1. **Overview of Register Transfer And Micro operations** Register Transfer Language, Register transfer. Bus and Memory transfer**,** Arithmetic Micro-operations. Logic Micro-operations**,** Shift Micro-operations, Arithmetic Logic Shift Unit

1. Explain the Register Transfer Language.

Definition: The symbolic notation used to describe the micro operation transfers among registers is called a register transfer language.

- The term "**register transfer**" implies the availability of hardware logic circuits that can perform a stated micro operation and transfer the result of the operation to the same or another register.
- The word "language" is borrowed from programmers, who apply this term to programming languages.
- A register transfer language is a system for expressing in symbolic form the microoperation sequences among the registers of a digital module.
- It is a convenient tool for describing the internal organization of digital computers in concise and precise manner.
- It can also be used to facilitate the design process of digital systems.
- Information transfer from one register to another is designated in symbolic form by means of a replacement operator.
- The statement below denotes a transfer of the content of register R1 into register R2.

$$R2 \leftarrow R1$$

- A statement that specifies a register transfer implies that circuits are available from the outputs of the destination register has a parallel load capability.
- Every statement written in a register transfer notation implies a hardware construction for implementing the transfer.

2. Explain the Register Transfer in detail with block diagram and timing diagram.

Definition: Information transfer from one register to another is designated in symbolic form by means of a replacement operator is known as Register Transfer.

$$R2 \leftarrow R1$$

Denotes a transfer of the content of register R1 into register R2.

- Computer registers are designated by capital letters (sometimes followed by numerals) to denote the function of the register.
  For example:

MAR	Holds address of memory unit
PC	Program Counter
IR	Instruction Register
R_1	Processor Register

- Below figure1.1 shows the representation of registers in block diagram form.

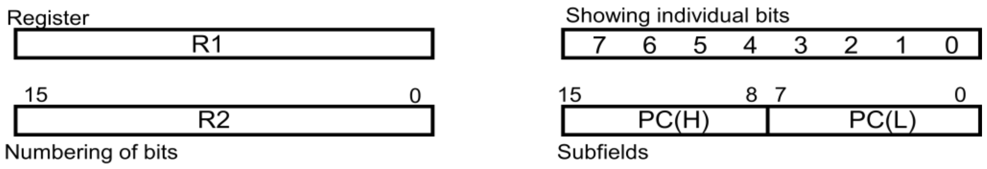


Figure 1.1: Block diagram of register

- The most common way to represent a register is by a rectangular box with the name of the register inside, as shown in figure.
- Bits 0 through 7 are assigned the symbol L (for low byte) and bits 8 through 15 are assigned the symbol H (for high byte). The name of the 16-bit register is PC. The symbol PC(0-7) or PC(L) refers to the low-order byte and PC(8-15) or PC(H) to the high-order byte.
- The statement that specifies a register transfer implies that circuits are available from the outputs of the source register to the inputs of the destination register and that the destination register has a parallel load capability.

Register Transfer with control function:

- If we want the transfer to occur only under a predetermined control condition. This can be shown by means of an if-then statement.

  $$\text{If } (P = 1) \text{ then } (R2 <\!- R1)$$

  where P is a control signal.
- It is sometimes convenient to separate the control variables from the register transfer operation control function by specifying a control function.
- A **control function** is a Boolean variable that is equal to 1 or 0. The control function is included in the statement as follows:

  $$P: R2 <\!-R1$$

- The control condition is terminated with a colon. It symbolizes the requirement that the transfer operation be executed by the hardware only if P = 1.
- Every statement written in a register transfer notation implies a hardware construction for implementing the transfer. Below figure shows the block diagram that depicts the transfer from R1 to R2.

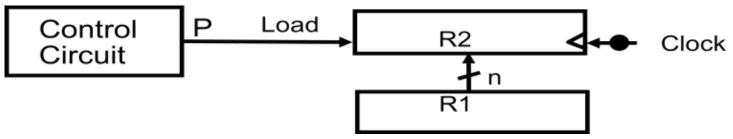


Figure 1.2: Transfer from R1 to R2 when P = 1

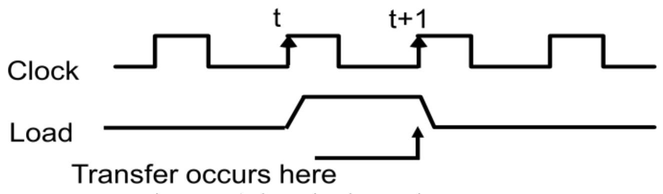


Figure 1.3: Timing diagram

- The n outputs of register R1 are connected to the n inputs of register R2. The letter n will be used to indicate any number of bits for the register.
- In the timing diagram, P is activated in the control section by the rising edge of a clock pulse at time t.
- The next positive transition of the clock at time t + 1 finds the load input active and the data inputs of R2 are then loaded into the register in parallel.
- P may go back to 0 at time t + 1; otherwise, the transfer will occur with every clock pulse transition while P remains active.
- The basic symbols of the register transfer notation are listed in Table below:

Symbol	**Description**	**Examples**
Letters (and numerals)	Denotes a register	MAR, R2
Parentheses ( )	Denotes a part of a register	R 2(0-7), R2(L)
Arrow ←	Denotes transfer of information	R2←R1
Comma ,	Separates two micro operations	R2←R1, R1←R2

Table 1.1: **Basic Symbols for Register Transfers**

- Registers are denoted by capital letters, and numerals may follow the letters.
- Parentheses are used to denote a part of a register by specifying the range of bits or by giving a symbol name to a portion of a register.
- The arrow denotes a transfer of information and the direction of transfer.
- A comma is used to separate two or more operations that are executed at the same time.
- The statement below, denotes an operation that exchanges the contents of two registers during one common clock pulse provided that T = 1.

  T: R2← R1, R1← R2
- This simultaneous operation is possible with registers that have edge-triggered flip-flops.

3. Design and explain a common bus system for four register.

- A typical digital computer has many registers, and paths must be provided to transfer information from one register to another.
- The number of wires will be excessive if separate lines are used between each register and all other registers in the system.
- A more efficient scheme for transferring information between registers in a multiple-register configuration is a common bus system.
- A bus structure consists of a set of common lines, one for each bit of a register, through which binary information is transferred one at a time.

- Control signals determine which register is selected by the bus during each particular register transfer.
- One way of constructing a common bus system is with multiplexers.
- The multiplexers select the source register whose binary information is then placed on the bus.
- The construction of a bus system for four registers is shown in figure below.
- Each register has four bits, numbered 0 through 3.
- The bus consists of four 4 x 1 multiplexers each having four data inputs, 0 through 3, and two selection inputs, S_1 and S_0.
- The diagram shows that the bits in the same significant position in each register are connected to the data inputs of one multiplexer to form one line of the bus.

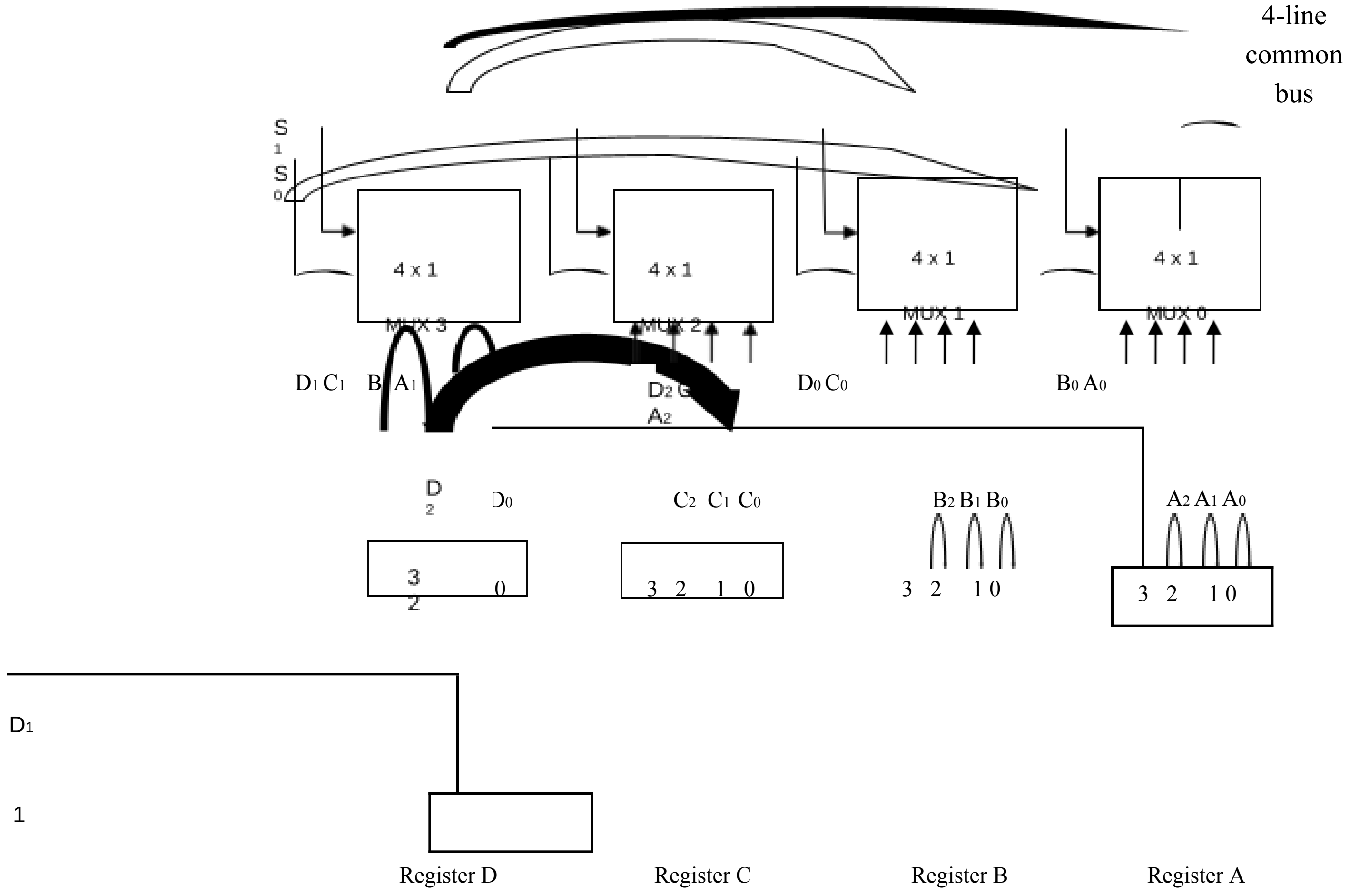


Figure 1.4: Bus system for four registers

- The two selection lines S_1 and S_0 are connected to the selection inputs of all four multiplexers.
- The selection lines choose the four bits of one register and transfer them into the four-

line common bus.

S_1	S_0	Register selected
0	0	A
0	1	B

1	0	C
1	1	D

Table 1.2: Function Table for Bus

- When $S_1S_0 = 00$, the 0 data inputs of all four multiplexers are selected and applied to the outputs that form the bus.
- This causes the bus lines to receive the content of register A since the outputs of this register are connected to the 0 data inputs of the multiplexers.
- Similarly, register B is selected if $S_1S_0 = 01$, and so on.
- Table shows the register that is selected by the bus for each of the four possible binary values of the selection lines.
- In general, a bus system will multiplex k registers of n bits each to produce an n-line common bus.
- The number of multiplexers needed to construct the bus is equal to n, the number of bits in each register.
- The size of each multiplexer must be K x 1 since it multiplexes K data lines.
  For example, a common bus for eight registers of 16 bits each requires 16 multiplexers, one for each line in the bus. Each multiplexer must have eight data input lines and three selection lines to multiplex one significant bit in the eight registers.

4. A digital computer has a common bus system for 16 registers of 32 bits each. (i) How many selection input are there in each multiplexer? (ii) What size of multiplexers is needed? (iii) How many multiplexers are there in a bus?

(i) How many selection input are there in each multiplexer?
2^n=No. of Registers; n=selection input of multiplexer
$2^n=16$; here n=4
Therefore 4 selection input lines should be there in each multiplexer.

(ii) What size of multiplexers is needed?
size of multiplexers= Total number of register X 1
= 16 X 1
Multiplexer of 16 x 1 size is needed to design the above defined common bus.

(iii) How many multiplexers are there in a bus? No. of multiplexers = bits of register
= 32
32 multiplexers are needed in a bus.

5. Explain three-state bus buffer. OR Explain the operation of three state bus buffers and show its use in design of common bus.

- A bus system can be constructed with three-state gates instead of multiplexers.
- A three-state gate is a digital circuit that exhibits three states.
  State 1: Signal equivalent to Logic 1
  State 2: Signal equivalent to Logic 0
  State 3: High Impedance State (behaves as open circuit)
- The high-impedance state behaves like an open circuit, which means that the output is disconnected and does not have logic significance.
- The most commonly used design of a bus system is the buffer gate.
- The graphic symbol of a three-state buffer gate is shown in figure 1.5 below:

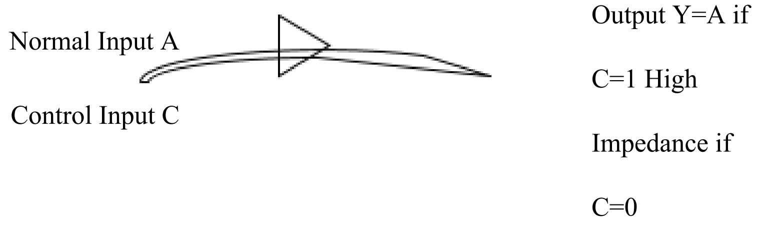


Figure 1.5: Graphic symbols for three-state buffer

- It is distinguished from a normal buffer by having both a normal input and a control input.
- The control input determines the output state. When the control input C is equal to 1, the output is enabled and the gate behaves like any conventional buffer, with the output equal to the normal input.

- When the control input C is 0, the output is disabled and the gate goes to a high-impedance state, regardless of the value in the normal input.
- The high-impedance state of a three-state gate provides a special feature not available in other gates.
- Because of this feature, a large number of three-state gate outputs can be connected with wires to form a common bus line without endangering loading effects.
- The construction of a bus system with three-state buffers is demonstrated in figure 1.6 below:

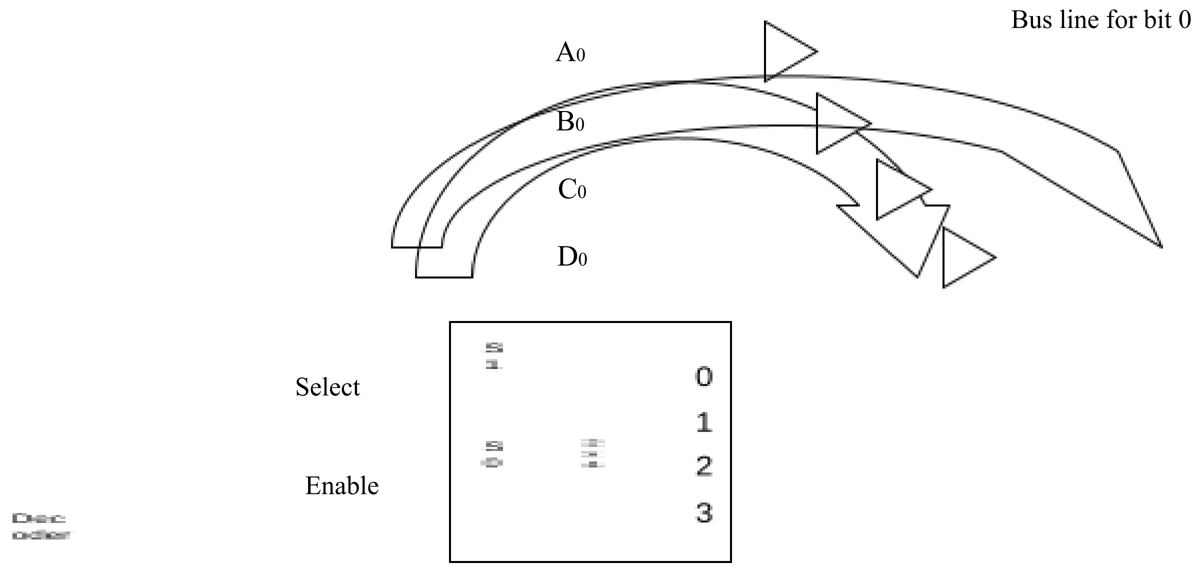
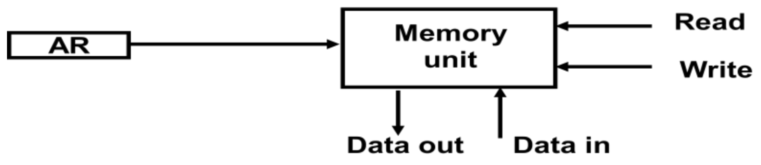


Figure 1.6: Bus line with three state-buffers

- The outputs of four buffers are connected together to form a single bus line.
- The control inputs to the buffers determine which of the four normal inputs will communicate with the bus line.
- No more than one buffer may be in the active state at any given time.
- The connected buffers must be controlled so that only one three-state buffer has access to the bus line while all other buffers are maintained in a high impedance state.
- One way to ensure that no more than one control input is active at any given time is to use a decoder, as shown in the figure: Bus line with three state-buffers.
- When the enable input of the decoder is 0, all of its four outputs are 0, and the bus line is in a high-impedance state because all four buffers are disabled.
- When the enable input is active, one of the three-state buffers will be active, depending on the binary value in the select inputs of the decoder.
- To construct a common bus for four registers of n bits each using three- state buffers, we need n circuits with four buffers in each as shown in figure: Bus line with three state-buffers,
- Each group of four buffers receives one significant bit from the four registers.
- Each common output produces one of the lines for the common bus for a total of n lines.
- Only one decoder is necessary to select between the four registers.

6. Explain Memory Transfer

- **Read Operation:** The transfer of information from a memory word to the outside environment is called a read operation.
- **Write Operation:** The transfer of new information to be stored into the memory is called a write operation.
- A memory word will be symbolized by the letter M.
- It is necessary to specify the address of M when writing memory transfer operations.
- This will be done by enclosing the address in square brackets following the letter M.
- Consider a memory unit that receives the address from a register, called the address register, symbolized by AR.
- The data are transferred to another register, called the data register, symbolized by DR. The read operation can be stated as follows:
  Read: DR $\leftarrow$ M[AR]
- This causes a transfer of information into DR from the memory word M selected by the address in AR.
- The write operation transfers the content of a data register to a memory word M selected by the address. Assume that the input data are in register R1 and the address is in AR.
- Write operation can be stated symbolically as follows:
  Write: M[AR] $\leftarrow$ R1
- This causes a transfer of information from R1 into memory word M selected by address AR.



7. Explain Arithmetic Micro-operation.

- The basic arithmetic micro-operations are:
    1. Addition
    2. Subtraction
    3. Increment
    4. Decrement
    5. Shift
- The additional arithmetic micro operations are:
    1. Add with carry
    2. Subtract with borrow
    3. Transfer/Load , etc.
- Summary of Typical Arithmetic Micro-Operations:

R3 ← R1 + R2	Contents of R1 plus R2 transferred to R3
R3 ← R1 - R2	Contents of R1 minus R2 transferred to R3
R2 ← R2’	Complement the contents of R2
R2 ← R2’+ 1	2's complement the contents of R2 (negate)
R3 ← R1 + R2’+ 1	subtraction
R1 ← R1 + 1	Increment
R1 ← R1 – 1	Decrement

8. Explain Binary Adder in detail

- To implement the add micro operation with hardware, we need :
    1. Registers : that hold the data
    2. Digital component: that performs the arithmetic addition.
- **Full-adder**
  The digital circuit that forms the arithmetic sum of two bits and a previous carry is called a full-adder.
- **Binary adder**
  The digital circuit that generates the arithmetic sum of two binary numbers of any lengths is called a binary adder.
- The binary adder is constructed with full-adder circuits connected in cascade, with the output carry from one full-adder connected to the input carry of the next full-adder.

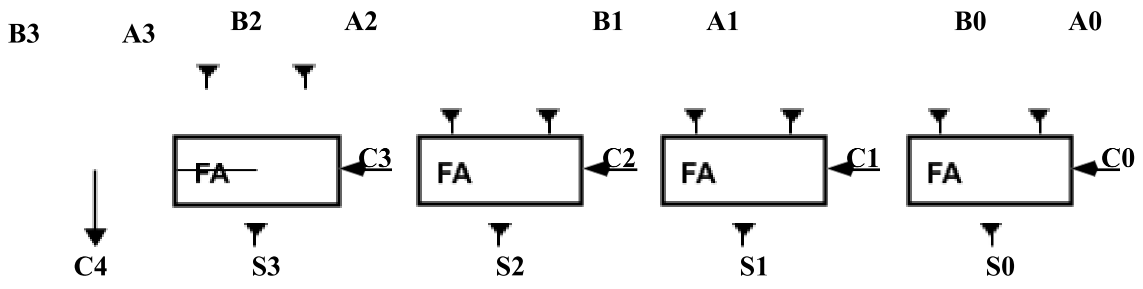


Figure 1.7: 4-bit binary adder

- Above figure 1.7 shows the interconnections of four full-adders (FA) to provide a 4-bit binary adder.
- The augends bits of A and the addend bits of B are designated by subscript numbers from right to left, with subscript 0 denoting the low-order bit.
- The carries are connected in a chain through the full-adders.
- The input carry to the binary adder is C0 and the output carry is C4.
- The S outputs of the full-adders generate the required sum bits.
- An n-bit binary adder requires n full-adders.
- The output carry from each full-adder is connected to the input carry of the next-high-order full-adder.
- The n data bits for the A inputs come from one register (such as R1), and the n data bits for the B inputs come from another register (such as R2). The sum can be transferred to a third register or to one of the source registers (R1 or R2), replacing its previous content.

9. Explain Binary Adder-Subtractor in detail.

- The subtraction of binary numbers can be done most conveniently by means of complements.
- Remember that the subtraction A - B can be done by taking the 2's complement of B and adding it to A.
- The 2's complement can be obtained by taking the l's complement and adding one to the least significant pair of bits. The l's complement can be implemented with inverters and a one can be added to the sum through the input carry.
- The addition and subtraction operations can be combined into one common circuit by including an exclusive-OR gate with each full-adder.
- The mode input M controls the operation.

  When **M = 0** the circuit is an **Adder**

  When **M = 1** the circuit becomes a **Subtractor**
- Each exclusive-OR gate receives input M and one of the inputs of B. When M = 0,

  We have $C_0=0$

  $B \oplus 0 = B$

The full-adders receive the value of B, the input carry is 0, and the circuit performs A

plus B.
When M =
1,
We have C0=1
$B \oplus 1 = B'$; B complement

The B inputs are all complemented and 1is added through the input carry. The circuit performs the operation A plus the 2's complement of B.

A + 2’s compliment of B

- A 4-bit adder-subtractor circuit is shown as follows:

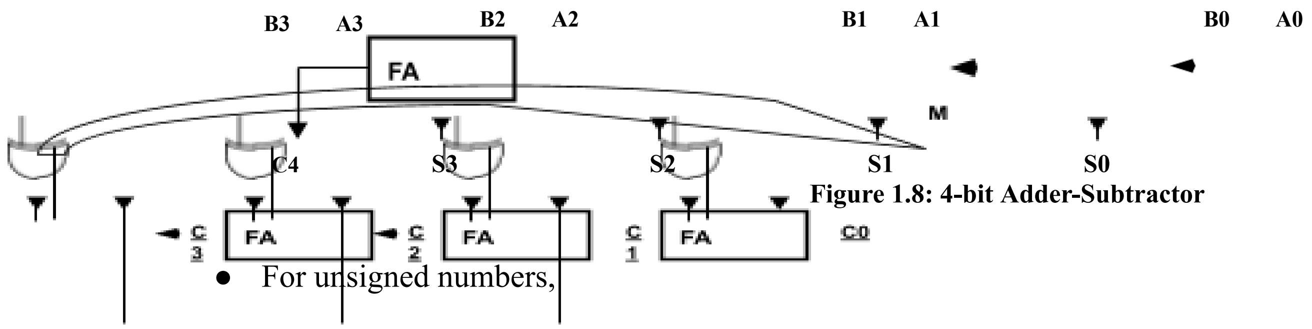


Figure 1.8: 4-bit Adder-Subtractor

- For unsigned numbers,

If A>=B, then A-B If
A<B, then B-A

For signed numbers,

Result is A-B, provided that there is no overflow.

10. Explain Binary Incrementer

- The increment micro operation adds one to a number in a register.
- For example, if a 4-bit register has a binary value 0110, it will go to 0111 after it is incremented.

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

- The diagram of a 4-bit combinational circuit incrementer is shown above. One of the inputs to the least significant half-adder (HA) is connected to logic-1 and the other input is connected to the least significant bit of the number to be incremented.
- The output carry from one half-adder is connected to one of the inputs of the next-higher-order half-adder.
- The circuit receives the four bits from A_0 through A_3, adds one to it, and generates the incremented output in S_0 through S_3.
- The output carry C_4 will be 1 only after incrementing binary 1111. This also causes

outputs S_0 through S_3 to go to 0.

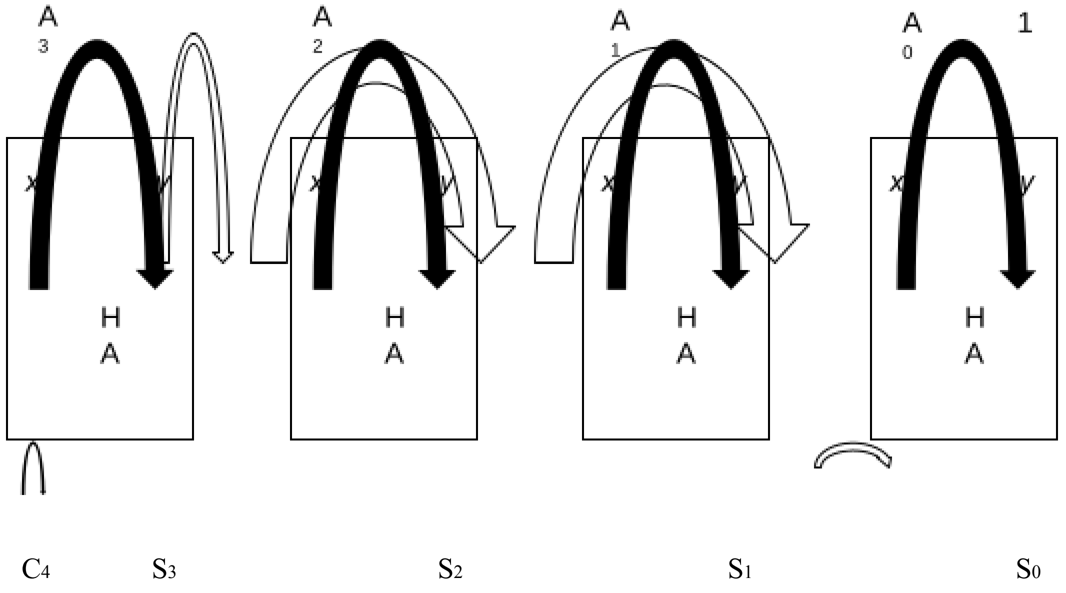


Figure 1.9: 4-bit binary incrementer

11. Draw block diagram of 4-bit arithmetic circuit and explain it in detail.

- The arithmetic micro operations can be implemented in one composite arithmetic circuit.
- The basic component of an arithmetic circuit is the parallel adder.
- By controlling the data inputs to the adder, it is possible to obtain different types of arithmetic operations.
- Hardware implementation consists of:
    1. 4 full-adder circuits that constitute the 4-bit adder and four multiplexers for choosing different operations.
    2. There are two 4-bit inputs A and B
       The four inputs from A go directly to the X inputs of the binary adder. Each of the four inputs from B is connected to the data inputs of the multiplexers. The multiplexer's data inputs also receive the complement of B.
    3. The other two data inputs are connected to logic-0 and logic-1. Logic-0 is a fixed voltage value (0 volts for TTL integrated circuits) and the logic-1 signal can be generated through an inverter whose input is 0.
    4. The four multiplexers are controlled by two selection inputs, S_1 and S_0.
    5. The input carry C_{in} goes to the carry input of the FA in the least significant position. The other carries are connected from one stage to the next.
    6. 4-bit output $D_0 \ldots D_3$

- The diagram of a 4-bit arithmetic circuit is shown below figure 1.10:

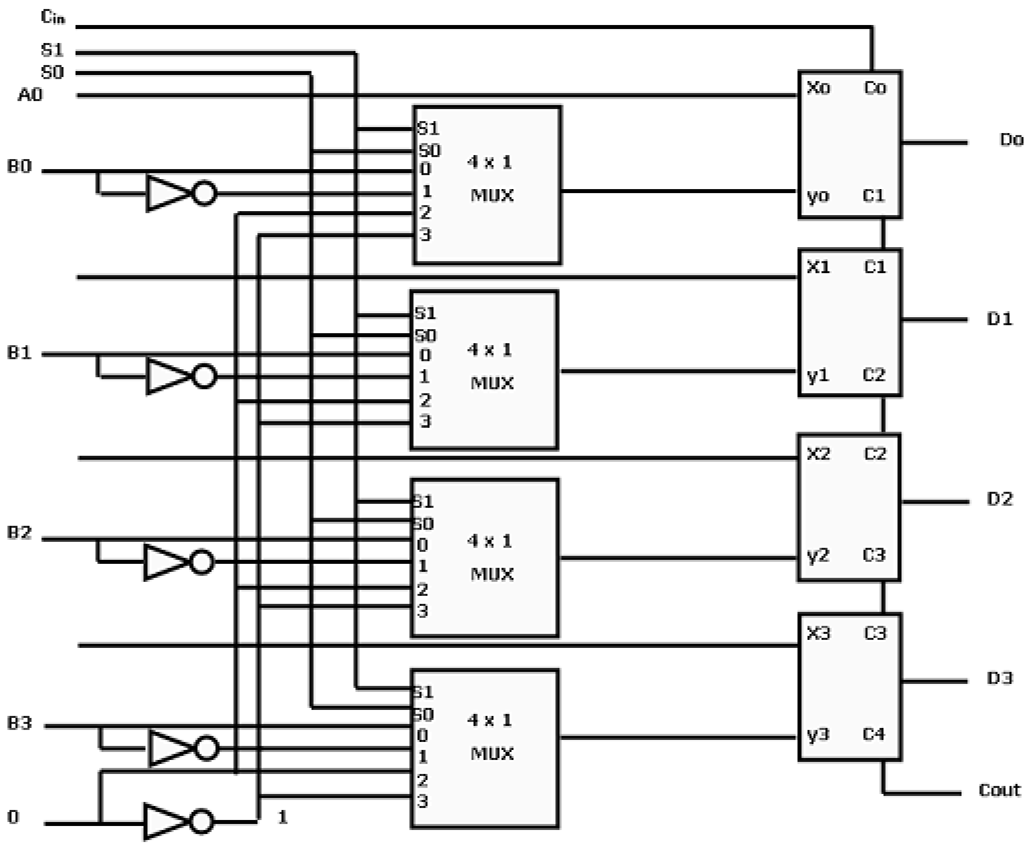


Figure 1.10: 4-bit arithmetic circuit

- The output of binary adder is calculated from arithmetic sum.

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

Select S_1	S_0	C_{in}	**Input** **Y**	**Output** **D = A + Y + Cin**	**Microoperation**
0	0	0	B	D = A + B	Add
0	0	1	B	D = A + B + 1	Add with Carry
0	1	0	B’	D = A + B’	Subtract with Borrow
0	1	1	B’	D = A + B’ + 1	Subtract
1	0	0	0	D = A	Transfer A
1	0	1	0	D = A + 1	Increment A
1	1	0	1	D = A – 1	Decrement A
1	1	1	1	D = A	Transfer A

TABLE 1.3: 4-4 Arithmetic Circuit Function Table

- When $S_1 S_0 = 0\ 0$
  - If Cin=0, D=A+B; Add
  - If Cin=1, D=A+B+1;Add with carry
- When $S_1 S_0 = 0\ 1$
  - If Cin=0, D=A+$\overline{B}$; Subtract with borrow
  - If Cin=1, D=A+$\overline{B}$+1;A+2's compliment of B i.e. A-B
- When $S_1 S_0 = 1\ 0$
  Input B is neglected and Y=> logic '0'
  D=A+0+ Cin
  - If Cin=0, D=A; Transfer A If Cin=1,
    D=A+1;Increment A
- When $S_1 S_0 = 1\ 1$
  Input B is neglected and Y=> logic '1'
  D=A-1+ Cin
  - If Cin=0, D=A-1; 2's compliment If Cin=1,
    D=A; Transfer A

- Note that the micro-operation D = A is generated twice, so there are only seven distinct micro-operations in the arithmetic circuit.

12. Draw and explain Logic Micro-operations in detail.

- Logic micro operations specify binary operations for strings of bits stored in registers.
- These operations consider each bit of the register separately and treat them as binary variables. For example, the exclusive-OR micro-operation with the contents of two registers R1 and R2 is symbolized by the statement:

$$P: R1 \leftarrow R1 \oplus R2$$

	1 0 1 0	Content of R1
$\oplus$	1 1 0 0	Content of R2
	0 1 1 0	Content of R1 after P = 1

- The logic micro-operations are seldom used in scientific computations, but they are very useful for bit manipulation of binary data and for making logical decisions.
- **Notation:**
  The symbol $\vee$ will be used to denote an **OR** microoperation and the symbol $\wedge$ to denote an **AND** microoperation. The complement microoperation is the same as the 1's complement and uses a bar on top of the symbol that denotes the register name.
- Although the + symbol has two meanings, it will be possible to distinguish between them by noting where the symbol occurs. When the symbol + occurs in a microoperation, it will denote an arithmetic plus. When it occurs in a control (or Boolean) function, it will denote an OR operation.

$$P + Q: R1 \leftarrow R2 + R3,\ R4 \leftarrow R5 \vee R6$$

- The + between P and Q is an OR operation between two binary variables of a control function. The + between R2 and R3 specifies an add microoperation. The OR microoperation is designated by the symbol V between registers R5and R6.

List of Logic Micro operations

- There are 16 different logic operations that can be performed with two binary variables.
- They can be determined from all possible truth tables obtained with two binary variables as shown in table below.

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 1.4: Truth Tables for 16 Functions of Two Variables

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 ← all 1's	Set to all l's

TABLE 1.5: Sixteen Logic Microoperation

Hardware Implementation

- The hardware implementation of logic microoperation requires that logic gates be inserted for each bit or pair of bits in the registers to perform the required logic function.
- Although there are 16 logic microoperation, most computers use only four—AND, OR, XOR (exclusive-OR), and complement from which all others can be derived.
- Below figure shows one stage of a circuit that generates the four basic logic micro operations.

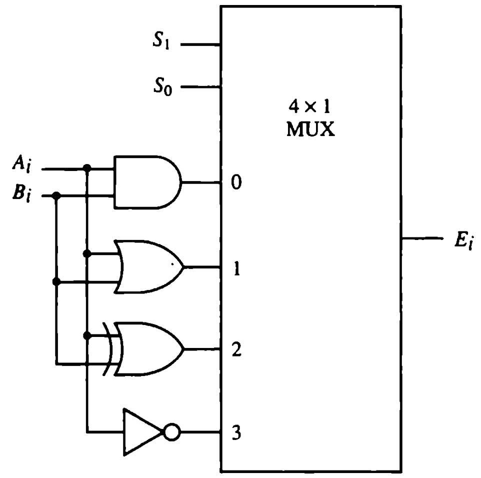


Figure 1.11 : One stage of logic circuit

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

Table 1.6: Function table

- Hardware implementation consists of four gates and a multiplexer.
- Each of the four logic operations is generated through a gate that performs the required logic.
- The outputs of the gates are applied to the data inputs of the multiplexer.
- The two selection inputs S_1 and S_0 choose one of the data inputs of the multiplexer and direct its value to the output.
- The diagram shows one typical stage with subscript i. For a logic circuit with n bits, the diagram must be repeated n times for i = 0, 1, 2, . . . n - 1. The selection variables are

applied to all stages.

13. Explain selective set, selective complement and selective clear.

Selective-Set operation:

- The selective-set operation sets to 1 the bits in register A where there are corresponding 1's in register B. It does not affect bit positions that have 0's in B. The following numerical example clarifies this operation:

1010	A before
$\underline{1100}$	B (logical operand)
1110	A after

- The two leftmost bits of B are 1's, so the corresponding bits of A are set to 1.
- One of these two bits was already set and the other has been changed from 0 to 1.
- The two bits of A with corresponding 0's in B remain unchanged. The example above serves as a truth table since it has all four possible combinations of two binary variables.
- The OR microoperation can be used to selectively set bits of a register.

Selective-Complement operation:

- The selective-complement operation complements bits in A where there are corresponding 1's in B. It does not affect bit positions that have O's in B. For example:

1010	A before
$\underline{1100}$	B (logical operand)
0110	A after

- Again the two leftmost bits of B are 1's, so the corresponding bits of A are complemented.
- The exclusive-OR microoperation can be used to selectively complement bits of a register.

Selective-Clear operation:

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

1010	A before
$\underline{1100}$	B (logical operand)
0010	A after

- Again the two leftmost bits of B are 1's, so the corresponding bits of A are cleared to 0.
- One can deduce that the Boolean operation performed on the individual bits is AB'.
- The corresponding logic microoperation is $A \leftarrow A \wedge B'$.

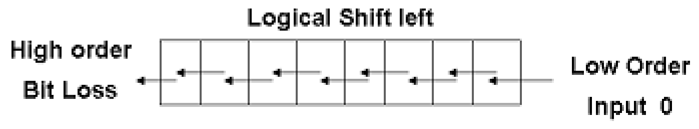
14. Explain shift micro operations and draw 4-bit combinational circuit shifter.

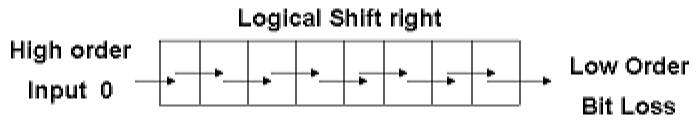
There are 3 types of shift micro-operations:

1. Logical Shift:
   - A logical shift is one that transfers 0 through the serial input. We will adopt the symbols shl and shr for logical shift-left and shift-right micro-operations.
   - For example:

     $$R1 \leftarrow \text{shl } R1$$
     $$R2 \leftarrow \text{shr}$$
     $$R2$$

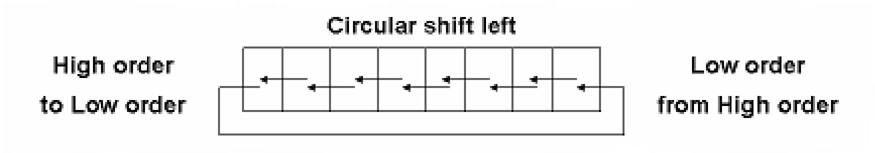
     are two micro-operations that specify a 1-bit shift to the left of the content of register R1 and a 1-bit shift to the right of the content of register R2.
   - The register symbol must be the same on both sides of the arrow.
   - The bit transferred to the end position through the serial input is assumed to be 0 during a logical shift.

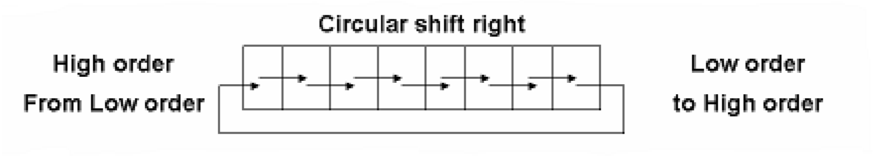




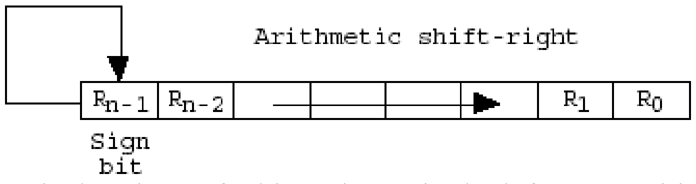
2. Circular Shift:
   - The circular shift (also known as a rotate operation) circulates the bits of the register around the two ends without loss of information.
   - This is accomplished by connecting the serial output of the shift register to its serial input. We will use the symbols cil and cir for the circular shift left and right, respectively.

     $$R1 \leftarrow \text{Cil}$$
     $$R1\ R2 \leftarrow \text{Cir}$$
     $$R2$$

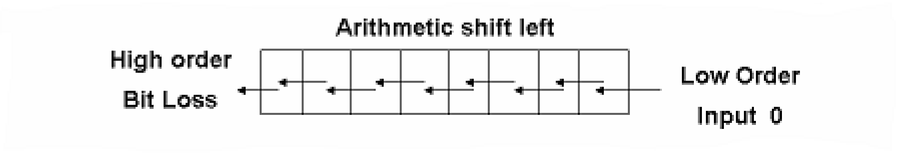


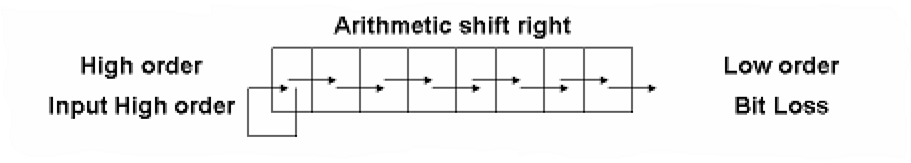


3. Arithmetic Shift:
    - An arithmetic shift is a micro-operation that shifts a signed binary number to the left or right.
    - An arithmetic shift-left multiplies a signed binary number by 2.
    - An arithmetic shift-right divides the number by 2.
    - Arithmetic shifts must leave the sign bit unchanged because the sign of the number remains the same when it is multiplied or divided by 2.
    - The leftmost bit in a register holds the sign bit, and the remaining bits hold the number. The sign bit is 0 for positive and 1 for negative.
    - Negative numbers are in 2's complement form.



    - Figure shows a typical register of n bits. Bit R_{n-1} in the leftmost position holds the sign bit.
    - R_{n-2} is the most significant bit of the number and R_0 is the least significant bit.
    - The arithmetic shift-right leaves the sign bit unchanged and shifts the number (including the sign bit) to the right.
    - Thus R_{n-1} remains the same; R_{n-2} receives the bit from R_{n-1}, and so on for the other bits in the register.
    - The bit in R_0 is lost.
    - The arithmetic shift-left inserts a 0 into R_0, and shifts all other bits to the left.
    - The initial bit of R_{n-1} is lost and replaced by the bit from R_{n-2}.
    - A sign reversal occurs if the bit in Rn-1 changes in value after the shift. This happens if the multiplication by 2 causes an overflow.





4-bit combinational circuit shifter

- A combinational circuit shifter can be constructed with multiplexers as shown in Figure below.

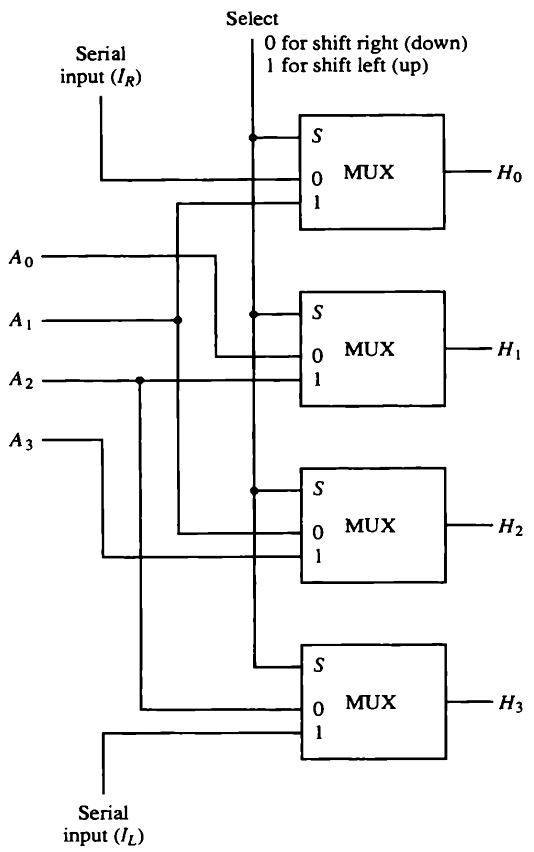


Figure 1.12: 4-bit combinational circuit shifter

- The 4-bit shifter has four data inputs, A0 through A3 and four data outputs, H0 through H3.
- There are two serial inputs, one for shift left (IL) and the other for shift right (IL).
- When the selection input S = 0, the input data are shifted right (down in the diagram).
- When S = 1, the input data are shifted left (up in the diagram).
- The function table in Figure shows which input goes to each output after the 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

- A shifter with n data inputs and outputs requires n multiplexers.
- The two serial inputs can be controlled by another multiplexer to provide the three possible types of shifts.

15. Draw and explain one stage of arithmetic logic shift unit.

- Instead of having individual registers performing the micro operations directly, computer systems employ a number of storage registers connected to a common operational unit called an arithmetic logic unit, abbreviated ALU.
- To perform a microoperation, the contents of specified registers are placed in the inputs of the common ALU.
- The ALU performs an operation and the result of the operation 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 dock pulse period.
- The arithmetic, logic, and shift circuits introduced in previous sections can be combined into one ALU with common selection variables.
- One stage of an arithmetic logic shift unit is shown in figure below:

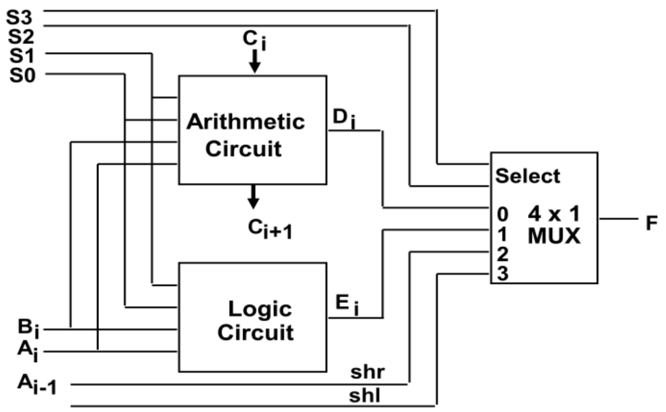


Figure 1.13: One stage of arithmetic logic shift unit

- The subscript i designates a typical stage. Inputs A_i and B_i are applied to both the arithmetic and logic units.
- A particular microoperation is selected with inputs S_1 and S_0.
- A 4 x 1 multiplexer at the output chooses between an arithmetic output in D_i and a logic
- output in E_i .
- The data in the multiplexer are selected with inputs S3 and S2.
- The other two data inputs to the multiplexer receive inputs A_{i-1} for the shift-right operation and A_{i+1} for the shift-left operation.
- Note that the diagram shows just one typical stage. The circuit shown in figure must be repeated n times for an n-bit ALU.

- The outputs carry C_{i+1} of a given arithmetic stage must be connected to the input carry C_{in} of the next stage in sequence.
- The input carry to the first stage is the input carry On, which provides a selection

variable for

- the arithmetic operations.
- The circuit whose one stage is specified in figure provides
  - 8 arithmetic operation
  - 4 logic operations
  - 2 shift operations
- Each operation is selected with the five variables S3, S2, Si, S0, and C_{in}. The input carry C_{in} is used for selecting an arithmetic operation only.
- Table below lists the 14 operations of the ALU.

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

Table 1.7: Function Table for Arithmetic Logic Shift Unit

- The first eight are arithmetic operations and are selected with $S_3S_2 = 00$.
- The next four are logic operations are selected with $S_3S_2 = 01$. The input carry has no effect during the logic operations and is marked with don't-care X's.
- The last two operations are shift operations and are selected with $S_3S_2 = 10$ and 11.
- The other three selection inputs have no effect on the shift.

- floating-point arithmetic operations.

1. Define the following:

Instruction Code

An instruction code is a group of bits that instruct the computer to perform a specific operation.

Operation Code

The operation code of an instruction is a group of bits that define such operations as add, subtract, multiply, shift, and complement. The number of bits required for the operation code of an instruction depends on the total number of operations available in the computer. The operation code must consist of at least n bits for a given 2^n (or less) distinct operations.

Accumulator (AC)

Computers that have a single-processor register usually assign to it the name accumulator (AC) accumulator and label it AC. The operation is performed with the memory operand and the content of AC.

2. Explain Stored Program Organization in detail.

- The simplest way to organize a computer is to have one processor register and an instruction code format with two parts.
- The first part specifies the operation to be performed and the second specifies an address.
- The memory address tells the control where to find an operand in memory.
- This operand is read from memory and used as the data to be operated on together with the data stored in the processor register.
- The following figure 2.1 shows this type of organization.

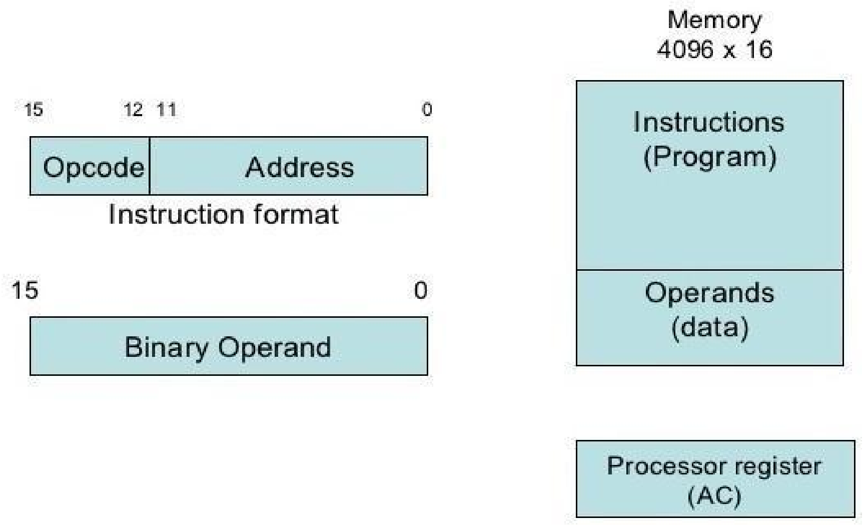


Figure 2.1: Stored Program Organization

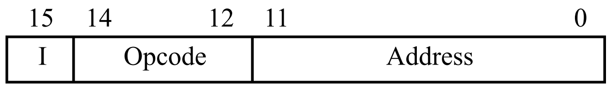
- Instructions are stored in one section of memory and data in another.
- For a memory unit with 4096 words, we need 12 bits to specify an address since 2^{12} = 4096.

- If we store each instruction code in one 16-bit memory word, we have available four bits for operation code (abbreviated opcode) to specify one out of 16 possible operations, and 12 bits to specify the address of an operand.
- The control reads a 16-bit instruction from the program portion of memory.
- It uses the 12-bit address part of the instruction to read a 16-bit operand from the data portion of memory.
- It then executes the operation specified by the operation code.
- Computers that have a single-processor register usually assign to it the name accumulator and label it AC.
- If an operation in an instruction code does not need an operand from memory, the rest of the bits in the instruction can be used for other purposes.
- For example, operations such as clear AC, complement AC, and increment AC operate on data stored in the AC register. They do not need an operand from memory. For these types of operations, the second part of the instruction code (bits 0 through 11) is not needed for specifying a memory address and can be used to specify other operations for the computer.

3. Explain Direct and Indirect addressing of basic computer.

- The second part of an instruction format specifies the address of an operand, the instruction is said to have a **direct address**.
- In **Indirect address**, the bits in the second part of the instruction designate an address of a memory word in which the address of the operand is found.
- One bit of the instruction code can be used to distinguish between a direct and an indirect address.
- It consists of a 3-bit operation code, a 12-bit address, and an indirect address mode bit designated by I.
- The mode bit is 0 for a direct address and 1 for an indirect address.
- A direct address instruction is shown in Figure 2.2. It is placed in address 22 in memory.
- The I bit is 0, so the instruction is recognized as a direct address instruction.
- The opcode specifies an ADD instruction, and the address part is the binary equivalent of 457.
- The control finds the operand in memory at address 457 and adds it to the content of AC.
- The instruction in address 35 shown in Figure 2.3 has a mode bit I = 1, recognized as an indirect address instruction.
- The address part is the binary equivalent of 300.
- The control goes to address 300 to find the address of the operand. The address of the operand in this case is 1350. The operand found in address 1350 is then added to the content of AC.

- The indirect address instruction needs two references to memory to fetch an operand.
    1. The first reference is needed to read the address of the operand
    2. Second reference is for the operand itself.
- The memory word that holds the address of the operand in an indirect address instruction is used as a pointer to an array of data.



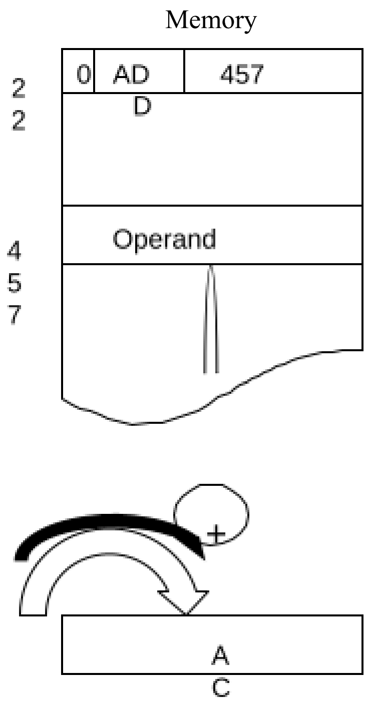


Figure 2.2: Direct Address

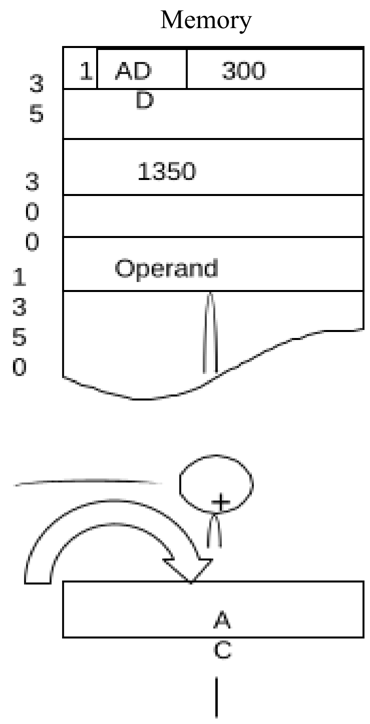


Figure 2.3: Indirect Address

Direct Address	Indirect Address
When the second part of an instruction code specifies the address of an operand, the instruction is said to have a direct address.	When the second part of an instruction code specifies the address of a memory word in which the address of the operand, the instruction is said to have a direct address.
For instance the instruction MOV R0 00H. R0, when converted to machine language is the physical address of register R0. The instruction moves 0 to R0.	For instance the instruction MOV @R0 00H, when converted to machine language, @R0 becomes whatever is stored in R0, and that is the address used to move 0 to. It can be whatever is stored in R0.

3. Explain Registers of basic computer.

- It is necessary to provide a register in the control unit for storing the instruction code after it is read from memory.
- The computer needs processor registers for manipulating data and a register for holding a memory address.
- These requirements dictate the register configuration shown in Figure 2.4.

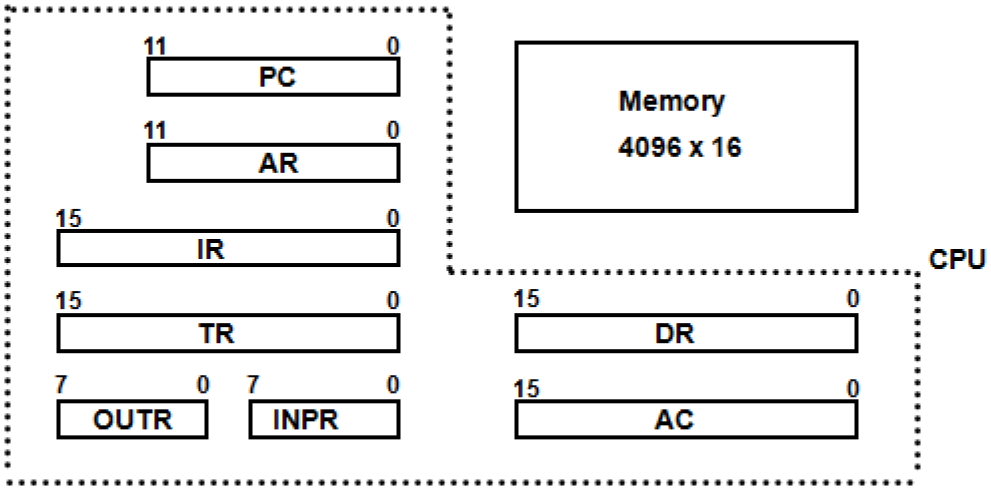


Figure 2.4: Basic Computer Register and Memory

- The data register (DR) holds the operand read from memory.
- The accumulator (AC) register is a general purpose processing register.
- The instruction read from memory is placed in the instruction register (IR).
- The temporary register (TR) is used for holding temporary data during the processing.
- The memory address register (AR) has 12 bits.
- The program counter (PC) also has 12 bits and it holds the address of the next instruction to be read from memory 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.
- Two registers are used for input and output. The input register (INPR) receives an 8-bit character from an input device. The output register (OUTR) holds an 8-bit character for an output device.

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

Table 2.1: List of Registers for Basic Computer

4. Draw and explain Common Bus System for basic computer register.

What is the requirement of common bus System?

- The basic computer has eight registers, a memory unit and a control unit.
- Paths must be provided to transfer information from one register to another and between memory and register.
- The number of wires will be excessive if connections are between the outputs of each register and the inputs of the other registers. An efficient scheme for transferring information in a system with many register is to use a common bus.
- The connection of the registers and memory of the basic computer to a common bus system is shown in figure 2.5.
- The outputs of seven registers and memory are connected to the common bus. The specific output that is selected for the bus lines at any given time is determined from the binary value of the selection variables S2, S1, and S0.
- The number along each output shows the decimal equivalent of the required binary selection.
- The particular register whose LD (load) input is enabled receives the data from the bus during the next clock pulse transition. The memory receives the contents of the bus when its write input is activated. The memory places its 16-bit output onto the bus when the read input is activated and S2 S1 S0 = 1 1 1.
- Four registers, DR, AC, IR, and TR have 16 bits each.
- Two registers, AR and PC, have 12 bits each since they hold a memory address.
- When the contents of AR or PC are applied to the 16-bit common bus, the four most significant bits are set to 0's. When AR and PC receive information from the bus, only the 12 least significant bits are transferred into the register.
- The input register INPR and the output register OUTR have 8 bits each and communicate with the eight least significant bits in the bus. INPR is connected to provide information to the bus but OUTR can only receive information from the bus.

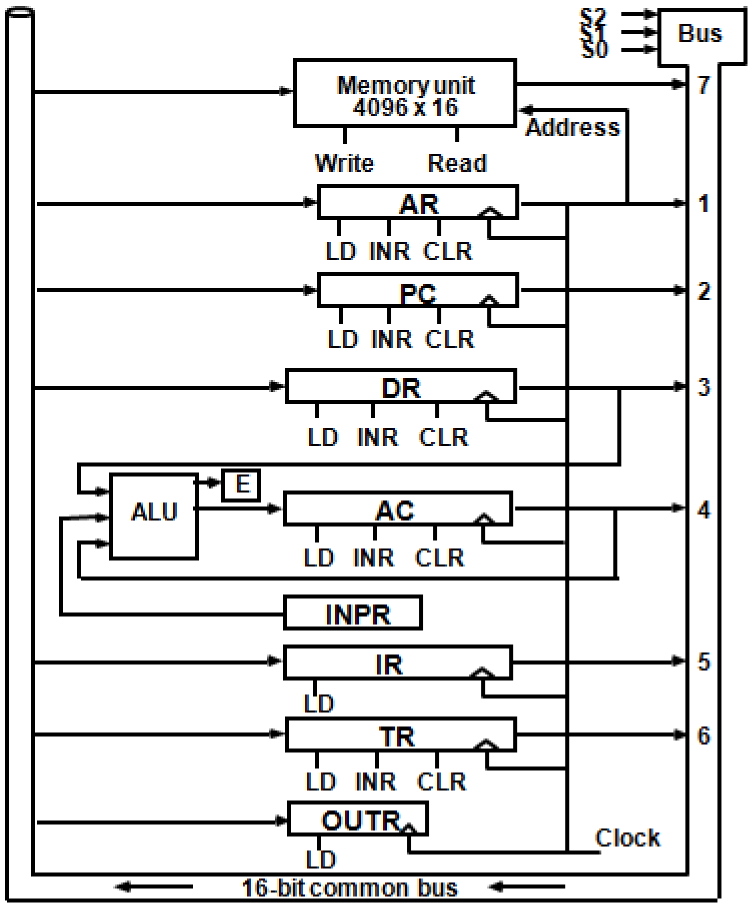


Figure 2.5: Basic computer registers connected to a common bus

- Five registers have three control inputs: LD (load), INR (increment), and CLR (clear). Two registers have only a LD input.
- AR must always be used to specify a memory address; therefore memory address is connected to AR.
- The 16 inputs of AC come from an adder and logic circuit. This circuit has three sets of inputs.
  1. Set of 16-bit inputs come from the outputs of AC.
  2. Set of 16-bits come from the data register DR.
  3. Set of 8-bit inputs come from the input register INPR.
- The result of an addition is transferred to AC and the end carry-out of the addition is transferred to flip-flop E (extended AC bit).
- The clock transition at the end of the cycle transfers the content of the bus into the designated destination register and the output of the adder and logic circuit into AC.

5. Explain Instruction Format with its types.

- The basic computer has three instruction code formats, as shown in figure 2.6.

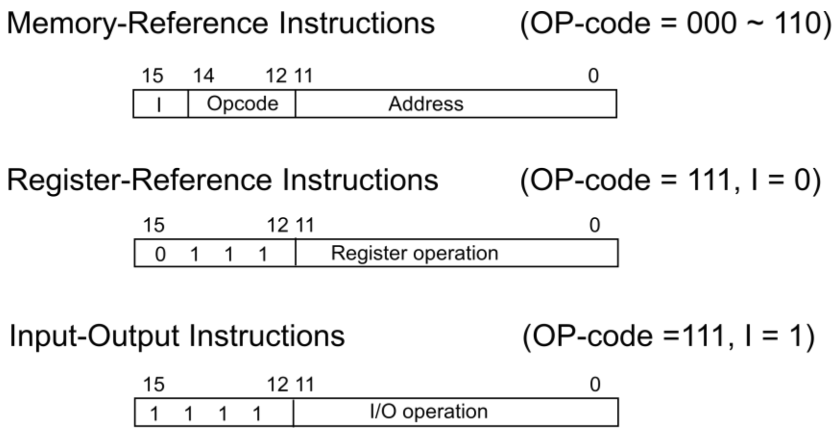


Figure 2.6: Basic computer instruction format

- Each format has 16 bits.

- The operation code (opcode) part of the instruction contains three bits and the meaning of the remaining 13 bits depends on the operation code encountered.
- A **memory-reference instruction** uses 12 bits to specify an address and one bit to specify the addressing mode I. I is equal to 0 for direct address and to 1 for indirect address.
- The **register reference instructions** are recognized by the operation code 111 with a 0 in the leftmost bit (bit 15) of the instruction. A register-reference instruction specifies an operation on or a test of the AC register. An operand from memory is not needed; therefore, the other 12 bits are used to specify the operation or test to be executed.
- An **input-output instruction** does not need a reference to memory and is recognized by the operation code 111 with a 1 in the leftmost bit of the instruction. The remaining 12 bits are used to specify the type of input-output operation or test performed.

6. Explain the basic working principle of the Control Unit with timing diagram.

- The block diagram of the control unit is shown in figure 2.7.
- Components of Control unit are
  1. Two decoders
  2. A sequence counter
  3. Control logic gates
- An instruction read from memory is placed in the instruction register (IR). In control unit the IR is divided into three parts: I bit, the operation code (12-14)bit, and bits 0 through 11.
- The operation code in bits 12 through 14 are decoded with a 3 X 8 decoder.

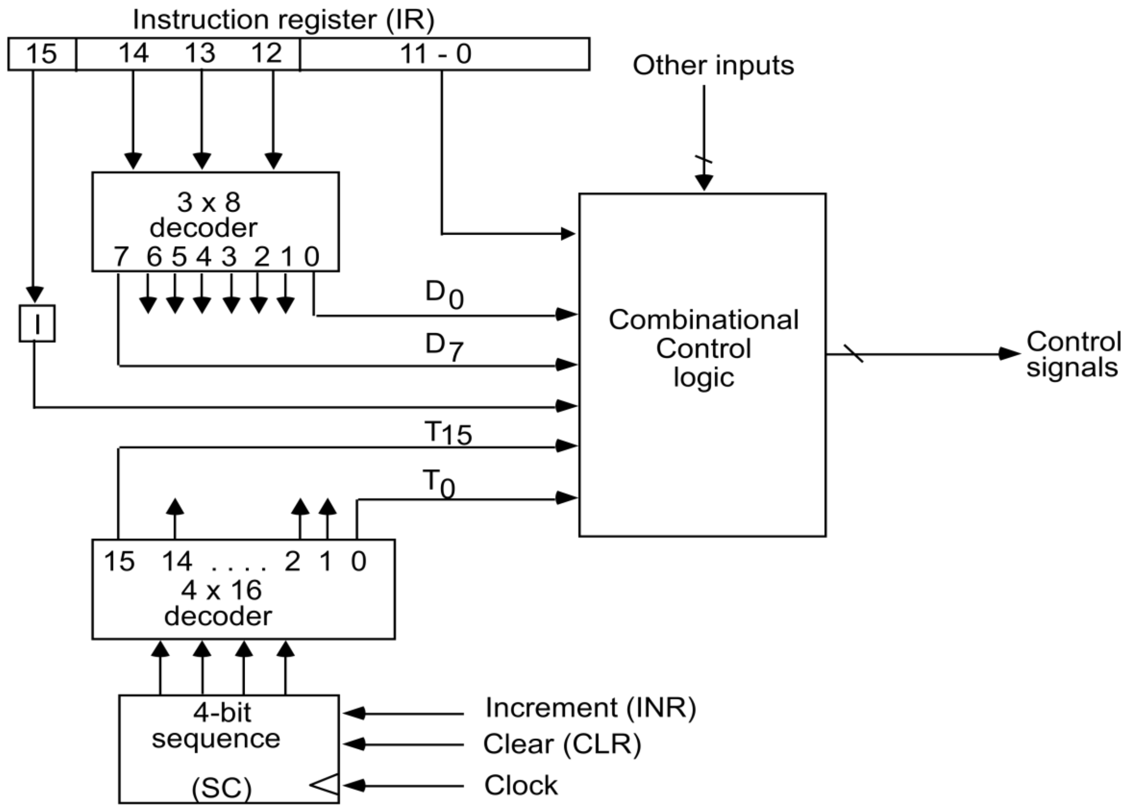


Figure 2.7: Control unit of basic computer

- Bit-15 of the instruction is transferred to a flip-flop designated by the symbol I.
- The eight outputs of the decoder are designated by the symbols D0 through D7. Bits 0 through 11 are applied to the control logic gates. The 4-bit sequence counter can count in binary from 0 through 15.The outputs of counter are decoded into 16 timing signals T0 through T_{15}.
- The sequence counter SC can be incremented or cleared synchronously. Most of the time, the counter is incremented to provide the sequence of timing signals out of 4 X 16 decoder. Once in awhile, the counter is cleared to 0, causing the next timing signal to be T0.
- As an example, consider the case where SC is incremented to provide timing signals T0, T_1, T_2, T_3 and T_4 in sequence. At time T_4, SC is cleared to 0 if decoder output D3 is active. This is expressed symbolically by the statement

$$D_3T_4: SC \leftarrow 0$$

Timing Diagram:

- The timing diagram figure2.8 shows the time relationship of the control signals.
- The sequence counter SC responds to the positive transition of the clock.
- Initially, the CLR input of SC is active.
- The first positive transition of the clock clears SC to 0, which in turn activates the timing T_0 out of the decoder. T_0 is active during one clock cycle. The positive clock transition

labeled T_0 in the diagram will trigger only those registers whose control inputs are connected to timing signal T_0.

- SC is incremented with every positive clock transition, unless its CLR input is active.
- This procedures the sequence of timing signals T_0, T_1, T_2, T_3 and T_4, and so on. If SC is not cleared, the timing signals will continue with T_5, T_6, up to T_{15} and back to T_0.

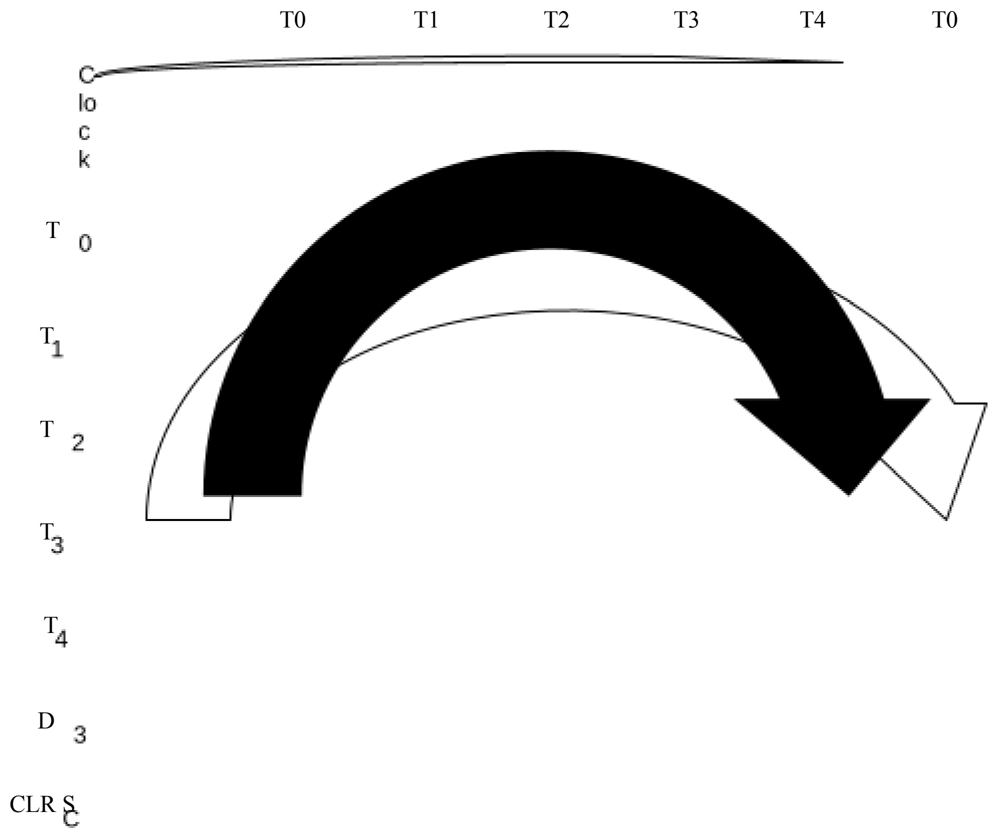


Figure 2.8: Example of control timing signals

- The last three waveforms shows how SC is cleared when D3T4 = 1. Output D3 from the operation decoder becomes active at the end of timing signal T_2. When timing signal T_4 becomes active, the output of the AND gate that implements the control function D_3T_4 becomes active.
- This signal is applied to the CLR input of SC. On the next positive clock transition the counter is cleared to 0. This causes the timing signal T_0 to become active instead of T_5 that would have been active if SC were incremented instead of cleared.

7. Draw and explain the flowchart for instruction cycle.

- A program residing in the memory unit of the computer consists of a sequence of instructions. In the basic computer each instruction cycle consists of the following phases:
  1. Fetch an instruction from memory.
  2. Decode the instruction.
  3. Read the effective address from memory if the instruction has an indirect address.
  4. Execute the instruction.
- After step 4, the control goes back to step 1 to fetch, decode and execute the nex instruction.
- This process continues unless a HALT instruction is encountered.

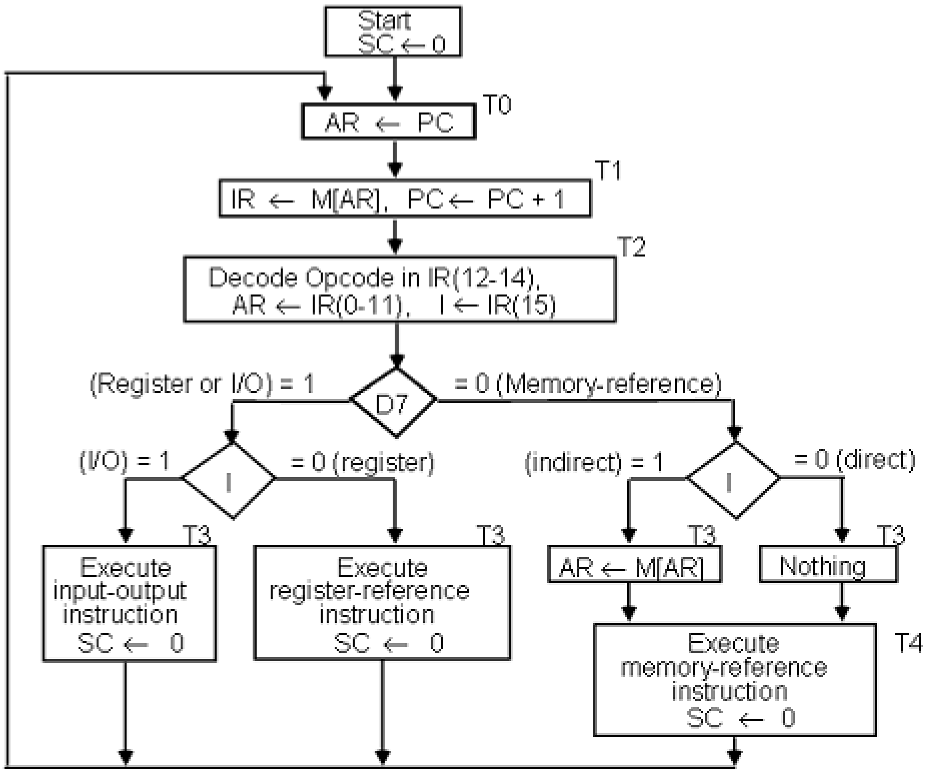

Start
SC ← 0
T0
AR ← PC
T1
IR ← M[AR], PC ← PC + 1
T2
Decode Opcode in IR(12-14),
AR ← IR(0-11), I ← IR(15)
(Register or I/O) = 1
D7
= 0 (Memory-reference)
(I/O) = 1
I
= 0 (register)
(indirect) = 1
I
= 0 (direct)
T3
Execute input-output instruction SC ← 0
T3
Execute register-reference instruction SC ← 0
T3
AR ← M[AR]
T3
Nothing
T4
Execute memory-reference instruction SC ← 0

Figure 2.9: Flowchart for instruction cycle (initial configuration)

- The flowchart presents an initial configuration for the instruction cycle and shows how the control determines the instruction type after the decoding.
- If D7 = 1, the instruction must be register-reference or input-output type. If D7 = 0, the operation code must be one of the other seven values 110, specifying a memory- reference instruction. Control then inspects the value of the first bit of the instruction, which now available in flip-flop I.
- If D7 = 0 and I = 1, we have a memory-reference instruction with an indirect address. It is then necessary to read the effective address from memory.
- The three instruction types are subdivided into four separate paths. The selected

operation is activated with the clock transition associated with timing signal T_3.This can be symbolized as follows:

D’7 I T3: AR $\leftarrow$ M [AR]
D’7 I’ T3: Nothing
D7 I’ T3: Execute a register-reference instruction
D7 I T3: Execute an input-output instruction

- When a memory-reference instruction with I = 0 is encountered, it is not necessary to do anything since the effective address is already in AR.
- However, the sequence counter SC must be incremented when D’7 I T3 = 1, so that the execution of the memory-reference instruction can be continued with timing variable T4.
- A register-reference or input-output instruction can be executed with the click associated with timing signal T3. After the instruction is executed, SC is cleared to 0 and control returns to the fetch phase with T0 =1. SC is either incremented or cleared to 0 with every positive clock transition.

8. List and explain register reference instruction.

- When the register-reference instruction is decoded, D7 bit is set to 1.
- Each control function needs the Boolean relation D7 I' T3

15 12	11 0
0 1 1 1	Register Operation

- There are 12 register-reference instructions listed below:

	r:	SC $\leftarrow$ 0	Clear SC
CLA	rB_{11}:	AC $\leftarrow$ 0	Clear AC
CLE	rB_{10}:	E $\leftarrow$ 0	Clear E
CMA	rB_{9}:	AC $\leftarrow$ AC’	Complement AC
CME	rB_{8}:	E $\leftarrow$ E’	Complement E
CIR	rB_{7}:	AC $\leftarrow$ shr AC, AC(15) $\leftarrow$ E, E $\leftarrow$ AC(0)	Circular Right
CIL	rB_{6}:	AC $\leftarrow$ shl AC, AC(0) $\leftarrow$ E, E $\leftarrow$ AC(15)	Circular Left
INC	rB_{5}:	AC $\leftarrow$ AC + 1	Increment AC
SPA	rB_{4}:	if (AC(15) = 0) then (PC $\leftarrow$ PC+1)	Skip if positive
SNA	rB_{3}:	if (AC(15) = 1) then (PC $\leftarrow$ PC+1	Skip if negative
SZA	rB_{2}:	if (AC = 0) then (PC $\leftarrow$ PC+1)	Skip if AC is zero
SZE	rB_{1}:	if (E = 0) then (PC $\leftarrow$ PC+1)	Skip if E is zero
HLT	rB_{0}:	S $\leftarrow$ 0 (S is a start-stop flip-flop)	Halt computer

- These 12 bits are available in IR (0-11). They were also transferred to AR during time T_2.
- These instructions are executed at timing cycle T_3.
- The first seven register-reference instructions perform clear, complement, circular shift, and increment microoperations on the AC or E registers.

- The next four instructions cause a skip of the next instruction in sequence when condition is satisfied. The skipping of the instruction is achieved by incrementing PC.

- The condition control statements must be recognized as part of the control conditions. The AC is positive when the sign bit in AC(15) = 0; it is negative when AC(15) = 1. The content of AC is zero (AC = 0) if all the flip-flops of the register are zero.
- The HLT instruction clears a start-stop flip-flop S and stops the sequence counter from counting. To restore the operation of the computer, the start-stop flip-flop must be set manually.

9. List and explain memory reference instructions.

- When the memory-reference instruction is decoded, D7 bit is set to 0.

15	14 12	11 0
I	000~110	Address

- The following table lists seven memory-reference instructions.

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

- The effective address of the instruction is in the address register AR and was placed there during timing signal T_2 when I = 0, or during timing signal T_3 when I = 1.
- The execution of the memory-reference instructions starts with timing signal T_4.

AND to AC

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

D_0T_4: $DR \leftarrow M[AR]$
D_0T_5: $AC \leftarrow AC \wedge DR, SC \leftarrow 0$

ADD to AC

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

D_1T_4: $DR \leftarrow M[AR]$
D_1T_5: $AC \leftarrow AC + DR, E \leftarrow C_{out}, SC \leftarrow 0$

LDA: Load to AC

This instruction transfers the memory word specified by the effective address to AC.

D_2T_4: DR $\leftarrow$ M[AR]
D_2T_5: AC $\leftarrow$ DR, SC $\leftarrow$ 0

STA: Store AC

This instruction stores the content of AC into the memory word specified by the effective address.

D_3T_4: M[AR] $\leftarrow$ AC, SC $\leftarrow$ 0

BUN: Branch Unconditionally

This instruction transfers the program to instruction specified by the effective address. The BUN instruction allows the programmer to specify an instruction out of sequence and the program branches (or jumps) unconditionally.

D_4T_4: PC $\leftarrow$ AR, SC $\leftarrow$ 0

BSA: Branch and Save Return Address

This instruction is useful for branching to a portion of the program called a subroutine or procedure. When executed, the BSA instruction stores the address of the next instruction in sequence (which is available in PC) into a memory location specified by the effective address.

M[AR] $\leftarrow$ PC, PC $\leftarrow$ AR + 1
M[135] $\leftarrow$21, PC $\leftarrow$135 + 1 =
136

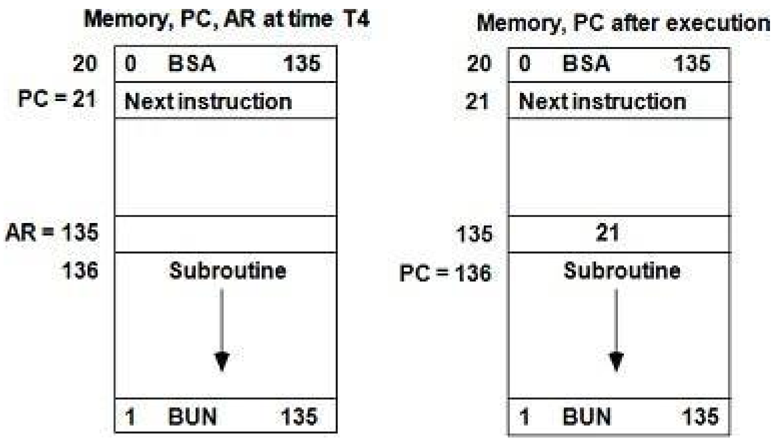


Figure2.10: Example of BSA instruction execution

It is not possible to perform the operation of the BSA instruction in one clock cycle when we use the bus system of the basic computer. To use the memory and the bus properly, the BSA instruction must be executed with a sequence of two microoperations:

D_5T_4: M[AR] $\leftarrow$ PC, AR $\leftarrow$ AR
+ 1 D_5T_5: PC $\leftarrow$ AR, SC $\leftarrow$ 0

ISZ: Increment and Skip if Zero

These instruction increments the word specified by the effective address, and if the incremented value is equal to 0, PC is incremented by 1. Since it is not possible to increment a word inside the memory, it is necessary to read the word into DR, increment DR, and store the word back into memory.

D_6T_4: DR $\leftarrow$
M[AR] D_6T_5:
DR $\leftarrow$ DR + 1
D_6T_4: M[AR] $\leftarrow$ DR, if (DR = 0) then (PC $\leftarrow$ PC + 1), SC $\leftarrow$ 0

Control Flowchart

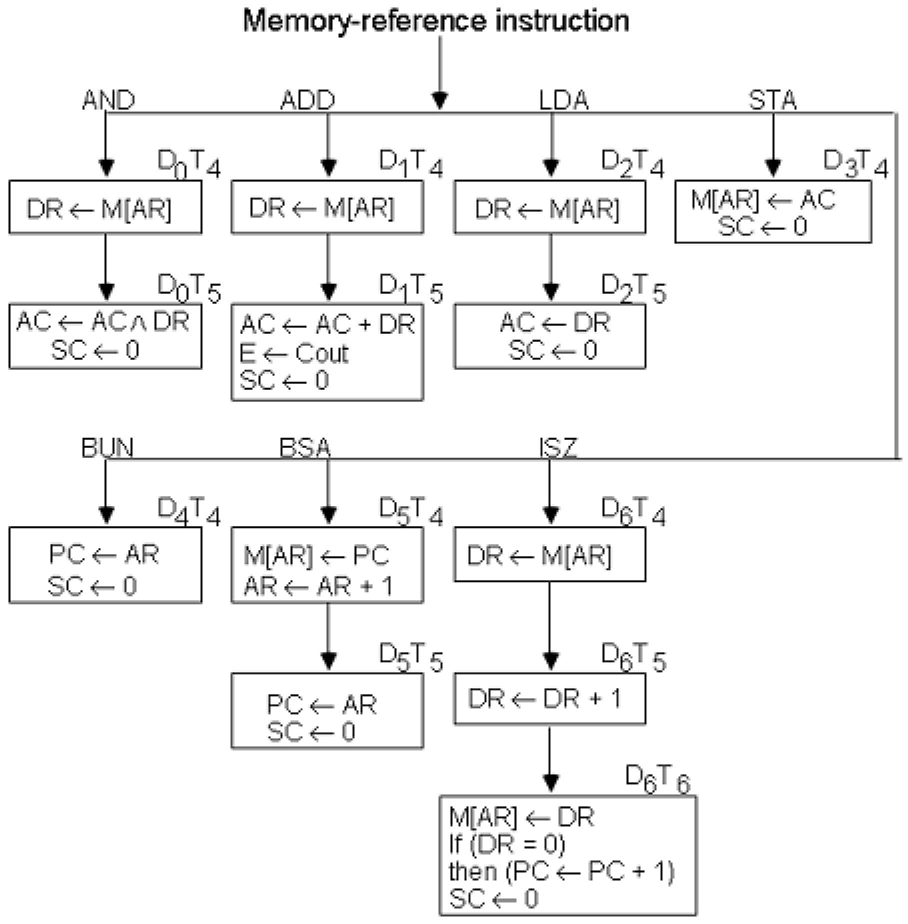


Figure 2.11: Flowchart for memory-reference instructions

10. Draw and explain input-output configuration of basic computer.

- A computer can serve no useful purpose unless it communicates with the external environment.
- To exhibit the most basic requirements for input and output communication, we will use a terminal unit with a keyboard and printer.

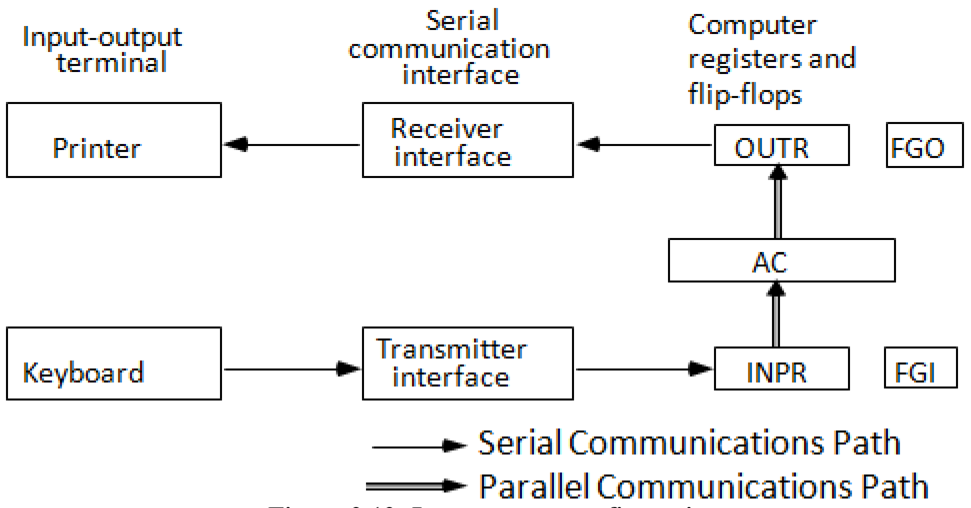

Input-output terminal
Serial communication interface
Computer registers and flip-flops
Printer
Receiver interface
OUTR
FGO
AC
Keyboard
Transmitter interface
INPR
FGI
Serial Communications Path
Parallel Communications Path

Figure 2.12: Input-output configuration

- The terminal sends and receives serial information and each quantity of information has eight bits of an alphanumeric code.
- The serial information from the keyboard is shifted into the input register INPR.
- The serial information for the printer is stored in the output register OUTR.
- These two registers communicate with a communication interface serially and with the AC in parallel.
- The transmitter interface receives serial information from the keyboard and transmits it to INPR. The receiver interface receives information from OUTR and sends it to the printer serially.
- The 1-bit input flag FGI is a control flip-flop. It is set to 1 when new information is available in the input device and is cleared to 0 when the information is accepted by the computer.
- The flag is needed to synchronize the timing rate difference between the input device and the computer.
- The process of information transfer is as follows:

The process of input information transfer:

- Initially, the input flag FGI is cleared to 0. When a key is struck in the keyboard, an 8-bit alphanumeric code is shifted into INPR and the input flag FGI is set to 1.
- As long as the flag is set, the information in INPR cannot be changed by striking another key. The computer checks the flag bit; if it is 1, the information from INPR is transferred in parallel into AC and FGI is cleared to 0.

- Once the flag is cleared, new information can be shifted into INPR by striking another key.

The process of outputting information:

- The output register OUTR works similarly but the direction of information flow is reversed.
- Initially, the output flag FGO is set to 1. The computer checks the flag bit; if it is 1, the information from AC is transferred in parallel to OUTR and FGO is cleared to 0. The output device accepts the coded information, prints the corresponding character, and when the operation is completed, it sets FGO to 1.
- The computer does not load a new character into OUTR when FGO is 0 because this condition indicates that the output device is in the process of printing the character.

11. Explain Input-Output instructions.

- Input and output instructions are needed for transferring information to and from AC register, for checking the flag bits, and for controlling the interrupt facility.
- Input-output instructions have an operation code 1111 and are recognized by the control when D7 = 1 and I = 1.
- The remaining bits of the instruction specify the particular operation.
- The control functions and microoperations for the input-output instructions are listed below.

INP	AC(0-7) ← INPR, FGI ← 0	Input char. to AC
OUT	OUTR ← AC(0-7), FGO ← 0	Output char. from AC
SKI	if(FGI = 1) then (PC ← PC + 1)	Skip on input flag
SKO	if(FGO = 1) then (PC ← PC + 1)	Skip on output flag
ION	IEN ← 1	Interrupt enable on
IOF	IEN ← 0	Interrupt enable off

Table 2.2: Input Output Instructions

- The INP instruction transfers the input information from INPR into the eight low-order bits of AC and also clears the input flag to 0.
- The OUT instruction transfers the eight least significant bits of AC into the output register OUTR and clears the output flag to 0.
- The next two instructions in Table 2.2 check the status of the flags and cause a skip of the next instruction if the flag is 1.
- The instruction that is skipped will normally be a branch instruction to return and check the flag again.
- The branch instruction is not skipped if the flag is 0. If the flag is 1, the branch instruction is skipped and an input or output instruction is executed.
- The last two instructions set and clear an interrupt enable flip-flop IEN. The purpose of IEN is explained in conjunction with the interrupt operation.

12. What is an Interrupt Cycle? Draw and explain flow chart of it.

- The way that the interrupt is handled by the computer can be explained by means of the flowchart shown in figure 2.13.
- An interrupt flip-flop R is included in the computer.
- When R = 0, the computer goes through an instruction cycle.
- During the execute phase of the instruction cycle IEN is checked by the control.
- If it is 0, it indicates that the programmer does not want to use the interrupt, so control continues with the next instruction cycle.
- If IEN is 1, control checks the flag bits.
- If both flags are 0, it indicates that neither the input nor the output registers are ready for transfer of information.
- In this case, control continues with the next instruction cycle. If either flag is set to 1 while IEN = 1, flip-flop R is set to 1.
- At the end of the execute phase, control checks the value of R, and if it is equal to 1, it goes to an interrupt cycle instead of an instruction cycle.

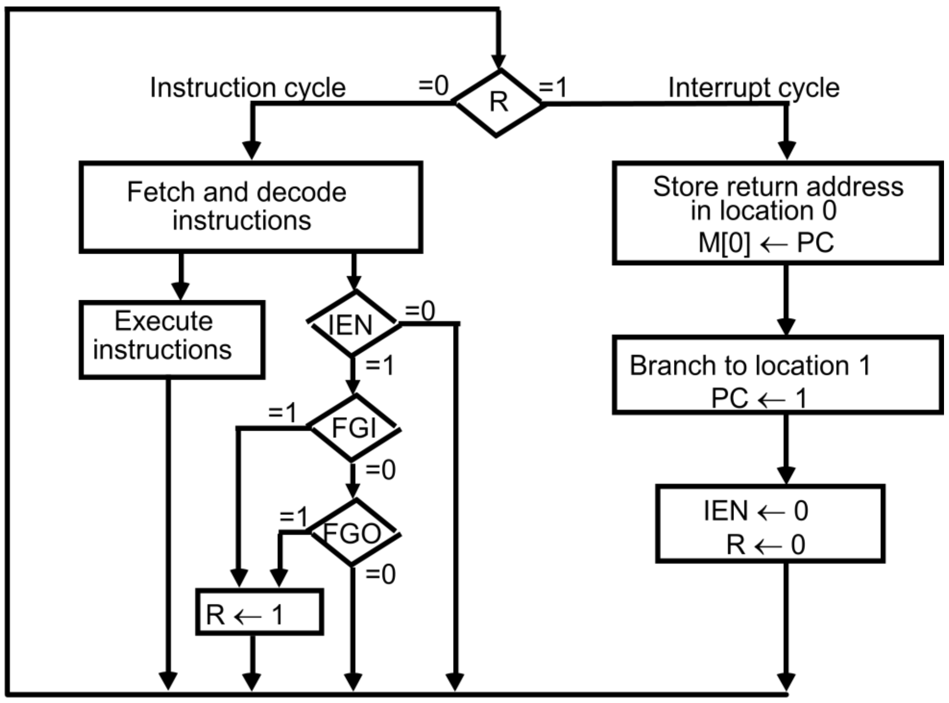


Figure 2.13: Flowchart for interrupt cycle

Interrupt Cycle

- The interrupt cycle is a hardware implementation of a branch and save return address operation.
- The return address available in PC is stored in a specific location where it can be found later when the program returns to the instruction at which it was interrupted. This location may be a processor register, a memory stack, or a specific memory location.
- Here we choose the memory location at address 0 as the place for storing the return

address.
- Control then inserts address 1 into PC and clears IEN and R so that no more interruptions can occur until the interrupt request from the flag has been serviced.
- An example that shows what happens during the interrupt cycle is shown in Figure 2.14:

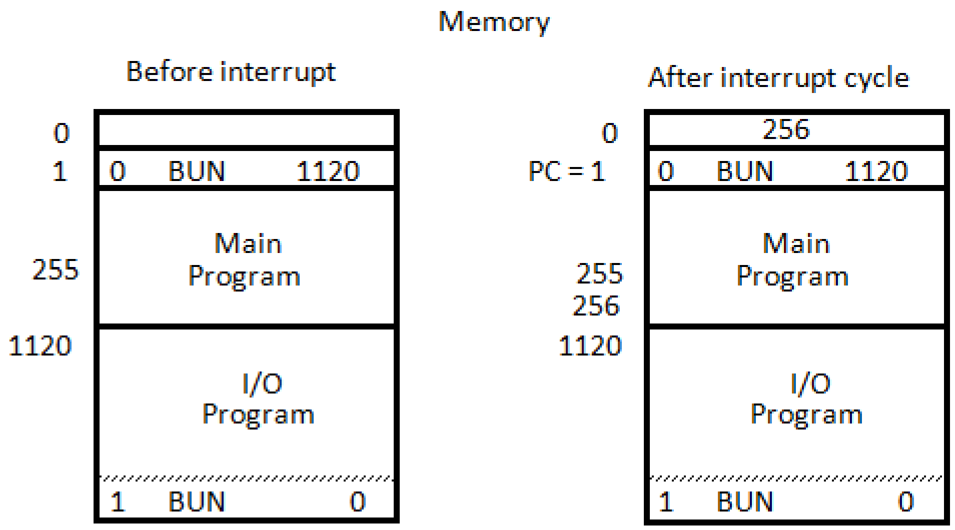


Figure 2.14: Demonstration of the interrupt cycle

- Suppose that an interrupt occurs and R = 1, while the control is executing the instruction at address 255. At this time, the return address 256 is in PC.
- The programmer has previously placed an input-output service program in memory starting from address 1120 and a BUN 1120 instruction at address 1.
- The content of PC (256) is stored in memory location 0, PC is set to 1, and R is cleared to 0.
- At the beginning of the next instruction cycle, the instruction that is read from memory is in address 1 since this is the content of PC. The branch instruction at address 1 causes the program to transfer to the input-output service program at address 1120.
- This program checks the flags, determines which flag is set, and then transfers the required input or output information. Once this is done, the instruction ION is executed to set IEN to 1 (to enable further interrupts), and the program returns to the location where it was interrupted.
- The instruction that returns the computer to the original place in the main program is a branch indirect instruction with an address part of 0. This instruction is placed at the end of the I/O service program.
- The execution of the indirect BUN instruction results in placing into PC the return address from location 0.

Register transfer statements for the interrupt cycle

- The flip-flop is set to 1 if IEN = 1 and either FGI or FGO are equal to 1. This can happen with any clock transition except when timing signals T_0, T_1 or T_2 are active.
- The condition for setting flip-flop R= 1 can be expressed with the following register transfer statement:

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

- The symbol + between FGI and FGO in the control function designates a logic OR operation. This is AND with IEN and $T_0'T_1' T_2'$.
- The fetch and decode phases of the instruction cycle must be modified and Replace T_0, T_1, T_2 with $R'T_0$, $R'T_1$, $R'T_2$
- Therefore the interrupt cycle statements are :
  RT_0: AR $\leftarrow$ 0, TR $\leftarrow$ PC
  RT_1: M[AR] $\leftarrow$ TR, PC $\leftarrow$ 0
  RT_2: PC $\leftarrow$ PC + 1, IEN $\leftarrow$ 0, R $\leftarrow$ 0, SC $\leftarrow$ 0
- During the first timing signal AR is cleared to 0, and the content of PC is transferred to the temporary register TR.
- With the second timing signal, the return address is stored in memory at location 0 and PC is cleared to 0.
- The third timing signal increments PC to 1, clears IEN and R, and control goes back to T_0 by clearing SC to 0.
- The beginning of the next instruction cycle has the condition RT_0 and the content of PC is equal to 1. The control then goes through an instruction cycle that fetches and executes the BUN instruction in location 1.

13. Draw and explain flow chart for computer operation.

- The final flowchart of the instruction cycle, including the interrupt cycle for the basic computer, is shown in Figure 2.15.
- The interrupt flip-flop R may be set at any time during the indirect or execute phases.
- The control returns to timing signal T_0 after SC is cleared to 0.
- If R = 1, the computer goes through an interrupt cycle. If R = 0, the computer goes through an instruction cycle.
- If the instruction is one of the memory-reference instructions, the computer first checks if there is an indirect address and then continues to execute the decoded instruction according to the flowchart.
- If the instruction is one of the register-reference instructions, it is executed with one of the microoperations register reference.
- If it is an input-output instruction, it is executed with one of the microoperation's input-output reference.

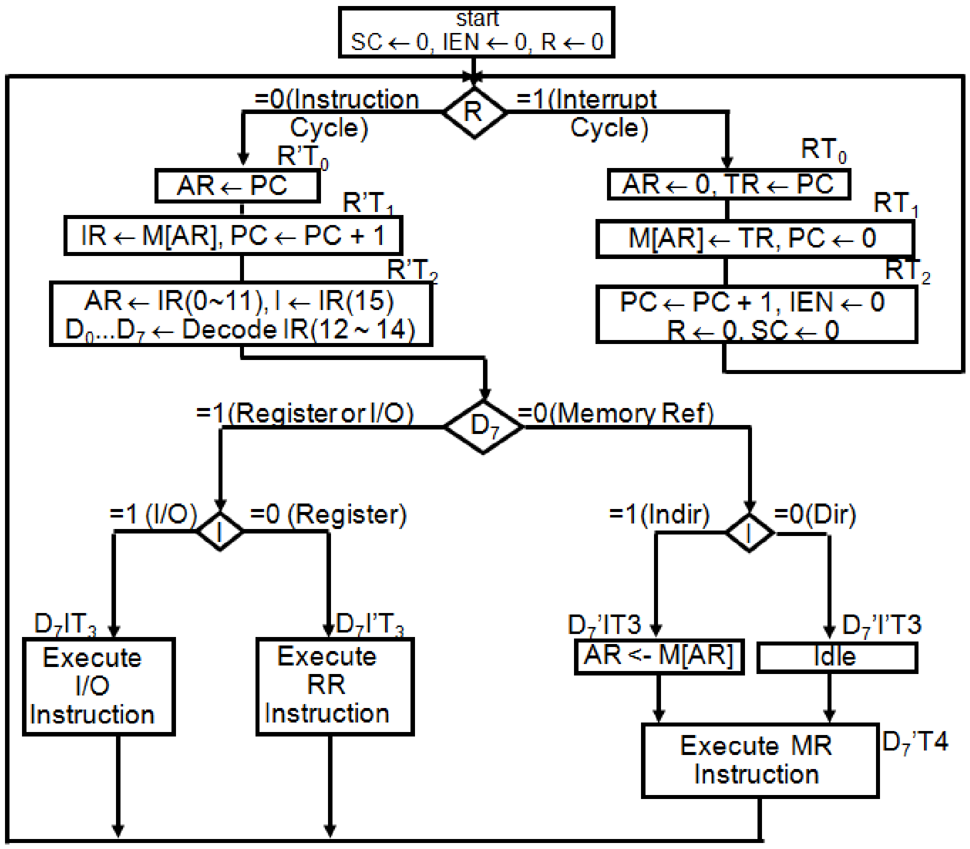


Figure 2.15: Flowchart for computer operation

14. Draw and explain design of Accumulator Logic

- The circuits associated with the AC register are shown in figure 2.16.

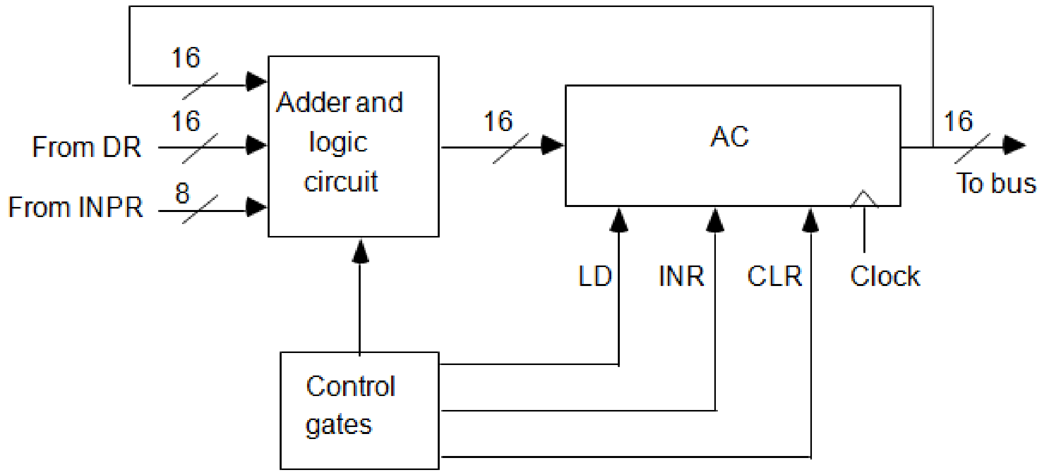


Figure 2.16: Circuits associated with AC

- The adder and logic circuit has three sets of inputs.
- One set of 16 inputs comes from the outputs of AC.
- Another set of 16 inputs comes from the data register DR.
- A third set of eight inputs comes from the input register INPR.
- The outputs of the adder and logic circuit provide the data inputs for the register.
- In addition, it is necessary to include logic gates for controlling the LD, INR, and CLR in the register and for controlling the operation of the adder and logic circuit.
- In order to design the logic associated with AC, it is necessary to extract all the statements that change the content of AC.

D_0T_5:	$AC \leftarrow AC \wedge DR$	AND with DR
D_1T_5:	$AC \leftarrow AC + DR$	Add with DR
D_2T_5:	$AC \leftarrow DR$	Transfer from DR
pB_{11}:	$AC(0\text{-}7) \leftarrow INPR$	Transfer from INPR
rB_9:	$AC \leftarrow AC'$	Complement
rB_7:	$AC \leftarrow \text{shr } AC, AC(15) \leftarrow E$	Shift right
rB_6:	$AC \leftarrow \text{shl } AC, AC(0) \leftarrow E$	Shift left
rB_{11}:	$AC \leftarrow 0$	Clear
rB_5:	$AC \leftarrow AC + 1$	Increment

- The gate structure that controls the LD, INR, and CLR inputs of AC is shown in figure 2.17.

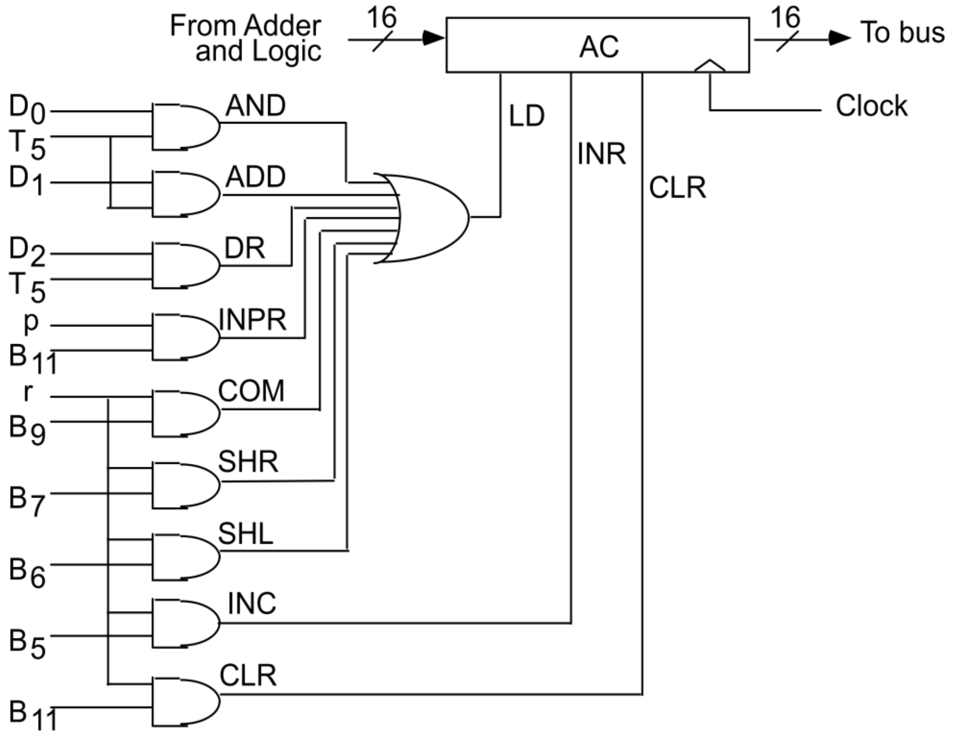


Figure 2.17: Gate structure for controlling the LD, INR, and CLR of AC

- The gate configuration is derived from the control functions in the list above.
- The control function for the clear microoperation is rB_{11}, where r = D7I'T3 and B_{11} = IR (11).
- The output of the AND gate that generates this control function is connected to the CLR input of the register.
- Similarly, the output of the gate that implements the increment microoperation is connected to the INR input of the register.
- The other seven microoperations are generated in the adder and logic circuit and are loaded into AC at the proper time.
- The outputs of the gates for each control function are marked with a symbolic name and used in the design of the adder and logic circuit.

15.

16. Draw and explain One stage of adder and logic circuit.

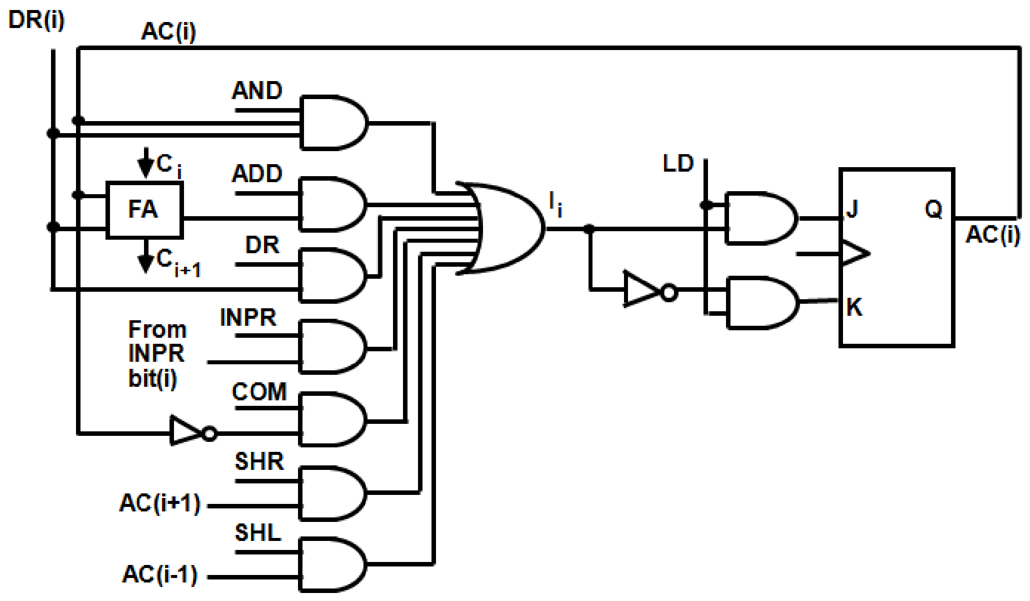


Figure 2.18: One stage of adder and logic circuit

- The adder and logic circuit can be subdivided into 16 stages, with each stage corresponding to one bit of AC. The internal construction of the register is as shown in Figure 2.18.
- We note that each stage has a JK flip-flop, two OR gates, and two AND gates. The load (LD) input is connected to the inputs of the AND gates.
- The input is labeled I_i, and the output AC(i).
- When the LD input is enabled, the 16 inputs I_i, for i = 0, 1, 2 , . . . , 15 are transferred to AC (0-15).
- One stage of the adder and logic circuit consists of seven AND gates, one OR gate and a full-adder (FA).
- The AND operation is achieved by ANDing AC(i) with the corresponding bit in the data register DR(i). The ADD operation is obtained using a binary adder.

- One stage of the adder uses a full-adder with the corresponding input and output carries.
- The transfer from INPR to AC is only for bits 0 through 7.
- The complement microoperation is obtained by inverting the bit value in AC.
- The shift-right operation transfers the bit from AC (i + 1), and the shift-left operation transfers the bit from AC (i - 1).
- The complete adder and logic circuit consists of 16 stages connected together.

1. Define Program and Categories of programs.

Program

A program is a list of instructions or statements for directing the computer to perform a required data-processing task.

Categories of programs

1. **Binary code:**
   This is a sequence of instructions and operands in binary that list the exact representation of instructions as they appear in computer memory.

2. **Octal or hexadecimal code:**
   This is an equivalent translation of the binary code to octal or hexadecimal representation.

3. **Symbolic code:**
   The user employs symbols (letters, numerals, or special characters) for the operation part, the address part, and other parts of the instruction code. Each symbolic instruction can be translated into one binary coded instruction. This translation is done by a special program called an *assembler*. Because an assembler translates the symbols, this type of symbolic program is referred to as an *assembly language program*.

4. **High-level programming languages:**
   These are special languages developed to reflect the procedures used in the solution of a problem rather than be concerned with the computer hardware behavior. An example of a high-level programming language is Fortran. It employs problem-oriented symbols and formats. The program is written in a sequence of statements in a form that people prefer to think in when solving a problem. However, each statement must be translated into a sequence of binary instructions before the program can be executed in a computer. The program that translates a high level language program to binary is called a *compiler*.

2. Explain Assembly language and also state the rules of language.

- The symbolic program (contains letters, numerals, or special characters) is referred to as an *assembly language program*.
- The basic unit of an assembly language program is a line of code.
- The specific language is defined by a set of rules that specify the symbols that can be used and they may be combined to form a line of code.

Rules of the Language

Each line of an assembly language program is arranged in three columns called fields. The fields specify the following information.

1. The ***label*** field may be empty or it may specify a symbolic address.
   A symbolic address consists of one, two, or three, but not more than three alphanumeric characters. The first character must be a letter; the next two may be letters or numerals. The symbol can be chosen arbitrarily by the programmer. A symbolic address in the label field is terminated by a comma so that it will be recognized as a label by the assembler.
2. The ***instruction*** field specifies a machine instruction or a pseudo instruction.
   The instruction field in an assembly language program may specify one of the following items:
   i. A memory-reference instruction (MRI)
   ii. A register-reference or input-output instruction (non-MRI)
   iii. A pseudo instruction with or without an operand
3. The ***comment*** field may be empty or it may include a comment.
   A line of code may or may not have a comment, but if it has, it must be preceded by a slash for the assembler to recognize the beginning of a comment field. Comments are useful for explaining the program and are helpful in understanding the step-by-step procedure taken by the program. Comments are inserted for explanation purpose only and are neglected during the binary translation process.

3. Explain pseudo instruction.

A pseudo instruction is not a machine instruction but rather an instruction to the assembler giving information about some phase of the translation. Four pseudo instructions that are recognized by the assembler are listed in Table 3.1.

Symbol	**Information for the Assembler**
ORG N	Hexadecimal number N is the memory location for the instruction or operand listed in the following line
END	Denotes the end of symbolic program
DEC N	Signed decimal number N to be converted to binary
HEX N	Hexadecimal number N to be converted to binary

Table3.1: Definition of Pseudo instructions

- The ORG (origin) pseudo instruction informs the assembler that the instruction or operand in the following line is to be placed in a memory location specified by the number next to ORG. It is possible to use ORG more than once in a program to specify more than one segment of memory.
- The END symbol is placed at the end of the program to inform the assembler that the program is terminated.
- The other two pseudo instructions (DEC and HEX) specify the radix of the operand and tell the assembler how to convert the listed number to a binary number.

4. Define Assembler and explain First Pass of an assembler with flow chart.

Assembler

- An assembler is a program that accepts a symbolic language program and produces its binary machine language equivalent.
- The input symbolic program is called the source program and the resulting binary program is called the object program.
- The assembler is a program that operates on character strings and produces an equivalent binary interpretation.

First Pass of an assembler

- During the first pass, it generates a table that correlates all user-defined address symbols with their binary equivalent value.
- The binary translation is done during the second pass.
- To keep track of the location of instructions, the assembler uses a memory word called a location counter (abbreviated LC).
- The content of LC stores the value of the memory location assigned to the instruction or operand presently being processed.
- The ORG pseudo instruction initializes the location counter to the value of the first location.
- Since instructions are stored in sequential locations, the content of LC is incremented by 1 after processing each line of code.
- To avoid ambiguity in case ORG is missing, the assembler sets the location counter to 0 initially.
- The tasks performed by the assembler during the first pass are described in the flowchart of figure 3.1.
- LC is initially set to 0.
- A line of symbolic code is analyzed to determine if it has a label (by the presence of a comma).
- If the line of code has no label, the assembler checks the symbol in the instruction field.
- If it contains an ORG pseudo instruction, the assembler sets LC to the number that follows ORG and goes back to process the next line.
- If the line has an END pseudo instruction, the assembler terminates the first pass and goes to the second pass.
- If the line of code contains a label, it is stored in the address symbol table together with its binary equivalent number specified by the content of LC Nothing is stored in the table if no label is encountered.
- LC is then incremented by 1 and a new line of code is processed.

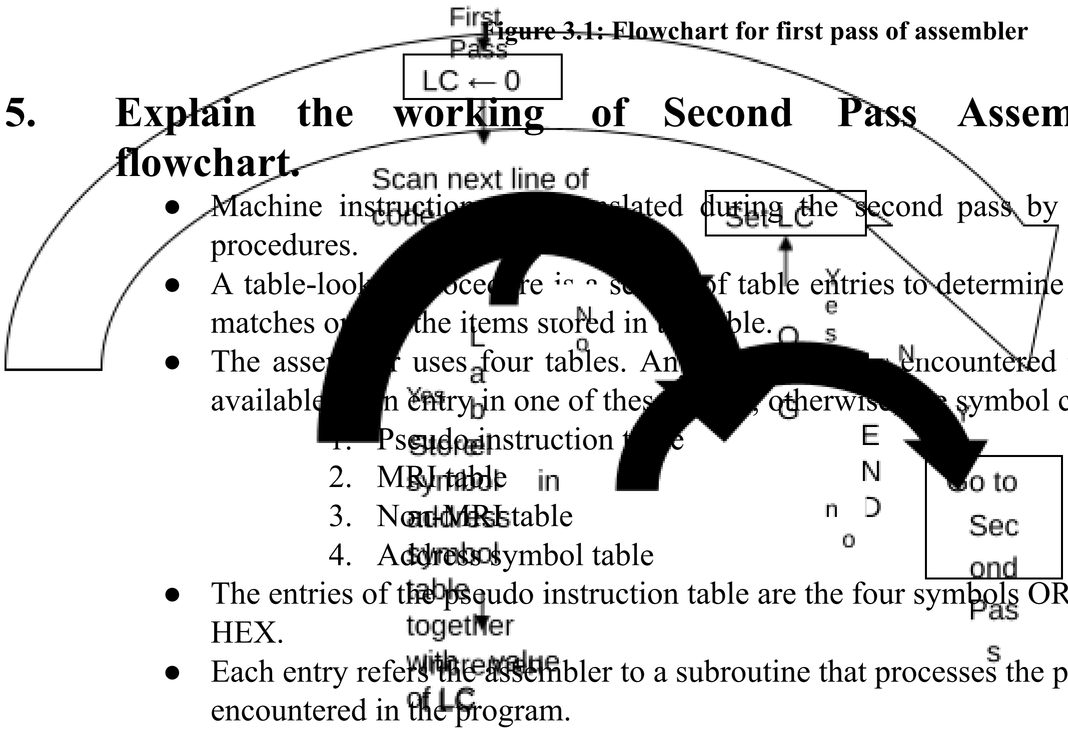


Figure 3.1: Flowchart for first pass of assembler

5. Explain the working of Second Pass Assembler with its flowchart.

- Machine instructions are translated during the second pass by means of table-lookup procedures.
- A table-lookup procedure is a search of table entries to determine whether a specific item matches one of the items stored in a table.
- The assembler uses four tables. Any symbol that is encountered in the program must be available as an entry in one of these tables; otherwise the symbol cannot be interpreted.
  1. Pseudo-instruction table
  2. MRI table
  3. Non-MRI table
  4. Address symbol table
- The entries of the pseudo instruction table are the four symbols ORG, END, DEC, and HEX.
- Each entry refers the assembler to a subroutine that processes the pseudo instruction when encountered in the program.
- The MRI table contains the seven symbols of the memory-reference instructions and their 3-bit operation code equivalent.
- The non-MRI table contains the symbols for the 18 register-reference and input-output

instructions and their 16-bit binary code equivalent.

- The address symbol table is generated during the first pass of the assembly process.
- The assembler searches these tables to find the symbol that it is currently processing in order to determine its binary value.
- The tasks performed by the assembler during the second pass are described in the flowchart of Figure 3.2.
- LC is initially set to 0.
- Lines of code are then analyzed one at a time.
- Labels are neglected during the second pass, so the assembler goes immediately to the instruction field and proceeds to check the first symbol encountered.
- It first checks the pseudo instruction table.
- A match with ORG sends the assembler to a subroutine that sets LC to an initial value.
- A match with END terminates the translation process. An operand pseudo instruction causes a conversion of the operand into binary.
- This operand is placed in the memory location specified by the content of LC.
- The location counter is then incremented by 1 and the assembler continues to analyze the next line of code.
- If the symbol encountered is not a pseudo instruction, the assembler refers to the MRI table.
- If the symbol is not found in this table, the assembler refers to the non-MRI table.
- A symbol found in the non-MRI table corresponds to a register reference or input-output instruction.
- The assembler stores the 16-bit instruction code into the memory word specified by LC.
- The location counter is incremented and a new line analyzed.
- When a symbol is found in the MRI table, the assembler extracts its equivalent 3-bit code and inserts it m bits 2 through 4 of a word.
- A memory reference instruction is specified by two or three symbols.
- The second symbol is a symbolic address and the third, which may or may not be present, is the letter I.
- The symbolic address is converted to binary by searching the address symbol table.
- The first bit of the instruction is set to 0 or 1, depending on whether the letter I is absent or present.
- The three parts of the binary instruction code are assembled and then stored in the memory location specified by the content of LC.
- The location counter is incremented and the assembler continues to process the next line.
- One important task of an assembler is to check for possible errors in the symbolic program. This is called ***error diagnostics***.
- One such error may be an invalid machine code symbol which is detected by its being absent in the MRI and non-MRI tables.
- The assembler cannot translate such a symbol because it does not know its binary equivalent value.

- In such a case, the assembler prints an error message to inform the programmer that his symbolic program has an error at a specific line of code.

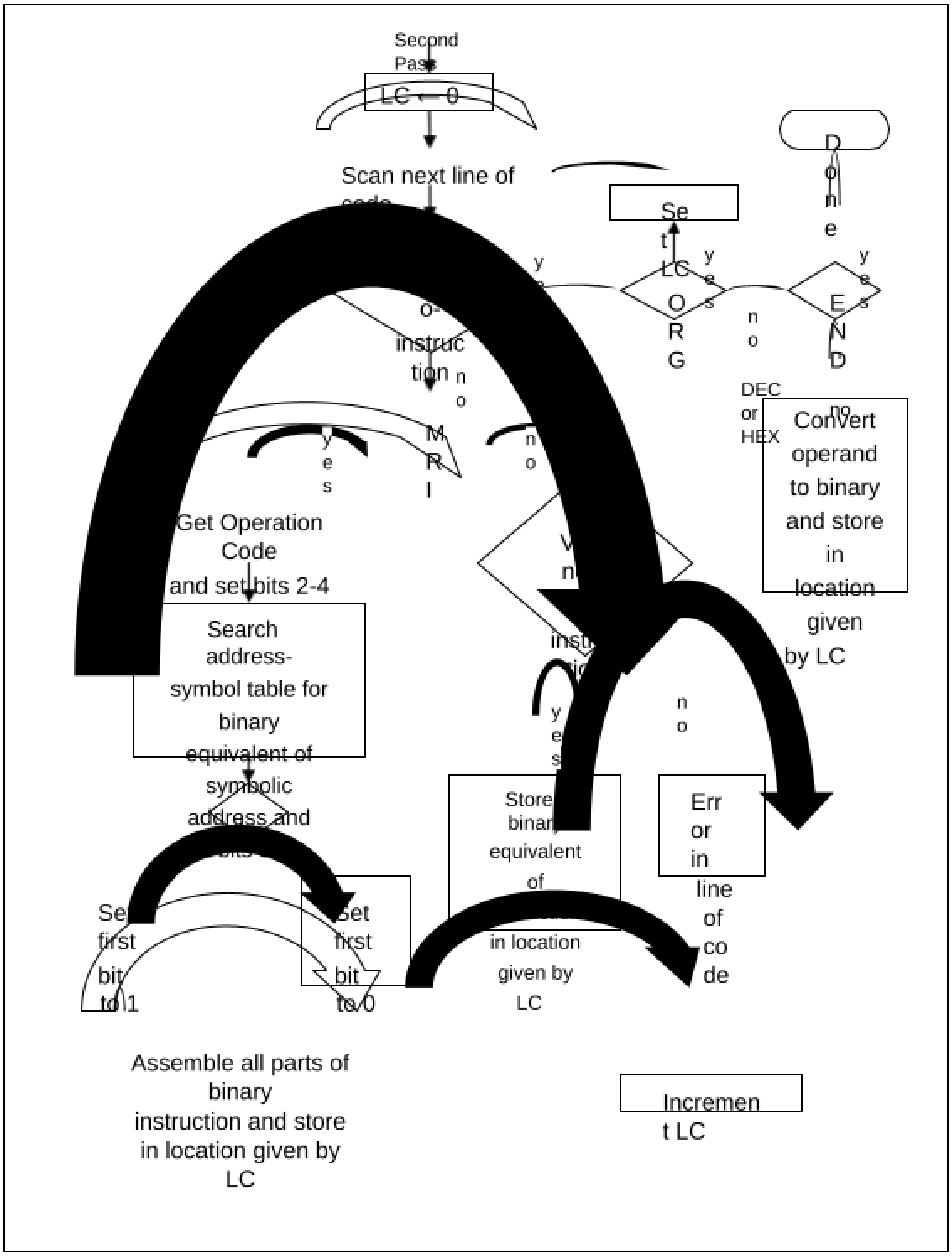

Second Pass
LC ← 0
Scan next line of code
Set LC
ORG
yes
no
END
yes
Done
Pseudo-instruction
no
yes
MRI
no
yes
Get Operation Code and set bits 2-4
Search address-symbol table for binary equivalent of symbolic address and set bits 5-16
Set first bit to 1
Set first bit to 0
Assemble all parts of binary instruction and store in location given by LC
DEC or HEX
no
Convert operand to binary and store in location given by LC
Valid non-MRI instruction
yes
no
Store binary equivalent of instruction in location given by LC
Error in line of code
Increment LC

Figure 3.2: Flowchart for second pass of assembler

6. Write short note on subroutine.

- The same piece of code must be written over again in many different parts of a program.
- Instead of repeating the code every time it is needed, there is an advantage if the common instructions are written only once.
- A set of common instructions that can be used in a program many times is called a ***subroutine***.
- Each time that a subroutine is used in the main part of the program, a branch is executed to the beginning of the subroutine.
- After the subroutine has been executed, a branch is returned to the main program.
- A subroutine consists of a self-contained sequence of instructions that carries out a given task.
- A branch can be made to the subroutine from any part of the main program.
- This poses the problem of how the subroutine knows which location to return to, since many different locations in the main program may make branches to the same subroutine.
- It is therefore necessary to store the return address somewhere in the computer for the subroutine to know where to return.
- Because branching to a subroutine and returning to the main program is such a common operation, all computers provide special instructions to facilitate subroutine entry and return.
- In the basic computer, the link between the main program and a subroutine is the BSA instruction (branch and save return address).

Example of Subroutine:

```
                ORG 100     /Main program
100             LDA X       /Load X
101             BSA SH4     /Branch to subroutine
102             STA X       /Store shifted number
103             LDA Y       /Load Y
104             BSA SH4     /Branch to subroutine again
105             STA Y       /Store shifted number
106             HLT
107     x,      HEX 1234
108     Y,      HEX 4321
                            /Subroutine to shift left 4 times
109     SH4,    HEX 0       /Store return address here
10A             CI          /Circulate left once
10B             L
10C             CI
10D             L           /Circulate left fourth time
10E             CI          /Set AC(13-16) to zero
10F             L           /Return to main program
110     MSK,    CI          /Mask operand
                L
                AND
                MSK
```

```
BUN
SH4     I
HEX
FFF0
END
```

7. Draw and explain flow chart for multiplication program.

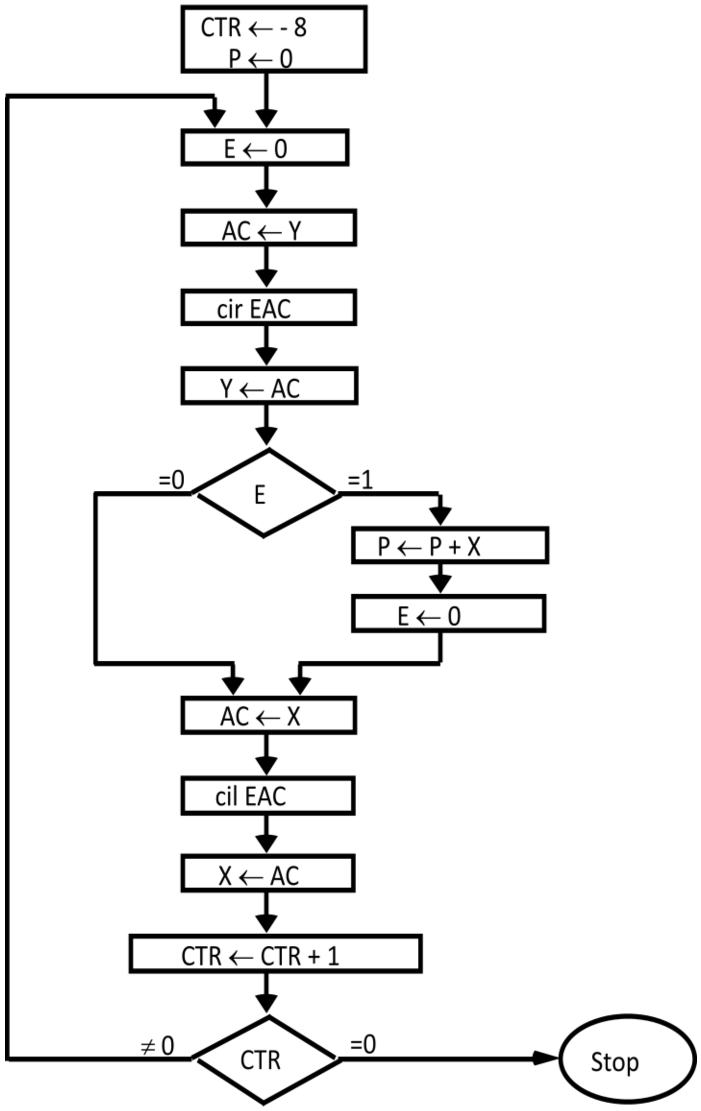


Figure 3.3: Flowchart for multiplication program

- The program for multiplying two numbers is based on the procedure we use to multiply numbers with paper and pencil.
- As shown in the numerical example of figure 3.3, the multiplication process consists of checking the bits of the multiplier Y and adding the multiplicand X as many times as there are 1 's in Y, provided that the value of X is shifted left from one line to the next.
- Since the computer can add only two numbers at a time, we reserve a memory location, denoted by P, to store intermediate sums.

- The intermediate sums are called partial products since they hold a partial product until all numbers are added.
- As shown in the numerical example under P, the partial product starts with zero.

- The multiplicand X is added to the content of P for each bit of the multiplier Y that is 1.
- The value of X is shifted left after checking each bit of the multiplier.
- The final value in P forms the product.
- The program has a loop that is traversed eight times, once for each significant bit of the multiplier.
- Initially, location X holds the multiplicand and location Y holds the multiplier.
- A counter CTR is set to - 8 and location P is cleared to zero.
- The multiplier bit can be checked if it is transferred to the E register.
- This is done by clearing E , loading the value of Y into the AC, circulating right E and AC and storing the shifted number back into location Y.
- This bit stored in E is the low-order bit of the multiplier.
- We now check the value of E. If it is 1, the multiplicand X is added to the partial product P. If it is 0, the partial product does not change.
- We then shift the value of X once to the left by loading it into the AC and circulating left E and AC.
- The loop is repeated eight times by incrementing location CTR and checking when it reaches zero.
- When the counter reaches zero, the program exits from the loop with the product stored in location P.

8. Write an assembly language program to Add two double precision numbers. (Sum 14)

```
         LDA AL          /Load A Low
         ADD BL          /Add B low , Carry in E
         STA CL          /Store in C low
         CLA             /Clear AC
         CIL             /Circulate to bring carry into
                         AC(16)
         ADD AH          /Add A high and carry
        ADD BH           /Add B high
         STA CH          /Store in C high
         HLT             /Increment Counter0
AL,      ------------    /Locations of operand
AH,      ------------
BL,      ------------
BH,      ------------
CL,      ------------
CH,      ------------
         END             /End of symbolic program
```

9. **Write an assembly language program to multiply two positive numbers. (Sum 14, Win11)**

```
        ORG 100
LOP,    CLE               /Clear E
        LDA Y             /Load multiplier
        CIR               /Transfer multiplier bit to E
        STA Y             /Store shifted multiplier
        SZE               /Check if bit is zero
       BUN ONE            /Bit is one, go to ONE
        BUN ZRO           /Bit is zero go to ZERO
ONE,   LDA X              /Load Multiplicand
        ADD P             /Add to partial product
        STA P             /Store partial product
        CLE               /Clear E
ZRO,    LDA X             /Load Multiplicand
        CIL               /Shift left
        STA X             /Store shifted multiplicand
        ISZ CTR          /Increment Counter
        BUN LOP           /Counter not zero ; repeat loop
        HLT               /Counter is zero halt
CTR,    DEC -8            /This location servers as a counter
X,      Hex 000F          /Multiplicand stored here
Y,      HEX 000B          /Multiplier stored here
P,      HEX 0             /Product formed here
        END               /End of symbolic program
```

10. **Write the program to multiply two positive numbers by a repeated addition method. For ex., to multiply 5 x 4, the program evaluates the product by adding 5 four times, or 5+5+5+5. (Win 10)**

```
        ORG 100       /Origin of program is HEX 100
        LDA X         /Load first address of operands
        STA PTR       /Store in pointer
        LDA NBR       /Load minus second operand
        STA Y         /Store in counter
        CLA           /Clear accumulator
LOP,    ADD PTR I     /Add an operand to AC
        ISZ Y         /Increment counter
        BUN LOP       /Repeat loop again
        STA MUL       /Store sum
        HLT           /Halt
X,      DEC 5         /First operand
PTR,    HEX 0         /This location reserved for a pointer
NBR,    DEC -4        /Second operand
CTR,    HEX 0         /This location reserved for a counter
MUL,    HEX 0         /Sum is stored here
        END           /End of symbolic program
```

11. **Write an assembly language program to take a character as input and outputs it. (Sum 13)**

Input a Character

```
CIF,    SKI           /Check input flag
        BUN CIF       /Flag=0,branch to check again
        INP           /Flag=1, input character
        OUT           /Print Character
        STA CHR       /Store Character
        HLT
CHR     --
```

Output a Character

```
        LDA CHR       /Load character into AC
COF,    SKO           /Check output flag
        BUN COF       /Flag=0, branch to check again
        OUT           /Flag=1, output character
        HLT
CHR,    HEX 0057      /Character is "W"
```

12. Write a Subroutine to Input and Pack Two Characters.

```
IN2,                        /Subroutine entry
FST,    SKI
        BUN FST
        INP                 /Input first character
        OUT
        BSA SH4             /Shift left four times
        BSA SH4             /Shift left four more times
SCD,    SKI
        BUN SCD
        INP                 /Input second character
        OUT
        BUN IN2 I           /Return
```

13. Write an ALP to transfer a block of 10 bytes from one location to other. (Win 14)

```
                            /Main program
        BSA MVE             /Branch to subroutine
        HEX 100             /First address of source data
        HEX 200             /First address of destination data
        DEC -10             /Number of items to move
        HLT
MVE,    HEX 0               /Subroutine MVE
        LDA MVE I           /Bring address of source
        STA PT1             /Store in first pointer
        ISZ MVE             /Increment return address
        LDA MVE I           /Bring address of destination
        STA PT2             /Store in second pointer
        ISZ MVE             /Increment return address
        LDA MVE I           /Bring number of items
        STA CTR             /Store in counter
        ISZ MVE             /Increment return address
LOP,    LDA PT1 I           /Load source item
        STA PT2 I           /Store in destination
        ISZ PT1             /Increment source pointer
        ISZ PT2             /Increment destination pointer
        ISZ CTR             /Increment Counter
        BUN LOP             /Repeat 10 times
        BUN MVE I           /Return to main program
PT1,    ---
PT2,    ---
CTR,    ---
```

14. Write an assembly level program for the following pseudo code. (Sum-10)
SUM = 0
SUM = SUM + A +
B DIF = DIF – C
SUM = SUM + DIF

```
         ORG 100
         LDA A          /Load value to accumulator
         ADD B          /addition AC=AC+B
         STA SUM        /Store result in destination
         LDA C          /Load value to accumulator
         CMA            /Complement accumulator
         STA C          /Store result in destination
         ISZ C          /Increment and skip if zero
         LDA DIF        /Load value to accumulator
         ADD C          /addition
         STA DIF        /Load value to accumulator
         HLT            /halt
A,       DEC 5
B,       DEC 4
C,       DEC 10
SUM,     HEX 0
DIF,     DEC 100
         END            /End of symbolic program
```

15. **Write the program to logically OR the two numbers. (Win-10)**

```
         ORG 100        /Origin of program is HEX 100
         LDA A          /Load first operand
         CMA            /Complement A
         STA AD         /Store at AD
         LDA B          /Load second operand
         CMA            /Complement B
         AND AD         /And A and B
         CMA            /Complement the result
         STA RES        /Store the result
         HLT            /Halt
A,       DEC 5          /First operand
AD,      HEX 0          /This location reserved A complement
B,       DEC 4          /Second operand
RES,     HEX 0          /This location reserved for Result
         END            /End of symbolic program
```

16. **Write a program loop using a pointer and a counter to clear the contents of hex locations 500 to 5FF with 0. (Win-09)**

```
        ORG 100      /Origin of program is HEX 100
        LDA ADS      /Load first address HEX 500
        STA PTR      /Store in pointer
        LDA NBR      /Load minus second operand
        STA CTR      /Store in counter
        CLA          /Clear accumulator
  LOP,  STA PTR I    /Clear memory location
        ISZ PTR      /Increment Pointer
        ISZ CTR      /Increment Counter
        BUN LOP      /Repeat loop again
        HLT          /Halt
  ADS,  DEC 500      /Starting memory location
  PTR,  HEX 0        /Starting pointer
  NBR,  DEC -256     /Second operand
  CTR,  HEX 0        /This location reserved for a counter
        END          /End of symbolic program
```

17. **Write a Symbolic Program to Add 100 Numbers.**

```
        ORG 100      /Origin of program is HEX 100
        LDA ADS      /Load first address of operands
        STA PTR      /Store in pointer
        LDA NBR      /Load minus 100
        STA CTR      /Store in counter
        CLA          /Clear accumulator
  LOP,  ADD PTR I    /Add an operand to AC
        ISZ PTR      /Increment pointer
        ISZ CTR      /Increment counter
        BUN LOP      /Repeat loop again
        STA SUM      /Store sum
        HLT          /Halt
  ADS,  HEX 150      /First address of operands
  PTR,  HEX 0        /This location reserved for a pointer
  NBR,  DEC -100     /Constant to initialized counter
  CTR,  HEX 0        /This location reserved for a counter
  SUM,  HEX 0        /Sum is stored here
        ORG 150      /Origin of operands is HEX 150
        DEC 75       /First operand
.
.
```

```
.
        DEC 23        /Last operand
        END           /End of symbolic program
```

18. Write an assembly language program for arithmetic right shift operation.

```
        CLE           /Clear E to 0
        SPA           /Skip if AC is positive; E remains
                      0
        CME           /AC is negative; set E to 1
        CIR           /Circulate E and AC
```

19. Write a Program to Store Input Characters in a Buffer.

```
        LDA ADS       /Load first address of buffer
        STA PTR       /Initialize pointer
  LOP,  BSA IN2       /Go to subroutine IN2 (see program no.12)
        STA PTR I     /Store double character word in buffer
        ISZ PTR       /Increment pointer
        BUN LOP       /Branch to input more characters
        HLT
  ADS,  HEX 500       /First address of buffer
  PTR,  HEX 0         /Location for pointer
```

20. Write a Program to Compare Two Words.

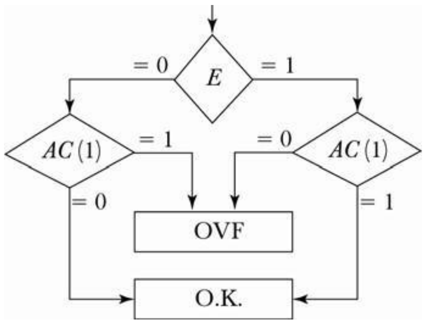
```
        LDA WD1       /Load first word
        CMA
        INC           /Form 2's complement
        ADD WD2       /Add second word
        SZA           /Skip if AC is zero
        BUN UEQ       /Branch to "unequal" routine
        BUN EQL       /Branch to "equal" routine
  WD1,  —
  WD2,  —
```

21. Write an assembly language program to subtract two double precision numbers. (Sum 13)

```
            CLE
            LDA BL          /Load Low Subtrahend B
            CMA             /Take one's complement
            INC             /Increment to form 2's complement
            ADD AL          /Add the Low Minuend
            STA CL          /Store low bit
            CLA             /Clear
            CIL             /left shift
            STA TMP         /Save Carry
            LDA BH          /Load High B
            CMA             /Complement BH
            ADD AH
            ADD TMP         /Add carry
            STA CH          /Store in C
            HLT
    TMP,    HEX 0
    C,      Hex 0
```

22. Write a program for the arithmetic shift-left operation. Branch to OVF if an overflow occurs.

```
            LDA X
            CLE
            CIL             /zero to low order bit; sign bit in E
            SZE
            BUN ONE
            SPA
            BUN OVF
            BUN EXT
    ONE,    SNA
            BUN OVF
    EXT,    HLT
```



23. **Write a program that evaluates the logic exclusive-OR of two logic operands.**

$z = x \oplus y = xy' + x'y = [(xy')' \cdot (x'y)']'$

```
        LDA Y
        CMA
        AND X
        CMA
        STA
        TMP
        LDA X
        CMA
        AND Y
        CMA
        AND
        TMP
        CMA
        STA Z
        HLT
X,      ---
Y,      ---
Z,      ---
TMP,    ---
```

1. Define the following.

Hardwired Control Unit:

When the control signals are generated by hardware using conventional logic design techniques, the control unit is said to be hardwired.

Micro programmed control unit:

A control unit whose binary control variables are stored in memory is called a micro programmed control unit.

Dynamic microprogramming:

A more advanced development known as *dynamic* microprogramming permits a microprogram to be loaded initially from an auxiliary memory such as a magnetic disk. Control units that use dynamic microprogramming employ a writable control memory. This type of memory can be used for writing.

Control Memory:

Control Memory is the storage in the microprogrammed control unit to store the microprogram.

Writeable Control Memory:

Control Storage whose contents can be modified, allow the change in microprogram and Instruction set can be changed or modified is referred as Writeable Control Memory**.**

Control Word:

The control variables at any given time can be represented by a control word string of 1 's and 0's called a control word.

2. Describe the following terms: Microoperation, Microinstruction, Micro program, Microcode.

Microoperations:

- In computer central processing units, micro-operations (also known as a micro- ops or μops) are detailed low-level instructions used in some designs to implement complex machine instructions (sometimes termed macro-instructions in this context).

Micro instruction:

- A symbolic microprogram can be translated into its binary equivalent by means of an assembler.
- Each line of the assembly language microprogram defines a symbolic microinstruction.
- Each symbolic microinstruction is divided into five fields: label, microoperations, CD, BR, and AD.

Micro program:

- A sequence of microinstructions constitutes a microprogram.
- Since alterations of the microprogram are not needed once the control unit is in operation, the control memory can be a read-only memory (ROM).
- ROM words are made permanent during the hardware production of the unit.
- The use of a micro program involves placing all control variables in words of ROM for use by the control unit through successive read operations.
- The content of the word in ROM at a given address specifies a microinstruction.

Microcode:

- Microinstructions can be saved by employing subroutines that use common sections of microcode.
- For example, the sequence of micro operations needed to generate the effective address of the operand for an instruction is common to all memory reference instructions.
- This sequence could be a subroutine that is called from within many other routines to execute the effective address computation.

3. Draw and explain the organization of micro programmed control unit.

- The general configuration of a micro-programmed control unit is demonstrated in the block diagram of Figure 4.1.
- The control memory is assumed to be a ROM, within which all control information is permanently stored.

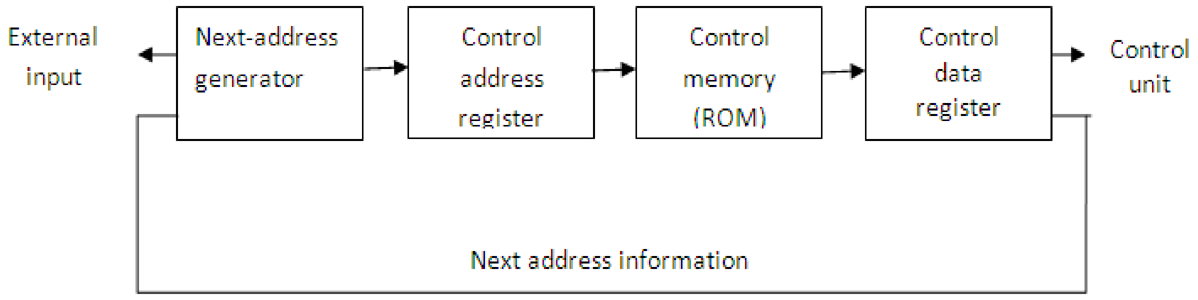


figure 4.1: Micro-programmed control organization

- The control memory address register specifies the address of the microinstruction, and 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. Once these operations are executed, the control must determine the next address.
- The location of the next microinstruction may be the one next in sequence, or it may be located somewhere else in the control memory.

- While the microoperations 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.
- Thus a microinstruction contains bits for initiating microoperations in the data processor part and bits that determine the address sequence for the control memory.
- The next address generator is sometimes called a ***micro-program sequencer**,* as it determines the address sequence that is read from control memory.
- Typical functions of a micro-program sequencer are incrementing the control address register by one, loading into the control address register an address from control memory, transferring an external address, or loading an initial address to start the control operations.
- The control data register holds the present microinstruction while the next address is computed and read from memory.
- The data register is sometimes called a ***pipeline register***.
- It allows the execution of the microoperations specified by the control word simultaneously with the generation of the next microinstruction.
- This configuration requires a two-phase clock, with one clock applied to the address register and the other to the data register.
- The main advantage of the micro programmed control is the fact that once the hardware configuration is established; there should be no need for further hardware or wiring changes.
- If we want to establish a different control sequence for the system, all we need to do is specify a different set of microinstructions for control memory.

4. Explain the steps of Address Sequencing in detail.

- Microinstructions are stored in control memory in groups, with each group specifying a ***routine***.
- To appreciate the address sequencing in a micro-program control unit, let us specify the steps that the control must undergo during the execution of a single computer instruction.

Step-1:

- An initial address is loaded into the control address register when power is turned on in the computer.
- This address is usually the address of the first microinstruction that activates the instruction fetch routine.
- The fetch routine may be sequenced by incrementing the control address register through the rest of its microinstructions.
- At the end of the fetch routine, the instruction is in the instruction register of the computer.

Step-2:

- The control memory next must go through the routine that determines the effective address of the operand.
- A machine instruction may have bits that specify various addressing modes, such as indirect address and index registers.
- The effective address computation routine in control memory can be reached through a branch microinstruction, which is conditioned on the status of the mode bits of the instruction.
- When the effective address computation routine is completed, the address of the operand is available in the memory address register.

Step-3:

- The next step is to generate the microoperations that execute the instruction fetched from memory.
- The microoperation steps to be generated in processor registers depend on the operation code part of the instruction.
- Each instruction has its own micro-program routine stored in a given location of control memory.
- The transformation from the instruction code bits to an address in control memory where the routine is located is referred to as a ***mapping*** process.
- A mapping procedure is a rule that transforms the instruction code into a control memory address.

Step-4:

- Once the required routine is reached, the microinstructions that execute the instruction may be sequenced by incrementing the control address register.
- Micro-programs that employ subroutines will require an external register for storing the return address.
- Return addresses cannot be stored in ROM because the unit has no writing capability.
- When the execution of the instruction is completed, control must return to the fetch routine.
- This is accomplished by executing an unconditional branch microinstruction to the first address of the fetch routine.

In summary, the address sequencing capabilities required in a control memory are:

1. Incrementing of the control address register.
2. Unconditional branch or conditional branch, depending on status bit conditions.
3. A mapping process from the bits of the instruction to an address for control memory.
4. A facility for subroutine call and return.

5. Draw and explain selection of address for control memory.

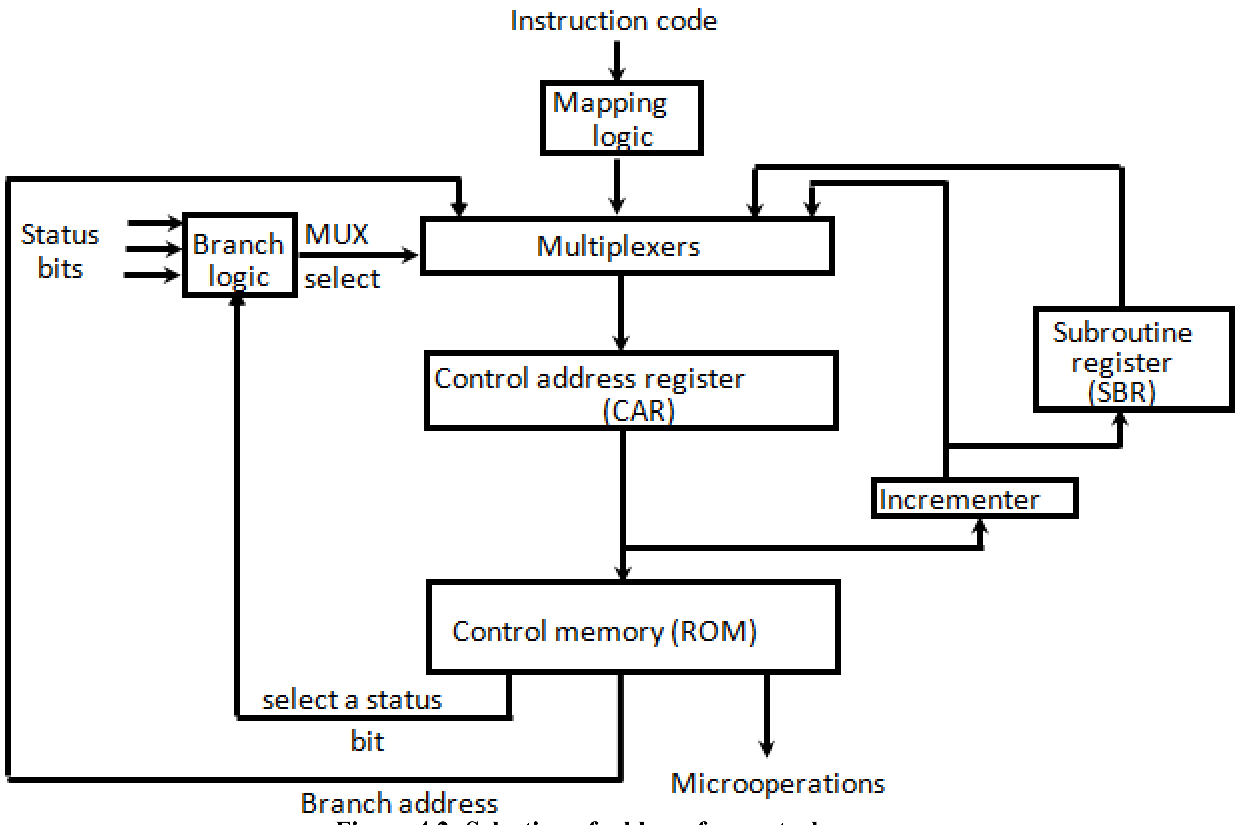


Figure 4.2: Selection of address for control memory

- Above figure 4.2 shows a block diagram of a control memory and the associated hardware needed for selecting the next microinstruction address.
- 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.
- The diagram shows four different paths from which the control address register (CAR) receives the address.
- The incrementer increments the content of the control address register by one, to select the next microinstruction in sequence.
- Branching is achieved by specifying the branch address in one of the fields of the microinstruction.
- Conditional branching is obtained by using part of the microinstruction to select a specific status bit in order to determine its condition.
- An external address is transferred into control memory via a mapping logic circuit.
- The return address for a subroutine is stored in a special register whose value is then used when the micro-program wishes to return from the subroutine.

- The branch logic of figure 4.2 provides decision-making capabilities in the control unit.
- 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 input or output status conditions.
- The status bits, together with the field in the microinstruction that specifies a branch address, control the conditional branch decisions generated in the branch logic.
- A 1 output in the multiplexer generates a control signal to transfer the branch address from the microinstruction into the control address register.
- A 0 output in the multiplexer causes the address register to be incremented.

6. Explain Mapping of an Instruction

- A special type of branch exists when a microinstruction specifies a branch to the first word in control memory where a microprogram routine for an instruction is located.
- The status bits for this type of branch are the bits in the operation code part of the instruction.
  For example, a computer with a simple instruction format as shown in figure 4.3 has an operation code of four bits which can specify up to 16 distinct instructions.
- Assume further that the control memory has 128 words, requiring an address of seven bits.
- One simple mapping process that converts the 4-bit operation code to a 7-bit address for control memory is shown in figure 4.3.
- This mapping consists of placing a 0 in the most significant bit of the address, transferring the four operation code bits, and clearing the two least significant bits of the control address register.
- This provides for each computer instruction a microprogram routine with a capacity of four microinstructions.
- If the routine needs more than four microinstructions, it can use addresses 1000000 through 1111111. If it uses fewer than four microinstructions, the unused memory locations would be available for other routines.

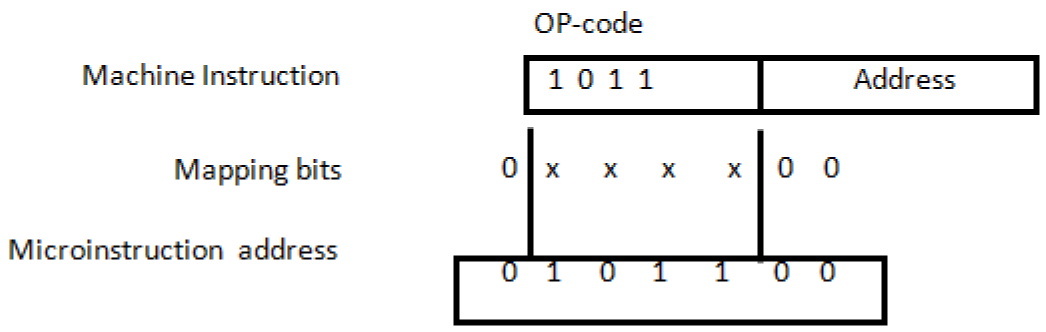


Figure 4.3: Mapping from instruction code to microinstruction address

- One can extend this concept to a more general mapping rule by using a ROM to specify the mapping function.
- The contents of the mapping ROM give the bits for the control address register.

- In this way the microprogram routine that executes the instruction can be placed in any desired location in control memory.
- The mapping concept provides flexibility for adding instructions for control memory as the need arises.

7. Draw and explain Computer Hardware Configuration in detail.

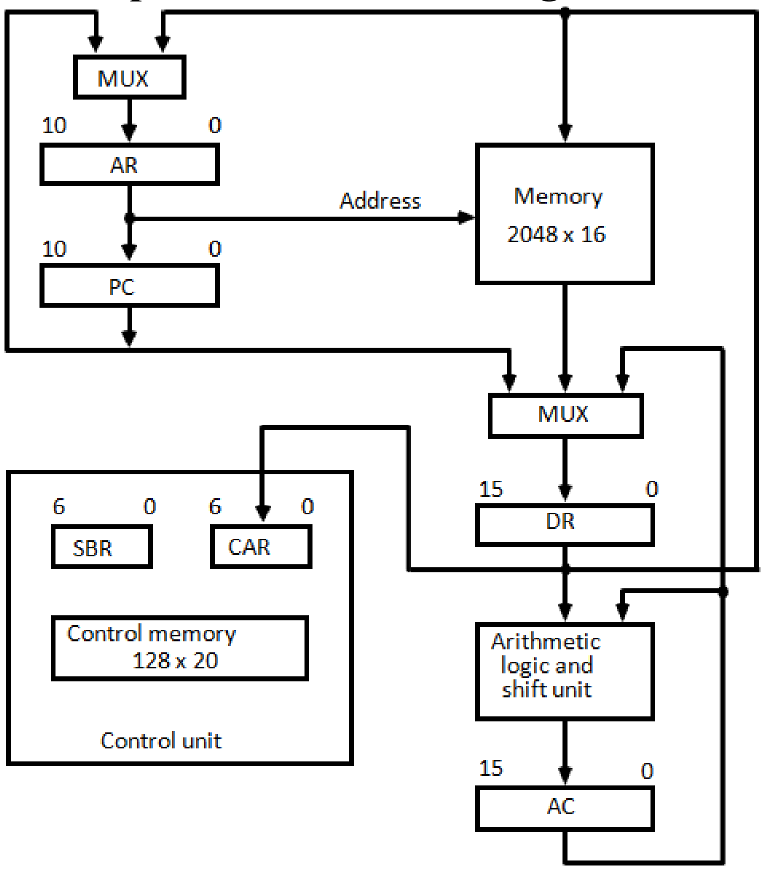


Figure 4.4: Computer hardware configuration

The block diagram of the computer is shown in Figure 4.4. It consists of

1. Two memory units:
   Main memory -> for storing instructions and data, and
   Control memory -> for storing the microprogram.
2. Six Registers:
   Processor unit register: AC(accumulator),PC(Program Counter), AR(Address Register), DR(Data Register)
   Control unit register: CAR (Control Address Register), SBR(Subroutine Register)
3. Multiplexers:
   The transfer of information among the registers in the processor is done through multiplexers rather than a common bus.
4. ALU:
   The arithmetic, logic, and shift unit performs microoperations with data from AC and DR and places the result in AC.

- DR can receive information from AC, PC, or memory.
- AR can receive information from PC or DR.
- PC can receive information only from AR.
- Input data written to memory come from DR, and data read from memory can go only to DR.

8. Explain Microinstruction Format in detail.

The microinstruction format for the control memory is shown in figure 4.5. The 20 bits of the microinstruction are divided into four functional parts as follows:

1. The three fields F1, F2, and F3 specify microoperations for the computer.
   The microoperations are subdivided into three fields of three bits each. The three bits in each field are encoded to specify seven distinct microoperations. This gives a total of 21 microoperations.
2. The CD field selects status bit conditions.
3. The BR field specifies the type of branch to be used.
4. The AD field contains a branch address. The address field is seven bits wide, since the control memory has 128 = 2^7 words.

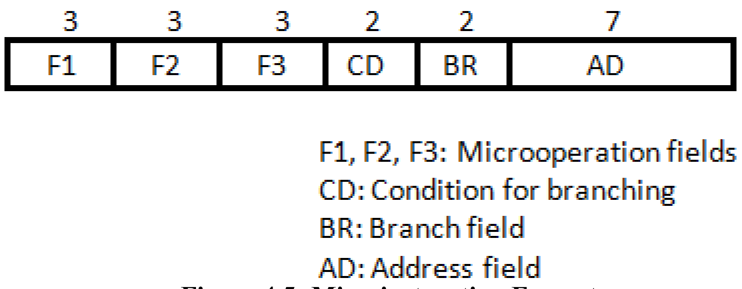

3
3
3
2
2
7
F1
F2
F3
CD
BR
AD
F1, F2, F3: Microoperation fields
CD: Condition for branching
BR: Branch field
AD: Address field

Figure 4.5: Microinstruction Format

- As an example, a microinstruction can specify two simultaneous microoperations from F2 and F3 and none from F1.

  DR $\leftarrow$ M[AR] with F2 =
  100 PC $\leftarrow$ PC + 1 with F3
  = 101
- The nine bits of the microoperation fields will then be 000 100 101.
- The CD (condition) field consists of two bits which are encoded to specify four status bit conditions as listed in Table 4.1.

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

Table 4.1: Condition Field

- The BR (branch) field consists of two bits. It is used, in conjunction with the address field AD, to choose the address of the next microinstruction shown in Table 4.2.

BR	Symbol	Function
00	JMP	CAR ← AD if condition = 1
		CAR ← CAR + 1 if condition = 0
01	CALL	CAR ← AD, SBR ← CAR + 1 if condition = 1
		CAR ← CAR + 1 if condition = 0
10	RET	CAR ← SBR (Return from subroutine)
11	MAP	CAR(2-5) ← DR(11-14), CAR(0,1,6) ← 0

Table 4.2: Branch Field

9. Explain Symbolic Microinstruction.

- Each line of the assembly language microprogram defines a symbolic microinstruction.
- Each symbolic microinstruction is divided into five fields: label, microoperations, CD, BR, and AD. The fields specify the following Table 4.3.

1.	Label	The label field may be empty or it may specify a symbolic address. A label is terminated with a colon (:).
2.	Microoperations	It consists of one, two, or three symbols, separated by commas, from those defined in Table 5.3. There may be no more than one symbol from each F field. The NOP symbol is used when the microinstruction has no microoperations. This will be translated by the assembler to nine zeros.
3.	CD	The CD field has one of the letters U, I, S, or Z.
4.	BR	The BR field contains one of the four symbols defined in Table 5.2.
5.	AD	The AD field specifies a value for the address field of the microinstruction in one of three possible ways: i. With a symbolic address, this must also appear as a label. ii. With the symbol NEXT to designate the next address in sequence. iii. When the BR field contains a RET or MAP symbol, the AD field is left empty and is converted to seven zeros by the assembler.

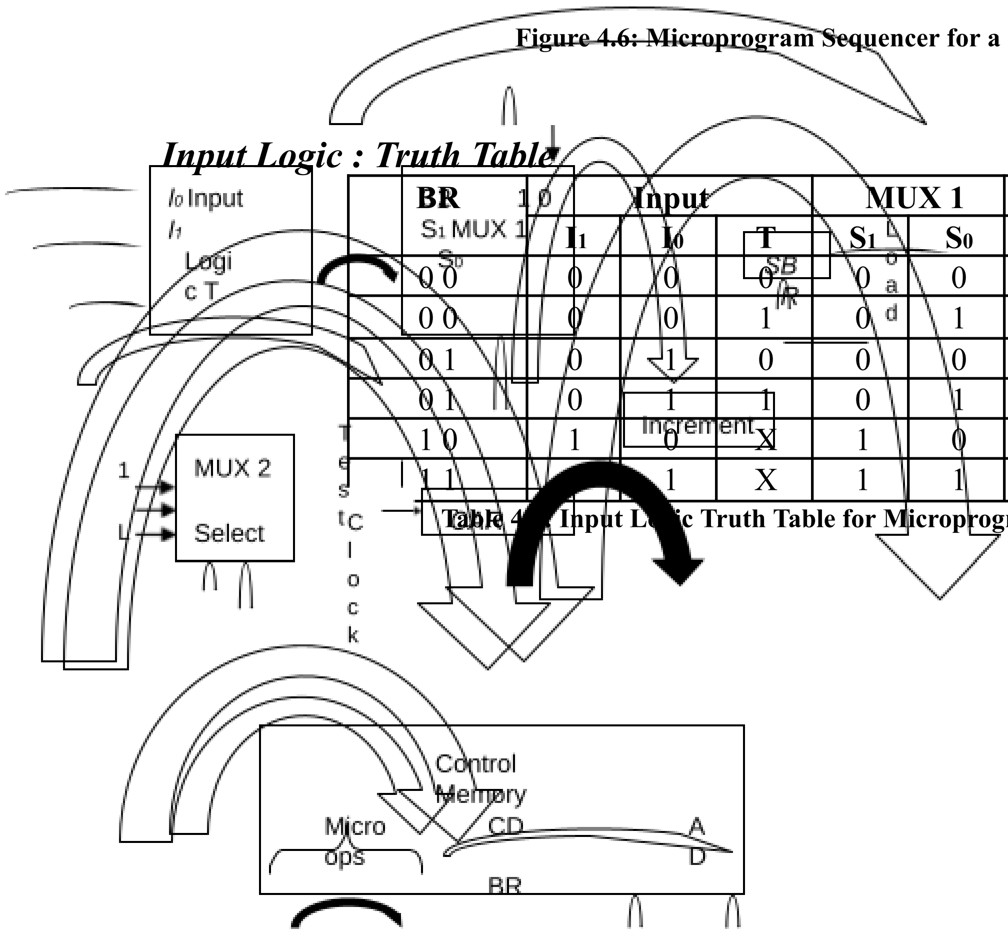
Table 4.3: Symbolic Microinstruction

10. Draw the diagram of Micro programmed sequencer for a control memory and explain it.

Microprogram sequencer:

- The basic components of a microprogrammed control unit are the control memory and the circuits that select the next address.
- The address selection part is called a microprogram sequencer.
- A microprogram sequencer can be constructed with digital functions to suit a particular application.
- To guarantee a wide range of acceptability, an integrated circuit sequencer must provide an internal organization that can be adapted to a wide range of applications.
- The purpose of a microprogram sequencer is to present an address to the control memory so that a microinstruction may be read and executed.
- Commercial sequencers include within the unit an internal register stack used for temporary storage of addresses during microprogram looping and subroutine calls.
- Some sequencers provide an output register which can function as the address register for the control memory.
- The block diagram of the microprogram sequencer is shown in figure 4.6.
- There are two multiplexers in the circuit.
- The first multiplexer selects an address from one of four sources and routes it into a control address register CAR.
- The second multiplexer tests the value of a selected status bit and the result of the test is applied to an input logic circuit.
- The output from CAR provides the address for the control memory.
- The content of CAR is incremented and applied to one of the multiplexer inputs and to the subroutine registers SBR.
- The other three inputs to multiplexer 1 come from the address field of the present microinstruction, from the output of SBR, and from an external source that maps the instruction.
- Although the figure 4.6 shows a single subroutine register, a typical sequencer will have a register stack about four to eight levels deep. In this way, a number of subroutines can be active at the same time.
- The CD (condition) field of the microinstruction selects one of the status bits in the second multiplexer.
- If the bit selected is equal to 1, the T (test) variable is equal to 1; otherwise, it is equal to 0.
- The T value together with the two bits from the BR (branch) field goes to an input logic circuit.
- The input logic in a particular sequencer will determine the type of operations that are available in the unit.

Figure 4.6: Microprogram Sequencer for a control memory



Input Logic : Truth Table

BR	Input			MUX 1		Load SBR
	I1	**I0**	**T**	**S1**	**S0**	**L**
0 0	0	0	0	0	0	0
0 0	0	0	1	0	1	0
0 1	0	1	0	0	0	0
0 1	0	1	1	0	1	1
1 0	1	0	X	1	0	0
1 1	[illegible]	1	X	1	1	0

Table 4.[illegible] Input Logic Truth Table for Microprogram Sequencer

Boolean Function:

$S_0 = I_0$

$S_1 = I_0 I_1 + I_0' T$

$L = I_0' I_1 T$

- Typical sequencer operations are: increment, branch or jump, call and return from subroutine, load an external address, push or pop the stack, and other address sequencing operations.
- With three inputs, the sequencer can provide up to eight address sequencing operations.
- Some commercial sequencers have three or four inputs in addition to the T input and thus provide a wider range of operations.

1. **What is stack? Give the organization of register stack with all necessary elements and explain the working of push and pop operations.** **(Win'13, Win'14, Win'15, Sum'15)**

Stack organization:

A stack is a storage device that stores information in such a manner that the item stored last is the first item retrieved.

The stack in digital computers is essentially a memory unit with an address register that can count only. The register that holds the address for the stack is called a stack pointer (SP) because its value always points at the top item in the stack.

The physical registers of a stack are always available for reading or writing. It is the content of the word that is inserted or deleted.

Register stack:

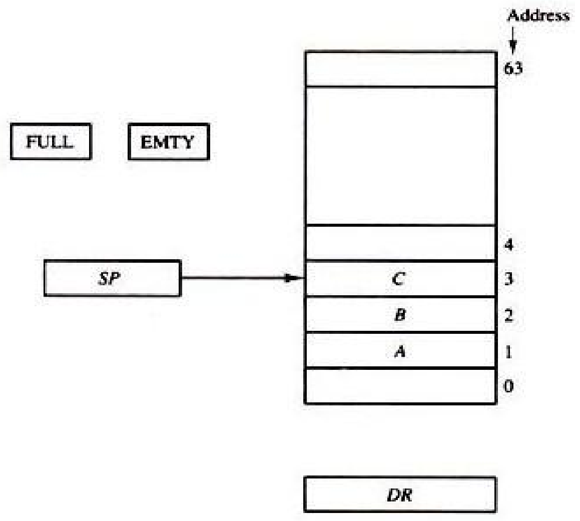


Figure 5.1: Block diagram of a 64-word stack

A stack can be placed in a portion of a large memory or it can be organized as a collection of a finite number of memory words or registers. Figure shows the organization of a 64-word register stack.

The stack pointer register SP contains a binary number whose value is equal to the address of the word that is currently on top of the stack. Three items are placed in the stack: A, B, and C, in that order. Item C is on top of the stack so that the content of SP is now 3.

To remove the top item, the stack is popped by reading the memory word at address 3 and decrementing the content of SP. Item B is now on top of the stack since SP holds address 2.

To insert a new item, the stack is pushed by incrementing SP and writing a word in the next-higher location in the stack.

In a 64-word stack, the stack pointer contains 6 bits because $2^6 = 64$.

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

PUSH:

If the stack is not full (FULL =0), a new item is inserted with a push operation. The push operation consists of the following sequences of microoperations:

$SP \leftarrow SP + 1$	Increment stack pointer
$M[SP] \leftarrow DR$	WRITE ITEM ON TOP OF THE STACK
IF ($SP = 0$) then ($FULL \leftarrow 1$)	Check is stack is full
$EMTY \leftarrow 0$	Mark the stack not empty

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

POP:

A new item is deleted from the stack if the stack is not empty (if EMTY = 0). The pop operation consists of the following sequences of microoperations:

$DR \leftarrow M[SP]$	Read item on top of the stack
$SP \leftarrow SP - 1$	Decrement stack pointer
IF ($SP = 0$) then ($EMTY \leftarrow 1$)	Check if stack is empty
$FULL \leftarrow 0$	Mark the stack not full

The top item is read from the stack into DR. The stack pointer is then decremented. If its value reaches zero, the stack is empty, so EMTY is set to 1.

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

2. Explain Memory Stack. (Win'14, Win'15)

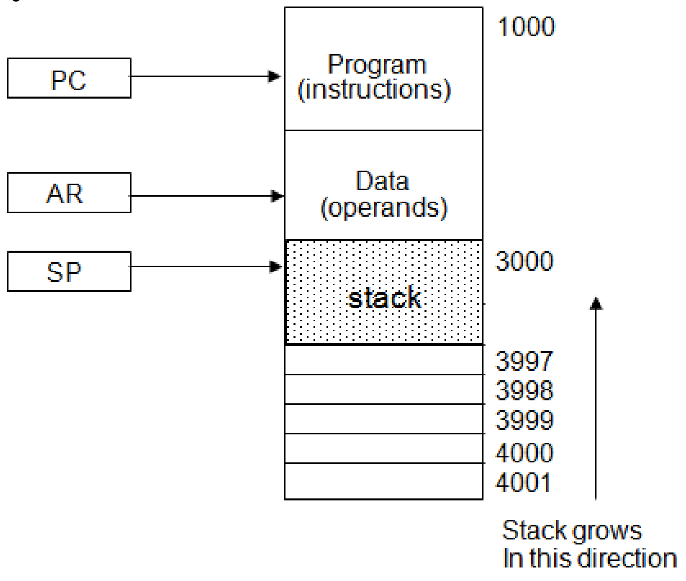


Figure 5.2: Computer memory with program, data, and stack segments

The implementation of a stack in the CPU is done by assigning a portion of memory to a stack operation and using a processor register as a stack pointer.
Figure 5.2 shows a portion of computer memory partitioned into three segments: program, data, and stack.
The program counter PC points at the address of the next instruction in the program which is used during the fetch phase to read an instruction.
The address registers AR points at an array of data which is used during the execute phase to read an operand.
The stack pointer SP points at the top of the stack which is used to push or pop items into or from the stack.
The three registers are connected to a common address bus, and either one can provide an address for memory.
As shown in Figure 5.2, the initial value of SP is 4001 and the stack grows with decreasing addresses. Thus the first item stored in the stack is at address 4000, the second item is stored at address 3999, and the last address that can be used for the stack is 3000.
We assume that the items in the stack communicate with a data register DR.

PUSH

A new item is inserted with the push operation as follows:

SP ← SP - 1
M[SP] ← DR

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

POP

A new item is deleted with a pop operation as follows:

DR ←
M[SP] SP
← SP + 1

The top item is read from the stack into DR.
The stack pointer is then incremented to point at the next item in the stack.
The two microoperations needed for either the push or pop are (1) an access to memory through SP, and (2) updating SP.
Which of the two microoperations is done first and whether SP is updated by incrementing or decrementing depends on the organization of the stack.
In figure. 5.2 the stack grows by decreasing the memory address. The stack may be constructed to grow by increasing the memory also.
The advantage of a memory stack is that the CPU can refer to it without having to specify an address, since the address is always available and automatically updated in the stack pointer.

3. Explain four types of instruction formats. (Sum'11, Win'13)

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.

```
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
```

The advantage of three-address format is that it results in short programs when evaluating arithmetic expressions.
The disadvantage is that the binary-coded instructions require too many bits to specify three addresses.
An example of a commercial computer that uses three-address instruction is the Cyber 170.

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 these is a need for a second register.
However, here we will neglect the second register and assume that the AC contains the result of all operations. The program to evaluate X = (A + B) * (C + D) is

LOAD	A	AC ← M [A]
ADD	B	AC ← AC + M [B]
STORE	T	M [T] ← AC
LOAD	C	AC ← M [C]
ADD	D	AC ← AC + M [D]
MUL	T	AC ← AC * M [T]
STORE	X	M [X] ← AC

All the operations are done between the AC register and a memory operand. T is the address of the 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 program to evaluate X = (A + B) * (C + D) will be written for a stack-organized computer.

PUSH	A	TOS ← A
PUSH	B	TOS ← B
ADD		TOS ← (A + B)
PUSH	C	TOS ← B
PUSH	D	TOS ← D

ADD		TOS ⋧ (C + D)
MUL		TOS ⋧ (C + D) * (A + B)
POP	X	M [X] ⋧ TOS

To evaluate arithmetic expressions in a stack computer, it is necessary to convert the expression into reverse polish notation.

RISC Instructions:

All other instructions are executed within the registers of the CPU without referring to memory. A program for a RISC type CPU consists of LOAD and STORE instructions that have one memory and one register address, and computational-type instructions that have three addresses with all three specifying processor registers.

The following is a program to evaluate X = (A + B) * (C + D).

LOA	R1,	R1 ≋ M
D	A	[A] R1 ≋
LOA	R1,	M [B] R1
D	B	≋ M [C]
LOA	R1,	R1 ≋ M [D]
D	C	
LOAD	R1, D	
ADD	R1, R1, R2	R1 ≋ R1 + R2
ADD	R3, R3, R2	R3 ≋ R3 + R4
MUL	R1, R1, R3	R1 ≋ R1 * R3
STORE	X, R1	M [X] ≋ R1

The load instructions transfer the operands from memory to CPU register.

Add and multiply operations are executed with data in the registers without accessing memory.

The result of the computations is then stored in memory with a store instruction.

4. Write a note on different Addressing Modes.

(Win'15, Sum'15, Win'14, Win'13)

The general addressing modes supported by the computer processor are as follows:

1) **Implied mode:**

In this mode the operands are specified implicitly in the definition of the definition of the instruction. For example, the instruction "complement accumulator" is an implied-mode instruction because the operand in the 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 later register is implied in the definition of the instruction. In fact, all register reference instructions that use an accumulator are implied mode instructions.

2) **Immediate Mode:**

In this mode the operand is specified in the instruction itself. In other 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 of instructions is useful for initializing register to constant value.

3) **Register Mode:**

In this mode the operands are in registers that within the CPU. The particular register is selected from a register field in the instruction.

A k-bit field can specify any one of 2^k registers.

4) **Register Indirect Mode:**

In this mode the instruction specifies a register in the CPU whose contents give the address of the operand in memory.

Before using a register indirect mode instruction, the programmer must ensure that the memory address of the operand is placed in the processor register with a previous instruction.

The advantage of this mode is that address field of the instruction uses fewer bits to select a register than would have been required to specify a memory address directly.

5) **Autoincrement or Autodecrement Mode:**

This is similar to the register indirect mode expect 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.

6) **Direct Address Mode:**

In this mode the effective address is equal to the address part of the instruction. The operand resides in memory and its address is given directly by the address field of the instruction.

7) **Indirect Address Mode:**

In this mode the address field of the instruction gives the address 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. The effective address in this mode is obtained from the following computational:

Effective address = address part of instruction + content of CPU register

8) **Relative Address Mode:**

In this mode the content of the program counter is added to the address of the instruction in order to obtain the effective address. The address part of the instruction is usually a signed number 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.

Relative addressing is often used with branch-type instruction when the branch address is in the area surrounding the instruction word itself.

9) **Indexed Addressing Mode:**
In this mode the content of an index register is added to the address part of the instruction to obtain the effective address.
The indexed register is a special CPU register that contain 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 begging address.
The distance between the beginning address and the address of the operand is the index value stored in the index register.

10) **Base Register Addressing Mode:**
In this mode the content of a base register is added to the address part of the instruction to obtain the effective address.
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.
With a base register, the displacement values of instruction do not have to change. Only the value of the base register requires updating to reflect the beginning of a new memory segment.

5. Explain Data Transfer Instructions.

Data transfer instructions move data from one place in the computer to another without changing the data content.
The most common transfers are between memory and processor registers, between processor registers and input or output, and between the processor registers themselves.
The ***load*** instruction has been used mostly to designate a transfer from memory to a processor register, usually an accumulator.
The ***store*** instruction designates a transfer from a processor register into memory.
The ***move*** instruction has been used in computers with multiple CPU registers to designate a transfer from one register to another. It has also been used for data transfers between CPU registers and memory or between two memory words.
The ***exchange*** instruction swaps information between two registers or a register and a memory word.
The ***input and output*** instructions transfer data among processor registers and input or output terminals.
The ***push and pop*** instructions transfer data between processor registers and a memory stack.

6. Explain Arithmetic instructions.

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

7. Explain Logical instructions.

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

8. Explain shift instructions.

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

9. What are status register bits? Draw and explain the block diagram showing all status registers. (Win'13)

It is sometimes convenient to supplement the ALU circuit in the CPU with a status register where status bit conditions be stored for further analysis. Status bits are also called condition-code bits or flag bits.

Figure 5.3 shows the block diagram of an 8-bit ALU with a 4-bit status register. The four status bits are symbolized by C, S, Z, and V. The bits are set or cleared as a result of an operation performed in the ALU.

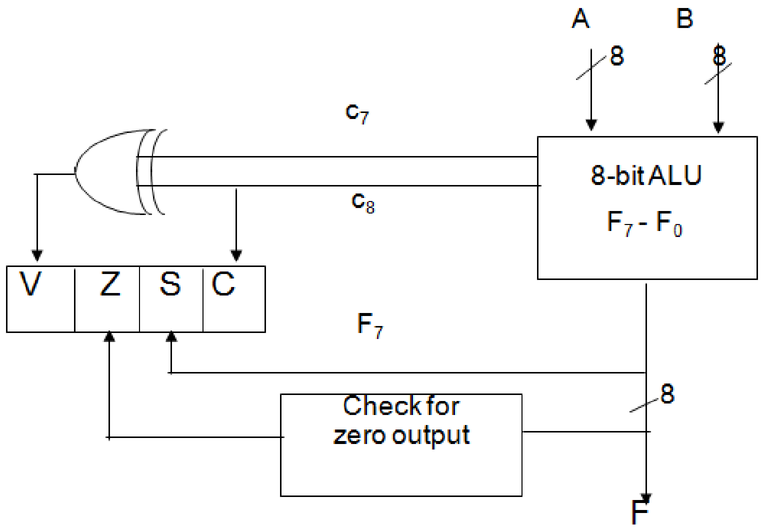


Figure 5.3: Status Register Bits

1. Bit C (carry) is set to 1 if the end carry C_8 is 1. It is cleared to 0 if the carry is 0.
2. Bit S (sign) is set to 1 if the highest-order bit F_7 is 1. It is set to 0 if set to 0 if the bit is 0.
3. Bit Z (zero) is set to 1 if the output of the ALU contains all 0's. it is cleared to 0 otherwise. In other words, Z = 1 if the output is zero and Z = 0 if the output is not zero.
4. Bit V (overflow) is set to 1 if the exclusives-OR of the last two carries is equal to 1, and cleared to 0 otherwise. This is the condition for an overflow when negative numbers are in 2's complement. For the 8-bit ALU, V = 1 if the output is greater than + 127 or less than -128.

The status bits can be checked after an ALU operation to determine certain relationships that exist between the vales of A and B.

If bit V is set after the addition of two signed numbers, it indicates an overflow condition.

If Z is set after an exclusive-OR operation, it indicates that A = B.

A single bit in A can be checked to determine if it is 0 or 1 by masking all bits except the bit in question and then checking the Z status bit.

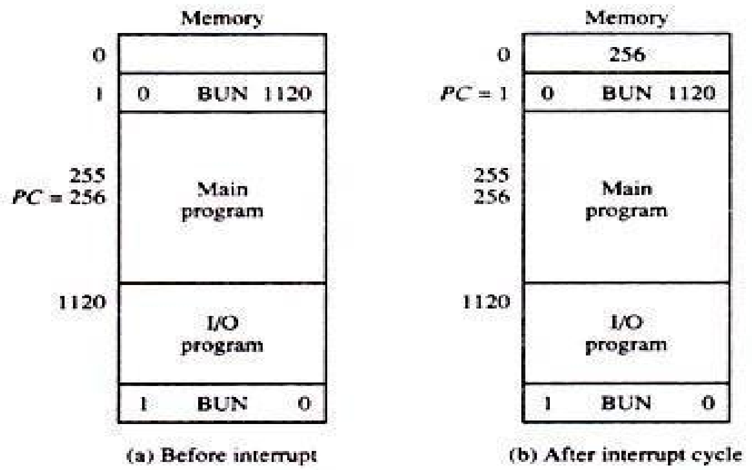
10. What is program interrupt? What happens when it comes? What are the tasks to be performed by service routine?
OR
Explain Program Interrupts. Explain clearly, discussing the role of stack, PSW and return from interrupt instruction, how interrupts are implemented on computers.

The concept of program interrupt is used to handle a variety of problems that arise out of normal program sequence.
Program interrupt refers to the transfer of program control from a currently running program to another service program as a result of an external or internal generated request. Control returns to the original program after the service program is executed.
After a program has been interrupted and the service routine been executed, the CPU must return to exactly the same state that it was when the interrupt occurred.
Only if this happens will the interrupted program be able to resume exactly as if nothing had happened.
The state of the CPU at the end of the execute cycle (when the interrupt is recognized) is determined from:

1. The content of the program counter
2. The content of all processor registers
3. The content of certain status conditions

The interrupt facility allows the running program to proceed until the input or output device sets its ready flag. Whenever a flag is set to 1, the computer completes the execution of the instruction in progress and then acknowledges the interrupt.
The result of this action is that the retune address is stared in location 0. The instruction in location 1 is then performed; this initiates a service routine for the input or output transfer. The service routine can be stored in location 1.
The service routine must have instructions to perform the following tasks:

1. Save contents of processor registers.
2. Check which flag is set.
3. Service the device whose flag is set.
4. Restore contents of processor registers.
5. Turn the interrupt facility on.
6. Return to the running program.



(a) Before interrupt (b) After interrupt cycle

11. Explain various types of interrupts.

There are three major types of interrupts that cause a break in the normal execution of a program. They can be classified as:

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

1) <u>External interrupts:</u>

External interrupts come from input-output (I/0) devices, from a timing device, from a circuit monitoring the power supply, or from any other external source.

Examples that cause external interrupts are I/0 device requesting transfer of data, I/o device finished transfer of data, elapsed time of an event, or power failure. Timeout interrupt may result from a program that is in an endless loop and thus exceeded its time allocation.

Power failure interrupt may have as its service routine a program that transfers the complete state of the CPU into a nondestructive memory in the few milliseconds before power ceases.

External interrupts are asynchronous. External interrupts depend on external conditions that are independent of the program being executed at the time.

2) <u>Internal interrupts:</u>

Internal interrupts arise from illegal or erroneous use of an instruction or data. Internal interrupts are also called traps.

Examples of interrupts caused by internal error conditions are register overflow, attempt to divide by zero, an invalid operation code, stack overflow, and protection violation. These error conditions usually occur as a result of a premature termination of the instruction execution. The service program that processes the internal interrupt determines the corrective measure to be taken.

Internal interrupts are synchronous with the program. . If the program is rerun, the internal interrupts will occur in the same place each time.

3) Software interrupts:

A software interrupt is a special call instruction that behaves like an interrupt rather than a subroutine call. It can be used by the programmer to initiate an interrupt procedure at any desired point in the program.

The most common use of software interrupt is associated with a supervisor call instruction. This instruction provides means for switching from a CPU user mode to the supervisor mode. Certain operations in the computer may be assigned to the supervisor mode only, as for example, a complex input or output transfer procedure. A program written by a user must run in the user mode.

When an input or output transfer is required, the supervisor mode is requested by means of a supervisor call instruction. This instruction causes a software interrupt that stores the old CPU state and brings in a new PSW that belongs to the supervisor mode.

The calling program must pass information to the operating system in order to specify the particular task requested.

12. What do you understand by Reduced Instruction Set Computers? What are Complex Instruction Set Computers? List important characteristics of CISC and RISC computers. Also in a tabular form compare their relative advantages / disadvantages. (Win'15, Sum'15, Win'14, Win'13)

Characteristics of RISC:

1. Relatively few instructions
2. Relatively few addressing modes
3. Memory access limited to load and store instructions
4. All operations done within the registers of the CPU
5. Fixed-length, easily decoded instruction format
6. Single-cycle instruction execution
7. Hardwired rather than microprogrammed control
8. A relatively large number of registers in the processor unit
9. Use of overlapped register windows to speed-up procedure call and return
10. Efficient instruction pipeline
11. Compiler support for efficient translation of high-level language programs into machine language programs

Characteristics of CISC:

1. A larger number of instructions – typically from 100 to 250 instructions
2. Some instructions that perform specialized tasks and are used infrequently
3. A large variety of addressing modes – typically from 5 to 20 different modes
4. Variable-length instruction formats
5. Instructions that manipulate operands in memory

13. What is overlapped register window? How the window size and register file size is calculated? (Win'13)

A characteristic of some RISC processors is their use of overlapped register windows to provide the passing of parameters and avoid the need for saving and restoring register values. Each procedure call results in the allocation of a new window consisting of a set of registers from the register file for use by the new procedure.

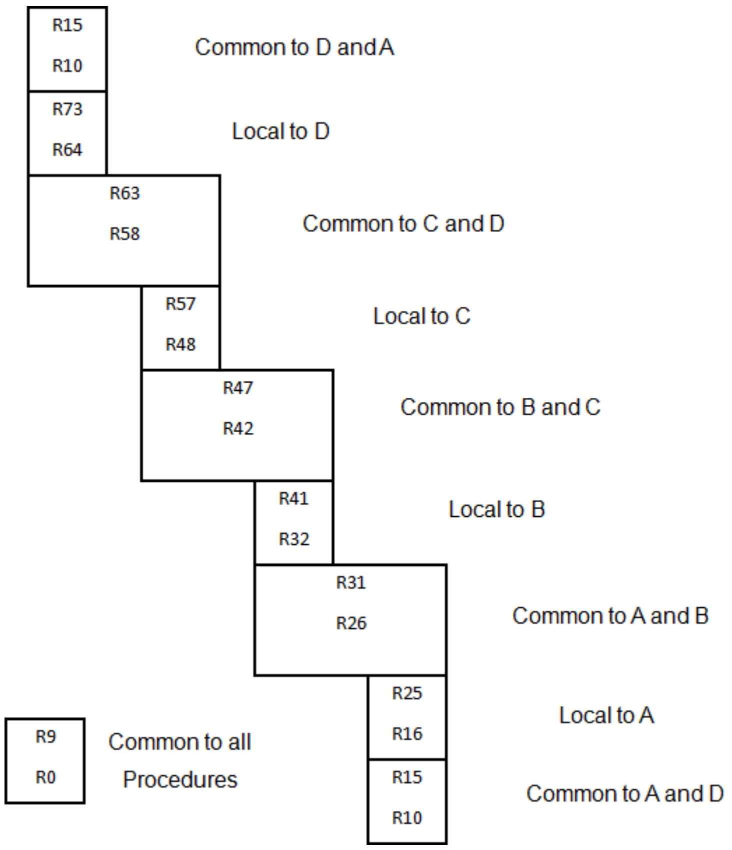


Figure 5.4: Overlapped Register Windows

Each procedure call activates a new register window by incrementing a pointer, while the return statement decrements the pointer and causes the activation of the previous window. Windows for adjacent procedures have overlapping registers that are shared to provide the passing of parameters and results.

The concept of overlade register windows is shown in figure. The system had a total of 74 registers. Register R0 through R9 are global registers that hold parameters shared by all procedures

The other 64 registers are divided into four windows to accommodate procedure A, B, C and D. Each register window consists of 10 local registers and two sets of six registers common to adjacent windows.

Only one register window is activated at any given time with a pointer indicating the active window.

The high register of the calling procedure overlap the low registers of the called procedure, and therefore the parameters automatically transfer from calling to called procedure.
The organization of register windows will have the following relationships:
Number of global registers = G
Number of local registers in each window = L
Number of register common to two windows = C
Number of windows = W
The number of registers available for each window is calculated as follows:

Window size = L + 2C + G

The total number of register needed in the processor is

Register file = (L + C) W + G

14. Explain Reverse Polish Notation (RPN) with appropriate example.

The postfix RPN notation, referred to as Reverse Polish Notation (RPN), places the operator after the operands.
The following examples demonstrate the three representations:

A + B	Infix notation
+ A B	Prefix or Polish notation
A B +	Postfix or reverse Polish notation

The reverse Polish notation is in a form suitable for stack manipulation.
The expression

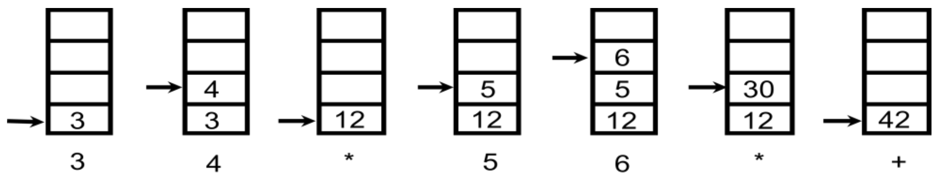
A * B + C * D is written in reverse Polish notation as
A B * C D * +

The conversion from infix notation to reverse Polish notation must take into consideration the operational hierarchy adopted for infix notation.
This hierarchy dictates that we first perform all arithmetic inside inner parentheses, then inside outer parentheses, and do multiplication and division operations before addition and subtraction operations.

Evaluation of Arithmetic Expressions

Any arithmetic expression can be expressed in parenthesis-free Polish notation, including reverse Polish notation

(3 * 4) + (5 * 6) ➳ 3 4 * 5 6 * +



1. Explain Flynn’s classification for computers. (Win’15)

Flynn's classification

- It is 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

		Number of Data Streams	
		Single	**Multiple**
Number of Instruction Streams	**Single**	SISD	SIMD
	Multiple	MISD	MIMD

Figure 6.1: Flynn’s Classification

SISD:

- Single instruction stream, single data stream.
- SISD represents the organization of a single computer containing a control unit, a processor unit, and a memory unit.
- Instructions are executed sequentially and the system may or may not have internal parallel processing capabilities.

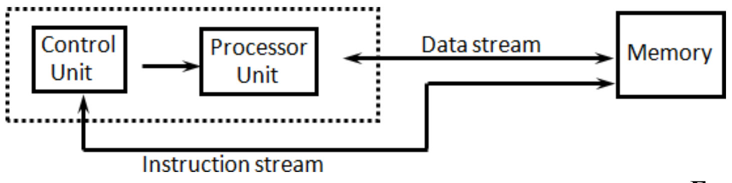


Figure 6.2: SISD Organ

SIMD:

i z a t **i o n**

- SIMD represents an organization that includes many processing units under the supervision of a common control unit.
- All processors receive the same instruction from the control unit but operate on different items of data.

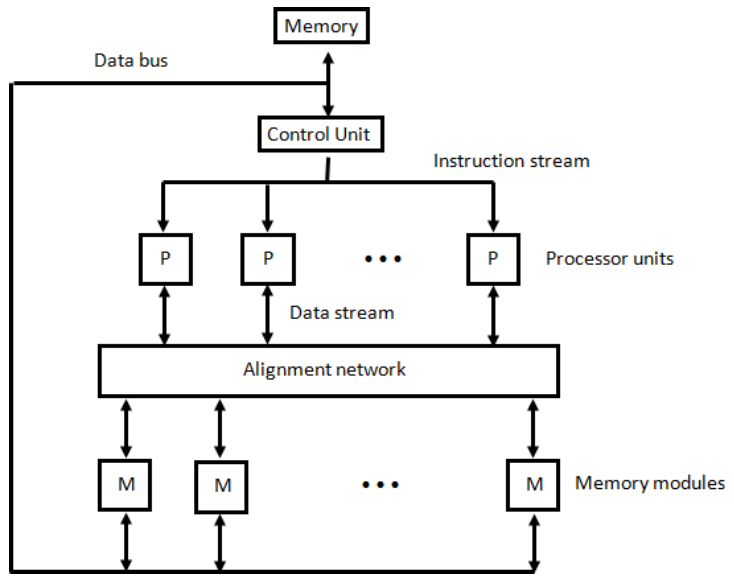


Figure 6.3: SIMD Organization

MISD:

- There is no computer at present that can be classified as MISD.
- MISD structure is only of theoretical interest since no practical system has been constructed using this organization.

MIMD:

- MIMD organization refers to a computer system capable of processing several programs at the same time.
- Most multiprocessor and multicomputer systems can be classified in this category.
- Contains multiple processing units.
- Execution of multiple instructions on multiple data.

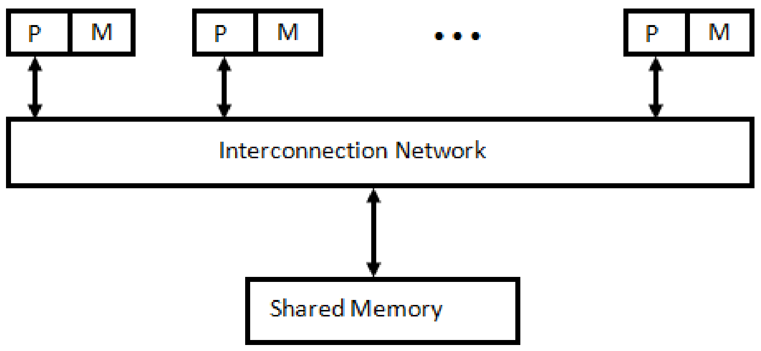


Figure 6.4: MIMD Organization

2. **Explain pipelining technique. Draw the general structure of four segment pipeline.** (Sum'15)

- Pipeline 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.
- A pipeline can be visualized as a collection of processing segments through which binary information flows.
- 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.
- It is characteristic of pipelines that several computations can be in progress in distinct segments at the same time.
- The overlapping of computation is made possible by associating a register with each segment in the pipeline.
- The registers provide isolation between each segment so that each can operate on distinct data simultaneously.
- Any operation that can be decomposed into a sequence of sub operations 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 times with different sets of data.
- The general structure of a four-segment pipeline is illustrated in Figure 6.5.

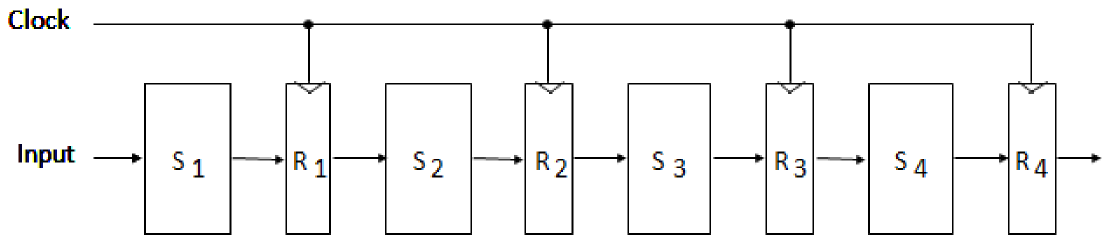


Figure 6.5: General Structure of Four-Segment Pipeline

- The operands pass through all four segments in a fixed sequence.
- Each segment consists of a combinational circuit S, which performs a sub operation over the data stream flowing through the pipe.
- The segments are separated by registers R, which hold the intermediate results between the stages.
- Information flows between adjacent stages under the control of a common clock applied to all the registers simultaneously.
- We define a task as the total operation performed going through all the segments in the pipeline.

3. Draw and explain Arithmetic Pipeline.

- The inputs to the floating-point adder pipeline are two normalized floating-point binary numbers.

  X = A x
  2^a Y = B
  x 2^b

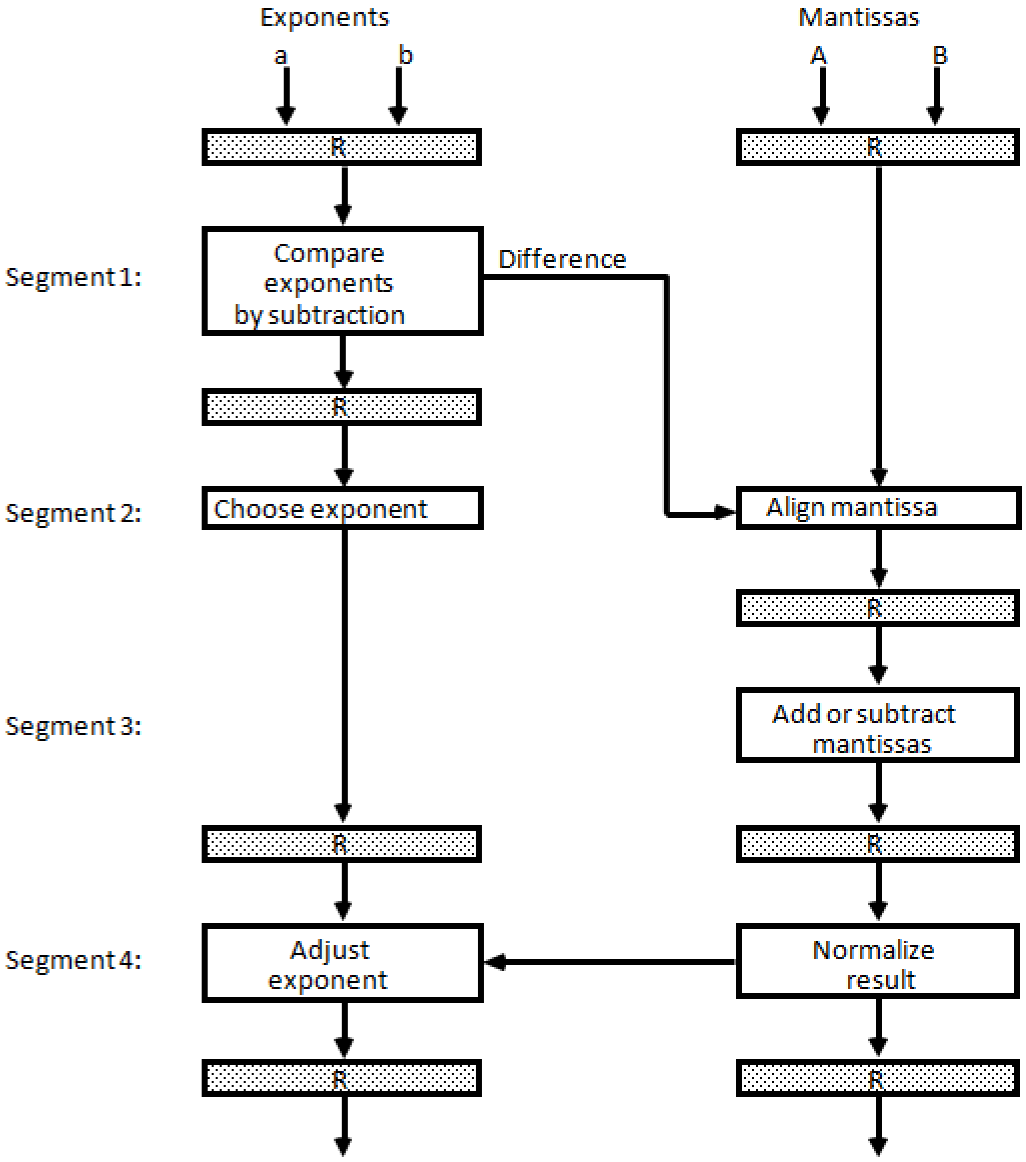


Figure 6.6: Pipeline for floating-point addition and subtraction

- A and B are two fractions that represent the mantissas and a and b are the exponents.
- The floating-point addition and subtraction can be performed in four segments, as shown in Figure 6.6.
- The registers labeled R are placed between the segments to store intermediate results.

- The sub-operations that are performed in the four segments are:
  1. Compare the exponents
  2. Align the mantissas
  3. Add or subtract the mantissas
  4. Normalize the result
- The following numerical example may clarify the sub-operations performed in each segment.
- For simplicity, we use decimal numbers, although Figure 6.6 refers to binary numbers.
- Consider the two normalized floating-point numbers: $X = 0.9504 \times 10^3$
  $Y = 0.8200 \times 10^2$
- The two exponents are subtracted in the first segment to obtain 3 - 2 = 1.
- The larger exponent 3 is chosen as the exponent of the result.
- The next segment shifts the mantissa of Y to the right to obtain $X = 0.9504 \times 10^3$
  $Y = 0.0820 \times 10^3$
- This aligns the two mantissas under the same exponent. The addition of the two mantissas in segment 3 produces the sum
  $Z = 1.0324 \times 10^3$

4. Explain the Instruction Pipelining with example. OR Discuss four-segment instruction pipeline with diagram(s).

(Sum'14, Win'15)

Pipeline processing can occur in data stream as well as in instruction stream.
An instruction pipeline reads consecutive instructions from memory while previous instructions are being executed in other segments.
This causes the instruction fetch and executes phases to overlap and perform simultaneous operations.
One possible digression associated with such a scheme is that an instruction may cause a branch out of sequence.
In that case the pipeline must be emptied and all the instructions that have been read from memory after the branch instruction must be discarded.
Consider a computer with an instruction fetch unit and an instruction execution unit designed to provide a two-segment pipeline.
The instruction fetch segment can be implemented by means of a first-in, first-out (FIFO) buffer.
The buffer acts as a queue from which control then extracts the instructions for the execution unit.

Instruction cycle:

The fetch and execute to process an instruction completely.

In the most general case, the computer needs to process each instruction with the following sequence of steps

1. Fetch the instruction from memory.
2. Decode the instruction.
3. Calculate the effective address.
4. Fetch the operands from memory.
5. Execute the instruction.
6. Store the result in the proper place.

There are certain difficulties that will prevent the instruction pipeline from operating at its maximum rate.

Different segments may take different times to operate on the incoming information.

Some segments are skipped for certain operations.

The design of an instruction pipeline will be most efficient if the instruction cycle is divided into segments of equal duration.

The time that each step takes to fulfill its function depends on the instruction and the way it is executed.

Example: Four-Segment Instruction Pipeline

Assume that the decoding of the instruction can be combined with the calculation of the effective address into one segment.

Assume further that most of the instructions place the result into a processor registers so that the instruction execution and storing of the result can be combined into one segment.

This reduces the instruction pipeline into four segments.

1. FI: Fetch an instruction from memory
2. DA: Decode the instruction and calculate the effective address of the operand
3. FO: Fetch the operand
4. EX: Execute the operation

Figure 6.7 shows, how the instruction cycle in the CPU can be processed with a four-segment pipeline.

While an instruction is being executed in segment 4, the next instruction in sequence is busy fetching an operand from memory in segment 3.

The effective address may be calculated in a separate arithmetic circuit for the third instruction, and whenever the memory is available, the fourth and all subsequent instructions can be fetched and placed in an instruction FIFO.

Thus up to four sub operations in the instruction cycle can overlap and up to four different instructions can be in progress of being processed at the same time.

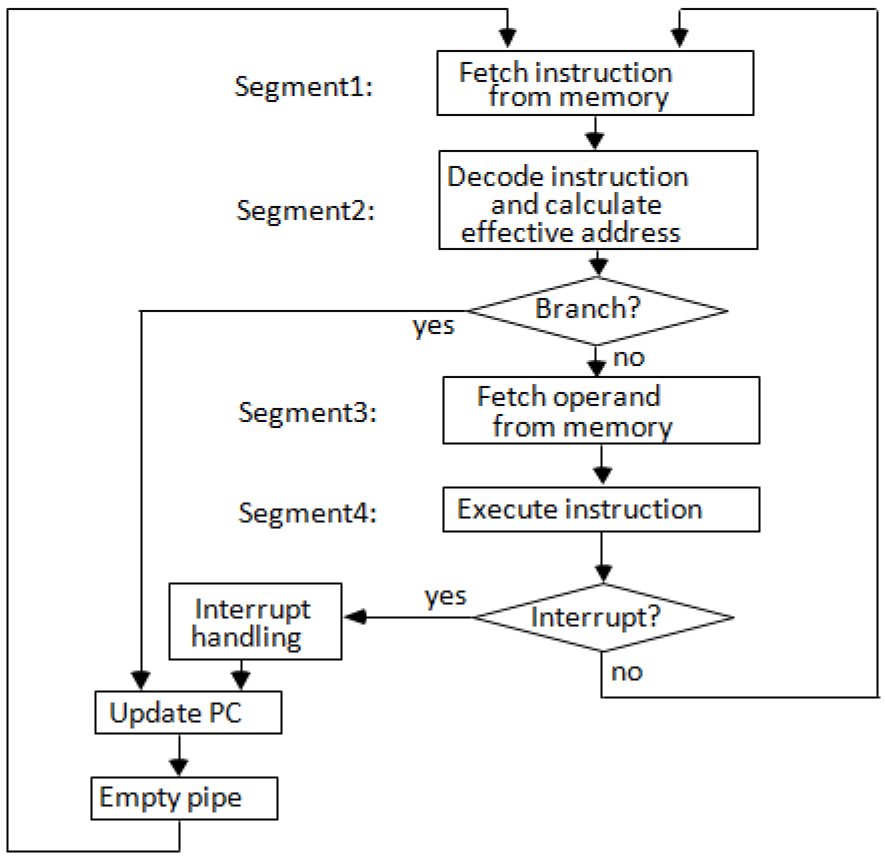
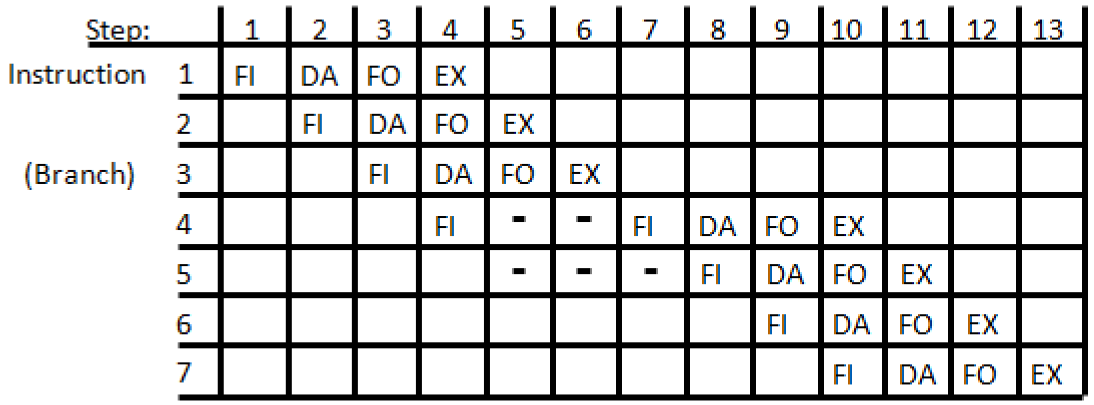


Figure 6.7: Four-segment CPU pipeline

Figure 6.8 shows the operation of the instruction pipeline. The time in the horizontal axis is divided into steps of equal duration. The four segments are represented in the diagram with an abbreviated symbol.



Step:		1	2	3	4	5	6	7	8	9	10	11	12	13
Instruction	1	FI	DA	FO	EX									
	2		FI	DA	FO	EX								
(Branch)	3			FI	DA	FO	EX							
	4				FI	-	-	FI	DA	FO	EX			
	5					-	-	-	FI	DA	FO	EX		
	6									FI	DA	FO	EX	
	7										FI	DA	FO	EX

Figure 6.8: Timing of Instruction Pipeline

It is assumed that the processor has separate instruction and data memories so that the operation in FI and FO can proceed at the same time.

Thus, in step 4, instruction 1 is being executed in segment EX; the operand for instruction 2 is being fetched in segment FO; instruction 3 is being decoded in segment DA; and instruction 4 is being fetched from memory in segment FI.
Assume now that instruction 3 is a branch instruction.
As soon as this instruction is decoded in segment DA in step 4, the transfer from FI to DA of the other instructions is halted until the branch instruction is executed in step 6.
If the branch is taken, a new instruction is fetched in step 7. If the branch is not taken, the instruction fetched previously in step 4 can be used.
The pipeline then continues until a new branch instruction is encountered.
Another delay may occur in the pipeline if the EX segment needs to store the result of the operation in the data memory while the FO segment needs to fetch an operand.
In that case, segment FO must wait until segment EX has finished its operation.

5. What is pipeline conflict? Explain data dependency and handling of branch instruction in detail.

Pipeline conflict:

There are three major difficulties that cause the instruction pipeline conflicts.

1. Resource conflicts caused by access to memory by two segments at the same time.
2. Data dependency conflicts arise when an instruction depends on the result of a previous instruction, but this result is not yet available.
3. Branch difficulties arise from branch and other instructions that change the value of PC.

Data Dependency:

A collision occurs when an instruction cannot proceed because previous instructions did not complete certain operations.
A data dependency occurs when an instruction needs data that are not yet available.
Similarly, an address dependency may occur when an operand address cannot be calculated because the information needed by the addressing mode is not available.
Pipelined computers deal with such conflicts between data dependencies in a variety of ways as follows.

Hardware interlocks:

An interlock is a circuit that detects instructions whose source operands are destinations of instructions farther up in the pipeline.
Detection of this situation causes the instruction whose source is not available to be delayed by enough clock cycles to resolve the conflict.
This approach maintains the program sequence by using hardware to insert the required delays.

Operand forwarding:

It uses special hardware to detect a conflict and then avoid it by routing the data through special paths between pipeline segments.
This method requires additional hardware paths through multiplexers as well as the circuit that detects the conflict.

Delayed load:

Sometimes compiler has the responsibility for solving data conflicts problems.
The compiler for such computers is designed to detect a data conflict and reorder the instructions as necessary to delay the loading of the conflicting data by inserting no-operation instruction, this method is called delayed load.

Handling of Branch Instructions:

One of the major problems in operating an instruction pipeline is the occurrence of branch instructions.
A branch instruction can be conditional or unconditional.
The branch instruction breaks the normal sequence of the instruction stream, causing difficulties in the operation of the instruction pipeline.
Various hardware techniques are available to minimize the performance degradation caused by instruction branching.

Pre-fetch target:

One way of handling a conditional branch is to prefetch the target instruction in addition to the instruction following the branch.
If the branch condition is successful, the pipeline continues from the branch target instruction.
An extension of this procedure is to continue fetching instructions from both places until the branch decision is made.

Branch target buffer:

Another possibility is the use of a branch target buffer or BTB.
The BTB is an associative memory included in the fetch segment of the pipeline.
Each entry in the BTB consists of the address of a previously executed branch instruction and the target instruction for that branch.
It also stores the next few instructions after the branch target instruction.
The advantage of this scheme is that branch instructions that have occurred previously are readily available in the pipeline without interruption.

Loop buffer:

A variation of the BTB is the loop buffer. This is a small very high speed register file maintained by the instruction fetch segment of the pipeline.
When a program loop is detected in the program, it is stored in the loop buffer in its entirety, including all branches.

Branch Prediction:

A pipeline with branch prediction uses some additional logic to guess the outcome of a conditional branch instruction before it is executed.
The pipeline then begins pre-fetching the instruction stream from the predicted path.
A correct prediction eliminates the wasted time caused by branch penalties.

Delayed branch:

A procedure employed in most RISC processors is the delayed branch.
In this procedure, the compiler detects the branch instructions and rearranges the machine language code sequence by inserting useful instructions that keep the pipeline operating without interruptions.

6. Explain (i) Vector Processing (ii) Vector Operations. Explain how matrix multiplication is carried out on a computer supporting Vector Computations. (Win'14)

Vector Processing

There is a class of computational problems that are beyond the capabilities of a conventional computer.
These problems are characterized by the fact that they require a vast number of computations that will take a conventional computer days or even weeks to complete.
In many science and engineering applications, the problems can be formulated in terms of vectors and matrices that lend themselves to vector processing.

Applications of Vector processing

1. Long-range weather forecasting
2. Petroleum explorations
3. Seismic data analysis
4. Medical diagnosis
5. Aerodynamics and space flight simulations
6. Artificial intelligence and expert systems
7. Mapping the human genome
8. Image processing

Vector Operations

Many scientific problems require arithmetic operations on large arrays of numbers.
These numbers are usually formulated as vectors and matrices of floating-point numbers.
A vector is an ordered set of a one-dimensional array of data items.
A vector V of length n is represented as a row vector by V = [V1 V2 V3 … Vn] that may be represented as a column vector if the data items are listed in a column.
A conventional sequential computer is capable of processing operands one at a time.
Consequently, operations on vectors must be broken down into single computations with subscripted variables.
The element of vector V is written as V(I) and the index I refers to a memory address or register where the number is stored.

Matrix Multiplication

Matrix multiplication is one of the most computational intensive operations performed in computers with vector processors.

An n x m matrix of numbers has n rows and m columns and may be considered as constituting a set of n row vectors or a set of m column vectors.

Consider, for example, the multiplication of two 3x3 matrices A and B.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} X \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{bmatrix}$$

The product matrix C is a 3 x 3 matrix whose elements are related to the elements of A and B by the inner product:

$$c_{ij} = \sum_{k=1}^{3} c_{ik} \times b_{kj}$$

For example, the number in the first row and first column of matrix C is calculated by letting I= 1, j = 1, to obtain

$$C_{11} = a_{11}\, b_{11} + a_{12}\, b_{21} + a_{13}\, b_{31}$$

This requires three multiplications and three additions.

The total number of multiplications or additions required to compute the matrix product is 9 x 3 = 27.

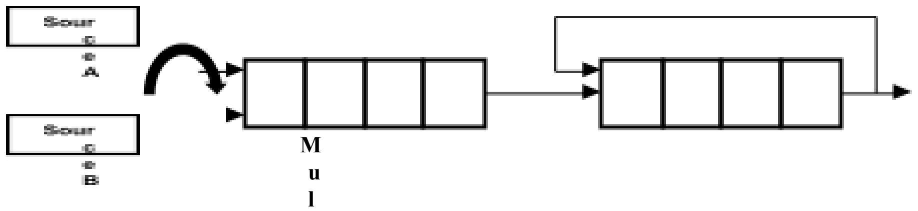
If we consider the linked multiply-add operation c + a x b as a cumulative operation, the product of two n x n matrices requires n^3 multiply-add operations.

The computation consists of n^2 inner products, with each inner product requiring n multiply-add operations, assuming that c is initialized to zero before computing each element in the product matrix.

In general, the inner product consists of the sum of k product terms of the form

$$C = A_1B_1 + A_2B_2 + A_3B_3 + A_4B_4 + \quad + A_kB_k$$

In a typical application k may be equal to 100 or even 1000. The inner product calculation on a pipeline vector processor is shown in following figure.



Multiplierpipeline

Figure 6.9: Pipeline for Inner Product

Adder pipeline

The values of A and B are either in memory or in processor registers.
The floating point multiplier pipeline and the floating point adder pipeline are assumed to have four segments each.
All segment registers in the multiplier and adder are initialized to 0.
Therefore, the output of the adder is 0 for the first eight cycles until both pipes are full.

$$\begin{aligned} C = {} & A_1B_1 + A_5B_5 + A_9B_3A_9 + A_{13}B_{13} + \cdots \\ & + A_2B_2 + A_6B_6 + A_{10}B_{10} + A_{14}B_{14} + \cdots \\ & + A_3B_3 + A_7B_7 + A_{11}B_{11} + A_{15}B_{15} + \cdots \end{aligned}$$

$+ A_4B_4 + A_8B_8 + A_{12}B_{12} + A_{16}B_{16} + \cdots$

When there are no more product terms to be added, the system inserts four zeros into the multiplier pipeline.

The adder pipeline will then have one partial product in each of its four segments, corresponding to the four sums listed in the four rows in the above equation.

The four partial sums are then added to form the final sum.

7. Explain Memory Interleaving.

Instead of using two memory buses for simultaneous access, the memory can be partitioned into a number of modules connected to a common memory address and data buses.

A memory module is a memory array together with its own address and data registers.

Figure 6.10 shows a memory unit with four modules.

Each memory array has its own address register AR and data register DR.

The address registers receive information from a common address bus and the data registers communicate with a bidirectional data bus.

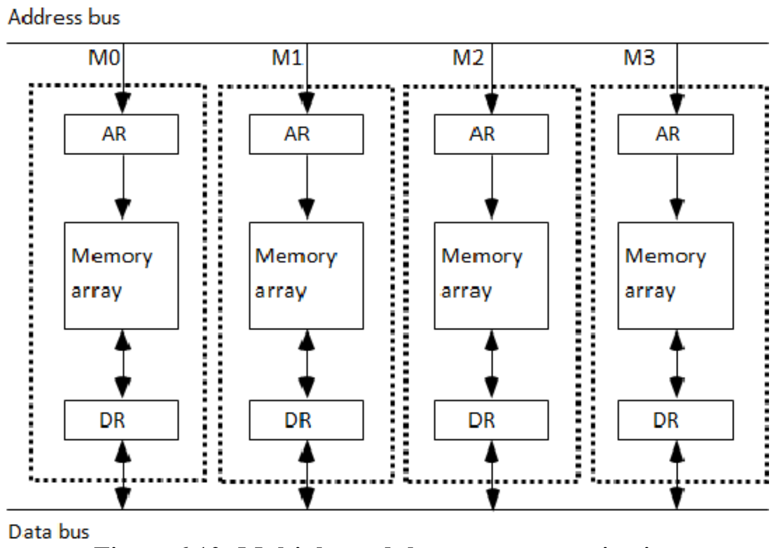


Figure 6.10: Multiple module memory organization

The modular system permits one module to initiate a memory access while other modules are in the process of reading or writing a word and each module can honor a memory request independent of the state of the other modules.

The advantage of a modular memory is that it allows the use of a technique called interleaving.

In an interleaved memory, different sets of addresses are assigned to different memory modules.

8. What is an array processor? Explain the different types of array processor.

An array processor is a processor that performs computations on large arrays of data.
The term is used to refer to two different types of processors, attached array processor and SIMD array processor.
An attached array processor is an auxiliary processor attached to a general-purpose computer.
It is intended to improve the performance of the host computer in specific numerical computation tasks.
An SIMD array processor is a processor that has a single-instruction multiple-data organization.
It manipulates vector instructions by means of multiple functional units responding to a common instruction.

Attached Array Processor

An attached array processor is designed as a peripheral for a conventional host computer, and its purpose is to enhance the performance of the computer by providing vector processing for complex scientific applications.
It achieves high performance by means of parallel processing with multiple functional units.
It includes an arithmetic unit containing one or more pipelined floating-point adders and multipliers.
The array processor can be programmed by the user to accommodate a variety of complex arithmetic problems.
Figure 6.11 shows the interconnection of an attached array processor to host computer.

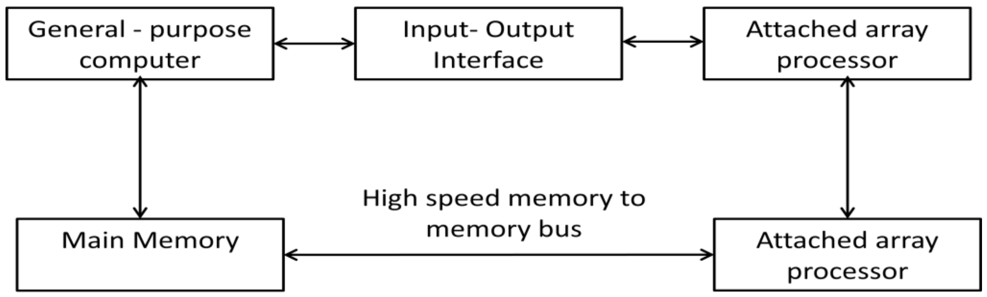


Figure 6.11: Attached array processor with host computer.

The host computer is a general-purpose commercial computer and the attached processor is a back-end machine driven by the host computer.
The array processor is connected through an input-output controller to the computer and the computer treats it like an external interface.
The data for the attached processor are transferred from main memory to a local memory through a high-speed bus.

The general-purpose computer without the attached processor serves the users that need conventional data processing.
The system with the attached processor satisfies the needs for complex arithmetic applications.

SIMD Array Processor

An SIMD array processor is a computer with multiple processing units operating in parallel.
The processing units are synchronized to perform the same operation under the control of a common control unit, thus providing a single instruction stream, multiple data stream (SIMD) organization.
A general block diagram of an array processor is shown in Figure

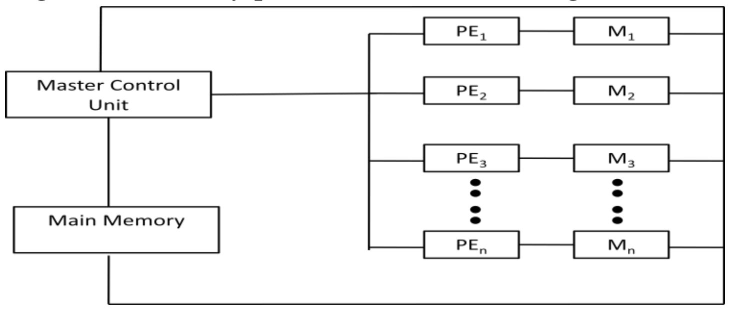


Figure 6.12: SIMD array processor organization

It contains a set of identical processing elements (PEs), each having a local memory M.
Each processor element includes an ALU, a floating-point arithmetic unit and working registers.
The master control unit controls the operations in the processor elements.
The main memory is used for storage of the program.
The function of the master control unit is to decode the instructions and determine how the instruction is to be executed.
Scalar and program control instructions are directly executed within the master control unit.
Vector instructions are broadcast to all PEs simultaneously.
Vector operands are distributed to the local memories prior to the parallel execution of the instruction.
Masking schemes are used to control the status of each PE during the execution of vector instructions.
Each PE has a flag that is set when the PE is active and reset when the PE is inactive.
This ensures that only that PE'S that needs to participate is active during the execution of the instruction.
SIMD processors are highly specialized computers.
They are suited primarily for numerical problems that can be expressed in vector matrix form.
However, they are not very efficient in other types of computations or in dealing with conventional data-processing programs.

1. Explain the procedure for Addition and Subtraction with signed-magnitude data with the help of flowchart.

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

  If the output of the gate is 0 the signs are identical;

  If it is 1, the signs are different.
- For an add operation, identical signs dictate that the magnitudes be added. For a subtract operation, different signs dictate that the magnitudes be added.
- The magnitudes are added with a microoperation $EA \leftarrow A + B$, where EA is a register that combines E and A. The carry in E after the addition constitutes an overflow if it is equal to 1. The value of E is transferred into the add-overflow flip-flop AVF.
- The two magnitudes are subtracted if the signs are different for an add operation or identical for a subtract operation. The magnitudes are subtracted by adding A to the 2's complemented B. No overflow can occur if the numbers are subtracted so AVF is cleared to 0.
- 1 in E indicates that A >= B and the number in A is the correct result. If this numbs is zero, the sign A must be made positive to avoid a negative zero.
- 0 in E indicates that A < B. For this case it is necessary to take the 2's complement of the value in A. The operation can be done with one microoperation $A \leftarrow A' +1$.
- However, we assume that the A register has circuits for microoperations complement and increment, so the 2's complement is obtained from these two microoperations.
- In other paths of the flowchart, the sign of the result is the same as the sign of A. so no change in A is required. However, when A < B, the sign of the result is the complement of the original sign of A. It is then necessary to complement A, to obtain the correct sign.
- The final result is found in register A and its sign in A_s. The value in AVF provides an overflow indication. The final value of E is immaterial.
- Figure 7.2 shows a block diagram of the hardware for implementing the addition and subtraction operations.
- It consists of registers A and B and sign flip-flops A_s and B_s.
- Subtraction is done by adding A to the 2's complement of B.
- The output carry is transferred to flip-flop E , where it can be checked to determine the relative magnitudes of two numbers.
- The add-overflow flip-flop *AVF* holds the overflow bit when A and B are added.
- The A register provides other microoperations that may be needed when we specify the sequence of steps in the algorithm.

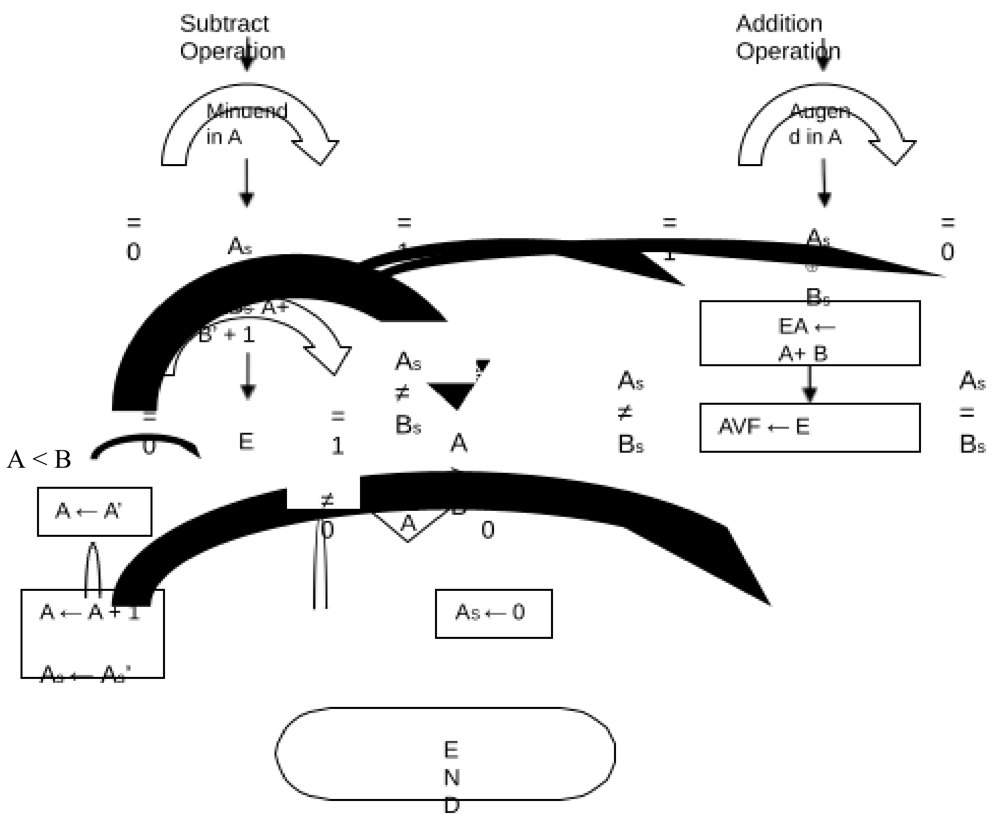


Figure 7.1: Flowchart for add and subtract operations.

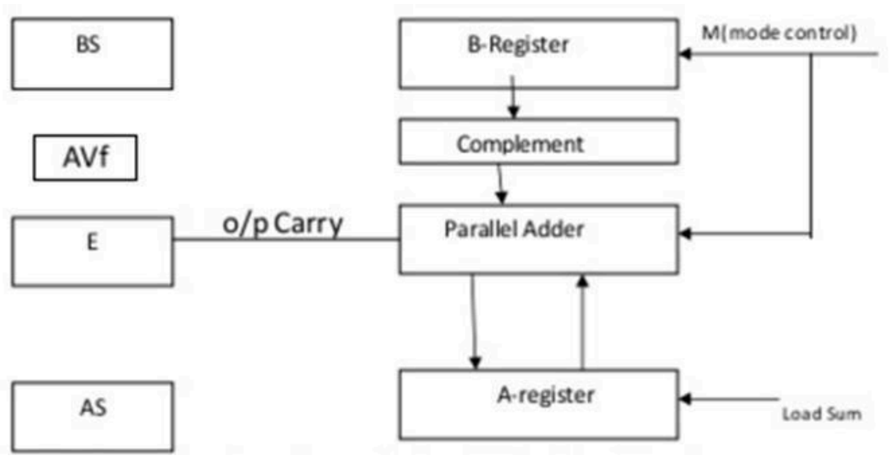


Figure 7.2: Hardware for signed-magnitude addition and subtraction

2. Explain the Booth's algorithm with the help of flowchart.

(Sum '14,Win '14, win'15)

- Booth algorithm gives a procedure for multiplying binary integers in signed- 2's complement representation.
- It operates on the fact that strings of 0's in the multiplier require no addition but just shifting, and a string of 1's in the multiplier from bit weight 2^k to weight 2^m can be treated as $2^{k+1} – 2^m$.
- For example, the binary number 001110 (+14) has a string 1's from 2^3 to 2^1 (k=3, m=1). The number can be represented as $2^{k+1} – 2^m$. = $2^4 – 2^1$ = 16 – 2 = 14. Therefore, the multiplication M X 14, where M is the multiplicand and 14 the multiplier, can be done as M X 2^4 – M X 2^1.
- Thus the product can be obtained by shifting the binary multiplicand M four times to the left and subtracting M shifted left once.

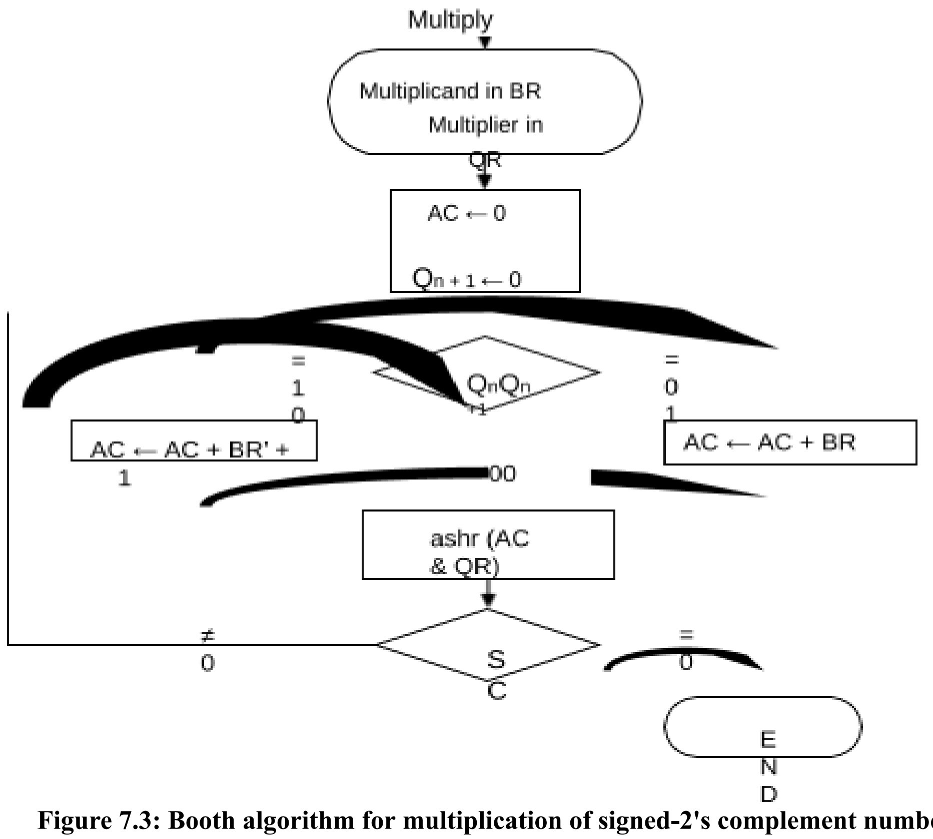


Figure 7.3: Booth algorithm for multiplication of signed-2's complement numbers

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

- Prior to the shifting, the multiplicand may be added to the partial product, subtracted from the partial, or left unchanged according to the following rules:
  1. The multiplicand is subtracted from the partial product upon encountering the first least significant 1 in a string of 1's in the multiplier.
  2. The multiplicand is added to the partial product upon encountering the first 0 in a string of 0's in the multiplier.
  3. The partial product does not change when multiplier bit is identical to the previous multiplier bit.
- The algorithm works for positive or negative multipliers in 2's complement representation.
- This is because a negative multiplier ends with a string of 1's and the last operation will be a subtraction of the appropriate weight.
- The two bits of the multiplier in Q_n and Q_{n+1} are inspected.
- If the two bits are equal to 10, it means that the first 1 in a string of 1 's has been encountered. This requires a subtraction of the multiplicand from the partial product in AC.
- If the two bits are equal to 01, it means that the first 0 in a string of 0's has been encountered. This requires the addition of the multiplicand to the partial product in AC.
- When the two bits are equal, the partial product does not change.

3. Explain with proper block diagram the Multiplication Operation on two floating point numbers.

- The multiplication of two floating-point numbers requires that we multiply the mantissas and add the exponents.
- No comparison of exponents or alignment of mantissas is necessary.
- The multiplication of the mantissas is performed in the same way as in fixed-point to provide a double-precision product.
- The double-precision answer is used in fixed-point numbers to increase the accuracy of the product.
- In floating-point, the range of a single-precision mantissa combined with the exponent is usually accurate enough so that only single-precision numbers are maintained.
- Thus the half most significant bits of the mantissa product and the exponent will be taken together to form a single-precision floating-point product.
- The multiplication algorithm can be subdivided into four parts:
  1. Check for zeros.
  2. Add the exponents.
  3. Multiply the mantissas.
  4. Normalize the product.
- The flowchart for floating-point multiplication is shown in Figure 7.4. The two operands are checked to determine if they contain a zero.
- If either operand is equal to zero, the product in the AC is set to zero and the operation is terminated.

- If neither of the operands is equal to zero, the process continues with the exponent addition.
- The exponent of the multiplier is in q and the adder is between exponents a and b.
- It is necessary to transfer the exponents from q to a, add the two exponents, and transfer the sum into a.

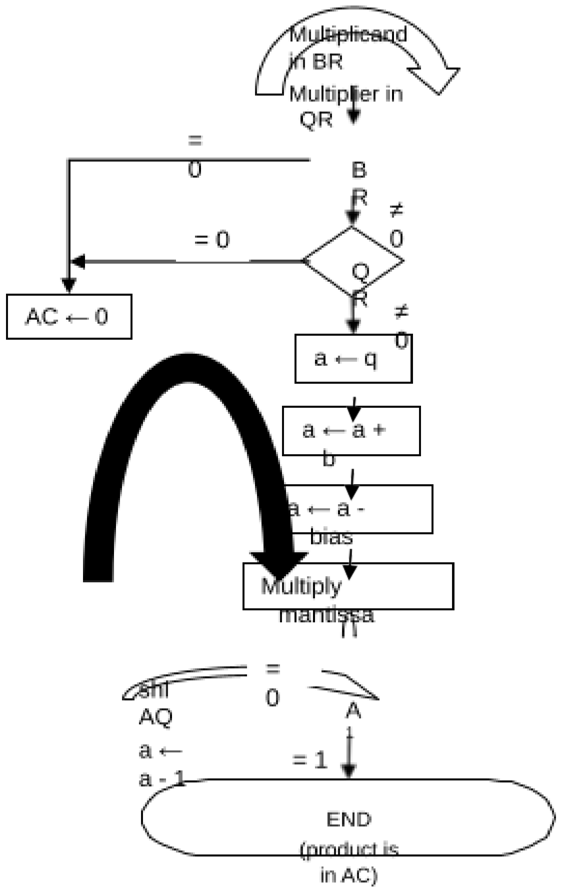


Figure 7.4: Multiplication of floating-point numbers

- Since both exponents are biased by the addition of a constant, the exponent sum will have double this bias.
- The correct biased exponent for the product is obtained by subtracting the bias number from the sum.
- The multiplication of the mantissas is done as in the fixed-point case with the product residing in A and Q.
- Overflow cannot occur during multiplication, so there is no need to check for it.
- The product may have an underflow, so the most significant bit in A is checked. If it is a 1, the product is already normalized.

- If it is a 0, the mantissa in AQ is shifted left and the exponent decremented.
- Note that only one normalization shift is necessary. The multiplier and multiplicand were originally normalized and contained fractions. The smallest normalized operand is 0.1, so the smallest possible product is 0.01.
- Therefore, only one leading zero may occur.
- Although the low-order half of the mantissa is in Q, we do not use it for the floating-point product. Only the value in the AC is taken as the product.

4. Multiply the (-9) with (-13) using Booth's algorithm. Give each step.

(Sum'14)

9:	01001	13:	01101
1's complement of 9:	10110	1's complement of 13:	10010

- A numerical example of booth algorithm is shown for n=5. It shows the step-by-step multiplication of (-9) X (-13) = +117.

	+ 1		+ 1
2's complement of 9:	10111 (-9)	2's complement of 13:	10011 (-13)

AC	QR(-13)	Q_{n+1}	M(BR)(-9)	SC	Comments
00000	1001$\underline{1}$	$\underline{0}$	10111	5	Initial value
01001	10011	0	10111	4	Subtraction: AC=AC+BR'+1
00100	1100$\underline{1}$	$\underline{1}$	10111		Arithmetic Shift Right
00010	0110$\underline{0}$	$\underline{1}$	10111	3	Arithmetic Shift Right
11001	01100	1	10111	2	Subtraction: AC=AC+BR'+1
11100	1011$\underline{0}$	$\underline{0}$	10111		Arithmetic Shift Right
11110	0101$\underline{1}$	$\underline{0}$	10111	1	Arithmetic Shift Right
00111	01011	0	10111	0	Subtraction: AC=AC+BR'+1
00011	**10101**	1	10111		Arithmetic Shift Right

Answer: -9 X -13 =117 => 001110101

5. Multiply the (7) with (3) using Booth's algorithm. Give each step.

7:	0111	3:	0011

AC	QR(3)	Q_{n+1}	M(BR)(7)	SC	Comments
0000	001$\underline{1}$	$\underline{0}$	0111	4	Initial value
1001	0011	0	0111	3	Subtraction: AC=AC+BR'+1
1100	100$\underline{1}$	$\underline{1}$	0111		Arithmetic Shift Right
1110	010$\underline{0}$	$\underline{1}$	0111	2	Arithmetic Shift Right

0101	0100	1	0111	1	Addition: AC=AC+BR
0010	1010	0	0111		Arithmetic Shift Right
0001	**0101**	0	0111	0	Arithmetic Shift Right

Answer: 7 X 3 =21 => 00010101

6. Multiply the (15) with (13) using Booth's algorithm. Give each step.

15: 01111 | 13: 01101

15X13=195

AC	QR(15)	Q_{n+1}	M(BR)(13)	SC	Comments
00000	0111<u>1</u>	<u>0</u>	01101	5	Initial value
10011	01111	0	01101	4	Subtraction: AC=AC+BR'+1
11001	1011<u>1</u>	<u>1</u>	01101		Arithmetic Shift Right
11100	1101<u>1</u>	<u>1</u>	01101	3	Arithmetic Shift Right
11110	0110<u>1</u>	<u>1</u>	01101	2	Arithmetic Shift Right
11111	0011<u>0</u>	<u>1</u>	01101	1	Arithmetic Shift Right
01100	00110	1	01101	0	Addition: AC=AC+BR
00110	**00011**	1	01101		Arithmetic Shift Right

Answer: 15X13=195 => 0011000011

7. Multiply the (+15) with (-13) using Booth's algorithm. Give each step.

15: 01111 | 13: 01101

1's complement of 13: 10010

$+\ \ 1$

2's complement of 13: 10011 (-13)

AC	QR(-13)	Q_{n+1}	M(BR)(+15)	SC	Comments
00000	1001<u>1</u>	<u>0</u>	01111	5	Initial value
10001	10011	0	01111	4	Subtraction: AC=AC+BR'+1
11000	1100<u>1</u>	<u>1</u>	01111		Arithmetic Shift Right
11100	0110<u>0</u>	<u>1</u>	01111	3	Arithmetic Shift Right
01011	01100	1	01111	2	Addition: AC=AC+BR
00101	1011<u>0</u>	<u>0</u>	01111		Arithmetic Shift Right
00010	1101<u>1</u>	<u>0</u>	01111	1	Arithmetic Shift Right
10011	11011	0	01111	0	Subtraction: AC=AC+BR'+1
11001	**11101**	1	01111		Arithmetic Shift Right

Answer: (+15) X (-13) = -195 => **1100111101**

```
To verify                 0011000010
                        +          1
                        ____________
                    +195=> 0011000011
```

8. Explain BCD adder with its block diagram. (Win'14)

- BCD representation is a class of binary encodings of decimal numbers where each decimal digit is represented by a fixed number of bits.
- BCD adder is a circuit that adds two BCD digits in parallel and produces a sum digit in BCD form.

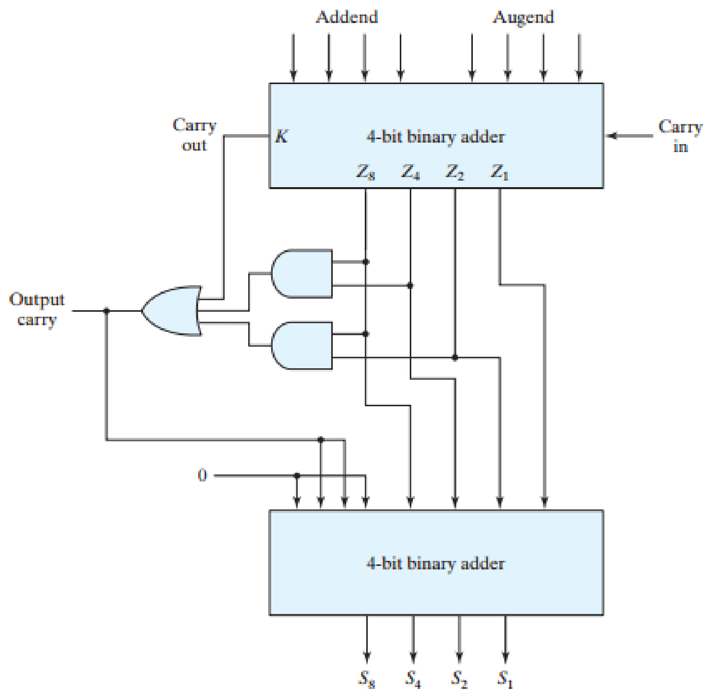


Figure 7.5: BCD Adder

- Since each input digit does not exceed 9, the output sum cannot be greater than 19(9+9+1). For example: suppose we apply two BCD digits to 4-bit binary adder.
- The adder will form the sum in binary and produce a result that may range from 0 to 19.
- In following figure 7.5, these binary numbers are represented by K, Z_8, Z_4, Z_2, and Z_1.
- K is the carry and subscripts under the Z represent the weights 8, 4, 2, and 1 that can be assigned to the four bits in the BCD code.
- When binary sum is equal to or less than or equal to 9, corresponding BCD number is identical and therefore no conversion is needed.
- The condition for correction and output carry can be expressed by the Boolean function:

$$C = K + Z_8Z_4 + Z_8 Z_2$$

- When it is greater than 9, we obtain non valid BCD representation, then additional binary 6 to binary sum converts it to correct BCD representation.
- The two decimal digits, together with the input-carry, are first added in the top 4-bit binary adder to produce the binary sum. When the output-carry is equal to 0, nothing is added to the binary sum.
- When C is equal to 1, binary 0110 is added to the binary sum using bottom 4-bit binary adder. The output carry generated from the bottom binary-adder may be ignored.

1. Define Peripherals. Explain I/O Bus and Interface Modules.

Peripherals:

Input-output device attached to the computer are also called peripherals.

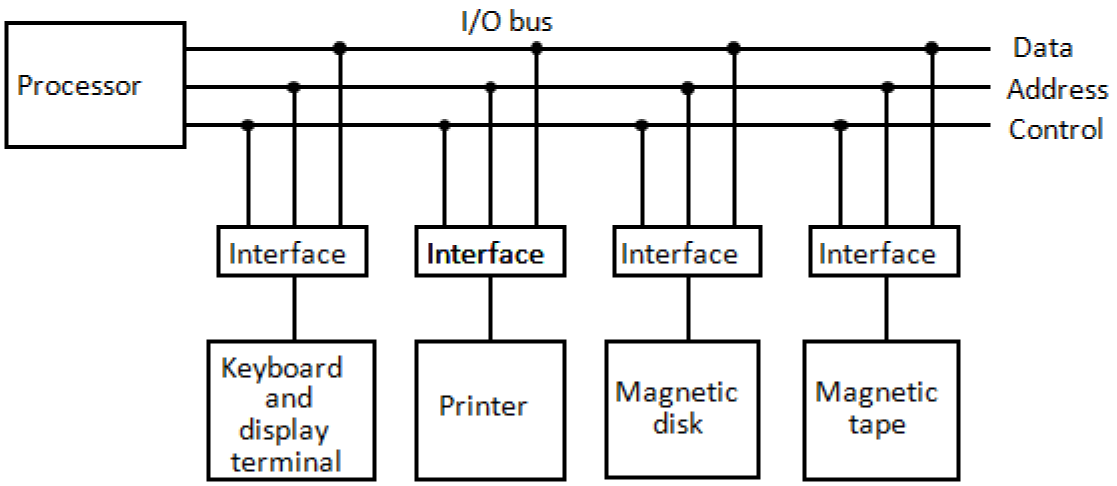


Figure 8.1: Connection of I/O bus to input-output device.

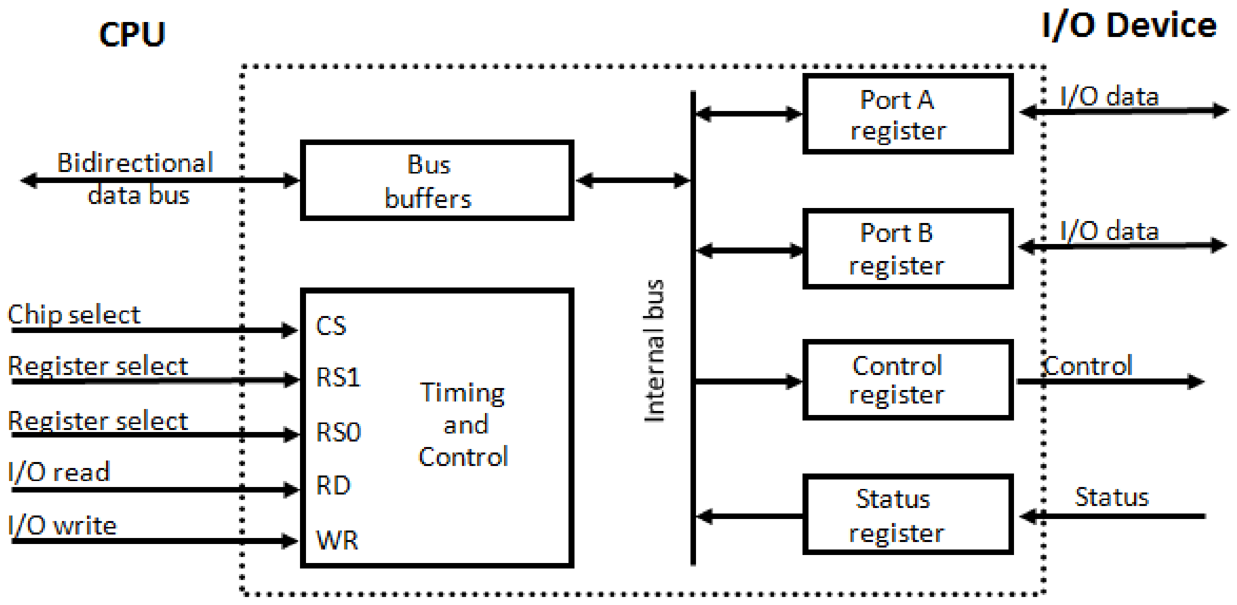
A typical communication link between the processor and several peripherals is shown in figure 8.1.
The I/O bus consists of data lines, address lines, and control lines.
The magnetic disk, printer, and terminal are employed in practically any general purpose computer.
Each peripheral device has associated with it an interface unit.
Each interface decodes the address and control received from the I/O bus, interprets them for the peripheral, and provides signals for the peripheral controller.
It also synchronizes the data flow and supervises the transfer between peripheral and processor.
Each peripheral has its own controller that operates the particular electromechanical device.
For example, the printer controller controls the paper motion, the print timing, and the selection of printing characters.
The I/O bus from the processor is attached to all peripheral interfaces.
To communicate with a particular device, the processor places a device address on the address lines.
Each interface attached to the I/O bus contains an address decoder that monitors the address lines.
When the interface detects its own address, it activates the path between the bus lines and the device that it controls.
All peripherals whose address does not correspond to the address in the bus are disabled by their interface selected responds to the function code and proceeds to execute it.

The function code is referred to as an I/O command.
There are four types of commands that an interface may receive. They are classified as control, status, data output, and data input.
A control command is issued to activate the peripheral and to inform it what to do.
For example, a magnetic tape unit may be instructed to backspace the tape by one record, to rewind the tape, or to start the tape moving in the forward direction.
A status command is used to test various status conditions in the interface and the peripheral.
For example, the computer may wish to check the status of the peripheral before a transfer is initiated.
During the transfer, one or more errors may occur which are detected by the interface.
These errors are designated by setting bits in a status register that the processor can read at certain intervals.
A data output command causes the interface to respond by transferring data from the bus into one of its registers.
The computer starts the tape moving by issuing a control command.
The processor then monitors the status of the tape by means of a status command.
When the tape is in the correct position, the processor issues a data output command.
The interface responds to the address and command and transfers the information from the data lines in the bus to its buffer register.
The interface that communicates with the tape controller and sends the data to be stored on tape.
The data input command is the opposite of the data output.
In this case the interface receives an item of data from the peripheral and places it in its buffer register.
The processor checks if data are available by means of a status command and then issues a data input command.
The interface places the data on the data lines, where they are accepted by the processor.

2. Explain I/O interface with example.

An example of an I/O interface units is shown in block diagram from in figure 8.2.
It consists of two data registers called ports, a control register, a status register, bus buffers, and timing and control circuit.
The interface communicates with the CPU through the data bus.
The chip select and register select inputs determine the address assigned to the interface.
The I/O read and writes are two control lines that specify an input or output, respectively.
The four registers communicate directly with the I/O device attached to the interface.
The I/O data to and from the device can be transferred into either port A or port B.

If the interface is connected to a printer, it will only output data, and if it services a character reader, it will only input data.
A magnetic disk unit transfers data in both directions but not at the same time, so the interface can use bidirectional lines.
A command is passed to the I/O device by sending a word to the appropriate interface register.
The control register receives control information from the CPU. By loading appropriate bits into the control register, the interface and the I/O device attached to it can be placed in a variety of operating modes.



CS	RS1	RS0	Register Selected
0	X	X	None: data bus in high impedance
1	0	0	Port A register
1	0	1	Port B register
1	1	0	Control register
1	1	1	Status register

Figure 8.2: Example of I/O interface unit

For example, port A may be defined as an input port and port B as an output port.
A magnetic tape unit may be instructed to rewind the tape or to start the tape moving in the forward direction.
The bits in the status register are used for status conditions and for recording errors that may occur during the data transfer.
For example, a status bit may indicate that port A has received a new data item from the I/O device.
Another bit in the status register may indicate that a parity error has occurred during the transfer.

The interface registers communicate with the CPU through the bidirectional data bus.
The address bus selects the interface unit through the chip select and the two register select inputs.
A circuit must be provided externally (usually, a decoder) to detect the address assigned to the interface registers.
This circuit enables the chip select (CS) input when the interface is selected by the address bus.
The two register select inputs RS1 and RS0 are usually connected to the two least significant lines of the address bus.
These two inputs select one of the four registers in the interface as specified in the table accompanying the diagram.
The content of the selected register is transfer into the CPU via the data bus when the I/O read signal is enables.
The CPU transfers binary information into the selected register via the data bus when the I/O write input is enabled.

3. What do you mean by Asynchronous data transfer? Explain Strobe control in detail.

Asynchronous data transfer

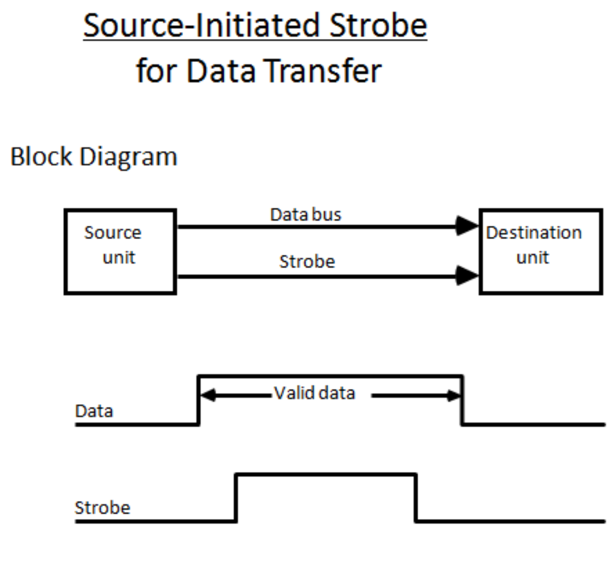
Data transfer between two independent units, where internal timing in each unit is independent from the other. Such two units are said to be asynchronous to each other.

Strobe Control

The Strobe control method of asynchronous data transfer employs a single control line to time each transfer.

Source-initiated strobe for data transfer

The strobe may be activated by either the source or the destination unit. Figure 8.3 shows a source-initiated transfer.
The data bus carries the binary information from source unit to the destination unit.
The strobe is a single line that informs the destination unit when a valid data word is available in the bus.
The source unit first places the data on the data bus.
After a delay to ensure that the data settle to a steady value, the source activates the strobe pulse.
The information on the data bus and the strobe signal remain in the active state for a sufficient time period to allow the destination unit to receive the data.
The source removes the data from the bus a brief period after it disables its strobe pulse.



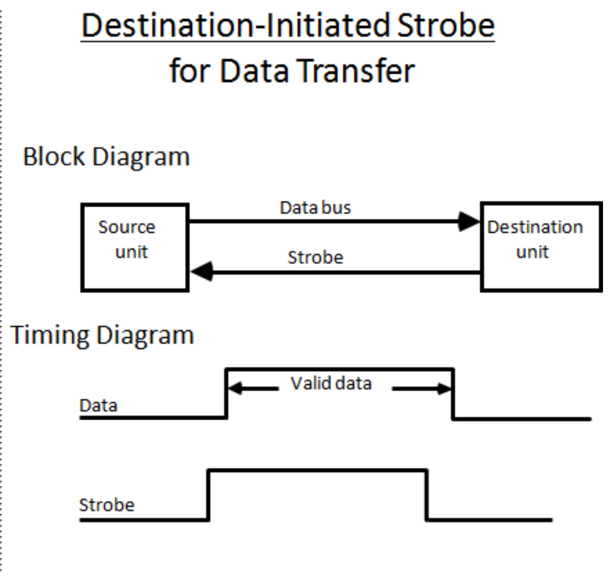


Figure 8.3: Source-initiated strobe for data transfer

Figure 8.4: Destination-initiated strobe for data transfer

Destination-initiated strobe for data transfer

Figure 8.4 shows a data transfer initiated by the destination unit. In this case the destination unit 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 the strobe pulse can be used again to trigger a destination register.
The destination unit then disables the strobe. The source removes the data from the bus after a predetermined time interval.
The transfer of data between the CPU and an interface unit is similar to the strobe transfer just described.

Disadvantage of Strobe method:

The disadvantage of the strobe method is that the source unit that initiates the transfer has no way of knowing whether the destination unit has actually received the data item that was placed in the bus
Similarly, a destination unit that initiates the transfer has no way of knowing whether the source unit has actually placed the data on the bus.

4. Explain Asynchronous data transfer with Handshaking method.

The handshake method solves the problem of Strobe method by introducing a second control signal that provides a reply to the unit that initiates the transfer.

Source-initiated transfer using handshaking

One control line is in the same direction as the data flow in the bus from the source to the destination.
It is used by the source unit to inform the destination unit whether there are valid data in the bus.

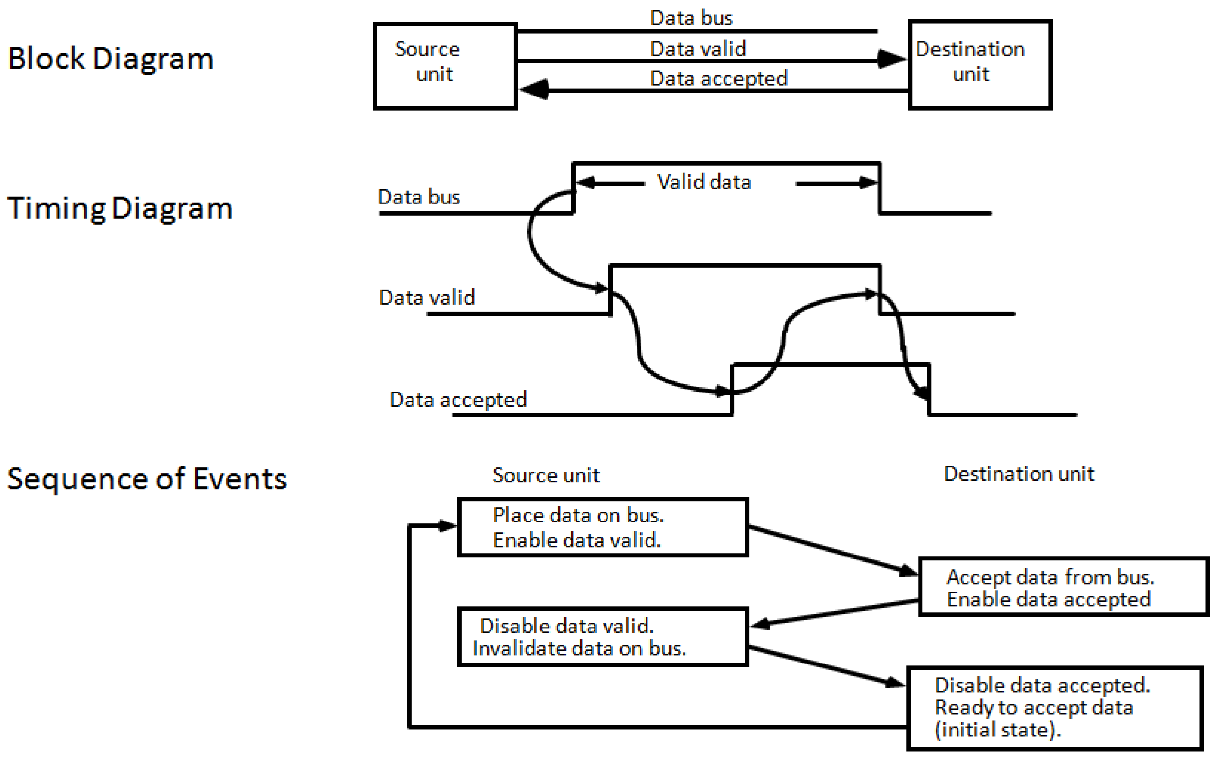

Block Diagram
Source unit
Data bus
Data valid
Data accepted
Destination unit
Timing Diagram
Data bus
Valid data
Data valid
Data accepted
Sequence of Events
Source unit
Destination unit
Place data on bus. Enable data valid.
Accept data from bus. Enable data accepted
Disable data valid. Invalidate data on bus.
Disable data accepted. Ready to accept data (initial state).

Figure 8.5: Source-initiated transfer using handshaking

The other control line is in the other direction from the destination to the source.
It is used by the destination unit to inform the source whether it can accept data.
The sequence of control during the transfer depends on the unit that initiates the transfer.
Figure 8.5 shows the data transfer procedure initiated by the source.
The two handshaking lines the data valid, which is generated by the source unit, and data accepted, generated by the destination unit, the timing diagram shows the exchange of signals between the two units.
The sequence of events listed in figure 8.5 shows the four possible states that the system can be at any given time.
The source unit initiates the transfer by placing the data on the bus and enabling its data valid signal.

The data accepted signal is activated by the destination unit after it accepts the data from the bus.
The source unit then disables its data valid signal, which invalidates the data on the bus.
The destination unit then disables its data accepted signal and the system goes into its initial state.
The source does not send the next data item until after the destination unit shows its readiness to accept new data by disabling its data accepted signal.
This scheme allows arbitrary delays from one state to the next and permits each unit to respond at its own data transfer rate.

Destination-initiated transfer using handshaking

The destination-initiated transfer using handshaking lines is shown in figure 8.6.
Note that the name of the signal generated by the destination unit has been changed to ready for data to reflect its new meaning.
The source unit in this case does not place data on the bus until after it receives the ready for data signal from the destination unit.
From there on, the handshaking procedure follows the same pattern as in the source-initiated case.
Note that the sequence of events in both cases would be identical if we consider the ready for data signal as the complement of data accepted.
In fact, the only difference between the source-initiated and the destination-initiated transfer is in their choice of initial state.

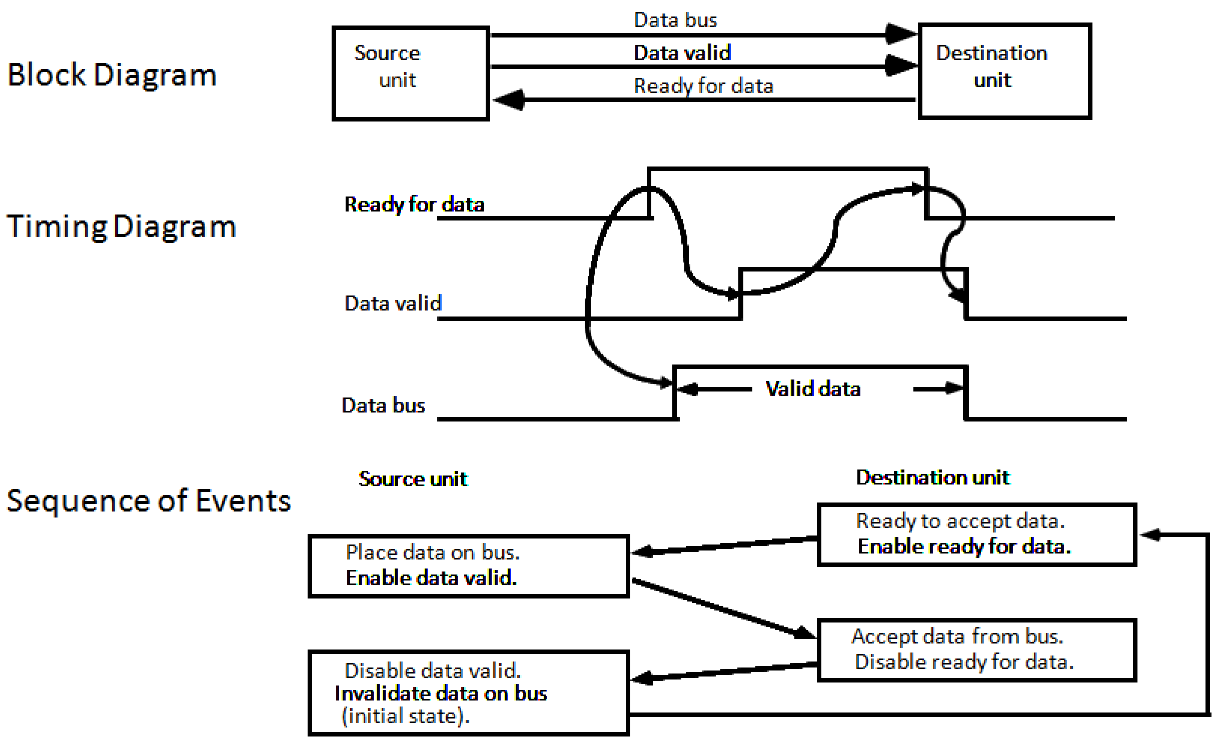

Block Diagram
Source unit
Data bus
Data valid
Ready for data
Destination unit
Timing Diagram
Ready for data
Data valid
Data bus
Valid data
Sequence of Events
Source unit
Destination unit
Ready to accept data. Enable ready for data.
Place data on bus. Enable data valid.
Accept data from bus. Disable ready for data.
Disable data valid. Invalidate data on bus (initial state).

Figure 8.6: Destination-initiated transfer using handshaking

5. Explain Programmed I/O with example.

Programmed I/O:

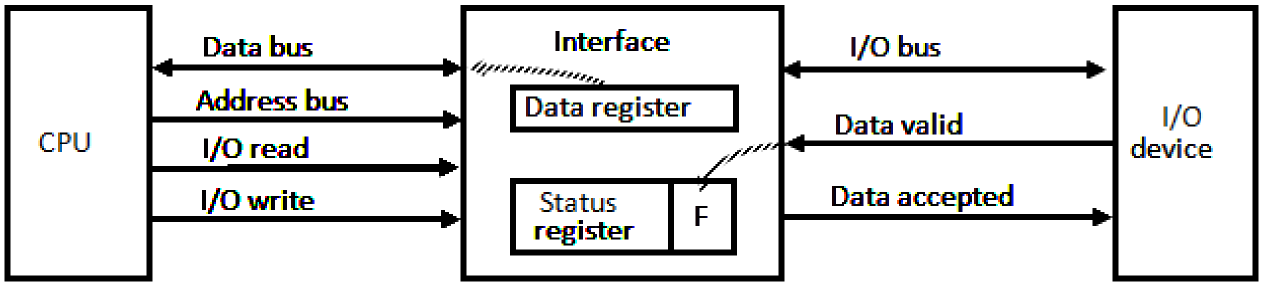


Figure 8.7: Data transfer from I/O device to CPU

In the programmed I/O method, the I/O device does not have direct access to memory.
An example of data transfer from an I/O device through an interface into the CPU is shown in figure 8.7.
When a byte of data is available, the device places it in the I/O bus and enables its data valid line.
The interface accepts the byte into its data register and enables the data accepted line.
The interface sets a bit in the status register that we will refer to as an F or "flag" bit.
The device can now disables the data valid line, but it will not transfer another byte until the data accepted line is disables by the interface.
A program is written for the computer to check the flag in the status register to determine if a byte has been placed in the data register by the I/O device.
This is done by reading the status register into a CPU register and checking the value of the flag bit.
Once the flag is cleared, the interface disables the data accepted line and the device can then transfer the next data byte.

Example of Programmed I/O:

A flowchart of the program that must be written for the CPU is shown in figure 8.8.
It is assumed that the device is sending a sequence of bytes that must be stored in memory.
The transfer of each byte requires three instructions :
1. Read the status register.
2. Check the status of the flag bit and branch to step 1 if not set or to step 3 if set.
3. Read the data register.

Each byte is read into a CPU register and then transferred to memory with a store instruction.
A common I/O programming task is to transfer a block of words from an I/O device and store them in a memory buffer.

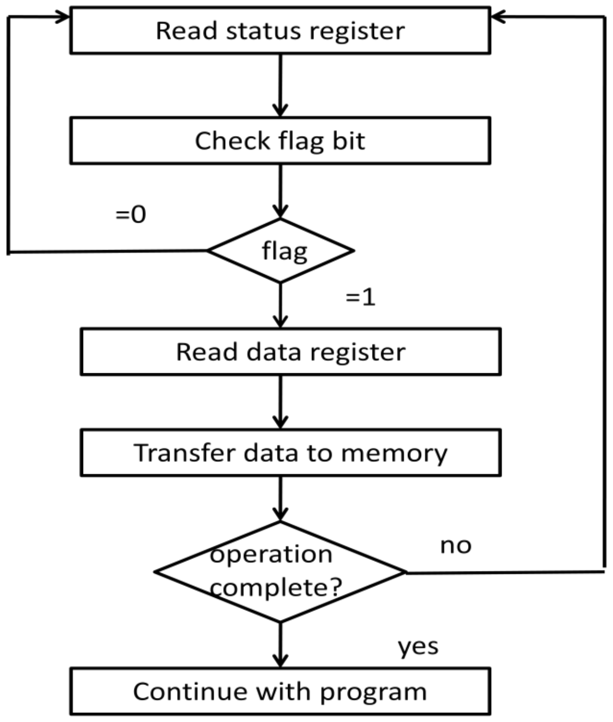


Figure 8.8: Flowchart for CPU program to input data

6. Write a note on Interrupt Initiated I/O

In programmed initiated, CPU stays in a program loop until the I/O unit indicates that it is ready for data transfer.
This is a time consuming process since it keeps the processor busy needlessly.
It can be avoided by using an interrupt facility and a special command to inform the interface to issue an interrupt request signal when data are available from the device.
In the meantime CPU can proceed to execute another program.
The interface meanwhile keeps monitoring the device.
When the interface determines that the device is ready for data transfer, it generates an interrupt request to the computer.
While the CPU is running a program, it does not check the flag. However, when the flag is set, the computer is momentarily interrupted from proceeding with the current program and is informed of the fact that the flag has been set.
The CPU deviates from what it is doing to take care of the input or output transfer.
After the transfer is completed, the computer returns to the previous program to continue what it was doing before the interrupt.
The CPU responds to the interrupt signal by storing the return address from the program counter into a memory stack and then control branches to a service routine that processes the required I/O transfer.

The way that the processor chooses the branch address of the service routine varies from one unit to another.
In **non-vectored interrupt**, branch address is assigned to a fixed location in memory.
In a **vectored interrupt**, the source that interrupts supplies the branch information to the computer. The information is called vector interrupt.
In some computers the interrupt vector is the first address of the I/O service routine.
In other computers the interrupt vector is an address that points to a location in memory where the beginning address of the I/O service routine is stored.

7. What is priority interrupt? Explain Daisy Chaining.

Determines which interrupt is to be served first when two or more requests are made simultaneously
Also determines which interrupts are permitted to interrupt the computer while another is being serviced
Higher priority interrupts can make requests while servicing a lower priority interrupt.

Daisy Chaining Priority

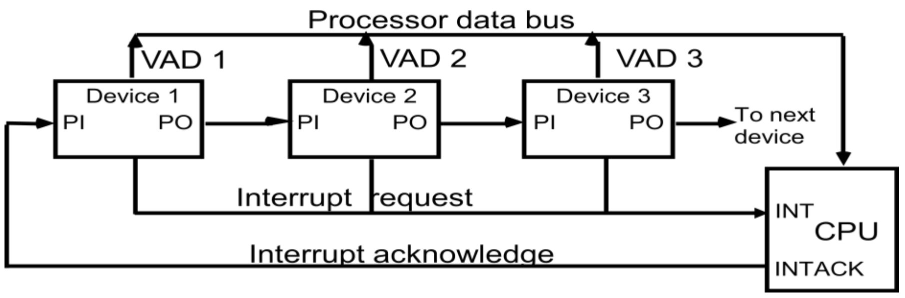


Figure 8.9: Daisy-chain priority interrupt

The daisy-chaining method of establishing priority consists of a serial connection of all devices that request an interrupt.
The device with the highest priority is placed in the first position, followed by lower-priority devices up to the device with the lowest priority, which is placed last in the chain.
This method of connection between three devices and the CPU is shown in figure 8.9.
If any device has its interrupt signal in the low-level state, the interrupt line goes to the low-level state and enables the interrupt input in the CPU.
When no interrupts are pending, the interrupt line stays in the high-level state and no interrupts are recognized by the CPU.
The CPU responds to an interrupt request by enabling the interrupt acknowledge line.
This signal passes on to the next device through the PO (priority out) output only if device 1 is not requesting an interrupt.
If device 1 has a pending interrupt, it blocks the acknowledge signal from the next device by placing a 0 in the PO output.

It then proceeds to insert its own interrupt vector address (VAD) into the data bus for the CPU to use during the interrupt cycle.
A device with a 0 in its Pl input generates a 0 in its PO output to inform the next-lower-priority device that the acknowledge signal has been blocked.
A device that is requesting an interrupt and has a 1 in its Pl input will intercept the acknowledge signal by placing a 0 in its PO output.
If the device does not have pending interrupts, it transmits the acknowledge signal to the next device by placing a 1 in its PO output.
Thus the device with Pl = 1 and PO = 0 is the one with the highest priority that is requesting an interrupt, and this device places its VAD on the data bus.
The daisy chain arrangement gives the highest priority to the device that receives the interrupt acknowledge signal from the CPU.
The farther the device is from the first position; the lower is its priority.

8. Write a detailed note on Direct Memory Access (DMA).

Direct Memory Access

Transfer of data under programmed I/O is between CPU and peripheral.
In direct memory access (DMA), Interface transfers data into and out of memory through the memory bus.
The CPU initiates the transfer by supplying the interface with the starting address and the number of words needed to be transferred and then proceeds to execute other tasks.
When the transfer is made, the DMA requests memory cycles through the memory bus.
When the request is granted by the memory controller, DMA transfers the data directly into memory.

DMA controller

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).

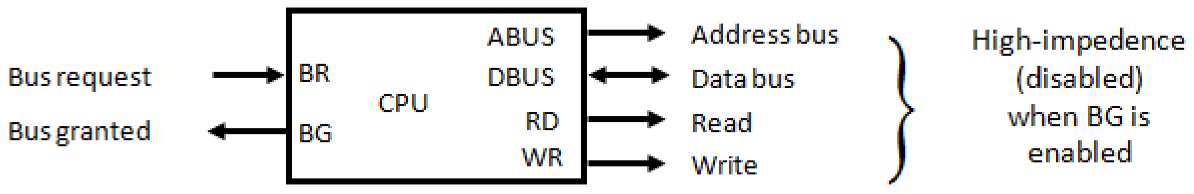


Figure 8.10: CPU bus signals for DMA transfer

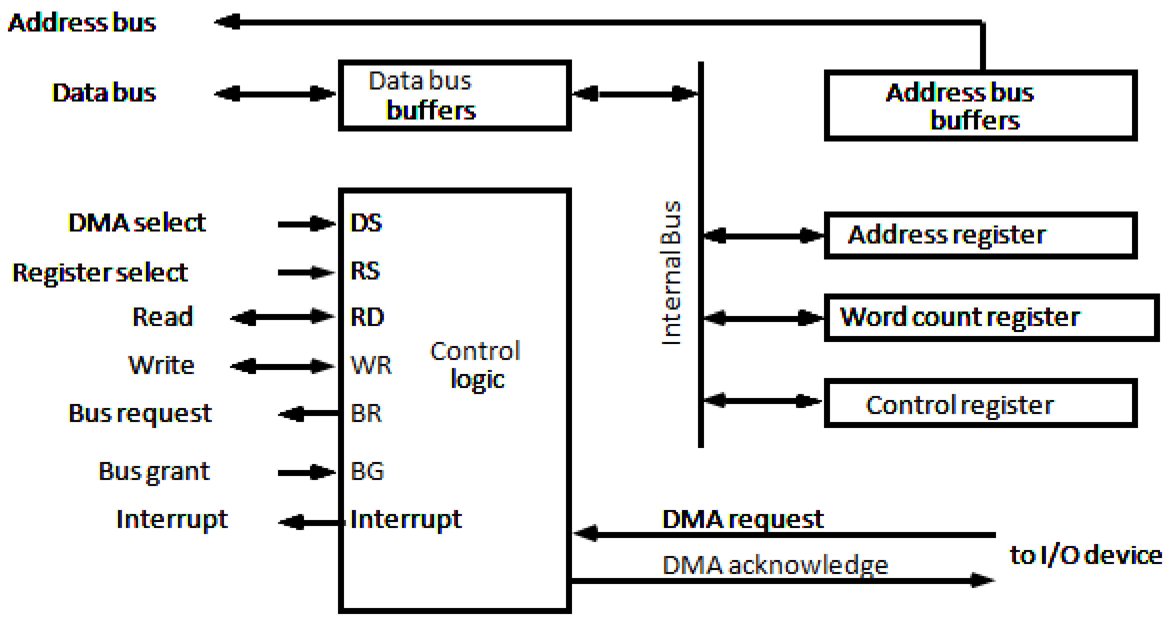


Figure 8.11: Block diagram of DMA controller

The DMA controller needs the usual circuits of an interface to communicate with the CPU and I/O device.
In addition, it needs an address register, a word count register, and a set of address lines.
The address register and address lines are used for direct communication with the memory.
The word count register specifies the number of words that must be transferred.
The data transfer may be done directly between the device and memory under control of the DMA.
Figure 8.11 shows the block diagram of a typical DMA controller.
The unit communicates with the CPU via the data bus and control lines.
The register in the DMA are selected by the CPU through the address bus by enabling the DS (DMA select) and RS (register select) inputs.
The RD (read) and WR (write) inputs are bidirectional.
When the BG (bus grant) input is 0, the CPU can communicate with the DMA registers through the data bus to read from or write to the DMA registers.
When BG= 1, the CPU has relinquished the buses and the DMA can communicate directly with the memory by specifying an address in the address but and activating the RD or WR control.
The DMA communicates with the external peripheral through the request and acknowledge lines by using a prescribed handshaking procedure.
The DMA controller has three registers: an address register, a word count register, and a control register.
The address register contains an address to specify the desired location in memory.
The word count register holds the number of words to be transferred.

This register is decremented by one after each word transfer and internally tested for zero.
The control register specifies the mode of transfer.
All registers in the DMA appear to the CPU as I/O interface registers.
Thus the CPU can read from or write into the DMA register under program control via the data bus.
The DMA is first initialized by the CPU.
After that, the DMA starts and continues to transfer data between memory and peripheral unit until an entire block is transferred.
The CPU initializes the DMA by sending the following information through the data bus

1. The staring address of the memory block where data are available (for read) or where data are to be stored (for write)
2. The word count, which is the number of words in the memory block.
3. Control to specify the mode of transfer such as read or write.
4. The starting address is stored in the address register.

9. Explain Input- Output Processor (IOP)

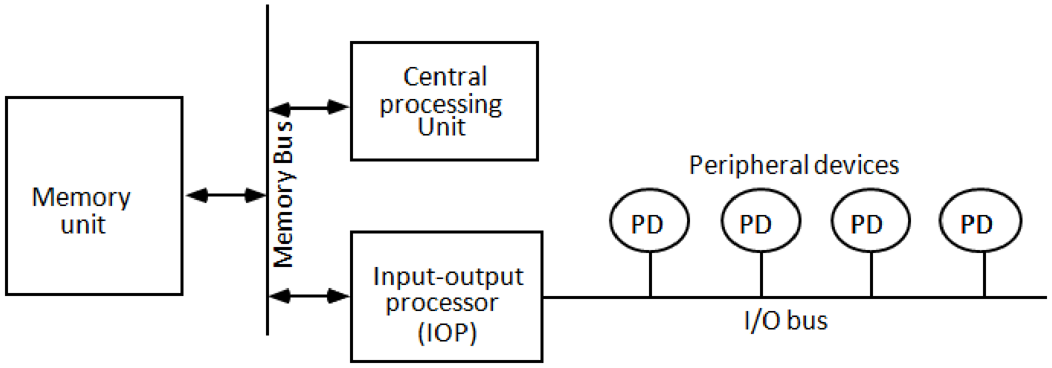


Figure 8.12: Block diagram of a computer with I/O processor

IOP is similar to a CPU except that it is designed to handle the details of I/O processing.
Unlike the DMA controller that must be setup entirely by the CPU, the IOP can fetch and execute its own instruction.
IOP instructions are specifically designed to facilitate I/O transfers.
In addition, IOP can perform other processing tasks, such as arithmetic, logic branching, and code translation.
The block diagram of a computer with two processors is shown in figure 8.12.
The memory unit occupies central position and can communicate with each processor by means of direct memory access.
The CPU is responsible for processing data needed in the solution of computational tasks.
The IOP provides a path of for transfer of data between various peripheral devices and memory unit.

The CPU is usually assigned the task of initiating the I/O program.
From then, IOP operates independent of the CPU and continues to transfer data from external devices and memory.
The data formats of peripheral devices differ from memory and CPU data formats. The IOP must structure data words from many different sources.
For example, it may be necessary to take four bytes from an input device and pack them into one 32-bit word before the transfer to memory.
Data are gathered in the IOP at the device rate and bit capacity while the CPU is executing its own program.
After the input data are assembled into a memory word, they are transferred from IOP directly into memory by "**stealing**" one memory cycle from the CPU.
Similarly, an output word transferred from memory to the IOP is directed from the IOP to the output word transferred from memory to the IOP.
In most computer systems, the CPU is the master while the IOP is a slave processor.
The CPU is assigned the task of initiating all operations, but I/O instructions are executed in the IOP.
CPU instructions provide operations to start an I/O transfer and also to test I/O status conditions needed for making decisions on various I/O activities.
The IOP, in turn, typically asks for CPU attention by means of an interrupt.
Instructions that are read from memory by an IOP are sometimes called commands, to distinguish them from instructions that are read by the CPU.

10. Explain CPU-IOP Communication.

The communication between CPU and IOP may take different forms, depending on the particular computer considered.
In most cases the memory unit acts.
The sequence of operations may be carried out as shown in the flowchart of figure 8.13.
The CPU sends an instruction to test the IOP path.
The IOP responds by inserting a status word in memory for the CPU to check.
The bits of the status word indicate the condition of the IOP and I/O device, such as IOP overload condition, device busy with another transfer, or device ready for I/O transfer.
The CPU refers to the status word in memory to device what do next.
If all is in order, the CPU sends the instruction to start I/O transfer.
The memory address received with this instruction tells the IOP where to find its program.
The CPU can now continue with another program while the IOP is busy with the I/O program.
Both programs refer to memory by means of DMA transfer.
When the IOP terminates the execution of its program, it sends an interrupt request to the CPU.
The CPU responds to the interrupt by issuing an instruction to read the status from the IOP.

The IOP responds by placing the contents of its status report into a specified memory location.
The status word indicates whether the transfer has been completed or if any errors occurred during the transfer.
From inspection of the bits in the status word, the CPU determines if the I/O operation was completed satisfactorily without errors.
The IOP takes care of all data transfers between several I/O units and the memory while the CPU is processing another program.
The IOP and CPU are competing for the use of memory, so the number of devices that can be in operation is limited by the access time of the memory.

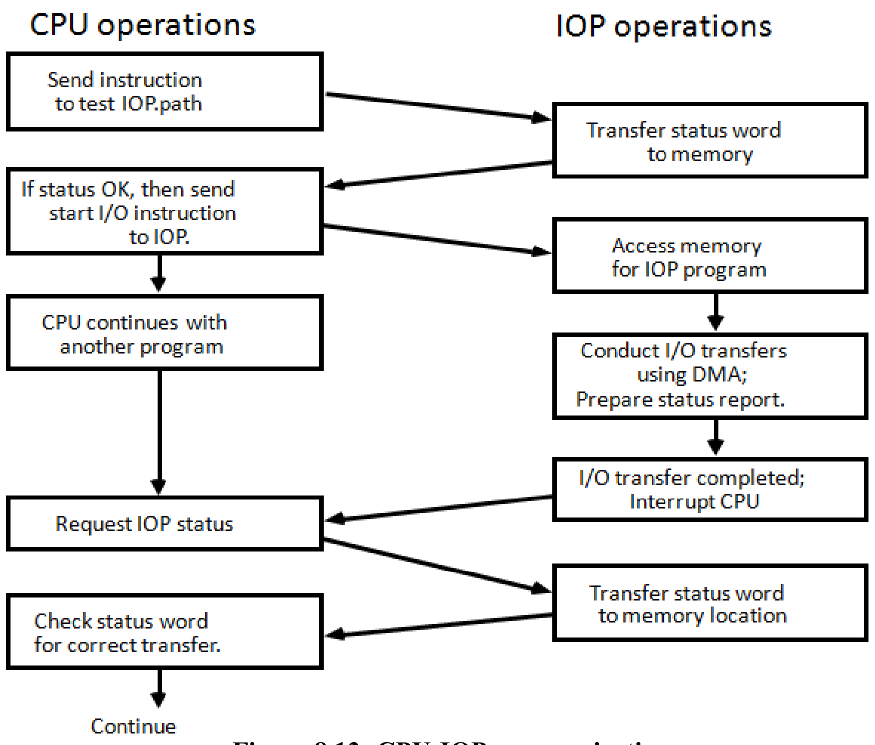

CPU operations
IOP operations
Send instruction to test IOP.path
Transfer status word to memory
If status OK, then send start I/O instruction to IOP.
Access memory for IOP program
CPU continues with another program
Conduct I/O transfers using DMA; Prepare status report.
I/O transfer completed; Interrupt CPU
Request IOP status
Transfer status word to memory location
Check status word for correct transfer.
Continue

Figure 8.13: CPU-IOP communication

1. How main memory is useful in computer system? Explain the memory address map of RAM and ROM. (Sum'15)

Main Memory

The main memory is the central storage unit in a computer system.
Primary memory holds only those data and instructions on which computer is currently working.
It has limited capacity and data is lost when power is switched off.
It is generally made up of semiconductor device.
These memories are not as fast as registers.
The data and instruction required to be processed reside in main memory.
It is divided into two subcategories RAM and ROM.

Memory address map of RAM and ROM

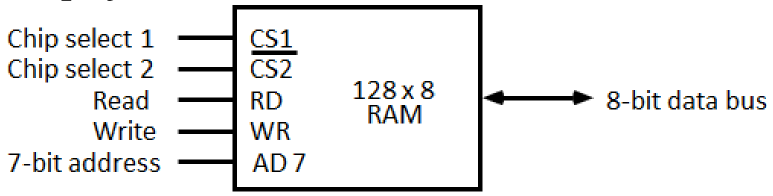


Figure 9.1: Typical RAM chip

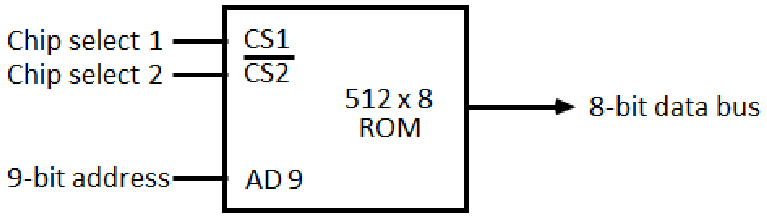


Figure 9.2: Typical ROM chip

The designer of a computer system must calculate the amount of memory required for the particular application and assign it to either RAM or ROM.
The interconnection between memory and processor is then established from knowledge of the size of memory needed and the type of RAM and ROM chips available.
The addressing of memory can be established by means of a table that specifies the memory address assigned to each chip.
The table, called a **memory address map**, is a pictorial representation of assigned address space for each chip in the system, shown in table 9.1.
To demonstrate with a particular example, assume that a computer system needs 512 bytes of RAM and 512 bytes of ROM.
The RAM and ROM chips to be used are specified in figure 9.1 and figure 9.2.

Component	Hexa address	Address bus									
		10	9	8	7	6	5	4	3	2	1
RAM 1	0000 - 007F	0	0	0	x	x	x	x	x	x	x
RAM 2	0080 - 00FF	0	0	1	x	x	x	x	x	x	x
RAM 3	0100 - 017F	0	1	0	x	x	x	x	x	x	x
RAM 4	0180 - 01FF	0	1	1	x	x	x	x	x	x	x
ROM	0200 - 03FF	1	x	x	x	x	x	x	x	x	x

Table 9.1: Memory Address Map for Micro-procomputer

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

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

The address bus lines are listed in the third column.

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

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

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

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

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

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

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

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

2. Explain Content Addressable Memory (CAM). (Sum'15)

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

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

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

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

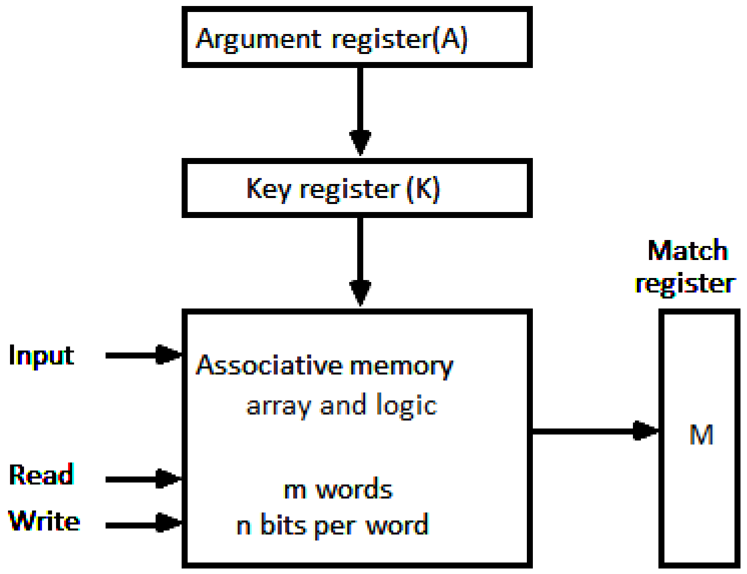


Figure 9.3: Block diagram of associative memory

It consists of a memory array and logic form words with n bits per word.
The argument register A and key register K each have n bits, one for each bit of a word.
The match register M has m bits, one for each memory word.
Each word in memory is compared in parallel with the content of the argument register.
The words that match the bits of the argument register set a corresponding bit in the match register.
After the matching process, those bits in the match register that have been set indicate the fact that their corresponding words have been matched.
Reading is accomplished by a sequential access to memory for those words whose corresponding bits in the match register have been set.

Hardware Organization

The key register provides a mask for choosing a particular field or key in the argument word.
The entire argument is compared with each memory word if the key register contains all 1's.
Otherwise, only those bits in the argument that have 1st in their corresponding position of the key register are compared.
Thus the key provides a mask or identifying piece of information which specifies how the reference to memory is made.
To illustrate with a numerical example, suppose that the argument register A and the key register K have the bit configuration shown below.
Only the three leftmost bits of A are compared with memory words because K has 1's in these position.

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

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

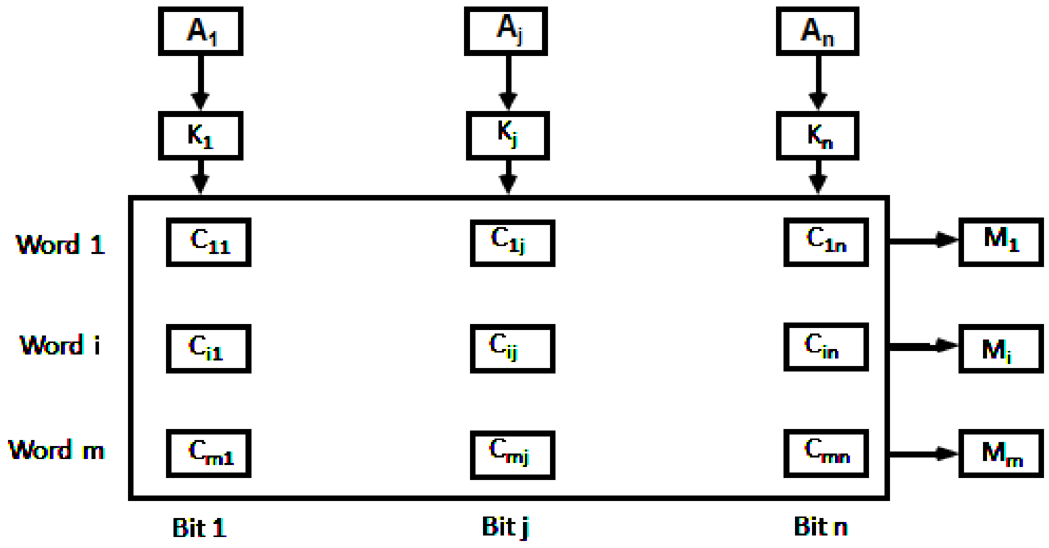


Figure 9.4: Associative memory of *m* word, n cells per word.

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

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

The first subscript gives the word number and the second specifies the bit position in the word.

Thus cell Cij is the cell for bit j in words i.

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

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

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

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

3. Define Cache memory. Discuss associative mapping in organization of cache memory. (Win'15)

Cache memory

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

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

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

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

Associative mapping

Consider the main memory can store 32K words of 12 bits each.

The cache is capable of storing 512 of these words at any given time.

For every word stored in cache, there is a duplicate copy in main memory.

The CPU communicates with both memories.

It first sends a 15-bit address to cache. If there is a hit, the CPU accepts the 12-bit data from cache, if there is miss, the CPU reads the word from main memory and the word is then transferred to cache.

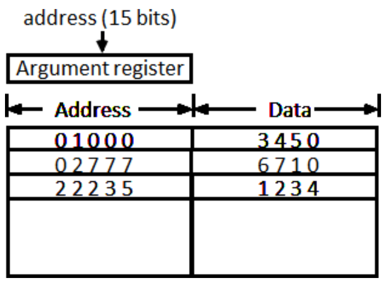


Figure 9.5: Associative mapping cache (all numbers in octal)

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

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

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

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

If the address is found the corresponding 12-bit data is read and sent to CPU.

If no match occurs, the main memory is accessed for the word.

The address data pairs then transferred to the associative cache memory.

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

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

4. Discuss direct mapping in organization of cache memory. (Win'15)

The CPU address of 15 bits is divided into two fields.
The nine least significant bits constitute the index field and the remaining six bits from the tag field.
The figure 9.6 shows that main memory needs an address that includes both the tag and the index.

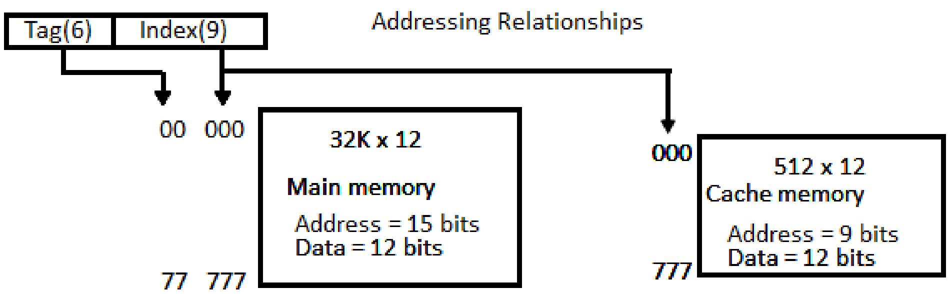


Figure 9.6: Addressing relationships between main and cache memories

The number of bits in the index field is equal to the number of address bits required to access the cache memory.
The internal organization of the words in the cache memory is as shown in figure 9.7.

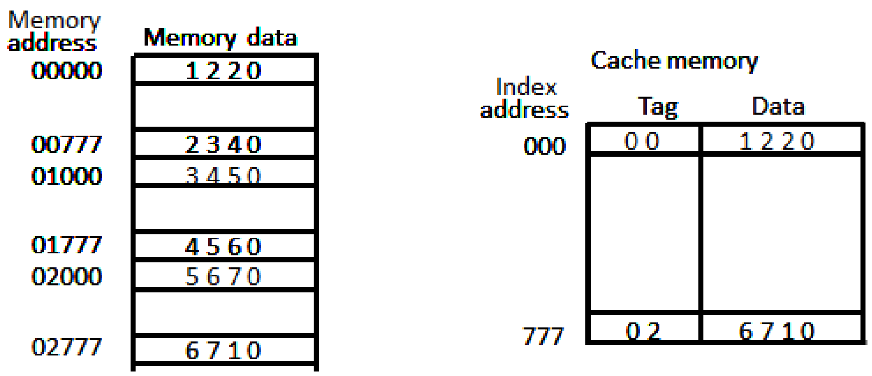


Figure 9.7: Direct mapping cache organization

Each word in cache consists of the data word and its associated tag.
When a new word is first brought into the cache, the tag bits are stored alongside the data bits.
When the CPU generates a memory request the index field is used for the address to access the cache.
The tag field of the CPU address is compared with the tag in the word read from the cache.
If the two tags match, there is a hit and the desired data word is in cache.
If there is no match, there is a miss and the required word is read from main memory.
It is then stored in the cache together with the new tag, replacing the previous value.
The word at address zero is presently stored in the cache (index = 000, tag = 00, data = 1220).

Suppose that the CPU now wants to access the word at address 02000.
The index address is 000, so it is used to access the cache. The two tags are then compared.
The cache tag is 00 but the address tag is 02, which does not produce a match.
Therefore, the main memory is accessed and the data word 5670 is transferred to the CPU.
The cache word at index address 000 is then replaced with a tag of 02 and data of 5670.
The **disadvantage** of direct mapping is that two words with the same index in their address but with different tag values cannot reside in cache memory at the same time.

5. Discuss set-associative mapping in organization of cache memory.

A third type of cache organization, called set associative mapping in that each word of cache can store two or more words of memory under the same index address.
Each data word is stored together with its tag and the number of tag-data items in one word of cache is said to form a set.
An example one set-associative cache organization for a set size of two is shown in figure 9.8.

Index	Tag	Data	Tag	Data
000	0 1	3 4 5 0	0 2	5 6 7 0
777	0 2	6 7 1 0	0 0	2 3 4 0

Figure 9.8: Two-way set-associative mapping cache

Each index address refers to two data words and their associated terms.
Each tag required six bits and each data word has 12 bits, so the word length is 2 (6+12) = 36 bits.
An index address of nine bits can accommodate 512 words.
Thus the size of cache memory is 512 × 36. It can accommodate 1024 words or main memory since each word of cache contains two data words.
In generation a set-associative cache of set size k will accommodate K word of main memory in each word of cache.
The octal numbers listed in figure 9.8 are with reference to the main memory contents.
The words stored at addresses 01000 and 02000 of main memory are stored in cache memory at index address 000.
Similarly, the words at addresses 02777 and 00777 are stored in cache at index address 777.
When the CPU generates a memory request, the index value of the address is used to access the cache.
The tag field of the CPU address is then compared with both tags in the cache to determine if a match occurs.

The comparison logic is done by an associative search of the tags in the set similar to an associative memory search: thus the name "set-associative”.
When a miss occurs in a set-associative cache and the set is full, it is necessary to replace one of the tag-data items with a new value.
The most common replacement algorithms used are: random replacement, first-in first-out (FIFO), and least recently used (LRU).

6. Explain Write-through and Write-back cache write method.

Write Through

The simplest and most commonly used procedure is to update main memory with every memory write operation.
The cache memory being updated in parallel if it contains the word at the specified address. This is called the *write-through* method.
This method has the advantage that main memory always contains the same data as the cache.
This characteristic is important in systems with direct memory access transfers.
It ensures that the data residing in main memory are valid at all times so that an I/O device communicating through DMA would receive the most recent updated data.

Write-Back (Copy-Back)

The second procedure is called the write-back method.
In this method only the cache location is updated during a write operation.
The location is then marked by a flag so that later when the word is removed from the cache it is copied into main memory.
The reason for the write-back method is that during the time a word resides in the cache, it may be updated several times.
However, as long as the word remains in the cache, it does not matter whether the copy in main memory is out of date, since requests from the word are filled from the cache.
It is only when the word is displaced from the cache that an accurate copy need be rewritten into main memory.

7. What do you mean by address space and memory space in virtual memory? Also explain the relation between address space and memory space in virtual memory. (Win’15)

Virtual Memory

Virtual memory is used to give programmers the illusion that they have a very large memory at their disposal, even though the computer actually has a relatively small main memory.
A virtual memory system provides a mechanism for translating program-generated addresses into correct main memory locations.

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

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

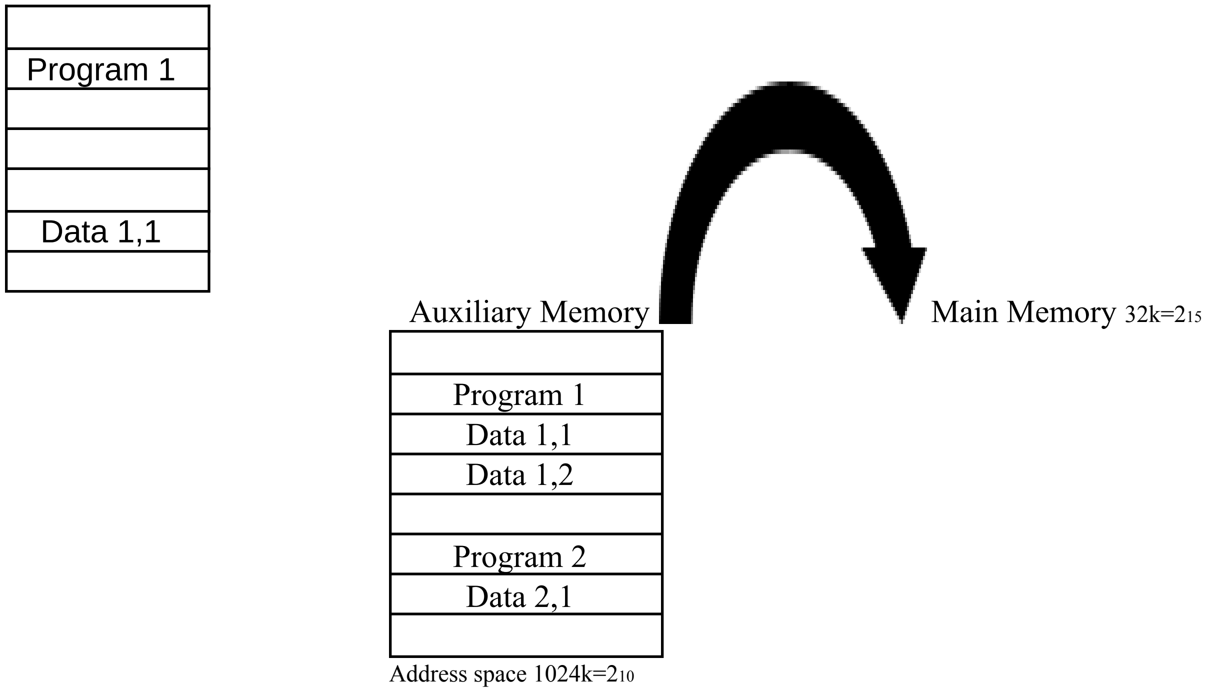


Figure 9.9: Relation between address and memory space in a virtual memory system

As an illustration, consider a computer with a main-memory capacity of 32K words (K = 1024). Fifteen bits are needed to specify a physical address in memory since 32K = 2^{15}. Suppose that the computer has available auxiliary memory for storing 2^{20} = 1024K words. Thus auxiliary memory has a capacity for storing information equivalent to the capacity of 32 main memories.

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

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

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

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

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

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

8. Explain address mapping using pages.

The table implementation of the address mapping is simplified if the information in the address space and the memory space are each divided into groups of fixed size.
The physical memory is broken down into groups of equal size called blocks, which may range from 64 to 4096 words each.
The term page refers to groups of address space of the same size.
Consider a computer with an address space of 8K and a memory space of 4K.
If we split each into groups of 1K words we obtain eight pages and four blocks as shown in figure 9.9
At any given time, up to four pages of address space may reside in main memory in any one of the four blocks.

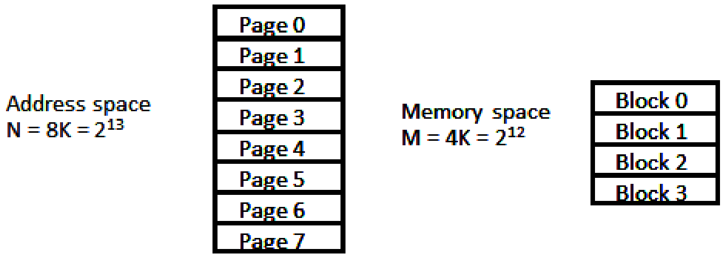


Figure 9.10 Address and Memory space split into group of 1K words

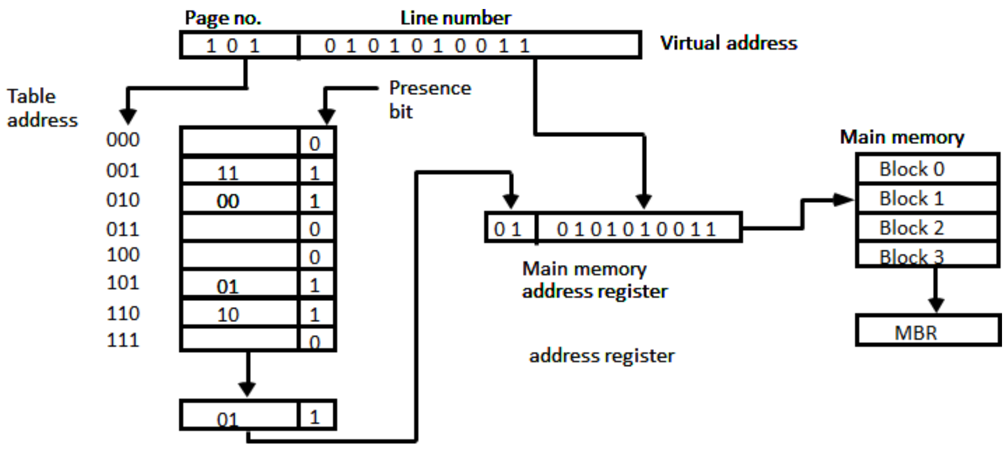


Figure 9.11: Memory table in paged system

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

The CPU references a word in memory with a virtual address of 13 bits.
The three high-order bits of the virtual address specify a page number and also an address for the memory-page table.
The content of the word in the memory page table at the page number address is read out into the memory table buffer register.
If the presence bit is a 1, the block number thus read is transferred to the two high-order bits of the main memory address register.
The line number from the virtual address is transferred into the 10 low-order bits of the memory address register.
A read signal to main memory transfers the content of the word to the main memory buffer register ready to be used by the CPU.
If the presence bit in the word read from the page table is 0, it signifies that the content of the word referenced by the virtual address does not reside in main memory.

9. What is segment? What is logical address? Explain segmented page mapping.

Segment

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

Logical address

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

Segmented page mapping

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

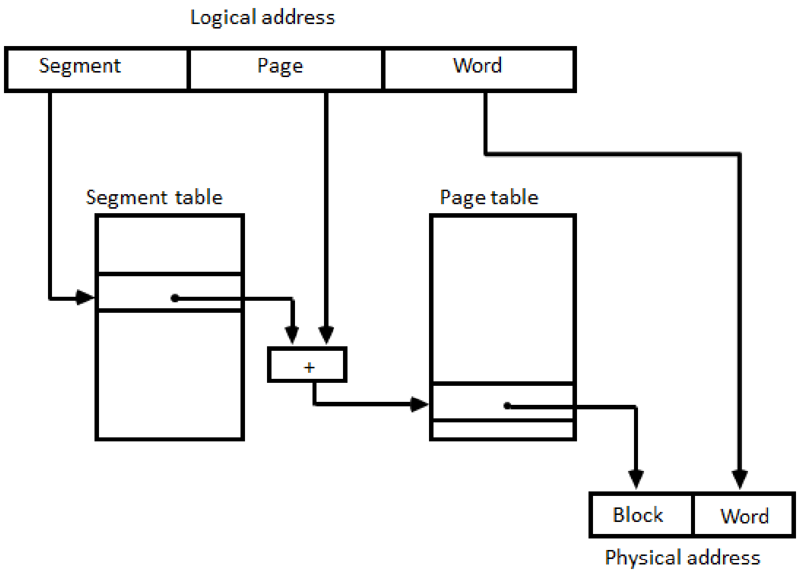


Figure 9.12: Logical to physical address mapping

The logical address is partitioned into three fields.
The segment field specifies a segment number.

The page field specifies the page within the segment and word field gives specific word within the page.
A page field of k bits can specify up to 2k pages.
A segment number may be associated with just one page or with as many as 2k pages.
Thus the length of a segment would vary according to the number of pages that are assigned to it.
The mapping of the logical address into a physical address is done by means of two tables, as shown in figure 9.12.
The segment number of the logical address specifies the address for the segment table.
The entry in the segment table is a pointer address for a page table base.
The page table base is added to the page number given in the logical address.
The sum produces a pointer address to an entry in the page table.
The concatenation of the block field with the word field produces the final physical mapped address.
The two mapping tables may be stored in two separate small memories or in main memory.
In either case, memory reference from the CPU will require three accesses to memory: one from the segment table, one from the page table and the third from main memory.
This would slow the system significantly when compared to a conventional system that requires only one reference to memory.

1. Differentiate Tightly coupled and Loosely coupled system.

Tightly Coupled System	Loosely Coupled System
Tasks and/or processors communicate in a highly synchronized fashion.	Tasks or processors do not communicate in a synchronized fashion.
Communicates through a common shared memory.	Communicates by message passing packets.
Shared memory system.	Distributed memory system.
Overhead for data exchange is lower comparatively.	Overhead for data exchange is higher comparatively.

2. Explain Time Shared Common bus Interconnection Structures.

- A common bus multiprocessor system consists of a number of processors connected through a common path to a memory unit.
- A time shared common bus for five processors is shown in figure 10.1.
- Only one processor can communicate with the memory or another processor at any given time.
- Transfer operations are conducted by the processor that is in control of the bus at the time.

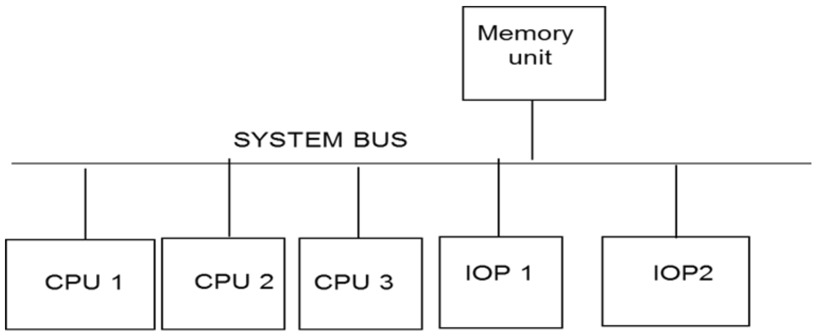


Figure 10.1: Time-shared common bus organization

- Any other processor wishing to initiate a transfer must first determine the availability status of the bus and only after the bus becomes available, the processor address the destination unit to initiate the transfer.
- A command is issued to inform the destination unit, what operation is to be performed.
- The receiving unit recognizes its address in the bus and responds to the control signals from the sender, after which the transfer is initiated.
- A single common-bus system is restricted to one transfer at a time.
- This means that when one processor is communicating with the memory, all other processors are either busy with internal operations or must be idle waiting for the bus.
- As a consequence, the total overall transfer rate within the system is limited by the speed of the single path.
- A more economical implementation of a dual bus structure is depicted in figure 10.2.

- Here we have a number of local buses each connected to its own local memory and to one or more processors.
- Each local bus may be connected to a CPU, an IOP, or any combination of processors.
- A system bus controller links each local bus to a common system bus.
- The I/O devices connected to the local IOP, as well as the local memory, are available to the local processor.

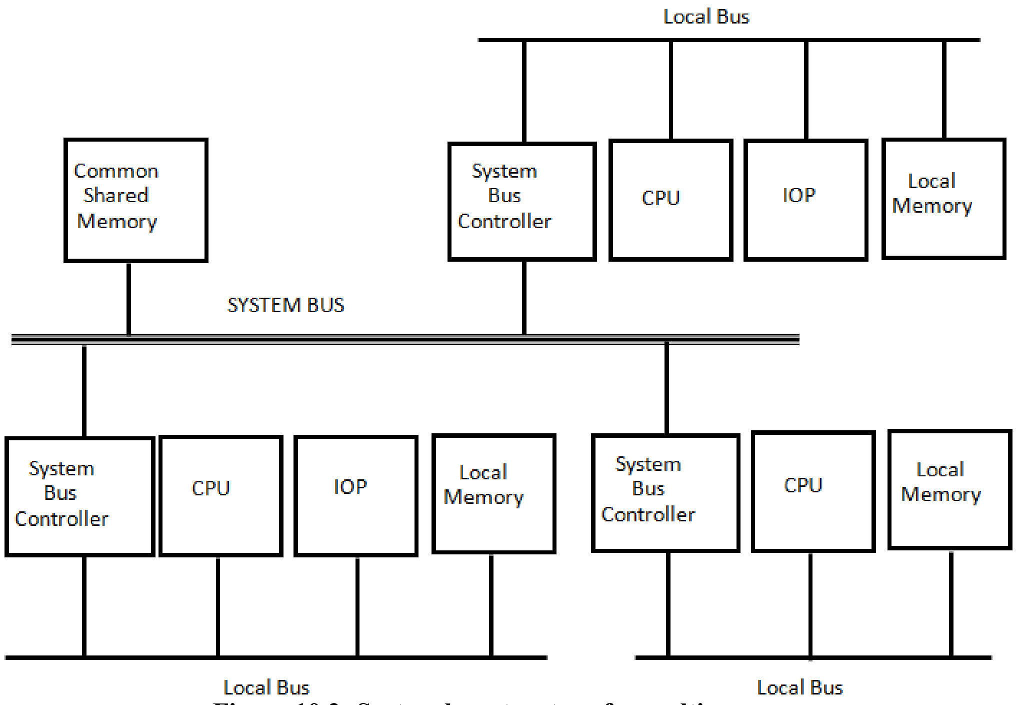


Figure 10.2: System bus structure for multiprocessors

- The memory connected to the common system bus is shared by all processors.
- If an IOP is connected directly to the system bus, the I/O devices attached to it may be made available to all processors.
- Only one processor can communicate shared memory with the shared memory and other common resources through the system bus at any given time.
- The other processors are kept busy communicating with their local memory and I/O devices.

2. Explain Multiport Memory Interconnection Structures.

A multiport memory system employs separate buses between each memory module and each CPU.
This is shown in figure 10.3 for four CPUs and four memory modules (MMs).
Each processor bus is connected to each memory module.
A processor bus consists of the address, data, and control lines required to communicate with memory.
The memory module is said to have four ports and each port accommodates one of the buses.

The module must have internal control logic to determine which port will have access to memory at any given time.
Memory access conflicts are resolved by assigning fixed priorities to each memory port.

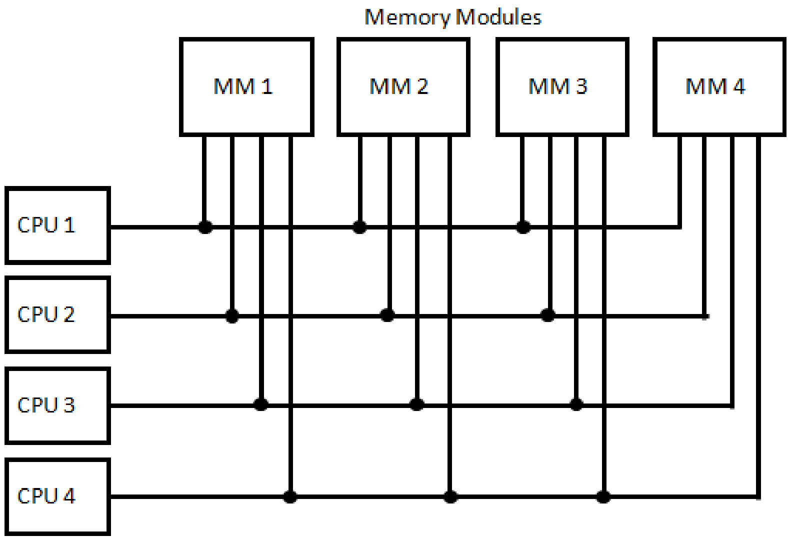


Figure 10.3: Multiport memory organization

The priority for memory access associated with each processor may be established by the physical port position that its bus occupies in each module.
Thus CPU 1 will have priority over CPU 2, CPU 2 will have priority over CPU 3, and CPU 4 will have the lowest priority.
The advantage of the multiport memory organization is the high transfer rate that can be achieved because of the multiple paths between processors and memory.
The disadvantage is that it requires expensive memory control logic and a large number of cables and connectors.

3. Explain Crossbar Switch Interconnection Structures.

Figure 10.4 shows a crossbar switch interconnection between four CPUs and four memory modules.
The small square in each cross point is a switch that determines the path from a processor to a memory module.
Each switch point has control logic to set up the transfer path between a processor and memory.
It examines the address that is placed in the bus to determine whether its particular module is being addressed.
It also resolves multiple requests for access to the same memory module on a predetermined priority basis, figure 10.5 shows the functional design of a crossbar switch connected to one memory module.
The circuit consists of multiplexers that select the data, address, and control from one CPU for communication with the memory module.

Priority levels are established by the arbitration logic to select one CPU when two or more CPUs attempt to access the same memory.

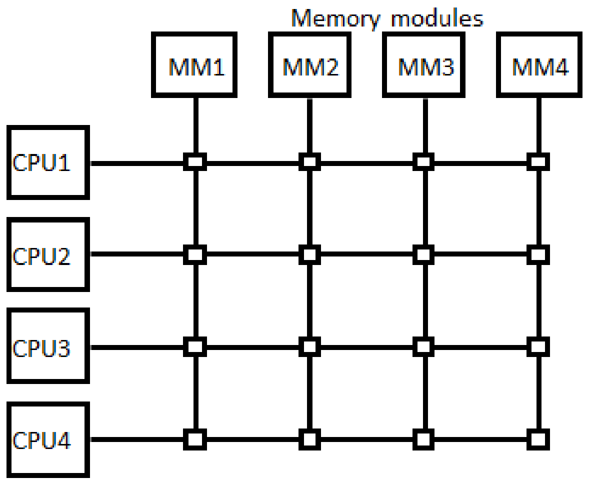


Figure 10.4: Crossbar switch

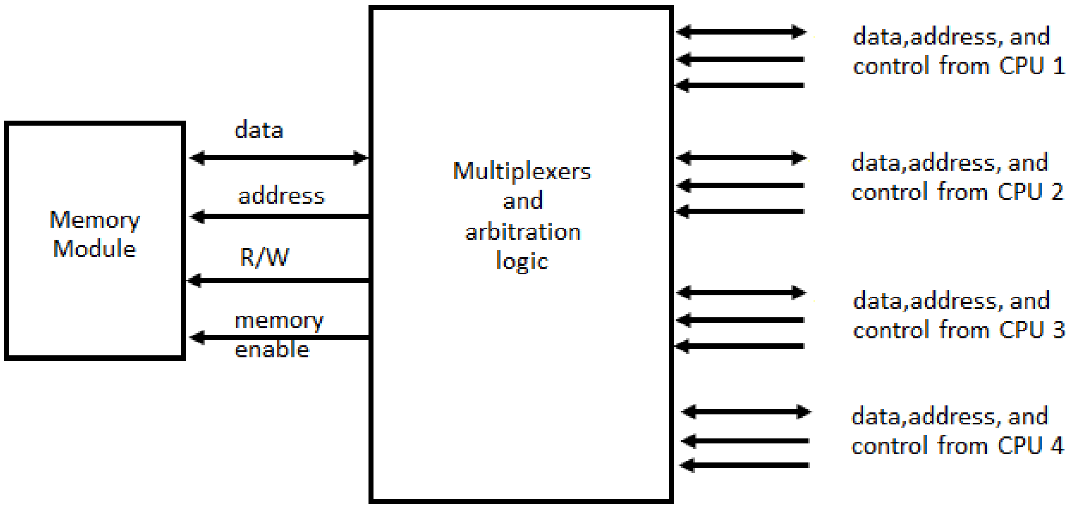

Memory Module
data
address
R/W
memory enable
Multiplexers and arbitration logic
data,address, and control from CPU 1
data,address, and control from CPU 2
data,address, and control from CPU 3
data,address, and control from CPU 4

Figure 10.5: Block diagram of crossbar switch

The multiplex are controlled with the binary code that is generated by a priority encoder with in the arbitration logic.
A crossbar switch organization supports simultaneous transfers from memory modules because there is a separate path associated with each module.
However, the hardware required to implement the switch can becomes quite large and complex.

4. Explain Multistage Switching Network Interconnection Structures.

The basic component of a multistage network is a two-input, two-on interchange switch interchange switch.

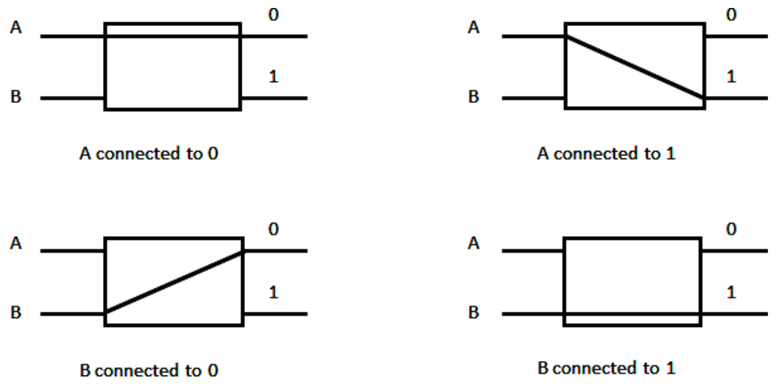


Figure 10.6: Operation of a 2 x 2 interchange switch

As shown in figure 10.6 the 2 X 2 switch has two input labeled A and B, and two outputs, labeled 0 and 1.

There are control sign (not shown) associated with the switch that establish the interconnection between the input and output terminals.

The switch has the capability connecting input A to either of the outputs. Terminal B of the switch behaves in a similar fashion.

The switch also has the capability to arbitrate between conflicting requests.

If inputs A and B both request the same output terminal only one of them will be connected; the other will be blocked.

Using the 2 X 2 switch as a building block, it is possible to build multistage network to control the communication between a number of source and destinations.

To see how this is done, consider the binary tree shown figure 10.7.

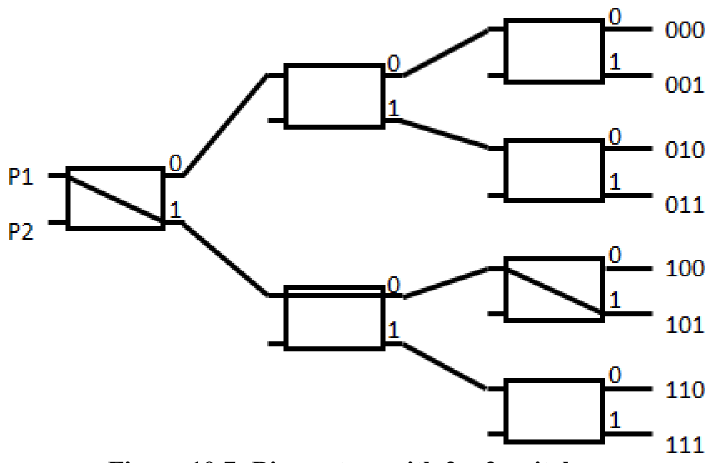


Figure 10.7: Binary tree with 2 x 2 switches

The two processors P1 and P2 are connected through switches to eight memory modules marked in binary from 000 through 111.

The path from source to a destination is determined from the binary bits of the destination number.
The first bit of the destination number determines the switch output in the first level.
The second bit specifies the output of the switch in the third level.
Example to connect P1 to memory 101, it is necessary to form a path from P1 to output 1 in first level switch, output 0 in second level switch and output 1 in the third level switch.

5. Explain Hypercube Interconnection Structures.

The hypercube or binary n cube multiprocessor structure is loosely coupled system composed of $N=2^n$ processors, interconnected in n-dimensional binary cube.
Each processor forms a node of the cube.
Each processor has direct communication paths to other neighbor processors.
These paths correspond to the edges of the cube.
There are 2^n distinct n-bit binary addresses that can be assigned to the processors.
Each processor address differs from that of each of its n neighbors by exactly one bit position.
Figure 10.8 shows the hypercube structure for n = 1, 2, and 3.
A one-cube structure has n = 1 and $2^n = 2$. It contains two processors interconnected by a single path.
A two-cube structure has $2^1 = 2$ and $2^n = 4$. It contains four nodes interconnected as a square.

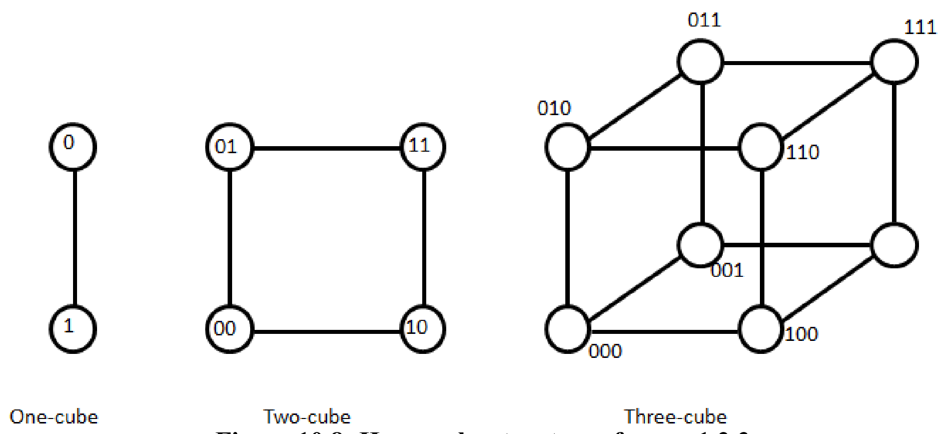


Figure 10.8: Hypercube structures for n = 1,2,3

A three-cube structure has eight nodes interconnected as a cube.
An n-cube structure has 2n nodes with a processor residing in each node.
Each node is assigned a binary address in such a way that the addresses of two neighbors differ in exactly one bit position.
For example, the three neighbors of the node with address 100 in a three-cube structure are 000,110, and 101.
Each of these binary numbers differs from address 100 by one bit value.
For example, in a three-cube structure, node 000 can communicate directly with node 001.
It must cross at least two links to communicate with 011 (from 000 to 001 to 011 or from 000 to 010 to 011).
It is necessary to go through at least three links to communicate from node 000 to node 111.

A routing procedure can be developed by computing the exclusive-OR of the source node address with the destination node address.
For example, in a three-cube structure, a message at 010 going to 001 produces an exclusive-OR of the two addresses equal to 011.
The message can be sent along the second axis to 000 and then through the third axis to 001.

6. Explain Daisy chain (Serial) arbitration.

Arbitration procedures service all processor requests on the basis of established priorities.
The serial priority resolving technique is obtained from a daisy-chain connection of bus arbitration circuits.
The processors connected to the system bus are assigned priority according to their position along the priority control line.
The device closest to the priority line is assigned the highest priority.
When multiple devices concurrently request the use of the bus, the device with the highest priority is granted access to it.

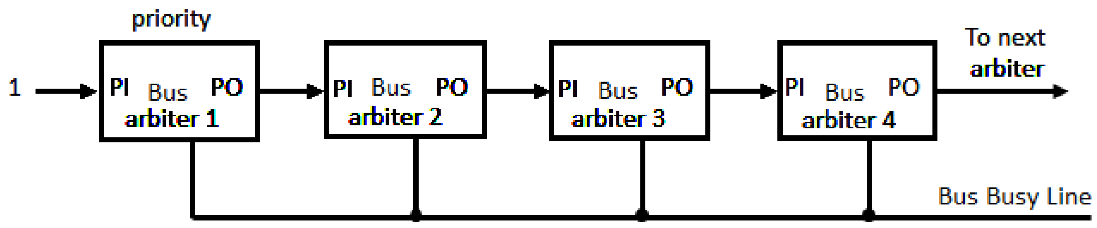


Figure 10.9: Serial (daisy-chain) arbitration

Figure 10.9 shows the daisy chaining connection of four arbiters.
It is assumed that each processor has its own bus arbiter logic with priority.
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 logic 1 value.
The highest-priority unit in the system will always receive access to the system bus when it requests it.
The PO output for a particular arbiter is equal to 1 if its PI input is equal to 1 and the processor associated with the arbiter logic is not requesting control of the bus.
This is the way that priority is passed to the next unit in the chain.
When processor requests control of the bus and the corresponding arbiter finds its Pl input equal to 1, it sets its PO output to 0.
Lower-priority arbiters receive a 0 in PI and generate a 0 in PO.
Thus the processor whose arbiter has a PI = 1 and PO = 0 is the one that is given control of the system bus.
The busy line comes from open-collector circuits in each unit and provides a wired-OR logic connection.
If the line is inactive, it means that no other processor is using the bus.

7. Explain Parallel Arbitration Logic.

The parallel bus arbitration technique uses an external priority encoder and a decoder as shown in figure 10.10.

Each bus arbiter in the parallel scheme has a bus request output line and a bus acknowledge input line.

Each arbiter enables the request line when its processor is requesting access to the system bus.

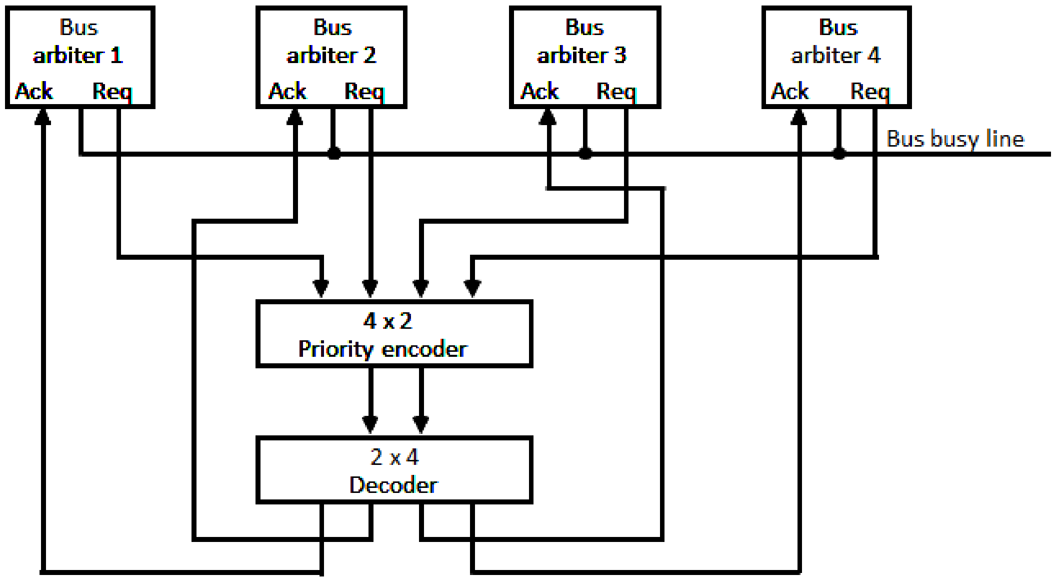


Figure 10.10: Parallel arbitration

The processor takes control of the bus if it acknowledge input line is enabled.

The bus busy line provides an orderly transfer of control, as in the Daisy chaining case.

Figure 10.10 shows the request lines from four arbiters going into a 4 X 2 priority encoder.

The output of the encoder generates a 2-bit code which represents the highest-priority unit among those requesting the bus.

The bus priority-in (BPRN) and bus priority-out (BPRO) are used for a daisy-chain connection of bus arbitration circuits.

The bus busy signal BUSY is an open-collector output used to instruct all arbiters when the bus is busy conducting a transfer.

The common bus request (CBRQ) is also an open-collector output that serves to instruct the arbiter if there are any other arbiters of lower-priority requesting use of the system bus.

The signals used to construct a parallel arbitration procedure are bus request (BREQ) and priority-in (BPRN), corresponding to the request and acknowledgement signals in figure 10.10.

The bus clock (BCLK) is used to synchronize all bus transactions.

8. Explain Dynamic Arbitration Algorithms

A dynamic priority algorithm gives the system the capability for changing the priority of the devices while the system is in operation.

Time slice

The time slice algorithm allocates a fixed-length time slice of bus time that is offered sequentially to each processor, in round-robin fashion.
The service is location independent.
No preference is given to any particular device since each is allotted the same amount of time to communicate with the bus.

Polling

In a bus system that uses polling, the bus grant signal is replaced by a set of lines called poll lines which are connected to all units.
These lines are used by the bus controller to define an address for each device connected to the bus.
The bus controller sequences through the addresses in a prescribed manner.
When a processor that requires access recognizes its address, it activates the bus busy line and then accesses the bus.
After a number of bus cycle, the polling process continues by choosing a different processor.
The polling sequence is normally programmable, and as a result, the selection priority can be altered under program control.

LRU

The least recently used (LRU) algorithm gives the highest priority to the requesting device that has not used the bus for the longest interval.
The priorities are adjusted after a number of bus cycles according to the LRU algorithm.
With this procedure, no processor is favored over any other since the priorities are dynamically changed to give every device an opportunity to access the bus.

FIFO

In the first-come first-serve scheme, requests are served in the order received.
To implement this algorithm the bus controller establishes a queue arranged according to the time that the bus requests arrive.
Each processor must wait for its turn to use the bus on a first-in first-out (FIFO) basis.

Rotating daisy-chain

The rotating daisy-chain procedure is a dynamic extension of the daisy chain algorithm.
Highest priority to the unit that is nearest to the unit that has most recently accessed the bus (it becomes the bus controller).

9. Describe cache coherence problem and its solutions in detail.

Cache coherence problem

To ensure the ability of the system to execute memory operations correctly, the multiple copies must be kept identical.
This requirement imposes a cache coherence problem.
A memory scheme is coherent if the value returned on a load instruction is always the value given by the latest store instruction with the same address.
Cache coherence problems exist in multiprocessors with private caches because of the need to share writable data.
Read-only data can safely be replicated without cache coherence enforcement mechanisms.
To illustrate the problem, consider the three-processor configuration with private caches shown in figure 10.11.
During the operation an element X from main memory is loaded into the three processors, P1, P2, and P3.
It is also copied into the private caches of the three processors.
For simplicity, we assume that X= 52.
The load on X to the three processors results in consistent copies in the caches and main memory.
If one of the processors performs a store to X, the copies of X in the caches become inconsistent.
A load by the other processors will not return the latest value.

Write-through policy

As shown in figure 10.12, a store to X (of the value of 120) into the cache of processor P1 updates memory to the new value in a write-through policy.
A write-through policy maintains consistency between memory and the originating cache, but the other two caches are inconsistent since they still hold the old value.

Write-back policy

In a write-back policy, main memory is not updated at the time of the store.
The copies in the other two caches and main memory are inconsistent.
Memory is updated eventually when the modified data in the cache are copied back into memory.
Another configuration that may cause consistency problems is a direct memory access (DMA) activity in conjunction with an IOP connected to the system bus.
In the case of input, the DMA may modify locations in main memory that also reside in cache without updating the cache.
During a DMA output, memory locations may be read before they are updated from the cache when using a write-back policy.

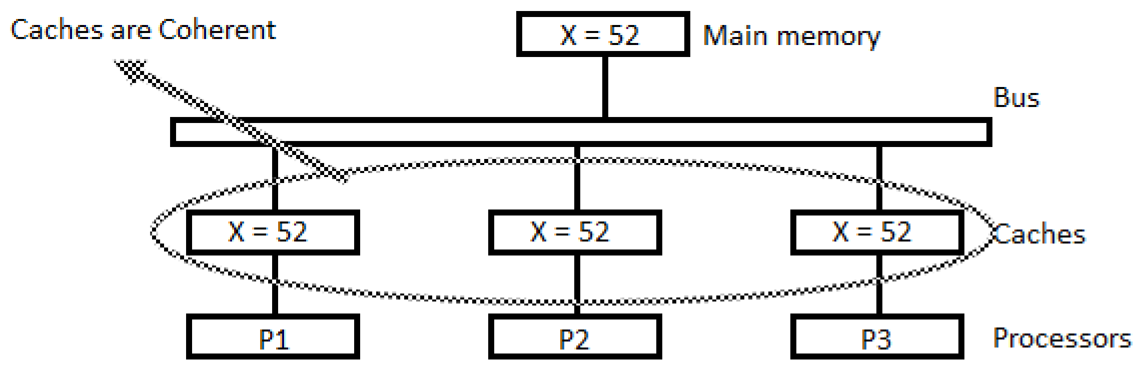


Figure 10.11: Cache configuration after a load on X.

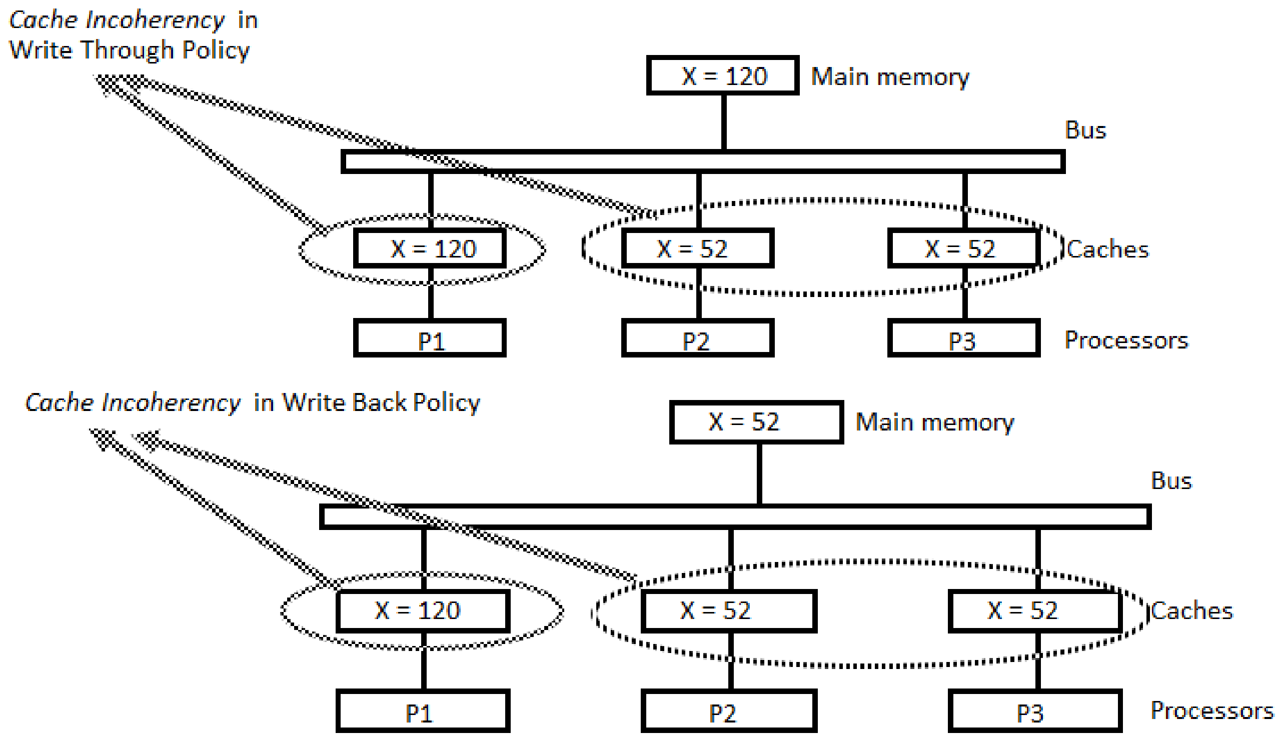


Figure 10.12: Cache configuration after a store to X by processor P1

Solution of cache coherence problem

Various schemes have been proposed to solve the cache coherence problem in shared memory multiprocessors.

Disallow private caches

A simple scheme is to disallow private caches for each processor and have a shared cache memory associated with main memory.

Every data access is made to the shared cache.

This method violates the principle of closeness of CPU to cache and increases the average memory access time.

In effect, this scheme solves the problem by avoiding it.

Software Approaches

Read-Only Data are Cacheable

The scheme that allows only nonshared and read-only data to be stored in caches. Such items are called cachable.

Shared writable data are noncachable.

The compiler must tag data as either cachable or noncachable, and the system hardware makes sure that only cachable data are stored in caches.

The noncachable data remain in main memory.

This method restricts the type of data stored in caches and introduces an extra software overhead that may degrades performance.

Centralized Global Table

A scheme that allows writable data to exist in at least one cache is a method that employs a centralized global table in it compiler.

The status of memory blocks is stored in the central global table.

Each block is identified as read-only (RO) or read and write (RW).

All caches can have copies of blocks identified as RO.

Only one cache can have a copy of an RW block.

Thus if the data are updated in the cache with an RW block, the other caches are not affected because they do not have a copy of this block.

Hardware Approaches

Hardware-only solutions are handled by the hardware automatically and have the advantage of higher speed and program transparency.

Snoopy Cache Controller

In the hardware solution, the cache controller is specially designed to allow it to monitor all bus requests from CPUs and IOPs.

All caches attached to the bus constantly monitor the network for possible write operations.

Depending on the method used, they must then either update or invalidate their own cache copies when a match is detected.

The bus controller that monitors this action is referred to as a snoopy cache controller.

This is basically a hardware unit designed to maintain a bus-watching mechanism over all the caches attached to the bus.

All the snoopy controllers watch the bus for memory store operations.

When a word in a cache is updated by writing into it, the corresponding location in main memory is also updated.

The local snoopy controllers in all other caches check their memory to determine if they have a copy of the word that has been overwritten.

If a copy exists in a remote cache, that location is marked invalid.

Because all caches snoop on all bus writes, whenever a word is written, the net effect is to update it in the original cache and main memory and remove it from all other caches.

If at some future time a processor accesses the invalid item from its cache, the response is equivalent to a cache miss, and the updated item is transferred from main memory. In this way, inconsistent versions are prevented.