprocessor system, the time shared bus interconnection provides a common

communication path connecting all the functional units like processor, I/O processor, memory unit etc. The figure below shows the multiple processors with common communication path (single bus).



Single-Bus Multiprocessor Organization

To communicate with any functional unit, processor needs the bus to transfer the data. To do so, the processor first need to see that whether the bus is available / not by checking the status of the bus. If the bus is used by some other functional unit, the status is busy, else free.

A processor can use bus only when the bus is free. The sender processor puts the address of the destination on the bus & the destination unit identifies it. In order to communicate with any functional unit, a command is issued to tell that unit, what work is to be done. The other processors at that time will be either busy in internal operations or will sit free, waiting to get bus.
We can use a bus controller to resolve conflicts, if any. (Bus controller can set priority of different functional units)
This Single-Bus Multiprocessor Organization is easiest to reconfigure & is simple. This interconnection structure contains only passive elements. The bus interfaces of sender & receiver units controls the transfer operation here.
To decide the access to common bus without conflicts, methods such as static & fixed priorities, First-In-Out (FIFO) queues & daisy chains can be used.

Advantages –

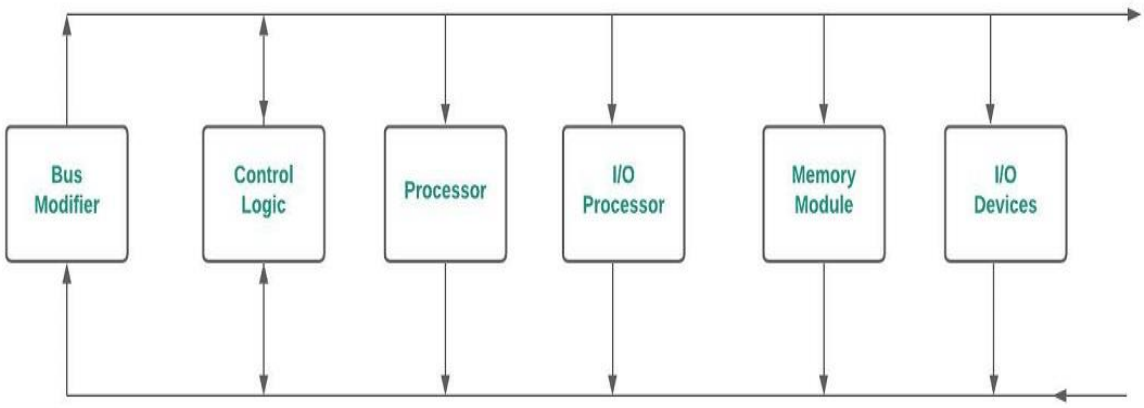
- Inexpensive as no extra hardware is required such as switch.
- Simple & easy to configure as the functional units are directly connected to the bus .

Disadvantages –

- Major fight with this kind of configuration is that if malfunctioning occurs in any of the bus interface circuits, complete system will fail.
- **Decreased throughput** —
  At a time, only one processor can communicate with any other functional unit.

- **Increased arbitration logic** —
  As the number of processors & memory unit increases, the bus contention problem increases.

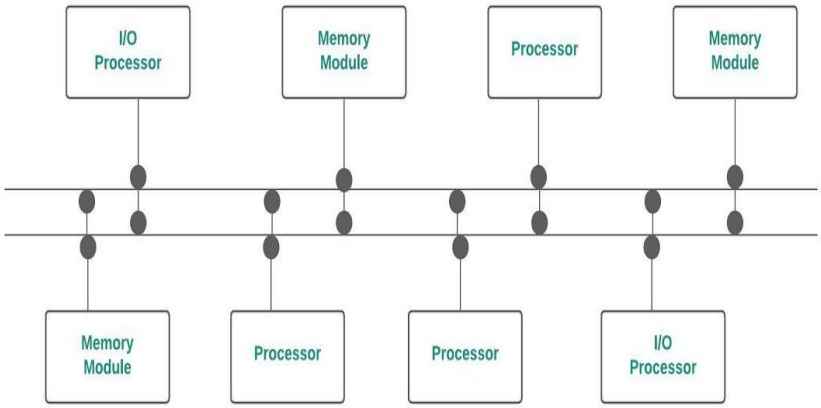
To solve the above disadvantages, we can use two uni-directional buses as :



Multiprocessor System with unidirectional buses

Both the buses are required in a single transfer operation. Here, the system complexity is increased & the reliability is decreased, The solution is to use multiple bi-directional buses.

Multiple bi-directional buses :
The multiple bi-directional buses means that in the system there are multiple buses that are bi-directional. It permits simultaneous transfers as many as buses are available. But here also the complexity of the system is increased.



Multiple Bi-Directional Multiprocessor System

Apart from the organization, there are many factors affecting the performance of bus. They are –

- Number of active devices on the bus.
- Data width
- Error Detection method
- Synchronization of data transfer etc.

Advantages of Multiple bi-directional buses –

- Lowest cost for hardware as no extra device is needed such as switch.
- Modifying the hardware system configuration is easy.
- Less complex when compared to other interconnection schemes as there are only 2 buses & all the components are connected via that buses.

Disadvantages of Multiple bi-directional buses –

- System Expansion will degrade the performance because as the number of functional unit increases, more communication is required but at a time only 1 transfer can happen via 1 bus.
- Overall system capacity limits the transfer rate & If bus fails, whole system will fail.
- Suitable for small systems only.

2. Crossbar Switch :

A point is reached at which there is a separate path available for each memory module, if the number of buses in common bus system is increased. Crossbar Switch (for multiprocessors) provides separate path fro each module.

3.Multiport Memory :

In Multiport Memory system, the control, switching & priority arbitration logic are distributed throughout the crossbar switch matrix which is distributed at the interfaces to the memory modules.

4.Hypercube Interconnection :

This is a binary n-cube architecture. Here we can connect 2n processors and each of the processor here forms a node of the cube. A node can be memory module, I/O interface also, not necessarily processor. The processor at a node has communication path that is direct goes to n other nodes (total 2n nodes). There are total 2n distinct n-bit binary addresses.

Conclusion :

Interconnection structure can decide overall system's performance in a multi processor environment. Although using common bus system is much easy & simple, but the availability of only 1 path is its major drawback & if the bus fails, whole system fails. To overcome this & improve overall performance, crossbar, multi port, hypercube & then multistage switch network evolved.

Computer systems contain a number of buses at various levels to facilitate the transfer of information between components. The CPU contains a number of internal buses for transferring information between processor registers and ALU.

Inter Processor Arbitration

The processor, main memory and I/O devices can be interconnected by means of a common bus. A bus is set of lines (wires) defined to transfer all bits of a word from a specified source to a specified destination. Thus, bus provides a communication path for the transfer of data.

The bus includes data lines, address lines and control lines. Such a bus known as system bus. Different types of arbitration: ***Serial (Daisy Chain) arbitration, Parallel arbitration, Dynamic arbitration***

Serial (Daisy Chain) arbitration

In this type of arbitration, processors can access bus based on priority. In serial arbitration, bus access priority resolving based on the serial connection of the processors. This technique is obtained from daisy chain (serial) connection of processors. The serial priority resolving technique is obtained from daisy-chain connection similar to the daisy chain priority interrupt logic. The processors connected to the system bus are assigned priority according to their position along the priority control line.

When multiple devices concurrently request the use of the bus, the device with the highest priority is granted access to it. Each processor has its own bus arbiter logic with priority-in and priority-out lines. The priority out (PO) of each arbiter is connected to the priority in (PI) of the next-lower-priority arbiter. The PI of the highest-priority unit is maintained at a logic value 1. The highest-priority unit in the system will always receive access to the system bus when it requests it. The processor whose arbiter has a PI = 1 and PO = 0. That processor accesses the system bus.

Advantages

Simple and cheaper method

Least number of lines.

Disadvantages

Higher delay

Priority of the processor is fixed

Not reliable

Inter Process Communication (IPC)

A process can be of two types:

- Independent process.
- Co-operating process.

An independent process is not affected by the execution of other processes while a co-operating process can be affected by other executing processes. Though one can think that those processes, which are running independently, will execute very efficiently, in reality, there are many situations when co-operative nature can be utilized for increasing computational speed, convenience, and modularity. Inter-process communication (IPC) is a mechanism that allows processes to

communicate with each other and synchronize their actions. The communication between these processes can be seen as a method of co-operation between them. Processes can communicate with each other through both:

1. Shared Memory
2. Message passing

Figure 1 below shows a basic structure of communication between processes via the shared memory method and via the message passing method.

An operating system can implement both methods of communication. First, we will discuss the shared memory methods of communication and then message passing. Communication between processes using shared memory requires processes to share some variable, and it completely depends on how the programmer will implement it. One way of communication using shared memory can be imagined like this: Suppose process1 and process2 are executing simultaneously, and they share some resources or use some information from another process. Process1 generates information about certain computations or resources being used and keeps it as a record in shared memory. When process2 needs to use the shared information, it will check in the record stored in shared memory and take note of the information generated by process1 and act accordingly. Processes can use shared memory for extracting information as a record from another process as well as for delivering any specific information to other processes.
Let's discuss an example of communication between processes using the shared memory method.

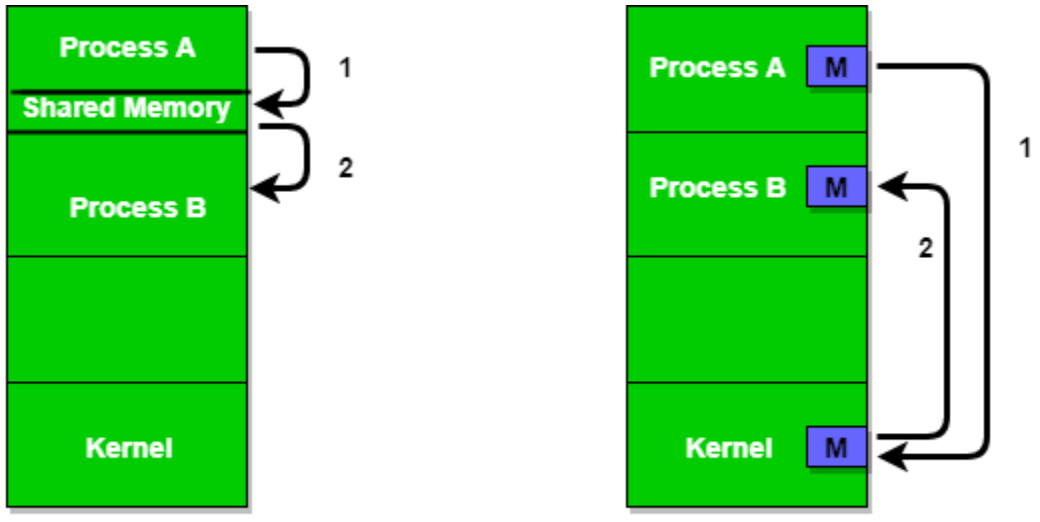


Figure 1 - Shared Memory and Message Passing

i) Shared Memory Method

Ex: Producer-Consumer problem

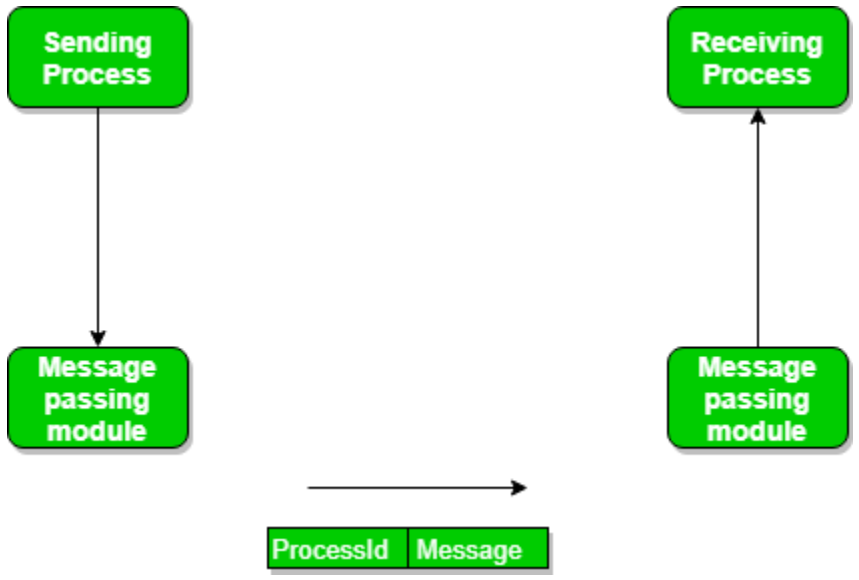
There are two processes: Producer and Consumer. The producer produces some items and the Consumer consumes that item. The two processes share a common space or memory location known as a buffer where the item produced by the Producer is stored and from which the Consumer consumes the item if needed. There are two versions of this problem: the first one is known as the unbounded buffer problem in which the Producer can keep on producing items and there is no limit on the size of the buffer, the second one is known as the bounded buffer problem in which the Producer can produce up to a certain number of items before it starts waiting for Consumer to consume it. We will discuss the bounded buffer problem. First, the Producer and the Consumer will share some common memory, then the producer will start producing items. If the total produced item is equal to the size of the buffer, the producer will wait to get it consumed by the Consumer. Similarly, the consumer will first check for the availability of the item. If no item is available, the Consumer will wait for the Producer to produce it. If there are items available, Consumer will consume them. The pseudo-code to demonstrate is provided below:

Shared Data between the two Processes

ii) Messaging Passing Method

Now, We will start our discussion of the communication between processes via message passing. In this method, processes communicate with each other without using any kind of shared memory. If two processes p1 and p2 want to communicate with each other, they proceed as follows:

- Establish a communication link (if a link already exists, no need to establish it again.)
- Start exchanging messages using basic primitives.
  We need at least two primitives:
  – **send**(message, destination) or **send**(message)
  – **receive**(message, host) or **receive**(message)



The message size can be of fixed size or of variable size. If it is of fixed size, it is easy for an OS designer but complicated for a programmer and if it is of variable size then it is easy for a programmer but complicated for the OS designer. A standard message can have two parts: **header and body.**

The **header part** is used for storing message type, destination id, source id, message length, and control information. The control information contains information like what to do if runs out of buffer space, sequence number, priority. Generally, message is sent using FIFO style.

Message Passing through Communication Link.

Direct and Indirect Communication link

Now, We will start our discussion about the methods of implementing communication links. While implementing the link, there are some questions that need to be kept in mind like :

1. How are links established?
2. Can a link be associated with more than two processes?
3. How many links can there be between every pair of communicating processes?
4. What is the capacity of a link? Is the size of a message that the link can accommodate fixed or variable?
5. Is a link unidirectional or bi-directional?

A link has some capacity that determines the number of messages that can reside in it temporarily for which every link has a queue associated with it which can be of zero capacity, bounded capacity, or unbounded capacity. In zero capacity, the sender waits until the receiver informs the sender that it has received the message. In non-zero capacity cases, a process does not know whether a message has been received or not after the send operation. For this, the sender must communicate with the receiver explicitly. Implementation of the link depends on the situation, it can be either a direct communication link or an in-directed communication link.

Direct Communication links are implemented when the processes use a specific process identifier for the communication, but it is hard to identify the sender ahead of time.

For example the print server.

In-direct Communication is done via a shared mailbox (port), which consists of a queue of messages. The sender keeps the message in mailbox and the receiver picks them up.

Message Passing through Exchanging the Messages.

Synchronous and Asynchronous Message Passing:

A process that is blocked is one that is waiting for some event, such as a resource becoming available or the completion of an I/O operation. IPC is possible between the processes on same computer as well as on the processes running on different computer i.e. in networked/distributed system. In both cases, the process may or may not be blocked while sending a message or attempting to receive a message so message passing may be blocking or non-blocking. Blocking is considered **synchronous** and **blocking send** means the sender will be blocked until the message is received by receiver. Similarly, **blocking receive** has the receiver block until a message is available. Non-blocking is considered **asynchronous** and Non-blocking send has the sender sends the message and continue. Similarly, Non-blocking receive has the receiver receive a valid message or null. After a careful analysis, we can come to a conclusion that for a sender it is more

natural to be non-blocking after message passing as there may be a need to send the message to different processes. However, the sender expects acknowledgment from the receiver in case the send fails. Similarly, it is more natural for a receiver to be blocking after issuing the receive as the information from the received message may be used for further execution. At the same time, if the message send keep on failing, the receiver will have to wait indefinitely. That is why we also consider the other possibility of message passing. There are basically three preferred combinations:

- Blocking send and blocking receive
- Non-blocking send and Non-blocking receive
- Non-blocking send and Blocking receive (Mostly used)

In Direct message passing, The process which wants to communicate must explicitly name the recipient or sender of the communication.
e.g. **send(p1, message)** means send the message to p1.
Similarly, **receive(p2, message)** means to receive the message from p2.
In this method of communication, the communication link gets established automatically, which can be either unidirectional or bidirectional, but one link can be used between one pair of the sender and receiver and one pair of sender and receiver should not possess more than one pair of links. Symmetry and asymmetry between sending and receiving can also be implemented i.e. either both processes will name each other for sending and receiving the messages or only the sender will name the receiver for sending the message and there is no need for the receiver for naming the sender for receiving the message. The problem with this method of communication is that if the name of one process changes, this method will not work.
In Indirect message passing, processes use mailboxes (also referred to as ports) for sending and receiving messages. Each mailbox has a unique id and processes can communicate only if they share a mailbox. Link established only if processes share a common mailbox and a single link can be associated with many processes. Each pair of processes can share several communication links and these links may be unidirectional or bi-directional. Suppose two processes want to communicate through Indirect message passing, the required operations are: create a mailbox, use this mailbox for sending and receiving messages, then destroy the mailbox. The standard primitives used are: **send(A, message)** which means send the message to mailbox A. The primitive for the receiving the message also works in the same way e.g. **received (A, message)**. There is a problem with this mailbox implementation. Suppose there are more than two processes sharing the same mailbox and suppose the process p1 sends a message to the mailbox, which process will be the receiver? This can be solved by either enforcing that only two processes can share a single mailbox or enforcing that only one process is allowed to execute the receive at a given time or select any process randomly and notify the sender about the receiver. A mailbox can be made private to a single sender/receiver pair and can also be shared between multiple sender/receiver pairs. Port is an implementation of such mailbox that can have multiple senders and a single receiver. It is used in client/server applications (in this case the server is the receiver). The port is owned by the receiving process and created by OS on the request of the receiver process and can be destroyed either on request of the same receiver processor when the receiver terminates itself. Enforcing that only one process is allowed to execute the receive can be done using the concept of mutual exclusion. **Mutex mailbox** is created which is shared by n process. The sender is non-

blocking and sends the message. The first process which executes the receive will enter in the critical section and all other processes will be blocking and will wait.
Now, let's discuss the Producer-Consumer problem using the message passing concept. The producer places items (inside messages) in the mailbox and the consumer can consume an item when at least one message present in the mailbox. The code is given below:

Examples of IPC systems

1. Posix : uses shared memory method.
2. Mach : uses message passing
3. Windows XP : uses message passing using local procedural calls

Communication in client/server Architecture:

There are various mechanism:

- Pipe
- Socket
- Remote Procedural calls (RPCs)

Cache Coherence.

In computer architecture, **cache coherence** is the uniformity of shared resource data that ends up stored in multiple local caches. When clients in a system maintain caches of a common memory resource, problems may arise with incoherent data, which is particularly the case with CPUs in a multiprocessing system.

In the illustration on the right, consider both the clients have a cached copy of a particular memory block from a previous read. Suppose the client on the bottom updates/changes that memory block, the client on the top could be left with an invalid cache of memory without any notification of the change. Cache coherence is intended to manage such conflicts by maintaining a coherent view of the data values in multiple caches.