HOLY CROSS HOME SCIENCE COLLEGE, THOOTHUKUDI

B.Sc., - II YEAR

SMCS32 - COMPUTER ARCHITECTURE

R.AME RAYAN,
ASSISTANT PROFESSOR,
DEPARTMENT OF COMPUTER SCIENCE

Basic Computer Organization and Design

Unit - I

Instruction Code

- The internal Organization of defined by the sequence of microoperations it performs on data stored in its registers.
- The general purpose digital computer is capable of executing various microoperations and to instruct the specify sequence of operation it should perform.
- The user controls the process by means of program
- Program – set of instructions that specify the operation and the operands and the sequence by which the process has to occur.

- Computer instruction – it is the binary code that specifies the sequence of microoperation for the computer.

(i) instruction code together with data are stored in memory.

(ii) the computer reads each instruction from memory and places it in the control register.

(iii)the control register then interprets the binary code of the instruction and proceeds to execute it by issuing a sequence of microoperation.

(iv) Every computer has a unique instruction set.

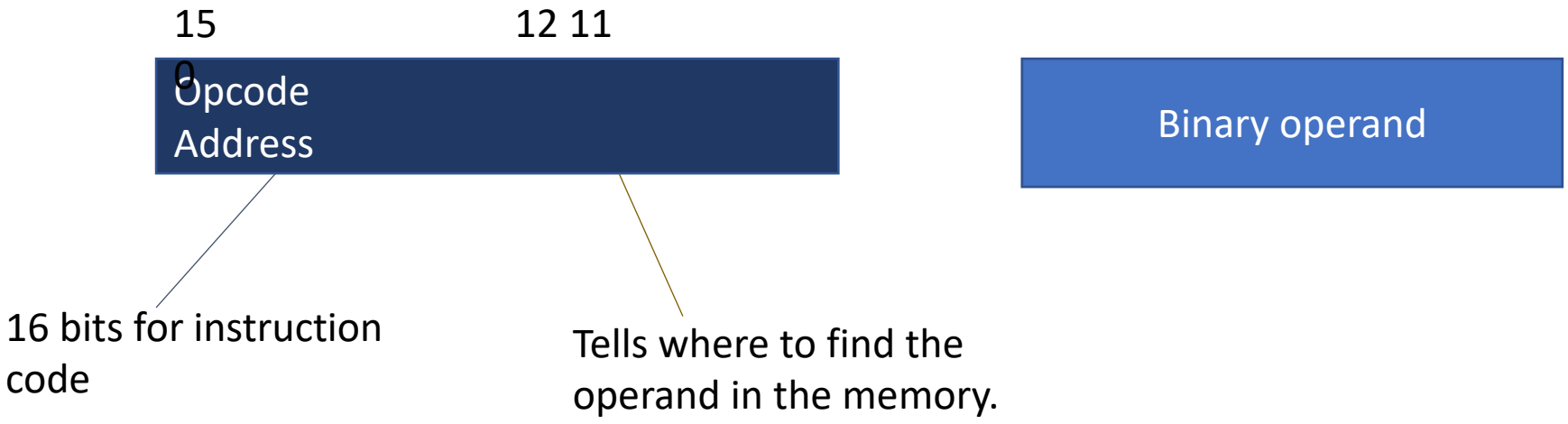
- Instruction code : is a group of bits that instructs the computer to perform a specific operation.
- The most basic part of the instruction code is the operation part.
- Operation code – is a group of bits that define operations such as add, subtract,multiply and divide.
- The number of bits required for the operation depends on the number of operations available in the computer.
- For 2^n operations we need n bits.

Relation between computer operation and microoperation

- The operation part of the instruction code specifies the operation to be performed on the data stored in processor register.
- So, the instruction code must also specify the memory address where the operands are found.
- The control unit receives the instruction from the memory and initiates a sequence of microoperation to be performed.
- The operation code is sometimes called as microoperation.

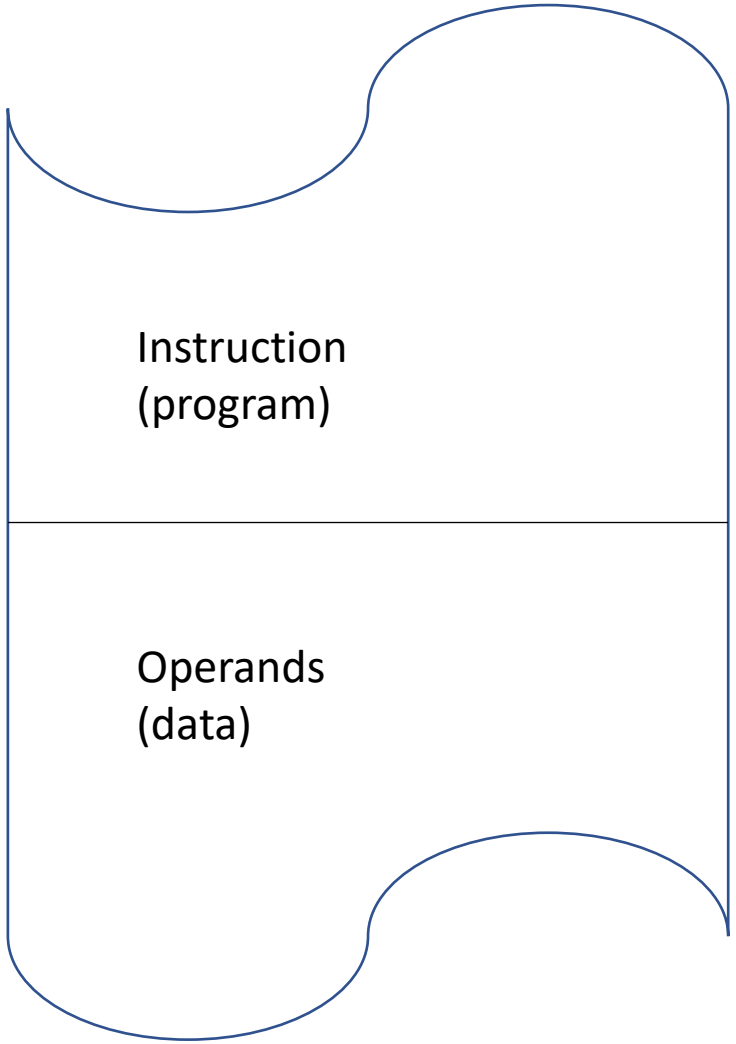
Stored program organization

- Simplest way to organize a computer is to have one processor register and instruction code with two parts.



Eg : memory unit – 4096 words
12 bits – address since, 2^{12} =4096

Memory 4096 x 16

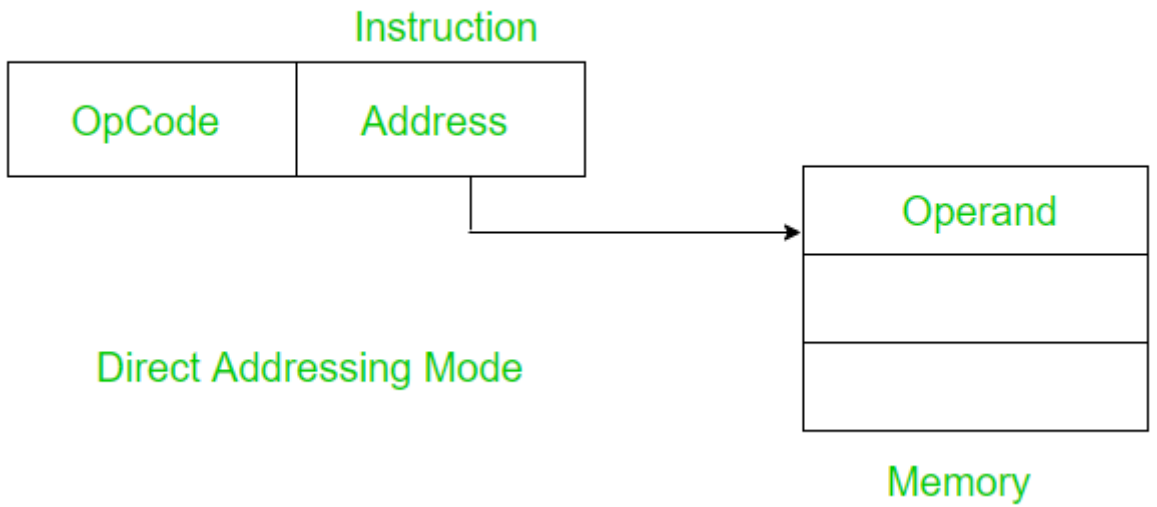


Processor Register (AC)

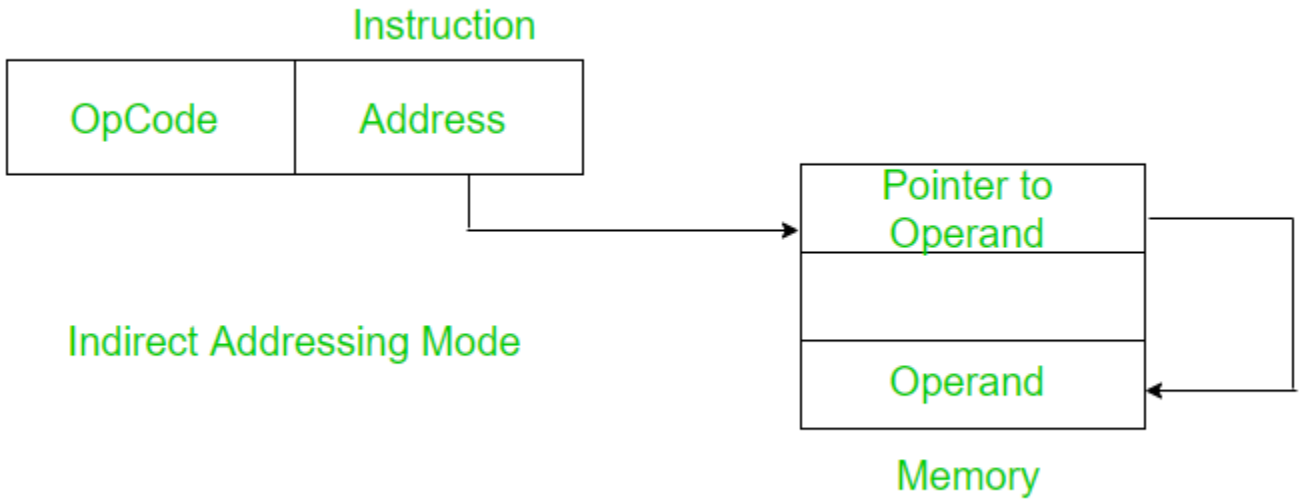
R.Ame Rayan, Holy Cross Home Science College, Thoothukudi.
09.07.2020

- Some instructions does not have the operand part , in such cases it can be used for other purpose.
- Eg: increment AC, clear AC, Complement AC
- Immediate instruction
- Direct addressing
- Indirect addressing
- Effective address

Direct Addressing

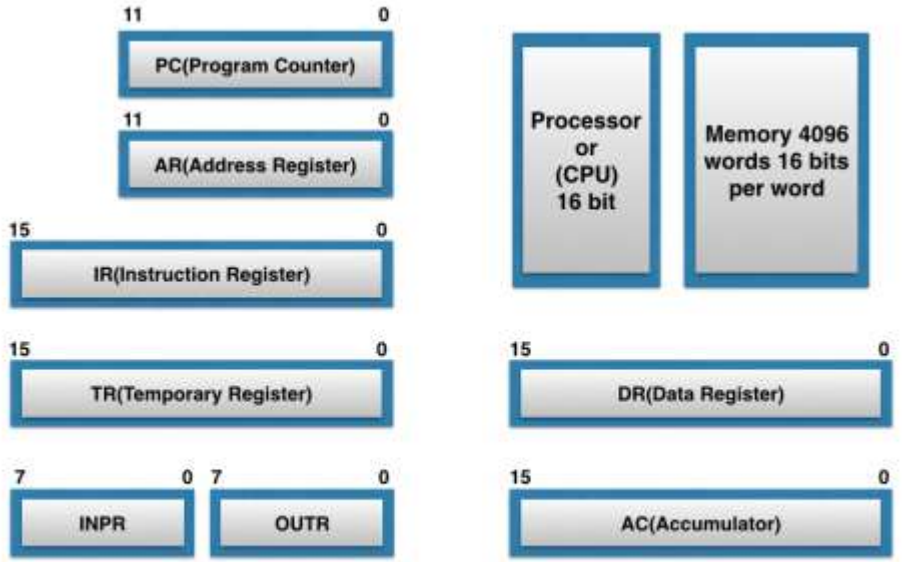


In-Direct Addressing



Computer Registers

Reg. Symbol	No. of Bits	Register	Register Function
DR	16	Data Register	Hold operand
AR	12	Address Register	Hold address for Mem
AC	16	Accumulator	Processor register
IR	16	Instruction Register	Hold instruction code
PC	12	Program Counter	Hold address of instruct
TR	16	Temp Register	Hold temp data
INPR	8	Input Register	Holds input char
OUTR	8	Output Register	Holds output char



Basic Computer Registers and Memory

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 registers.

The number of wires will be excessive if connections are made between the outputs of each register and the inputs of the other registers.

A more efficient scheme for transferring information in a system with many registers is to use a common bus.

The connection of the registers and memory of the basic computer to a common bus system. 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 S_2, S_1, and S_0.

The number along each output shows the decimal equivalent of the required binary selection. For example, the number along the output of DR is 3.

The 16-bit outputs of DR are placed on the bus lines when $S_2S_1S_0 = 011$ since this is the binary value of decimal 3.

The lines from the common bus are connected to the inputs of each register and the data inputs of the memory. 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 $S_2S_1S_0 = 111$.

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 or 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.

This is because INPR receives a character from an input device which is then transferred to AC. OUTR receives a character from AC and delivers it to an output device. There is no transfer from OUTR to any of the other registers.

The 16 lines of the common bus receive information from six registers and the memory unit. The bus lines are connected to the inputs of six registers and the memory. Five registers have three control inputs: LD (load), INR (increment), and CLR (clear).

This type of register is equivalent to a binary counter with parallel load and synchronous clear. The increment operation is achieved by enabling the count input of the counter. Two registers have only a LD input.

•**The input data and output data** of the memory are connected to the common bus, but the memory address is connected to AR. Therefore, AR must always be used to specify a memory address.

•**By using a single register** for the address, we eliminate the need for an address bus that would have been needed otherwise. The content of any register can be specified for the memory data input during a write operation. Similarly, any register can receive the data from memory after a read operation except AC .

•**The 16 inputs of AC** come from an adder and logic circuit. This circuit has three sets of inputs. One set of 16-bit inputs come from the outputs of AC . They are used to implement register microoperations such as complement AC and shift AC .

•**Another set of 16-bit inputs** come from the data register DR. The inputs from DR and AC are used for arithmetic and logic rnlcrooperations, such as add DR to AC or AND DR to AC.

•**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). A third set of 8-bit inputs come from the input register INPR.

•**Note that the content of any register** can be applied onto the bus and an operation can be performed in the adder and logic circuit during the same clock cycle. 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.

•**For example**, the two microoperations

•DR ← AC and AC ← DR

•**can be executed at the same time**. This can be done by placing the content of AC on the bus (with $S_2S_1S_0$ = 100), enabling the LD (load) input of DR, transferring the content of DR through the adder and logic circuit into AC, and enabling the LD (load) input of AC, all during the same clock cycle.

•**The two transfers** occur upon the arrival of the clock pulse transition at the end of the clock cycle.

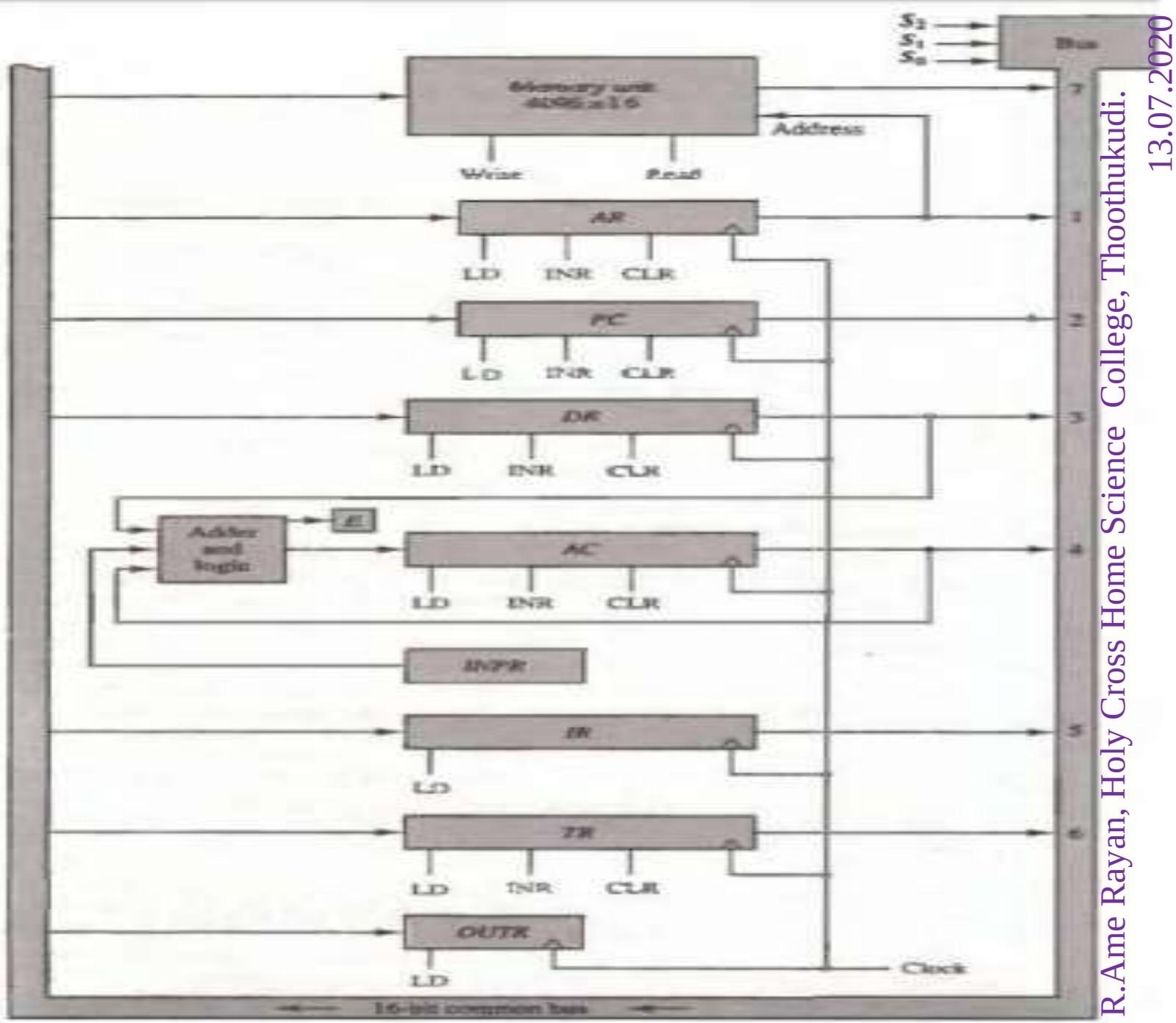


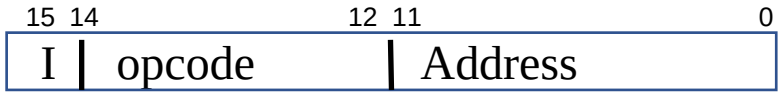
Figure 5-4 Basic computer registers connected to a common bus.

Computer Instructions

All Basic Computer instruction codes are 16 bits wide.
The operation part of the instruction contains 3 bits and the meaning of the remaining 13 bits depend on the operation code encountered.

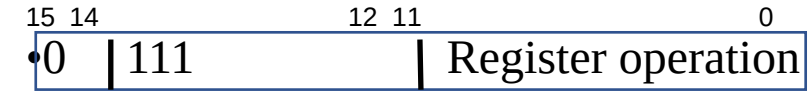
There are 3 instruction code formats:

•Memory-reference instructions take a single memory address as an operand, and have the format:



If I = 0, the instruction uses direct addressing. If I = 1, addressing in indirect.

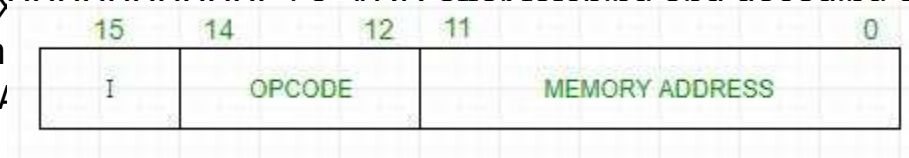
Register-reference instructions operate solely on the AC register, and have the following format:



•Input/output instructions have the following format:

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

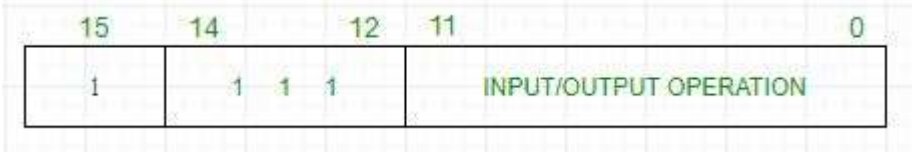
Example –
IR register contains = 0001XX[illegible] i.e. ADD after fetching and decoding of instruction we find out that it is a memory reference instruction
Hence, DR ← M[AR] AC ← A[illegible]



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

Example –
IR register contains = 011[illegible] i.e. CMA after fetch and decode cycle we find out that it is a register reference instruction for c[illegible]

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



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

The set of instructions incorporated in16 bit IR register are:

1.Arithmetic, logical and shift instructions
(and, add, complement, circulate left, right, etc)

2.To move information to and from memory
(store the accumulator, load the accumulator)

3.Program control instructions with status conditions
(branch, skip)

4.Input output instructions

5. (input character, output character)

TABLE 5-2 Basic Computer Instructions

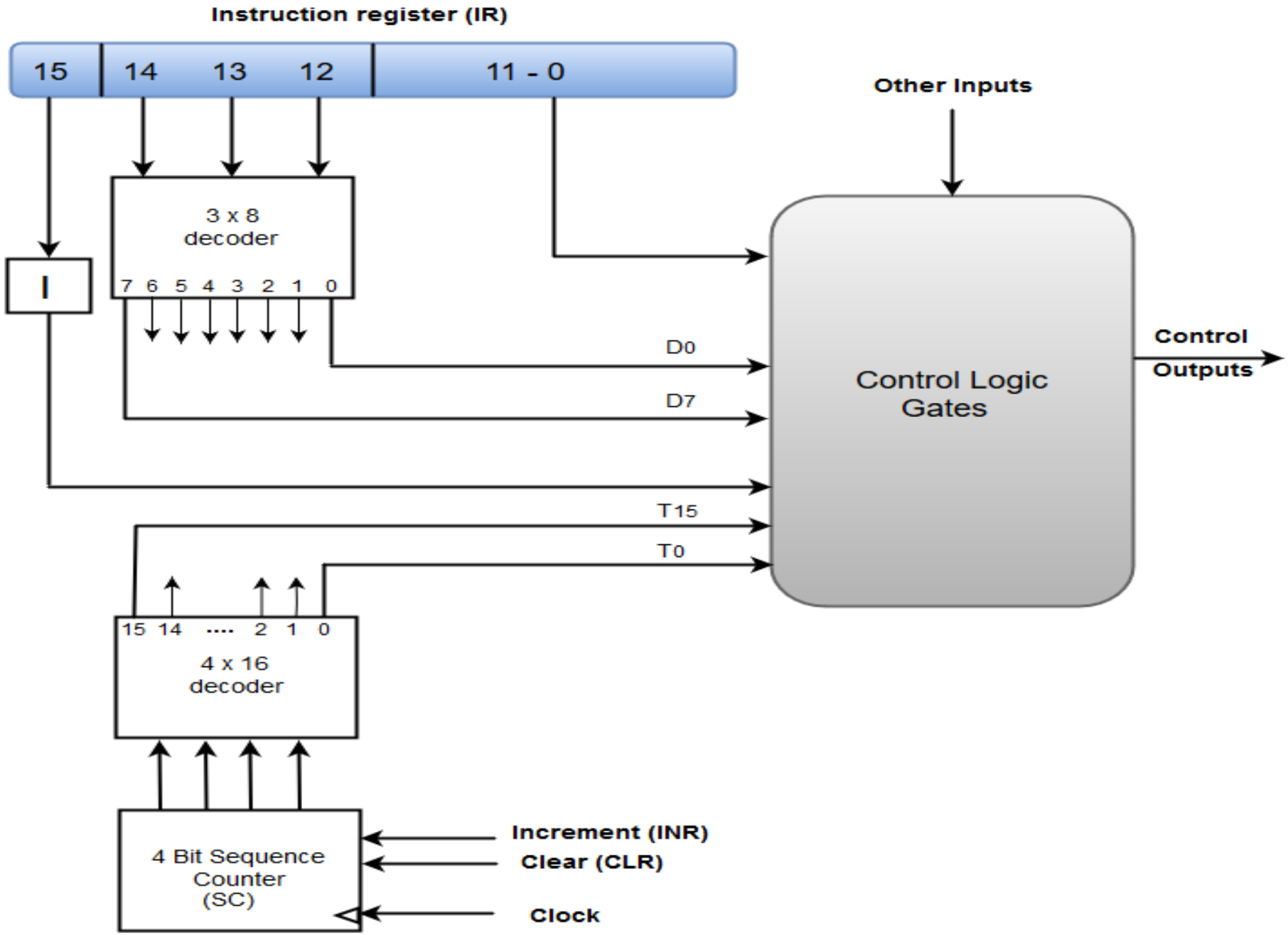
Symbol	Hexadecimal code I = 0	Hexadecimal code I = 1	Description
AND	0xxx	8xxx	AND memory word to AC
ADD	1xxx	9xxx	Add memory word to AC
LDA	2xxx	Axxx	Load memory word to AC
STA	3xxx	Bxxx	Store content of AC in memory
BUN	4xxx	Cxxx	Branch unconditionally
BSA	5xxx	Dxxx	Branch and save return address
ISZ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

Timing and Control

All sequential circuits in the Basic Computer CPU are driven by a master clock,
with the exception of the INPR register.
At each clock pulse, the control unit sends control signals to control inputs of the bus, the registers, and the ALU.
Control unit design and implementation can be done by two general methods:
•A ***hardwired*** control unit is designed from scratch using traditional digital logic design techniques
to produce a minimal, optimized circuit.
In other words, the control unit is like an ASIC (application-specific integrated circuit).

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

Control Unit of a Basic Computer:



Instruction Cycle

- The program residing in the memory unit of the computer consists of a sequence of instructions.
- Each instruction cycle in a basic computer consists of the following phases:

(i)Fetch an instruction from memory.

(ii) Decode the instruction

(iii) Read the effective address from memory if the instruction has an indirect address.

(iv) execute the instruction.

Fetch and Decode

- The sequence counter SC is cleared to 0, providing a decoded timing signal T0.

Initially, the program counter PC is loaded with the address of the first instruction in the program.

- After each clock pulse the SC is incremented by 1, so that it goes through a sequence of clock pluses T0,T1,T2, and so on.
- The microoperations for the fetch and decode phases can be specified by the following register transfer statements.

T0 : AR<- PC

T1: IR <-M[AR],PC=PC+1

T2:D0, ……D7 <-Decode IR(12-14), AR<-IR(0-11), I<-IR(15)

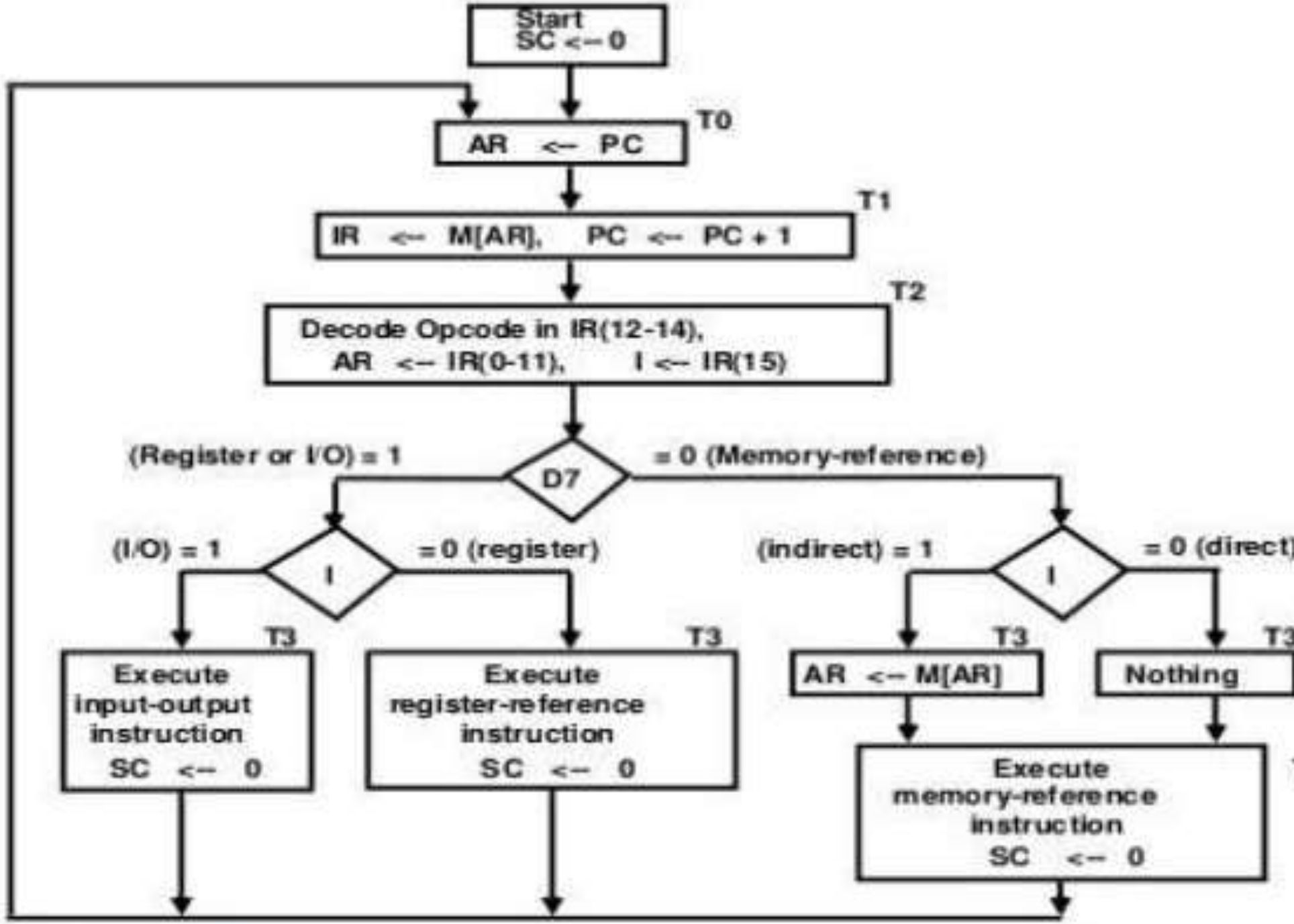
To provide data path for the transfer of PC to AR at timing signal T0 to achieve the following connection:

1. Place the content of PC onto the bus by making the bus selection inputs S1,S2,S0equal to 010.
2. Transfer the content of the bus to AR by enabling the LD input of AR.

The next clock transaction initiates the transfer from PC to AR since T0 =1. in order to implement the second statement.
T1: IR <-M[AR],PC=PC+1

It is necessary to use timing signal T1 to provide the following connection in the bus system.
1. Enable the read input of memory.
2. Place the content of memory unto the bus by making S2S1S0 = 111.
3. TRANSFER the content of the bus to IR by enabling the LD INPUT OF IR
4. Increment PC by enabling the INR input of PC.

THE next clock pulse initiates the read and increment operations since T1 = 1.



Register reference instruction

REGISTER REFERENCE INSTRUCTIONS

Register Reference Instructions are identified when

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

$r = D_7\, I'\, T_3$ => Register Reference Instruction
$B_i = IR(i)$, i=0,1,2,...,11

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

Address Sequencing

- Microinstructions are stored in control memory in groups, with each group specifying a routine.
- Each computer instruction has its own microprogram routine in control memory to generate the microoperations that execute the instruction.
- The 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.
- The next step is to generate the microoperations that execute the instruction fetched from memory.
- Each instruction has its own microprogram routine stored in a given location of control memory.
- The mapping procedure is a rule that transforms the instruction code into a control memory address

The address sequencing capabilities required in a control memory are:

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

•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.

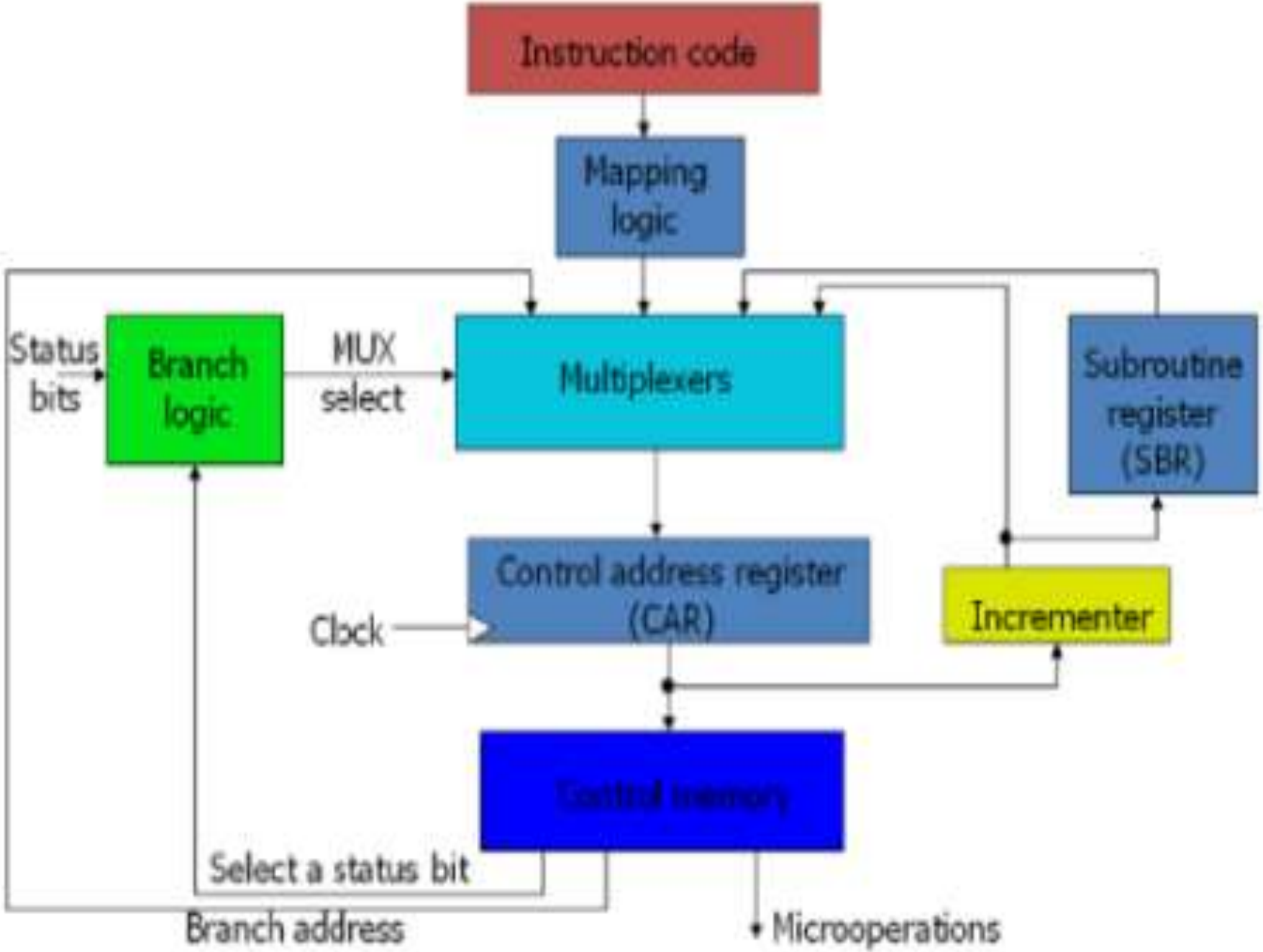


Fig 3-2: Selection of address for control memory

Above figure 3.2 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 3.2 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.

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.

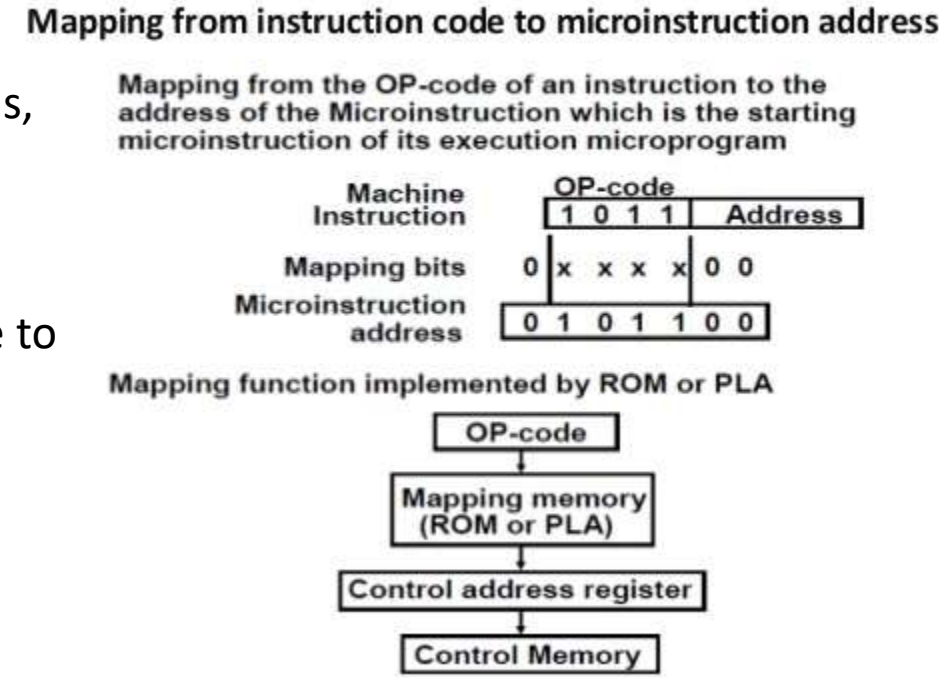
•If the routine needs more than four microinstructions, it can use addresses 1000000 through 1111111.

• 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



•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.
Microprogrammed Control

• 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.

Subroutines:

- subroutines are programs that are used by other routines to accomplish a particular task.

•A subroutine can be called from any point within the main body of the microprogram.
•Many microprograms contain identical sections of code.
•Microinstructions can be saved by employing subroutines that use a common sections of microcode.
•Microprograms that use subroutines must have a provision for storing the return address during a subroutine call and restoring the address during a subroutine return.

CENTRAL PROCESSING UNIT

UNIT - II

General Register Organization:—

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

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

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

$$R_1 \rightarrow R_2 + R_3$$

(1) MUX A selection (SEC A): to place the content of R2 into bus A
(2) MUX B selection (sec B): to place the content of R3 into bus B
(3) ALU operation selection (OPR): to provide the arithmetic addition (A + B)
(4) Decoder destination selection (SEC D): to transfer the content of the output bus into R_1

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

Control Word:
There are 14 binary selection inputs in the units, and their combined value specified a control word. It consists of four fields three fields contain three bits each, and one field has five bits. The three bits of SEL A select a source register for the A input of the ALU. The three bits of SEL B select a source register for the B input of the ALU. The three bit of SEC D select a destination register using the decoder and its seven load outputs. The five bits of OPR select one of the operations in the ALU. The 14-bit control word when applied to the selection inputs specify a particular micro-operation.

Table: Encoding of Register selection fields.

Binary Code	SEL A	SEL B	SEL D
000	Input	Input	None
001	R_1		
010	R_2		
011	R_3	S	S
100	R_4	A	A
101	R_5	M	M
110	R_6	E	E
111	R_7		

Table: Encoding of ALU operation

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

Examples of Micro-operation for the CPU
Symbolic Designation

Micro Operation	SECA	SEC B	SEL D	OPR	Control Word			
$R_1 \rightarrow R_2 - R_3$	R_2	R_3	R_1	SUB	010	011	001	00101
$R_4 \rightarrow R_5$ V R_5	R_4	R	R_4	OR	100	101	100	0101
$R_6 \rightarrow R_6 + 1$	R_6	-	R_6	MCA	110	000	110	00001
$R_7 \rightarrow R_1$	R_1	-	R_7	TSFA	001	000	111	00000
Output $\rightarrow R_2$	R_2	-	None	TSFA	010	000	000	00000
Output $\rightarrow$ Input	Input	-	None	TSFA	000	000	000	00000
$R_4 \rightarrow$ SHL R4	R_4	-	R_4	SHLA	100	000	100	11000
$R_5 \rightarrow 0$	R_5	R_5	R_5	XOR	101	101	101	01100

Stack organization

Stack Organization

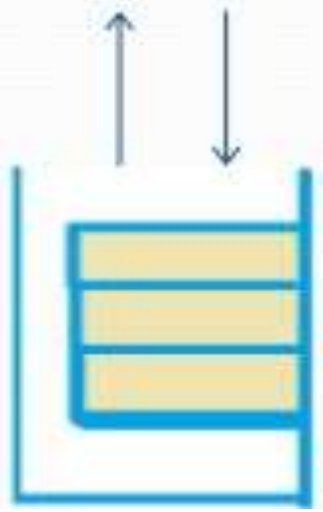
- **Stack**:- A stack storage device that stores information in such a manner that the item stored last is the first item retrieved.

- Also called **last-in first-out(LIFO) list.** Useful for compound arithmetic operations and nested subroutine calls.

Stack Organization

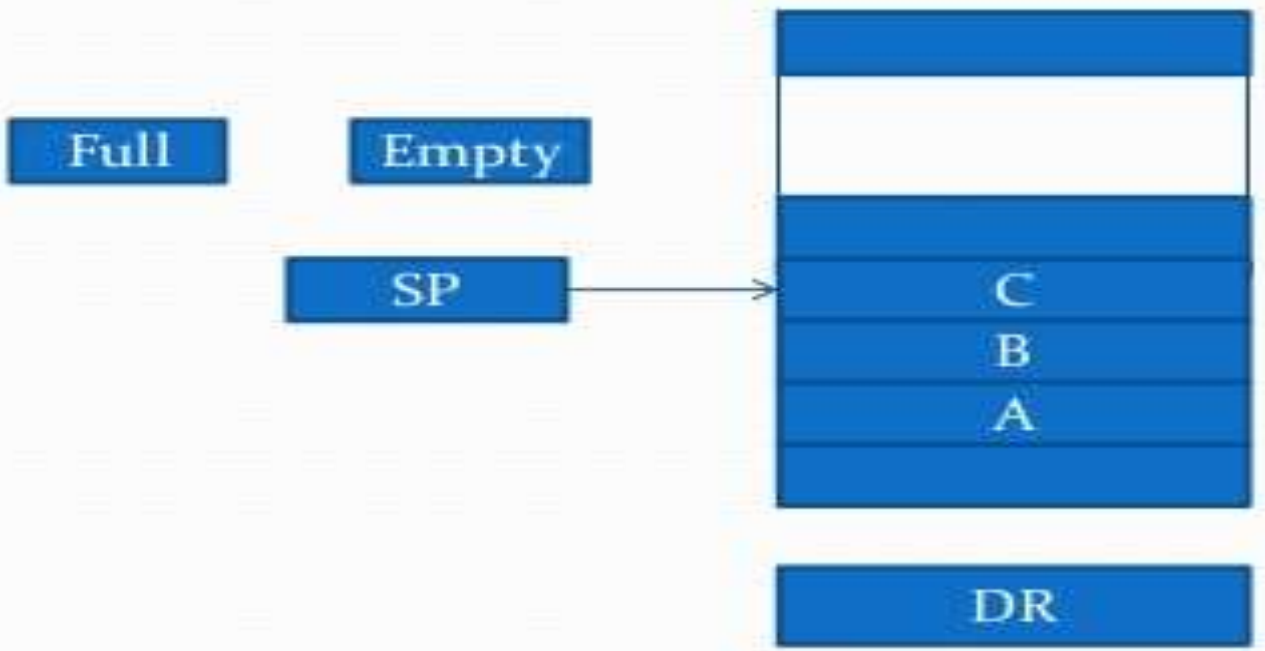
- **Stack Pointer(SP):-** A register that holds the address of the top item in the stack.

  SP always points at the top item in the stack.
- Push:- Operation to insert an item into the stack.
- Pop:- Operation to retrieve an item from the stack.



Register Stack

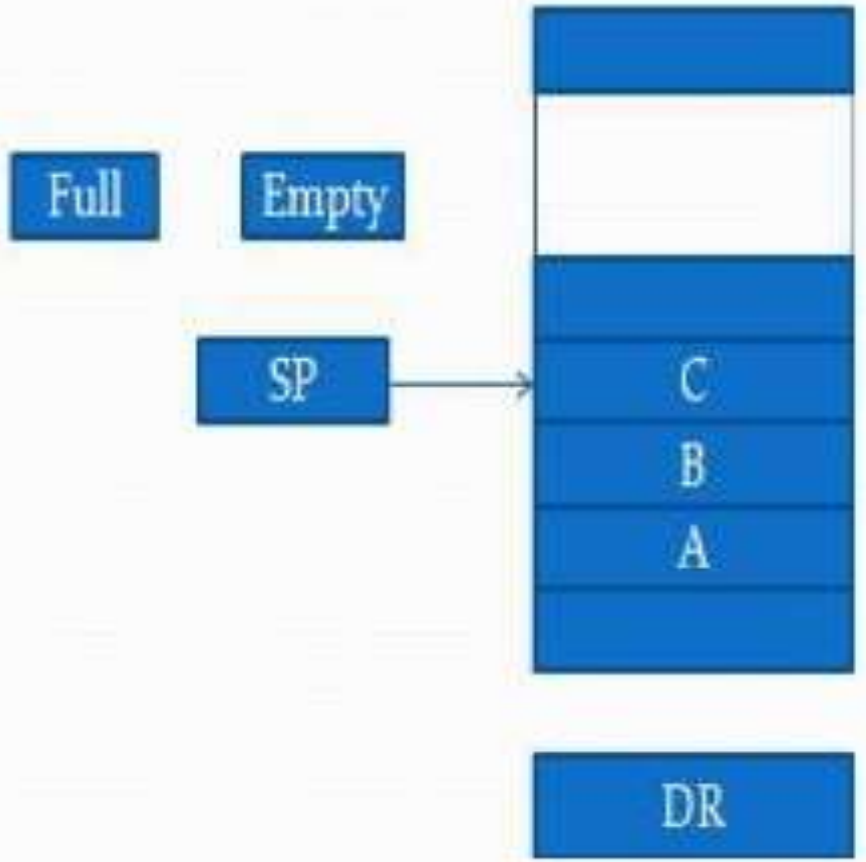
- A stack can be organized as a collection of a finite number of registers.



Register Stack

- In a 64- word stack, the stack pointer contains 6 bits.
- The one-bit register FULL is set to 1 when the stack is full;

  EMPTY register is 1 when the stack is empty.
- The data register DR holds the data to be written into or read from the stack.



Memory Stack

A stack can be implemented in a random access memory (RAM) attached to a CPU. The implementation of a stack in the CPU is done by assigning a portion of memory to a stack operation and using a processor register as a stack pointer. The starting memory location of the stack is specified by the processor register as *stack pointer*.

Stack Operation Reverse Polish Notation (postfix)

- **Reverse polish notation:** is a **postfix notation**(places operators after operands).
- **(Example)**

  Infix notation A + B

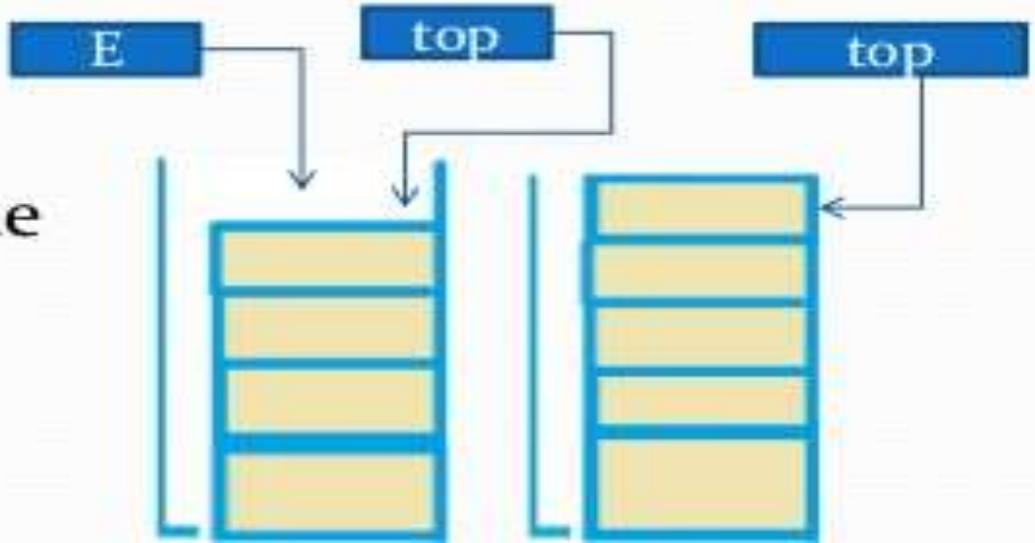
  Reverse Polish notation AB+ also called postfix.

Stack Operation Reverse Polish Notation (postfix)

- A stack organization is very effective for evaluating arithmetic expression.
- **A * B + C * D → (AB*)+(CD*) → AB* CD*+**
- **(3 * 4) + (5 * 6) → 34 * 56 *+**
- first convert the arithmetic expression into the equal to polish notation.
- Push the operands into stack in the order in which they appear.
- Use the 2 top most operation for evaluation.
- The stack is pop and the result often the operation is again push into the stack
- Finally only the result of the operation is left on stack top.

Stack operation

- **Push Operation:-** The process of putting a new data element onto stack is know as a push operation. Push operation involves a serise of step –
  - **Step 1** -> checks if the stack is full.
  - **Step 2**-> If the stack is full, produces an error and exit.
  - **Step 3**-> If the stack is not full, increments top to point next empty space.
  - **Step 4**-> Adds data element to the stack location, where top is pointing.
  - **Step 5**-> Returns success.



Pop Operation

- Accessing the content while removing it from the stack, is known as a pop operation. In an array implementation of pop() operation, the data element is not actually removed, instead top is decremented to a lower position in the actually removes data element and deallocates memory space.

➢ Pop operation may involve the following steps.

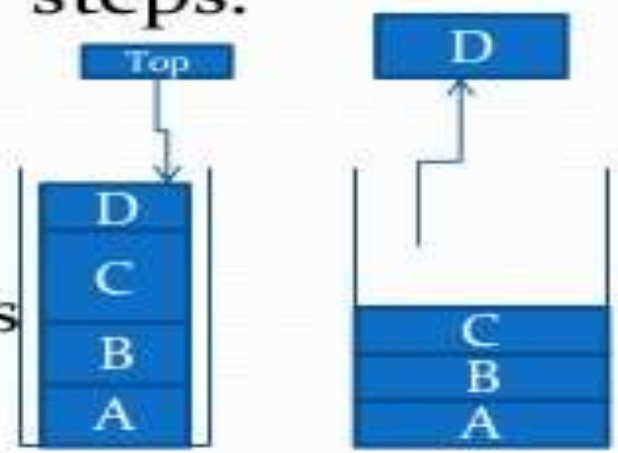
➢ **Step 1** -> checks if the stack is empty.

➢ **Step 2**-> If the stack is empty, produces an error and exit.

➢**Step 3**-> If the stack is not empty, accesses the data element at which top is pointing.

➢**Step 4**-> Decreases the value of top by 1.

➢**Step 5**-> Returns success



Instruction Formats

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

. The bits of the instruction are divided into groups called fields. The most common fields found in instruction formats are:

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

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

Operands residing in processor registers are specified with a register address. A register address is a binary number of k bits that defines one of 2k registers in the CPU. Thus a CPU with 16 processor registers R0 through R15 will have a register address field of four bits. The binary number 0101, for example, will designate register R5.

The three types of CPU organizations:
1 Single accumulator organization.
2 General register organization.
3 Stack organization.

All operations are performed with an implied accumulator register. The instruction format in this type of computer uses one address field. For example, the instruction that specifies an arithmetic addition is defined by an assembly language instruction as ADD X.

Where X is the address of the operand.
The ADD instruction in this case results in the operation AC ← AC + M[X]. AC is the accumulator register and M[X] symbolizes the memory word located at address X.

ADD R1, R2, R3 To denote the operation R1 ← R2 + R3.
The instruction format in this type of computer needs three register address fields.

the instruction ADD R1, R2 Would denote the operation R1 ← R1 + R2. Only register addresses for R1 and R2 need be specified in this instruction.

Computers with multiple processor registers use the move instruction with a mnemonic MOV to symbolize a transfer instruction. Thus the instruction MOV R1, R2 Denotes the transfer R1 ← R2 (or R2 ← R1, depending on the particular computer). Thus transfer-type instructions need two address fields to specify the source and the destination.

Each address field may specify a processor register or a memory word. An instruction symbolized by ADD R1, X Would specify the operation R1 ← R + M [X]. It has two address fields, one for register R1 and the other for the memory address X.

Computers with stack organization would have PUSH and POP instructions which require an address field. Thus the instruction PUSH X

Will push the word at address X to the top of the stack. The stack pointer is updated automatically. Operation-type instructions do not need an address field in stack-organized computers. This is because the operation is performed on the two items that are on top of the stack.

The instruction

ADD

In a stack computer consists of an operation code only with no address field. This operation has the effect of popping the two top numbers from the stack, adding the numbers, and pushing the sum into the stack. There is no need to specify operands with an address field since all operands are implied to be in the stack.

To illustrate the influence of the number of addresses on computer programs, we will evaluate the arithmetic statement X = (A + B) ∗ (C + D). . Using zero, one, two, or three address instruction.

We will use the symbols ADD, SUB, MUL, and DIV for the four arithmetic operations; MOV for the transfer-type operation; and LOAD and STORE for transfers to and from memory and AC register

THREE-ADDRESS INSTRUCTIONS

Computers with three-address instruction formats can use each address field to specify either a processor register or a memory operand.
The program in assembly language that evaluates X = (A + B) ∗ (C + D) is shown below, together with comments that explain the register transfer operation of each instruction.

ADD R1, A, B
R1 ← M [A] + M [B]
ADD R2, C, D
R2 ← M [C] + M [D]
MUL X, R1, R2
M [X] ← R1 ∗ R2

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

TWO-ADDRESS INSTRUCTIONS

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

The program to evaluate X = (A + B) ∗ (C + D) is as follows:

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

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

ONE-ADDRESS INSTRUCTIONS

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

The program to evaluate X = (A + B) ∗ (C + D) is

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

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

ZERO-ADDRESS INSTRUCTIONS

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

X = (A + B) ∗ (C + D) will be written for a stack organized computer.

(TOS stands for top of stack)

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

To evaluate arithmetic expressions in a stack computer, it is necessary to convert the expression into reverse Polish notation. The name “zero-address” is given to this type of computer because of the absence of an address field in the computational instructions

Addressing Modes

- The operation field of an instruction specifies the operation to be performed.
- This operation must be executed on some data stored in computer registers or memory words.
- The way the operands are chosen during program execution in dependent on the addressing mode of the instruction.

The instruction cycle is divided into three major phases:

1. Fetch the instruction from memory
2. Decode the instruction.
3. Execute the instruction.

- .. There is one register in the computer called the program counter (PC) that keeps track of the instructions in the program stored in memory.
- PC holds the address of the instruction to be executed next and is incremented each time an instruction is fetched from memory.

The operation code specifies the operation to be performed.
The mode field is sued to locate the operands needed for the operation.
An address field, it may designate a memory address or a processor register.

The instruction may have more than one address field, and each address field may be associated with its own particular addressing mode.

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

1 Implied Mode:

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

Figure 1: Instruction format with mode field

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

2 Immediate Mode:

In this mode the operand is specified in the instruction itself. The operand field contains the actual operand to be used in conjunction with the operation specified in the instruction. Immediate-mode instructions are useful for initializing registers to a constant value.

3 Register Mode:

In this mode the operands are in registers that reside within the CPU.The particular register is selected from a register field in the instruction. A 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. In other words, the selected register contains the address of the operand rather than the Op code Mode Address operand itself. 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. A reference to the register is then equivalent to specifying a memory address. The advantage of a register indirect mode instruction is that the address field of the instruction sues fewer bits to select a register than would have been required to specify a memory address directly.

5 Auto increment or Auto decrement Mode:

This is similar to the register indirect mode except 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.

8 Relative Address Mode:

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

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 index register is a special CPU register that contains an index value. The address field of the instruction defines the beginning address of a data array in memory. Each operand in the array is stored in memory relative to the beginning address. The distance between the beginning address and the address of the operand is the index value stores in the index register. Any operand in the array can be accessed with the same instruction provided that the index register contains the correct index value. The index register can be incremented to facilitate access to consecutive operands. Note that if an index-type instruction does not include an address field in its format, the instructionconverts to the register indirect mode of operation. Some computers dedicate one CPU register to function solely as an index register.

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. This is similar to the indexed addressing mode except that the register is now called a base register instead of an index register. The difference between the two modes is in the way they are used rather than in the way that they are computed. An index register is assumed to hold an index number that is relative to the address part of the instruction. A base register is assumed to hold a base address and the address field of the instruction gives a displacement relative to this base address.

Numerical Example

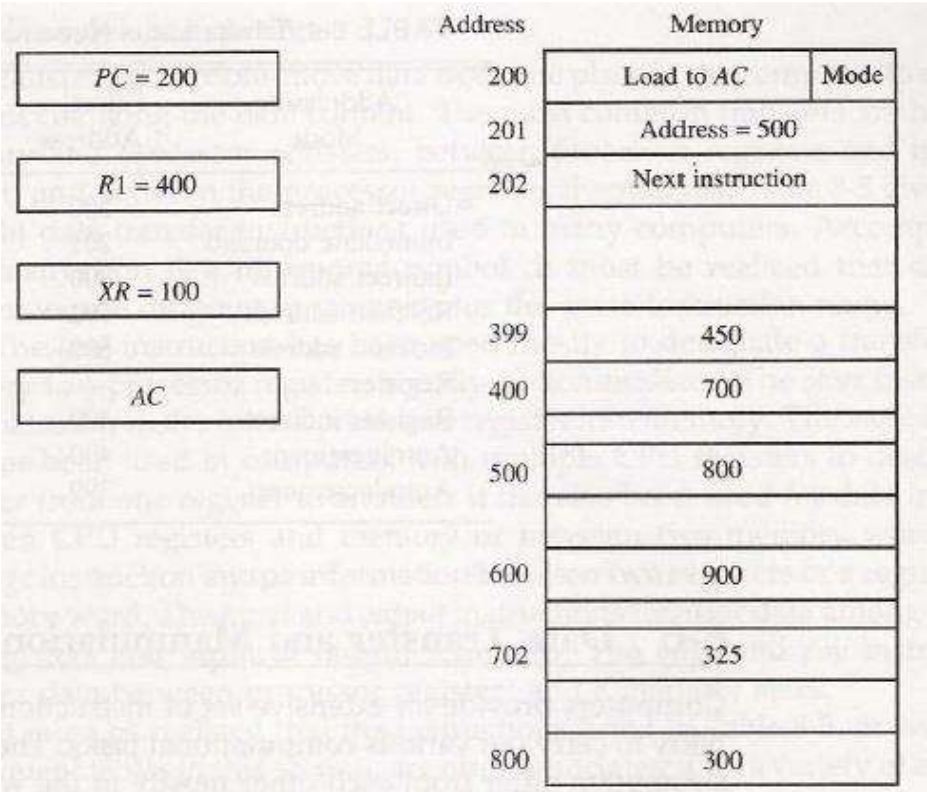


Figure 8-7 Numerical example for addressing modes.

TABLE 8-4 Tabular List of Numerical Example

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

Data Transfer and Manipulation

Data manipulation instructions perform operations on **data** and provide the computational capabilities for the computer. These instructions perform arithmetic, logic and shift operations. A subroutine call instruction consists of an operation code together with an address that specifies the beginning of the subroutine.

Most computer instructions can be classified into three categories:

2)Data transfer,

2) Data manipulation,

3) Program control instructions

Data Transfer Instruction

- 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.

Data Manipulation Instruction

- Data Manipulation Instructions perform operations on data and provide the computational capabilities for the computer.
- It is divided into three basic types:
  1) Arithmetic,
  2) Logical and bit manipulation,
  3) Shift Instruction

Arithmetic Insructions

- The four basic arithmetic operations are addition,subtraction, multiplication,and division.

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

Logical and bit manipulation Insructions

- Logical instructions perform binary operations on strings of bits stored in registers.
- They are useful for manipulating indvidual bits or a group of bits that represent binary-coded information.

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

Program control

Program control instructions specify conditions for altering the content of the program counter , while data transfer and manipulation instructions specify condtions for data-processing operations.

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

Program Interrupt - 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. - The interrupt procedure is, in principle, quite similar to a subroutine call except for three Variations: (1) The interrupt is usually initiated by an internal or external signal rather than from the Execution of an instruction (except for software interrupt as explained later);
(2) The address of the interrupt service program is determined by the hardware rather than from the address field of an instruction.
(3) An interrupt procedure usually stores all the information –

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 - Program status word the collection of all status bit conditions in the CPU is sometimes called a program status word or PSW.

The PSW is stored in a separate hardware register and contains the status information that characterizes the state of the CPU.

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

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

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. –

A software interrupt is initiated by executing an instruction. 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.

COMPUTER ARITHMETIC

UNIT - III

COMPUTER ARITHMETIC

1. Addition and Subtraction of Unsigned Numbers

The direct method of subtraction taught in elementary schools uses the borrowconcept. In this method we borrow a 1 from a higher significant position when theminuend digit is smaller than the corresponding subtrahend digit. This seems to beeasiest when people perform subtraction with paper and pencil. When subtraction isimplemented with digital hardware, this method is found to be less efficient than themethod that uses complements.

The subtraction of two ***n*** digit unsigned numbers ***M – N (N ≠ 0)*** in base ***r*** can bedone as follows:

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

Consider, for example, the subtraction 72532 – 13250 = 59282. The 10'scomplement of 13250 is 86750. Therefore:M = 72532. 10's complement of N = +86750. Sum = 159282Discard end carry, and the Answer = 59282

Since we are dealing with unsigned numbers, there is really no way to get anunsigned result for the second example. When working with paper and pencil, werecognize that the answer must be changed to a signed negative number. Whensubtracting with complements, the negative answer is recognized by the absence ofthe end carry and the complemented result. Subtraction with complements is donewith binary numbers in a similar manner using the same procedure outlined above.Using the two binary numbers X = 1010100 and Y = 1000011, we perform thesubtraction X – Y and Y – X using 2's complements: X = 1010100. 2's complement of Y = +0111101, Sum = 10010001, Discard end carry 2^7 = -10000000. Answer: X – Y = 0010001

Y = 1000011
2's complement of X = +0101100
Sum = 1101111

There is no end carry. Answer is negative 0010001 = 2's complement of 1101111.

***Addition and Subtraction* With Signed –Magnitude Data:**

We designate the magnitude of the two numbers by A and B. Where the signed numbers are added or subtracted, we find that there are eight different conditions to consider, depending on the sign of the numbers and the operation performed.

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

Table: Addition and Subtraction of Signed-Magnitude Numbers

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

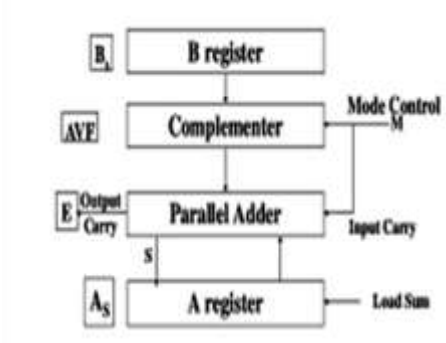


Figure: Hardware Architecture for Addition and Subtraction of Signed-Magnitude Numbers

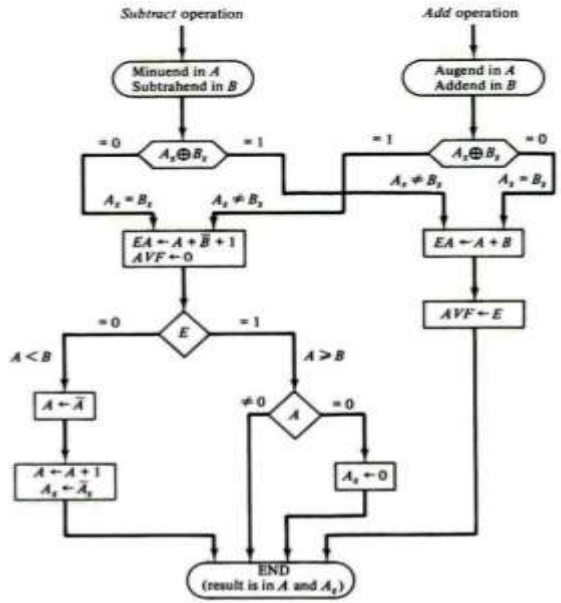


Figure: Flowchart

2. Multiplication

With Signed –2's Complement Data:

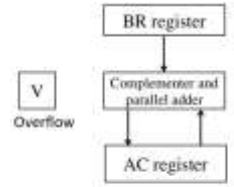


Figure: Hardware for signed-2's complement addition and subtraction.

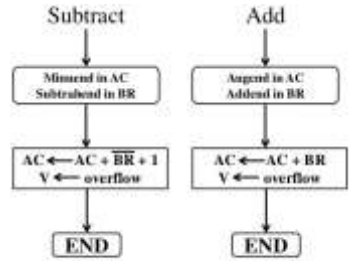


Figure: Algorithm for adding and subtracting numbers in signed-2's complement representation.

Signed-Magnitude complements:

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

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

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

When multiplication is implemented in a digital computer, we change the process slightly. Here, instead of providing registers to store and add simultaneously as many binary numbers as there are bits in the multiplier, it is convenient to provide an adder for the summation of only two binary numbers, and successively accumulate the partial products in a register. Second, instead of shifting the multiplicand to left, the partial product is shifted to the right, which results in leaving

the partial product and the multiplicand in the required relative positions. Now, when the corresponding bit of the multiplier is 0, there is no need to add all zeros to the partial product since it will not alter its value.

The hardware for multiplication consists of the equipment given in below Figure.

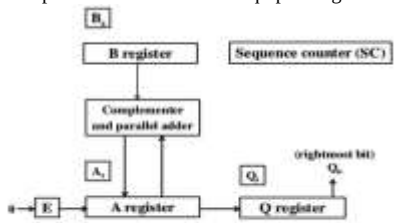


Figure: Hardware for multiply operation

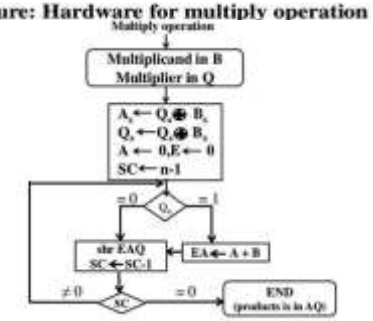


Figure: Flowchart for multiply operation.

Multiplicand B = 10111	E	A	Q	SC
Multiplier in Q	0	00000	10011	101
Q_n = 1; add B		10111		
First partial product	0	10111		
Shift right EAQ	0	01011	11001	100
Q_n = 1; add B		10111		
Second partial product	1	00010		
Shift right EAQ	1	00001	01100	011
Q_n = 0; shift right EAQ	0	01000	10110	010
Q_n = 0; shift right EAQ	0	00100	01011	001
Qn = 1; add B		10111		
Fifth partial product	0	11011		
Shift right EAQ	0	11011		

Final product in AQ = 0110110101.

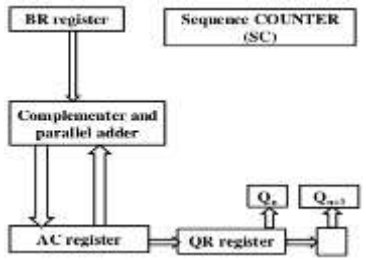
Table: Numerical Example for Binary Multiplier

Booth's Multiplication Algorithm (signed-2's complement):

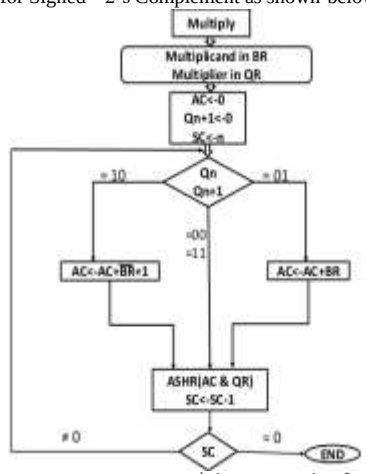
If the numbers are represented in signed 2's complement then we can multiply them by using Booth algorithm. In fact the strings of 0's in the multiplier need no addition but just

shifting, and a string of l's in the multiplier from bit weight 2^k to weight 2^m can be treated as $2^{k+1} - 2^m$. For example, the binary number 001111 (+15) has a string of 1's from 2^3 to 2^0(k = 3, m = 0).

The hardware architecture for Signed – 2's Complement as shown below



The Flowchart for Signed – 2's Complement as shown below



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

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

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

The algorithm applies to both positive and negative multipliers in 2's complement representation. This is because a negative multiplier ends with a string of l's and the last operation will be a subtraction of the appropriate weight. For example, a multiplier equal to -14 is represented in 2's complement as 110010 and is treated as $-2^4 + 2^2 - 2^1 = -14$.

A numerical example of Booth algorithm is given in Table for n = 5. It gives the multiplication of (-9) x (-13) = +117.

	BR = 10111 BR+1 = 01001		AC	QR	Qn+1	SC
1 0	Initial	00000	10011	0	101	
	Subtract BR	01001				
		01001				
	ashr	00100	11001	1	100	
1 1	ashr	00010	01100	1	011	
0 1	Add BR		10111			
		11001				
	ashr	11100	10110	0	010	
0 0	ashr	11110	01011	0	001	
1 0	Subtract BR	01001				
		00111				
ashr		00011	10101	1	000	

Table : Example of Multiplication with Booth Algorithm

3. Division Algorithms

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

Division:	11010	Quotient = Q
B = 10001	0111000000	Dividend = A
	01110	5 bits of A < B, quotient has 5 Bits
	011100	6 bits of A VVB
	- 10001	Shift right B and subtract; enter 1 in Q
	- 010110	
	- -10001	
	- - 001010	7 bits of remainder V B
	- - - 010100	Shift right B and subtract; enter 1 in Q
	- - - -10001	
	- - - -000110	Remainder < B; enter 0 in Q;
	- - - - -00110	shift right B; Remainder V B
		Shift right B and subtract; enter 1 in Q
		Remainder < B; enter 0 in Q; final remainder

Figure : Example of Binary Division

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

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

Hardware Implementation for Signed-Magnitude Data

In hardware implementation for signed-magnitude data in a digital computer, it is convenient to change the process slightly. Subtraction is achieved by adding A to the 2's complement of B. End carry gives the information about the relative magnitudes.

Register EAQ is now shifted to the left with 0 inserted into Qn and the previous value of E is lost. The example is given in Figure to clear the proposed division process. The divisor is stored in the B register and the double-length dividend is stored in registers A and Q. The dividend is shifted to the left and the divisor is subtracted by adding its 2's complement value. E keeps the information about the relative magnitude. A quotient bit 1 is inserted into Qn and the partial remainder is shifted to the left to repeat the process when E = 1. If E = 0, it signifies that A < B so the quotient in Qn remains a 0 (inserted during the shift).

Divisor $\bar{B} + 1 = 01111$

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

Figure: Example of Binary Division with Digital Hardware

Below is a flowchart of the hardware multiplication algorithm. In the beginning, the multiplicand is in B and the multiplier in Q. Their corresponding signs are in Bs and Qs respectively. We compare the signs of both A and Q and set to corresponding sign of the product since a double-length product will be stored in registers A and Q. Registers A and E are cleared and the sequence counter SC is set to the number of bits of the multiplier. Since an operand must

be stored with its sign, one bit of the word will be occupied by the sign and the magnitude will consist of n-1 bits.

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

The hardware divide algorithm is given in Figure. A and Q contain the dividend and B has the divisor. The sign of the result is transferred into Q. A constant is set into the sequence counter SC to specify the number of bits in the quotient. As in multiplication, we assume that operands are transferred to registers from a memory unit that has words of n bits. Since an operand must be stored with its sign, one bit of the word will be occupied by the sign and the magnitude will haven-1 bit.

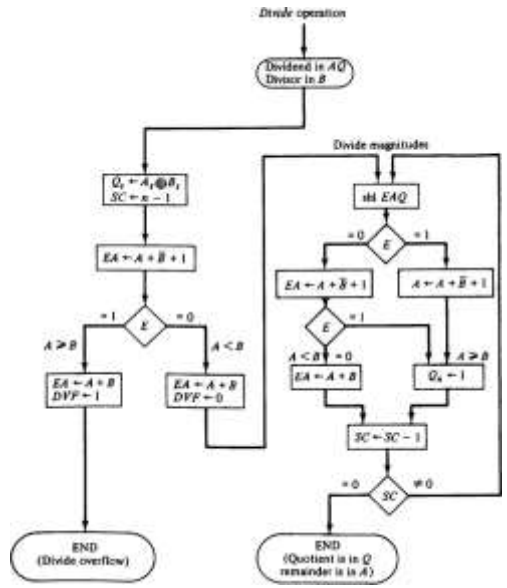


Fig: Signed- Magnitude Division

An overflow may occur in the division operation, which may be easy to handle if we are using paper and pencil but is not easy when using hardware. This is because the length of registers is finite and will not hold a number that exceeds the standard length. To see this, let us consider a system that has 5-bit registers. We use one register to hold the divisor and two registers to hold the dividend. From the example of Figure, the quotient will consist of six bits if the five most significant bits of the dividend constitute a number greater than the divisor. The quotient is to be stored in a standard 5-bit register, so the overflow bit will require one more flip-flop for storing the sixth bit. This divide-overflow condition must be avoided in normal computer operations because the entire quotient will be too long for transfer into a memory unit that has words of standard length, that is, the same as the length of registers. Provisions to ensure that this condition is detected must be included in either the hardware or the software of the computer, or in a combination of the two.

When the dividend is twice as long as the divisor, we can understand the condition for overflow as follows:

A divide-overflow occurs if the high-order half bits of the dividend makes a number greater than or equal to the divisor. Another problem associated with division is the fact that a division by zero must be avoided. The divide-overflow condition takes care of this condition as well. This occurs because any dividend will be greater than or equal to a divisor, which is equal to zero. Overflow condition is usually detected when a special flip-flop is set. We will call it a divide-overflow flip-flop and label it DVF.

Signed-2's Complement Division Algorithm as

1. M <- Divisor, A:Q <- dividend sign extended to 2n bits; for example 0111 -> 00000111 ; 1001-> 11111001
   (note that 0111 = 7 and 1001 = -3)
2. Shift A:Q left 1 bit
3. If M and A have same signs, perform A <- A-M otherwise perform A <- A + M
4. The preceding operation succeeds if the sign of A is unchanged
   - If successful, or (A==0 and Q==0) set Q_0 <- 1
   - If not successful, and (A!=0 or Q!=0) set Q_0 <- 0 and restore the previous value of A
5. Repeat steps 2,3,4 for n bit positions in Q
6. Remainder is in A. If the signs of the divisor and dividend were the same then the quotient is in Q, otherwise the correct quotient is 0-Q

Examples:

A	Q	M = 0011
0000	0111	Initial value
0000	1110	shift
1101		subtract
0000	1110	restore
0001	1100	shift
1110		subtract
0001	1100	restore
0011	1000	shift
0000		subtract
0000	1001	set $Q_0 = 1$
0001	0010	shift
1110		subtract
0001	0010	restore

(a) (7)/(3)

A	Q	M = 1101
0000	0111	Initial value
0000	1110	shift
1101		add
0000	1110	restore
0001	1100	shift
1110		add
0001	1100	restore
0011	1000	shift
0000		add
0000	1001	set $Q_0 = 1$
0001	0010	shift
1110		add
0001	0010	restore

(b) (7)/(−3)

A	Q	M = 0011
1111	1001	Initial value
1111	0010	shift
0010		add
1111	0010	restore
1110	0100	shift
0001		add
1110	0100	restore
1100	1000	shift
1111		add
1111	1001	set $Q_0 = 1$
1111	0010	shift
0010		add
1111	0010	restore

(c) (−7)/(3)

A	Q	M = 1101
1111	1001	Initial value
1111	0010	shift
0010		subtract
1111	0010	restore
1110	0100	shift
0001		subtract
1110	0100	restore
1100	1000	shift
1111		subtract
1111	1001	set $Q_0 = 1$
1111	0010	shift
0010		subtract
1111	0010	restore

(d) (−7)(−3)

4. Floating-point Arithmetic operations

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

Basic Considerations: There are two part of a floating-point number in a computer - a mantissa m and an exponent e. The two parts represent a number generated from multiplying m times a radix r raised to the value of e. Thus $\boldsymbol{m \times r^e}$

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

$.53725 \times 10^3$

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

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

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

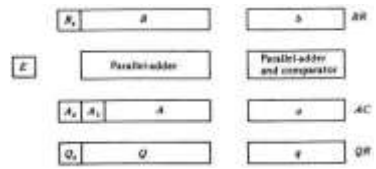


Figure: Registers for Floating Point arithmetic operations

The AC has a mantissa whose sign is in As, and a magnitude that is in A. The diagram shows the most significant bit of A, labelled by A1. The bit in his position must be a 1 to normalize

the number. Note that the symbol AC represents the entire register, that is, the concatenation of As, A and a.

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

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

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

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

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

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

For adding or subtracting two floating-point binary numbers, if BR is equal to zero, the operation is stopped, with the value in the AC being the result. If AC = 0, we transfer the content of BR into AC and also complement its sign we have to subtract the numbers. If neither number is equal it to zero, we proceed to align the mantissas.

The addition and subtraction of the two mantissas is similar to the fixed-point addition and subtraction algorithm presented in Fig. The magnitude part is added or subtracted depends on the operation and the signs of the two mantissas. If an overflow occurs when the magnitudes are added, it is transferred into flip-flop E. If E = 1, the bit is transferred into A1 and all other bits of A are shifted right. The exponent must be incremented so that it can maintain the correct number.

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

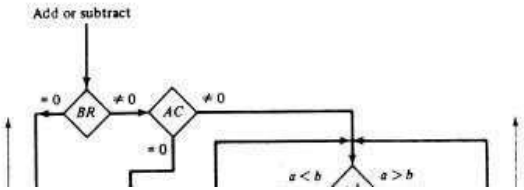


Figure: Addition and Subtraction of floating –point numbers

Multiplication of Floating Point Numbers:

The procedure for multiplication is divided in to below things:

1. Check for zeros.
2. Add the exponents.
3. Multiply the mantissas
4. Normalize the result

The procedure is as showed in below flowcharts.

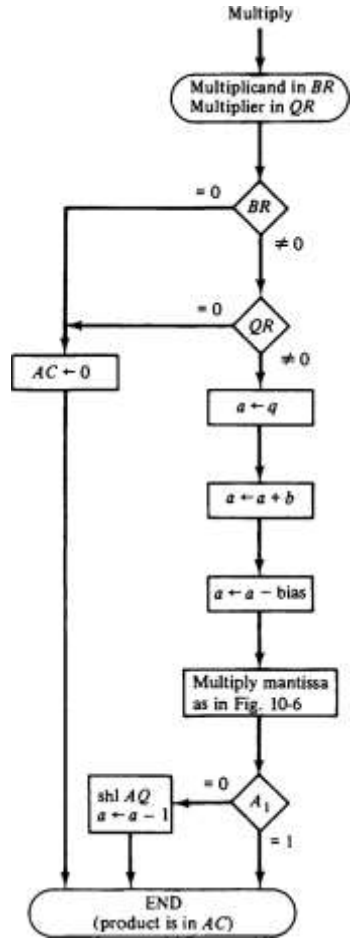


Fig: Flowchart for floating-point multiplication

Division of Floating Point Numbers:

The procedure for division is divided in to below things:

1. Check for zeros.
2. Subtract the exponents.
3. Divide the mantissas
4. Normalize the result

The procedure is as showed in below flowcharts.

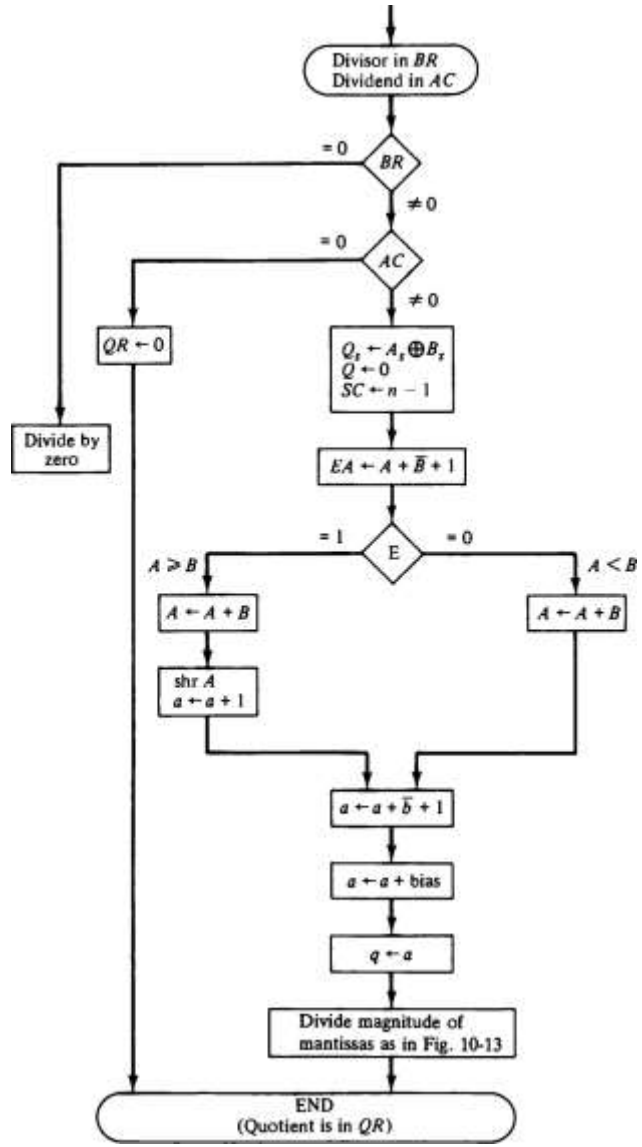


Fig: Flowchart for floating-point division

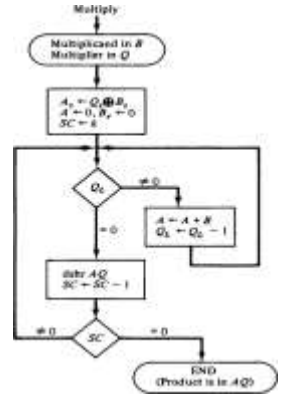


Fig: Flowchart Decimal Arithmetic Multiplication

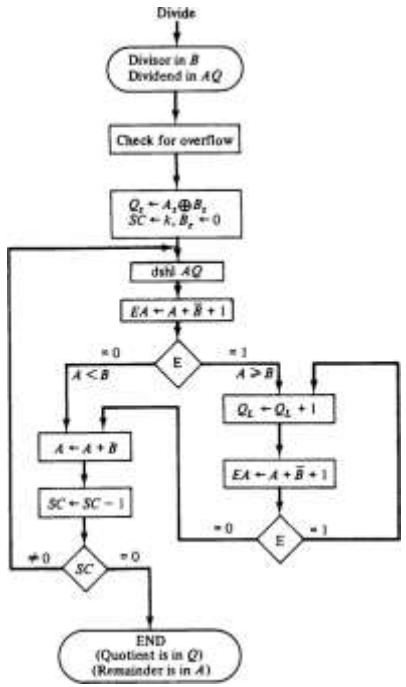


Fig: Flowchart Decimal Arithmetic Division

INPUT OUTPUT ORGANIZATION

UNIT 4

Overview

- **Peripheral Devices**
- **Input-Output Interface**
- **Asynchronous Data Transfer**
- **Modes of Transfer**
- **Priority Interrupt**
- **Direct Memory Access**
- **Input-Output Processor**
- **Serial Communication**

Input Output Organization

- **I/O Subsystem**
  - Provides an efficient mode of communication between the central system and the outside environment

- Programs and data must be entered into computer memory for processing and results obtained from computer must be recorded and displayed to user.

Peripheral Devices

- **Devices that are under direct control of computer are said to be connected on-line.**

- **Input or output devices attached to the computer are also called peripherals.**

- **There are three types of peripherals :**
  - **Input peripherals**
  - **Output peripherals**
  - **Input-output peripherals**

Peripheral (or I/O Device)
Monitor (*Visual Output Device*) : CRT, LCD
KeyBoard (*Input Device*) : light pen, mouse, touch screen, joy stick
Printer (Hard Copy Device) : **Daisy wheel**, **dot matrix** and **laser printer**
Storage Device : Magnetic tape, magnetic disk

Peripheral Devices

Input Devices

- Keyboard
- Optical input devices
  - Card Reader
  - Paper Tape Reader
  - Bar code reader
  - Optical Mark Reader
- Magnetic Input Devices
  - Magnetic Stripe Reader
- Screen Input Devices
  - Touch Screen
  - Light Pen
  - Mouse
- •

Output Devices

- Card Puncher, Paper Tape Puncher
- CRT
- Printer (Daisy Wheel, Dot Matrix, Laser)
- Plotter

I/O Interface

- **Provides a method for transferring information between internal storage (such as memory and CPU registers) and external I/O devices.**
- **They are special hardware components between CPU and peripheral to supervise and synchronize all input and output transfer.**
- **They are called interface units because they interface between the processor bus processor bus and the peripheral device.**
- **Resolves the *differences* between the computer and peripheral devices**

  (1) Peripherals – Electromechanical or Electromagnetic Devices

  CPU or Memory - Electronic Device

  – **Conversion of signal values required**

  (2). Data Transfer Rate

  - **Peripherals - Usually slower**
  - **CPU or Memory - Usually faster than peripherals**
    – **Some kinds of Synchronization mechanism may be needed**

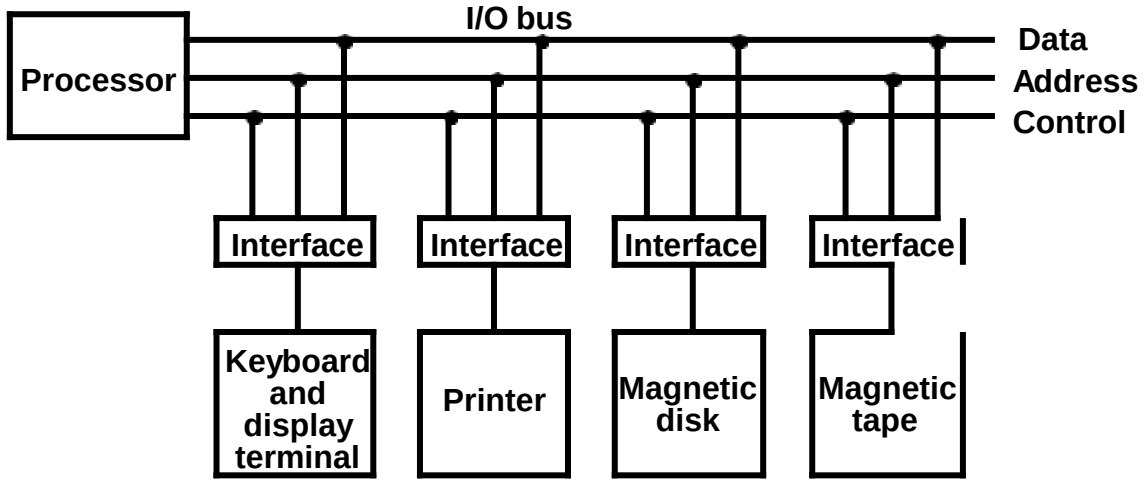
  (3) Data formats or Unit of Information

  - **Peripherals – Byte, Block, …**
  - **CPU or Memory – Word**

  (4) Operating modes of peripherals may differ

  - **must be controlled so that not to disturbed other peripherals connected to CPU**

I/O Bus and Interface



Interface :

- **Decodes the device address (device code)**
- **Decodes the commands (operation)**
- **Provides signals for the peripheral controller**
- **Synchronizes the data flow and supervises the transfer rate between peripheral and CPU or Memory**

4 types of command interface can receive : control, status, data o/p and data i/p

I/O Command is an instruction that is executed in the interface and its attached peripheral units.

- **Control command :** is issued to activate peripheral and to inform what to do
- **Status command :** used to test various status condition in the interface and the peripherals
- **Data o/p command :** causes the interface to respond by transferring data from the bus into one of its registers
- **Data i/p command :** interface receives an item of data from the peripheral and places it in its buffer register.

I/O Bus and Memory Bus

- Functions of Buses

  - *MEMORY BUS* is for information transfers between CPU and the
- *I/O BUS* is for information transfers between CPU and I/O devices through their I/O interface

  - Three ways , bus can communicate with memory and I/O :

(1) **use two separate buses, one to communicate with memory and the other with I/O interfaces**

(2) **Use one common bus for memory and I/O but separate control lines for each**

(3) **Use one common bus for memory and I/O with common control**

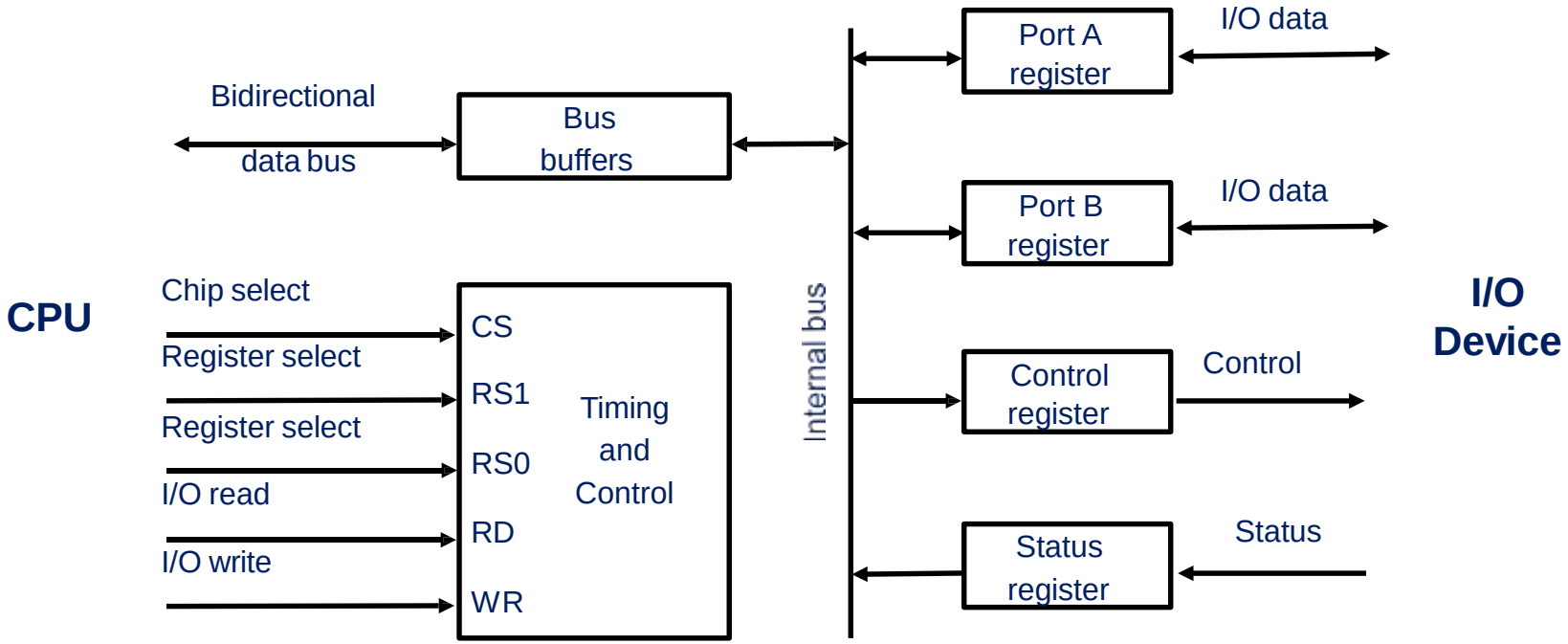
- lines for both

Isolated I/O

- Many computers use common bus to transfer information between memory or I/O.
- Separate I/O read/write control lines in addition to memory read/write control lines
- Separate (isolated) memory and I/O address spaces
- Distinct input and output instructions
  - each associated with address of interface register

Memory-mapped I/O

- A single set of read/write control lines (no distinction between memory and I/O transfer)
- Memory and I/O addresses share the common address space
  -> reduces memory address range available
- No specific input or output instruction
  -> The same memory reference instructions can be used for I/O transfers
- Considerable flexibility in handling I/O operations



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

ASYNCHRONOUS DATA TRANSFER

- In a computer system, CPU and an I/O interface are designed independently of each other.
- When internal timing in each unit is independent from the other and when registers in interface and registers of CPU uses its own private clock.
- In that case the two units are said to be asynchronous to each other. CPU and I/O device must coordinate for data transfers.

METHODS USED IN ASYNCHRONOUS DATA TRANSFER

- Strobe Control: This is one way of transfer i.e. by means of strobe pulse supplied by one of the units to indicate to the other unit when the transfer has to occur.

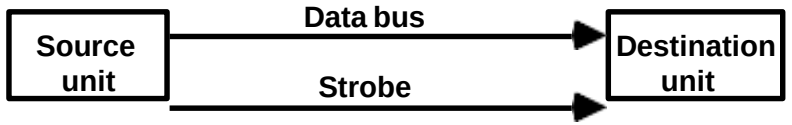
- Handshaking: This method is used to accompany each data item being transferred with a control signal that indicates the presence of data in the bus. The unit receiving the data item responds with another control signal to acknowledge receipt of the data.

STROBE CONTROL

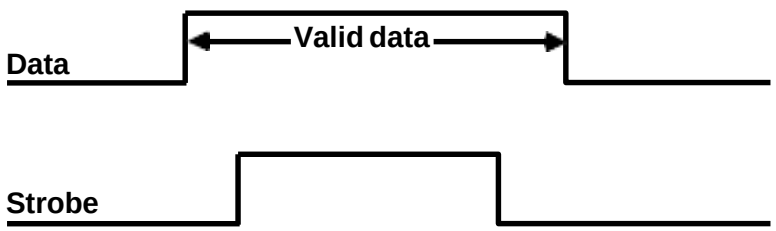
* **Employs a single control line to time each transfer**
* **The strobe may be activated by either the source or the destination unit**

Source-Initiated Strobe for Data Transfer

Block Diagram

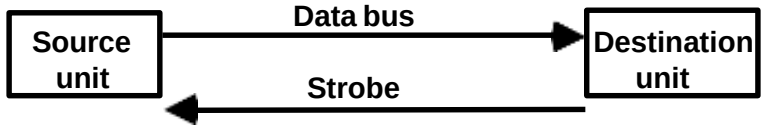


Timing Diagram

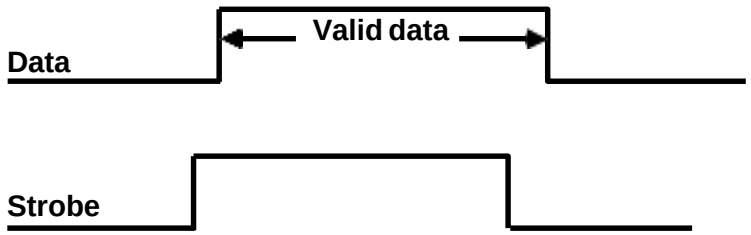


Destination-Initiated Strobe for Data Transfer

Block Diagram



Timing Diagram



HANDSHAKING

Problems in Strobe Methods

Source-Initiated

The source unit that initiates the transfer has no way of knowing whether the destination unit has actually received data

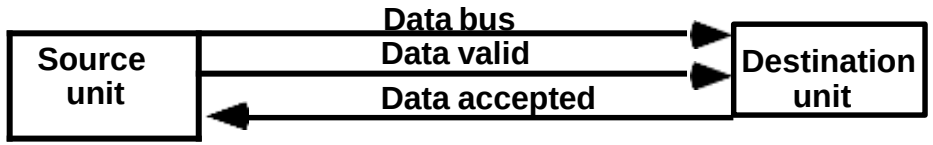
Destination-Initiated

The destination unit that initiates the transfer no way of knowing whether the source has actually placed the data on the bus

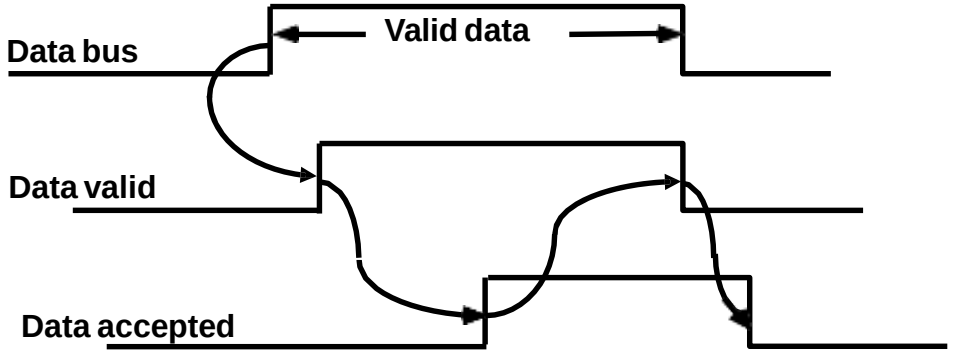
To solve this problem, the ***HANDSHAKE*** **method** introduces a second control signal to provide a *Reply* to the unit that initiates the transfer

SOURCE-INITIATED TRANSFER USING HANDSHAKE

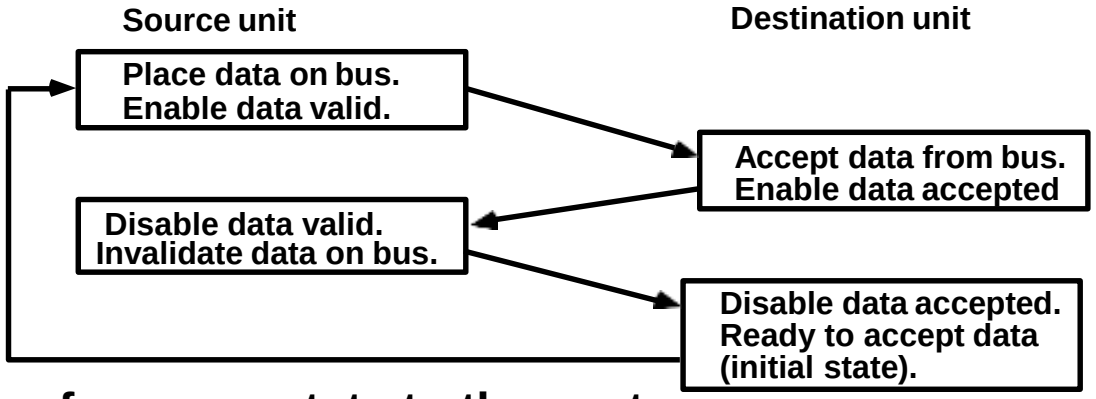
Block Diagram



Timing Diagram



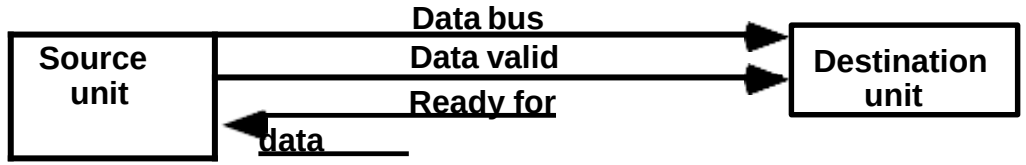
Sequence of Events



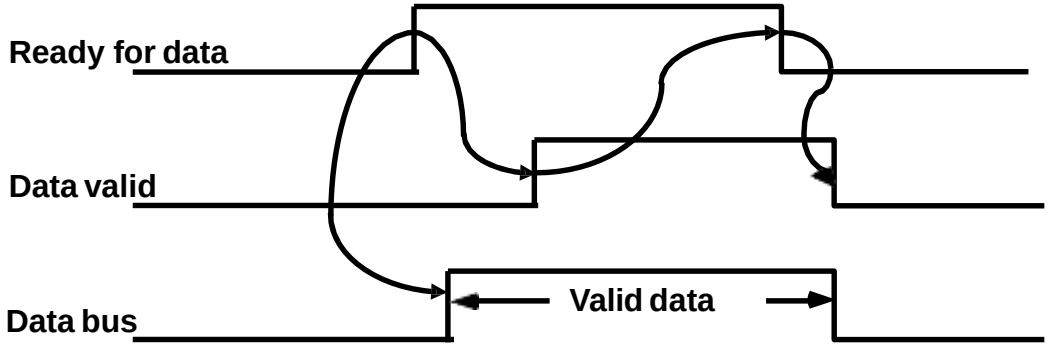
* Allows arbitrary delays from one state to the next
* Permits each unit to respond at its own data transfer rate
* The rate of transfer is determined by the slower unit

DESTINATION-INITIATED TRANSFER USING HANDSHAKE

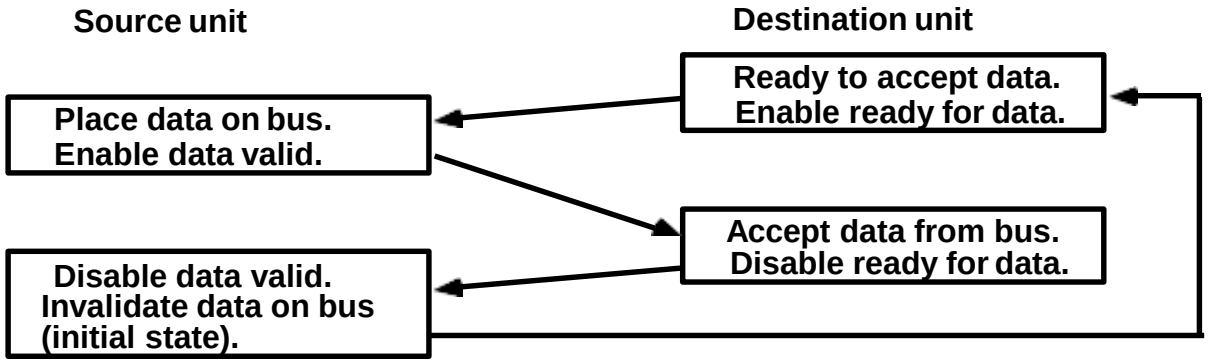
Block Diagram



Timing Diagram



Sequence of Events



* Handshaking provides a high degree of flexibility and reliability because the successful completion of a data transfer relies on active participation by both units
* If one unit is faulty, data transfer will not be completed
  -> Can be detected by means of a *timeout* mechanism

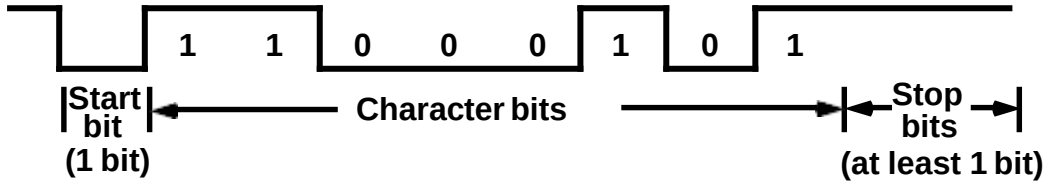
ASYNCHRONOUS SERIAL TRANSFER

Four Different Types of Transfer

Asynchronous serial transfer
Synchronous serial transfer
Asynchronous parallel transfer
Synchronous parallel transfer

Asynchronous Serial Transfer

- **Employs special bits which are inserted at both ends of the character code**
- **Each character consists of three parts; Start bit; Data bits; Stop bits.**



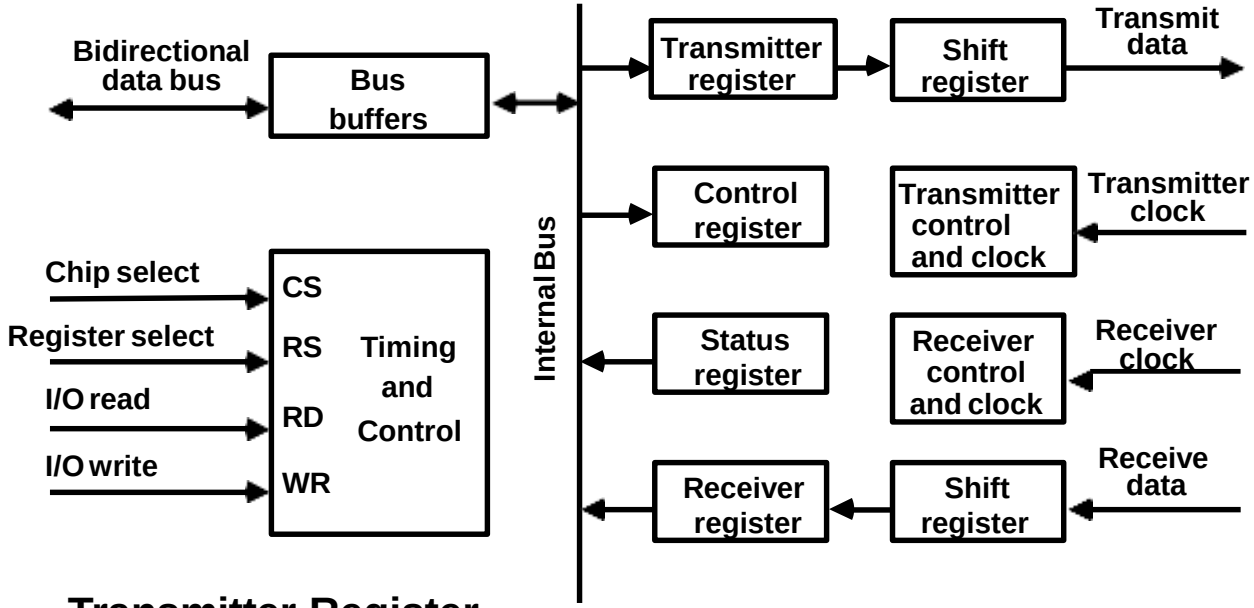
A character can be detected by the receiver from the knowledge of 4 rules;

- **When data are not being sent, the line is kept in the 1-state (idle state)**
- **The initiation of a character transmission is detected by a *Start Bit* , which is always a 0**
- **The character bits always follow the *Start Bit***
- **After the last character , a *Stop Bit* is detected when the line returns to the 1-state for at least 1 bit time**

The receiver knows in advance the transfer rate of the bits and the number of information bits to expect

UNIVERSAL ASYNCHRONOUS RECEIVER-TRANSMITTER - UART -

A typical asynchronous communication interface available as an IC



CS	RS	Oper.	Register selected
0	x	x	None
1	0	WR	Transmitter register
1	1	WR	Control register
1	0	RD	Receiver register
1	1	RD	Status register

Transmitter Register

- **Accepts a data byte(from CPU) through the data bus**
- **Transferred to a shift register for serial transmission**

Receiver

- **Receives serial information into another shift register**
- **Complete data byte is sent to the receiver register**

Status Register Bits

- **Used for I/O flags and for recording errors**

Control Register Bits

- **Define baud rate, no. of bits in each character, whether to generate and check parity, and no. of stop bits**

Modes of transfer (11.4)

- Data transfer between the central computer and the I/O devices may be handled in a variety of modes.

- The modes of transfer are:

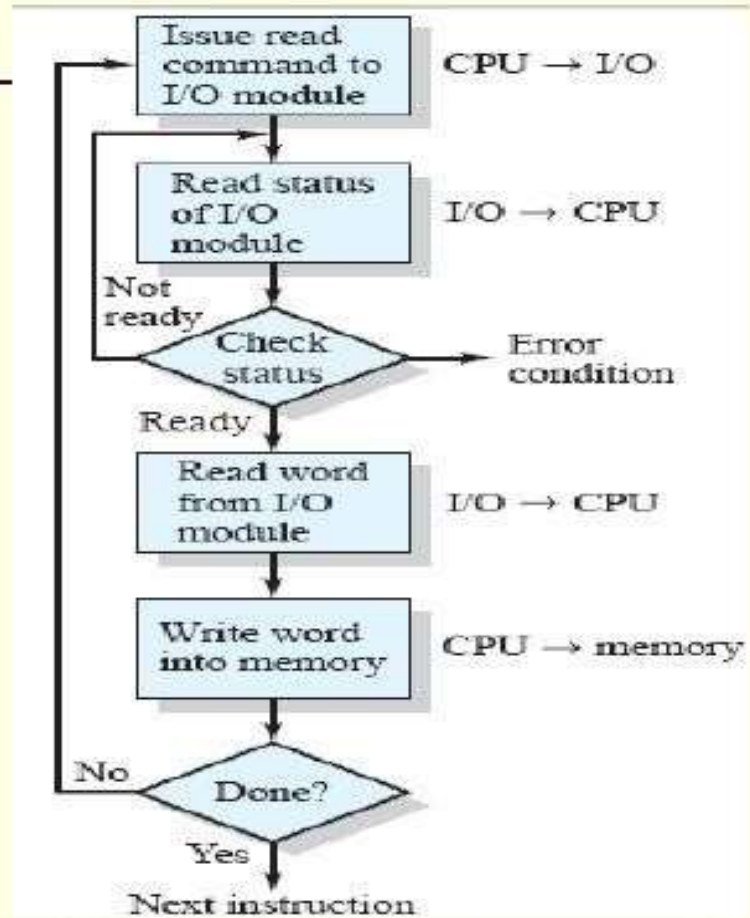
4. Programmed I/O.
5. Interrupt-initiated I/O
6. Direct memory access (DMA)

Programmed I/O

- Programmed I/O operations are the result of I/O instructions written in the computer program.
- Each data item transfer is initiated by an instruction in the program.
- Usually, the transfer is to and from CPU register and peripheral.
- Other instructions are needed to transfer the data to and from CPU and memory.
- Transferring data under program control requires constant monitoring of the peripheral by the CPU.
- Once a data transfer is initiated, the CPU is required to monitor the interface to see when a transfer can again be made.
- It is up to the programmed instructions executed in the CPU to keep close tabs on everything that is taking place in the interface unit and the I/O device.
- In this method, the 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.

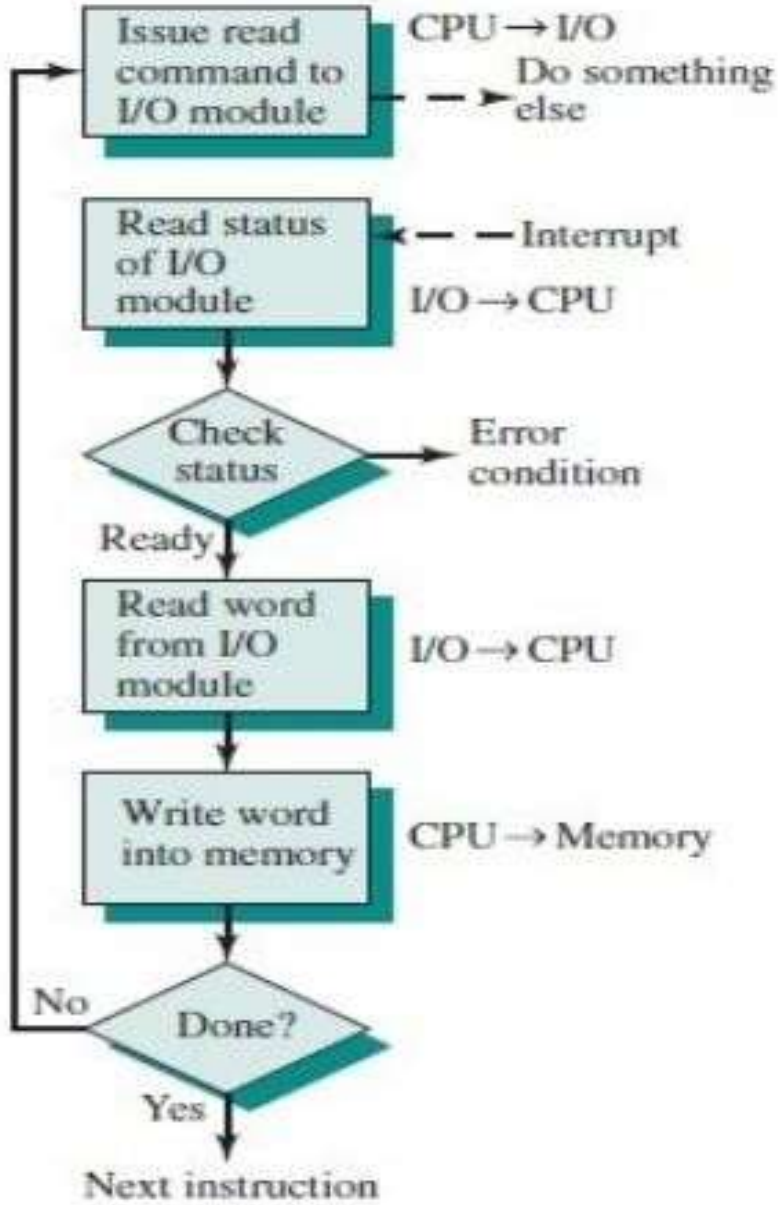
Programmed I/O

- CPU while executing a program encounters an I/O instruction
- CPU issues I/O command to I/O module
- I/O module performs the requested action & set status registers
- CPU is responsible to check status registers periodically to see if I/O operation is complete. **SO**
- No Interrupt to alert the processor



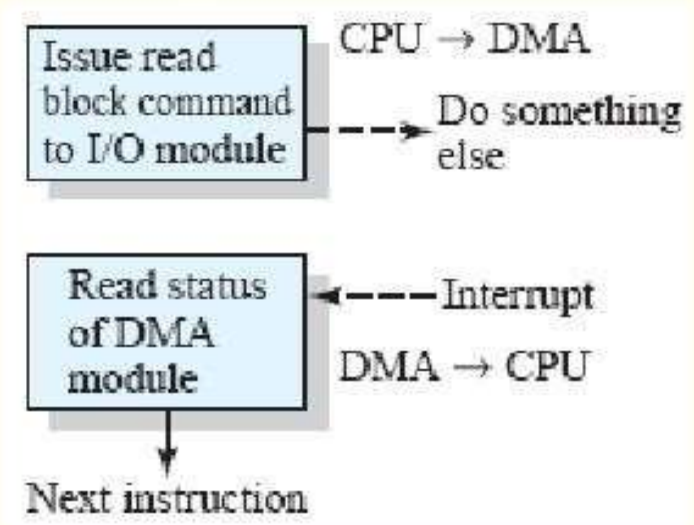
Interrupt Initiated IO

- In the programmed I/O method, the 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 interrupt facility and special commands to inform the interface to issue an interrupt request signal when the data are available from the device.
- In the mean-time the CPU can proceed to execute another program.
- The interface meanwhile keeps monitoring the device.
- When the interface determines that the device is ready for the data transfer, it generates an interrupt request to the computer.
- Upon detecting the external interrupt signal, the CPU momentarily stops the task it is processing, branches to a service program to process the I/O transfer, and then returns to the task it was originally performing.


Issue read command to I/O module
CPU → I/O
Do something else
Read status of I/O module
Interrupt
I/O → CPU
Check status
Error condition
Ready
Read word from I/O module
I/O → CPU
Write word into memory
CPU → Memory
No
Done?
Yes
Next instruction

Direct Memory Access (DMA)

- I/O exchanges occur directly with memory
  - Requires DMA module on system bus
  - Capable of mimicking CPU and taking over control of system from CPU
  - DMA will use bus when
    - Processor does not require it OR
    - Must force processor to suspend operation temporarily– called cycle stealing
- An interrupt is sent when the task is complete
- The processor is only involved at the beginning and end of the transfer



DMA (Direct Memory Access – 11.6)

- Direct memory access is an I/O technique used for high speed data transfer.
- In DMA, the interface transfers data into and out of the memory unit through the memory bus.
- In DMA, the CPU releases the control of the buses to a device called a DMA controller.
- Removing the CPU from the path and letting the peripheral device manage the memory buses directly would improve the speed of transfer.
- 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, the DMA transfers the data directly into memory.
- The CPU merely delays its memory access operation to allow the direct memory I/O transfer.

Direct Memory Access (DMA)

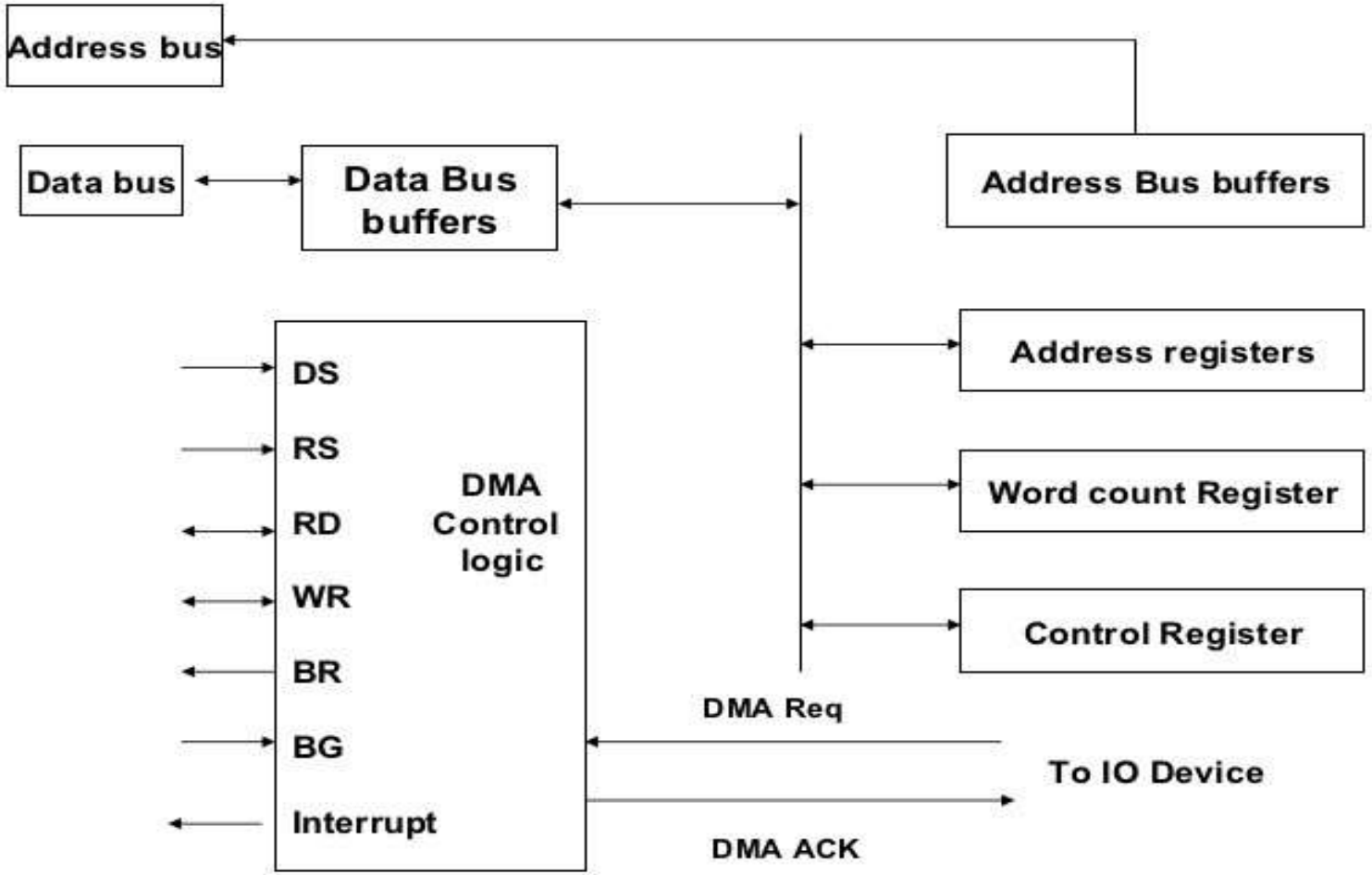
Cycle Stealing

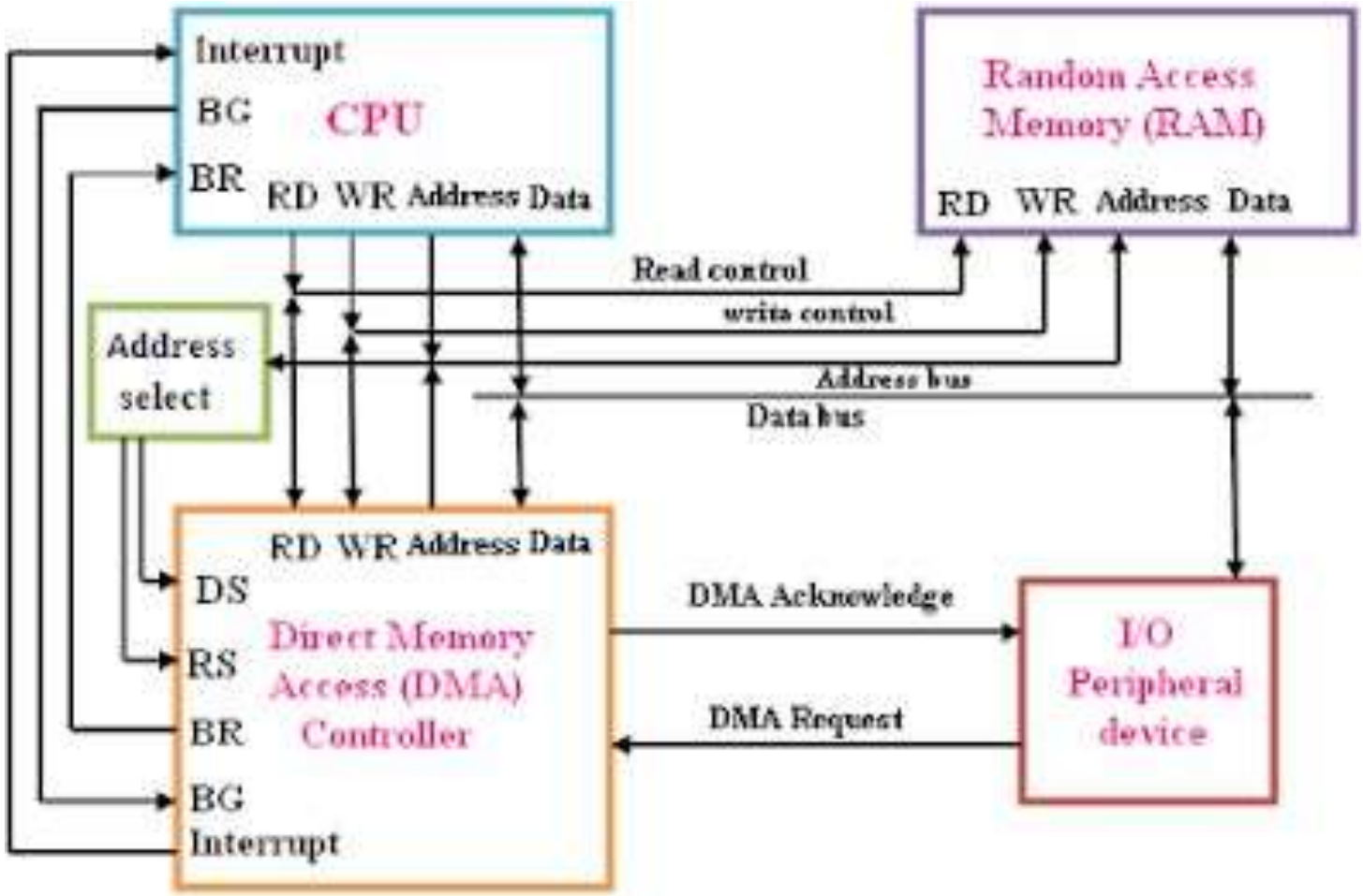
- DMA Controller acquires control of bus
- Transfers a single byte (or word)
- Releases the bus
- The CPU is slowed down due to bus contention

Burst Mode

- DMA Controller acquires control of bus
- Transfers all the data
- Releases the bus
- The CPU operation is temporarily suspended

Block diagram of DMA Controller





Direct Memory Access(DMA)

BURST TRANSFER / CYCLE STEALING

When DMA takes control of the bus system, it communicate directly with The memory. The transfer can be made in several ways.

BURST TRANSFER

In DMA burst transfer, a block sequence consisting of a number of memory word is transferred in a continuous burst. This mode is needed for fast devices.

CYCLE STEALING

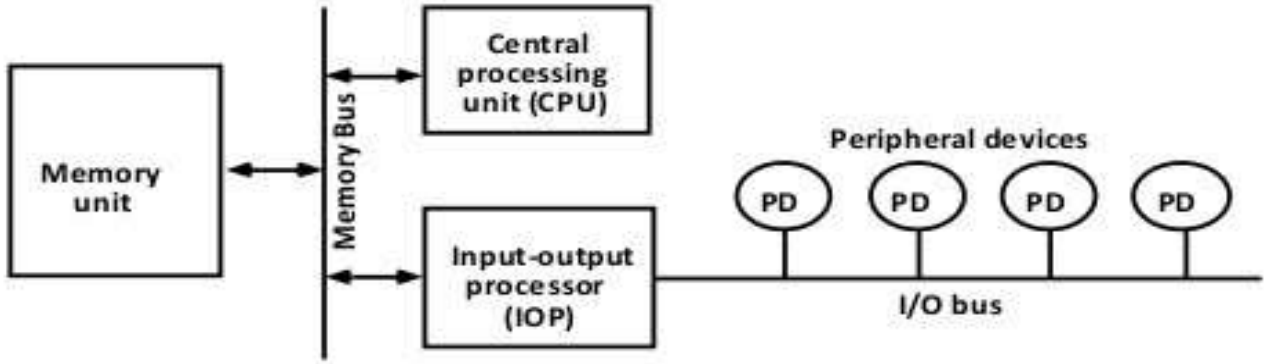
An alternative technique called Cycle Stealing allows the DMA controller to transfer one data word at a time, after which it must return control of the buses to the CPU.

- CPU is usually much faster than I/O(DMA), thus
  CPU uses the most of the memory cycles
- DMA Controller steals the memory cycles from CPU
- For those stolen cycles, CPU remains idle
- DMA Controller may steal most of the memory cycles which may cause CPU remain idle long time

I/O Processor - Channel

Channel

- Processor with direct memory access capability that communicates with I/O devices
- Channel accesses memory by cycle stealing
- Channel can execute a Channel Program
  - Stored in the main memory
  - Consists of Channel Command Word(CCW)
  - Each CCW specifies the parameters needed by the channel to control the I/O devices and perform data transfer operations
- CPU initiates the channel by executing an channel I/O class instruction and once initiated, channel operates independently of the CPU



I/O Processor

- Many computers combines the interface logic with the requirements for direct memory access into one unit and call it an I/O processor. The IOP can handle many peripherals through a DMA and interrupt facility. The computer is divided into three separate modules in such a system.

✓ Memory unit

✓ CPU

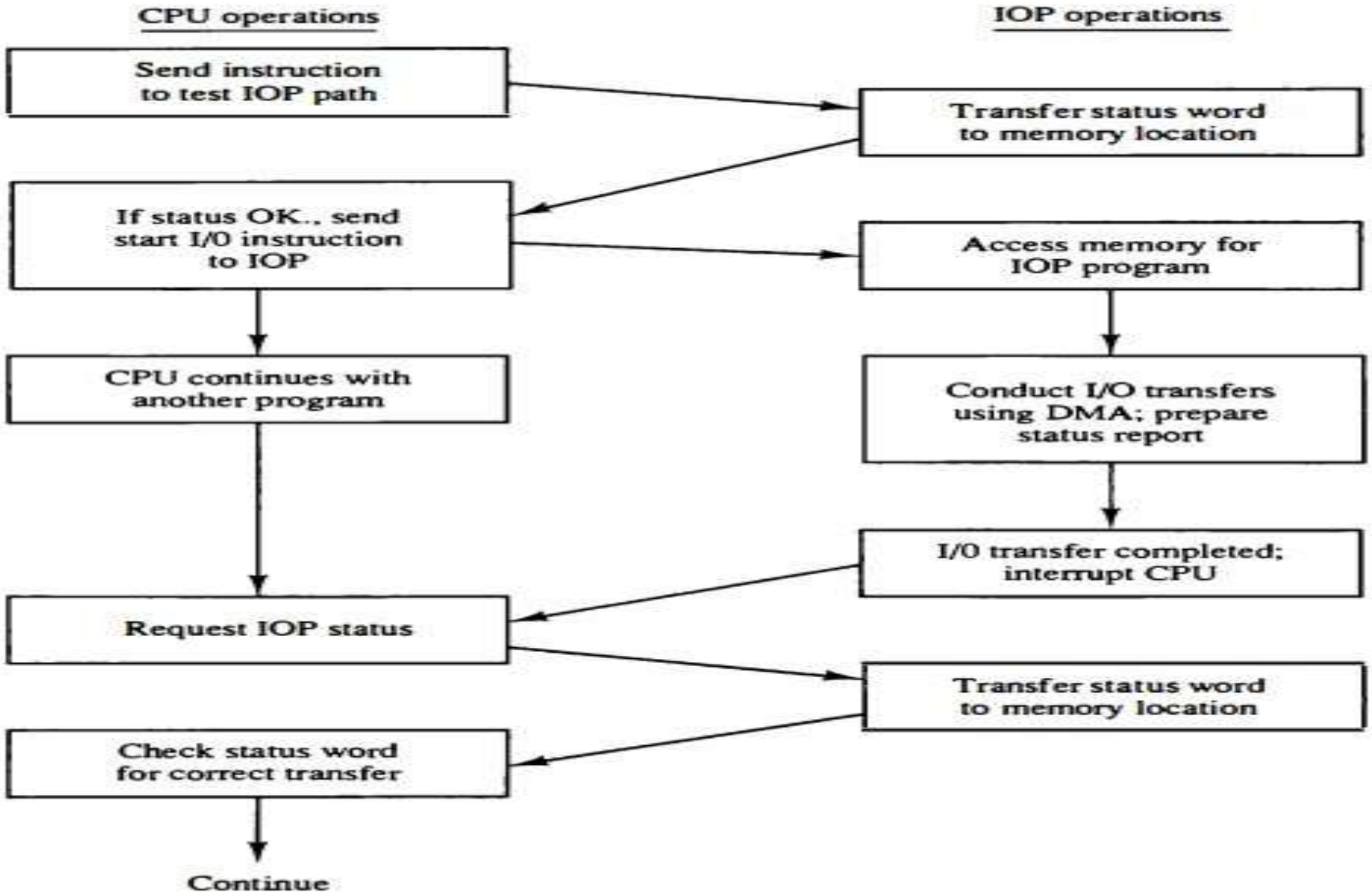
✓ IOP

- CPU is the master while the IOP is a slave processor. The CPU performs the tasks of initiating all operations.
- The operations include

✓ Starting an I/O transfer

✓ Testing I/O status conditions needed for making decisions on various I/O activities.

Figure 20 CPU-IOP communication.



Memory Organization

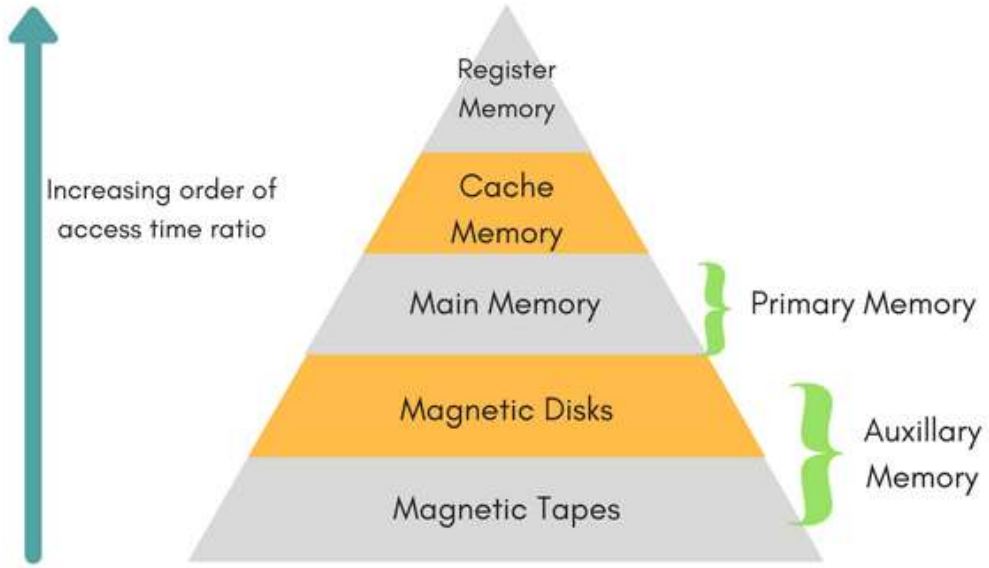
Unit -V

Memory Organization in Computer Architecture

A memory unit is the collection of storage units or devices together. The memory unit stores the binary information in the form of bits. Generally, memory/storage is classified into 2 categories:

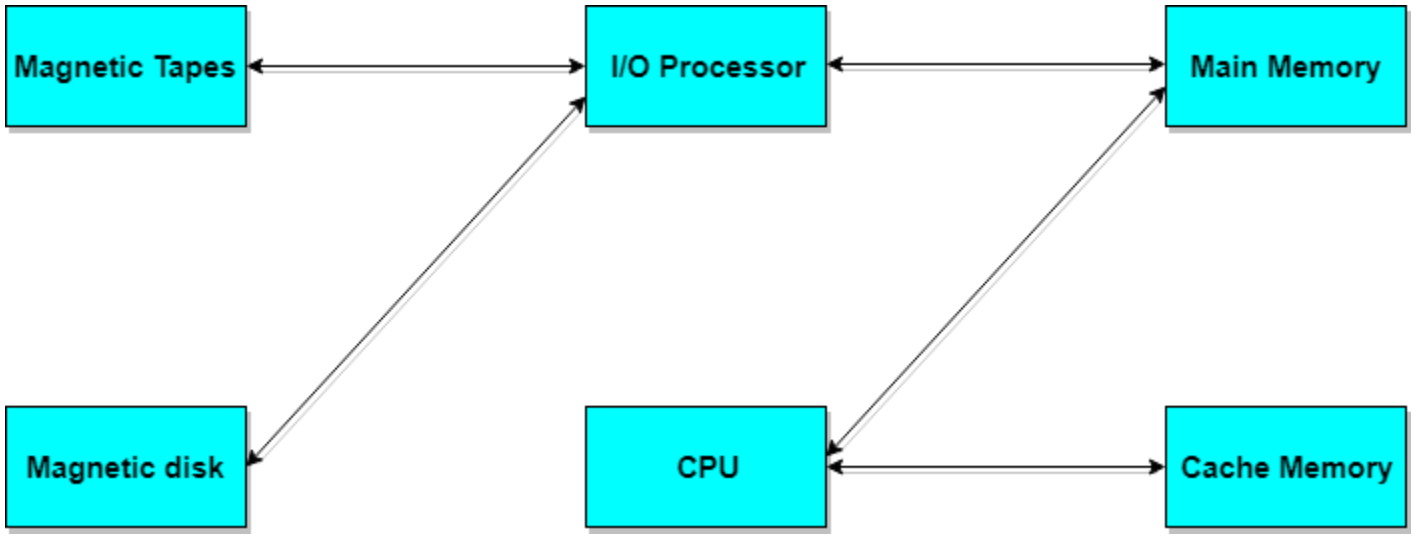
- **Volatile Memory**: This loses its data, when power is switched off.
- **Non-Volatile Memory**: This is a permanent storage and does not lose any data when power is switched off.

Memory Hierarchy



The total memory capacity of a computer can be visualized by hierarchy of components.
The memory hierarchy system consists of all storage devices contained in a computer system from the slow Auxiliary Memory to fast Main Memory and to smaller Cache memory.

Auxillary memory access time is generally **1000 times** that of the main memory, hence it is at the bottom of the hierarchy.
The **main memory** occupies the central position because it is equipped to communicate directly with the CPU and with auxiliary memory devices through Input/output processor (I/O).
When the program not residing in main memory is needed by the CPU, they are brought in from auxiliary memory.
Programs not currently needed in main memory are transferred into auxiliary memory to provide space in main memory for other programs that are currently in use.
The **cache memory** is used to store program data which is currently being executed in the CPU.
Approximate access time ratio between cache memory and main memory is about **1 to 7~10**



Memory Access Methods

Each memory type, is a collection of numerous memory locations. To access data from any memory, first it must be located and then the data is read from the memory location. Following are the methods to access information from memory locations:

1.Random Access: Main memories are random access memories, in which each memory location has a unique address. Using this unique address any memory location can be reached in the same amount of time in any order.

2.Sequential Access: This methods allows memory access in a sequence or in order.

3.Direct Access: In this mode, information is stored in tracks, with each track having a separate read/write head.

Main Memory

The memory unit that communicates directly within the CPU, Auxillary memory and Cache memory, is called main memory. It is the central storage unit of the computer system. It is a large and fast memory used to store data during computer operations. Main memory is made up of **RAM** and **ROM**, with RAM integrated circuit chips holing the major share.

•RAM: Random Access Memory

- **DRAM**: Dynamic RAM, is made of capacitors and transistors, and must be refreshed every 10~100 ms. It is slower and cheaper than SRAM.
- **SRAM**: Static RAM, has a six transistor circuit in each cell and retains data, until powered off.
- **NVRAM**: Non-Volatile RAM, retains its data, even when turned off. Example: Flash memory.

•ROM: Read Only Memory, is non-volatile and is more like a permanent storage for information. It also stores the **bootstrap loader** program, to load and start the operating system when computer is turned on. **PROM**(Programmable ROM), **EPROM**(Erasable PROM) and **EEPROM**(Electrically Erasable PROM) are some commonly used ROMs.

Auxiliary Memory

Devices that provide backup storage are called auxiliary memory.

For example: Magnetic disks and tapes are commonly used auxiliary devices.

Other devices used as auxiliary memory are magnetic drums, magnetic bubble memory and optical disks.

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

Cache Memory

The data or contents of the main memory that are used again and again by CPU, are stored in the cache memory so that we can easily access that data in shorter time.

Whenever the CPU needs to access memory, it first checks the cache memory.

If the data is not found in cache memory then the CPU moves onto the main memory.

It also transfers block of recent data into the cache and keeps on deleting the old data in cache to accommodate the new one.

Hit Ratio

The performance of cache memory is measured in terms of a quantity called **hit ratio**.

When the CPU refers to memory and finds the word in cache it is said to produce a **hit**.

If the word is not found in cache, it is in main memory then it counts as a **miss**.

The ratio of the number of hits to the total CPU references to memory is called hit ratio.

Hit Ratio = Hit/(Hit + Miss)

Associative Memory

It is also known as **content addressable memory (CAM)**. It is a memory chip in which each bit position can be compared. In this the content is compared in each bit cell which allows very fast table lookup. Since the entire chip can be compared, contents are randomly stored without considering addressing scheme. These chips have less storage capacity than regular memory chips.

Memory Mapping and Concept of Virtual Memory

The transformation of data from main memory to cache memory is called mapping. There are 3 main types of mapping:

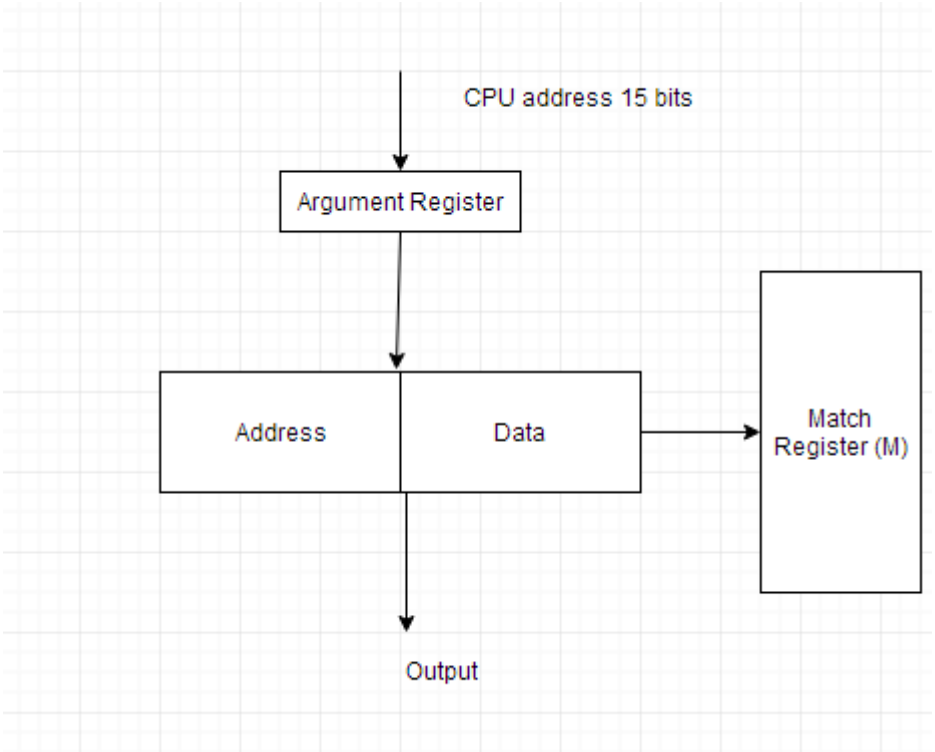
- Associative Mapping
- Direct Mapping
- Set Associative Mapping

Associative Mapping

The associative memory stores both address and data. The address value of 15 bits is 5 digit octal numbers and data is of 12 bits word in 4 digit octal number. A CPU address of 15 bits is placed in **argument register** and the associative memory is searched for matching address.

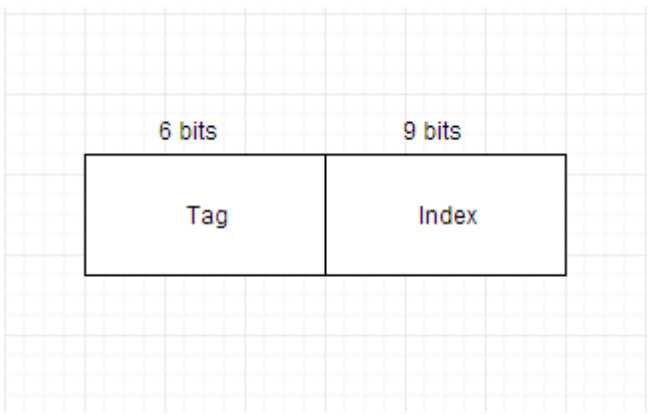
Associative Mapping

The associative memory stores both address and data. The address value of 15 bits is 5 digit octal numbers and data is of 12 bits word in 4 digit octal number. A CPU address of 15 bits is placed in **argument register** and the associative memory is searched for matching address.



Direct Mapping

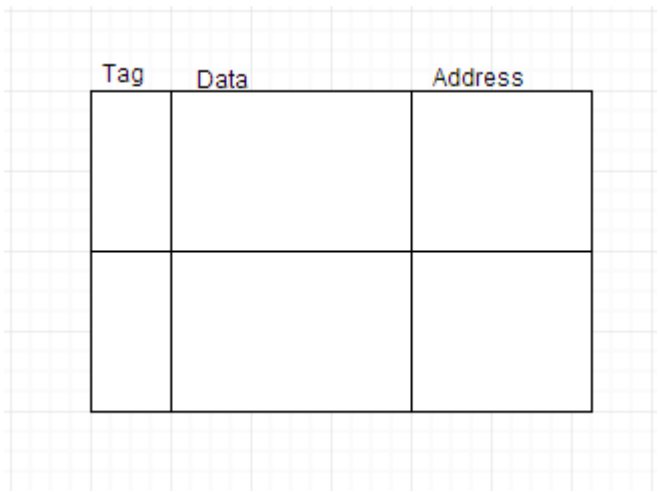
The CPU address of 15 bits is divided into 2 fields. In this the 9 least significant bits constitute the **index** field and the remaining 6 bits constitute the **tag** field. The number of bits in index field is equal to the number of address bits required to access cache memory.



Set Associative Mapping

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

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



Replacement Algorithms

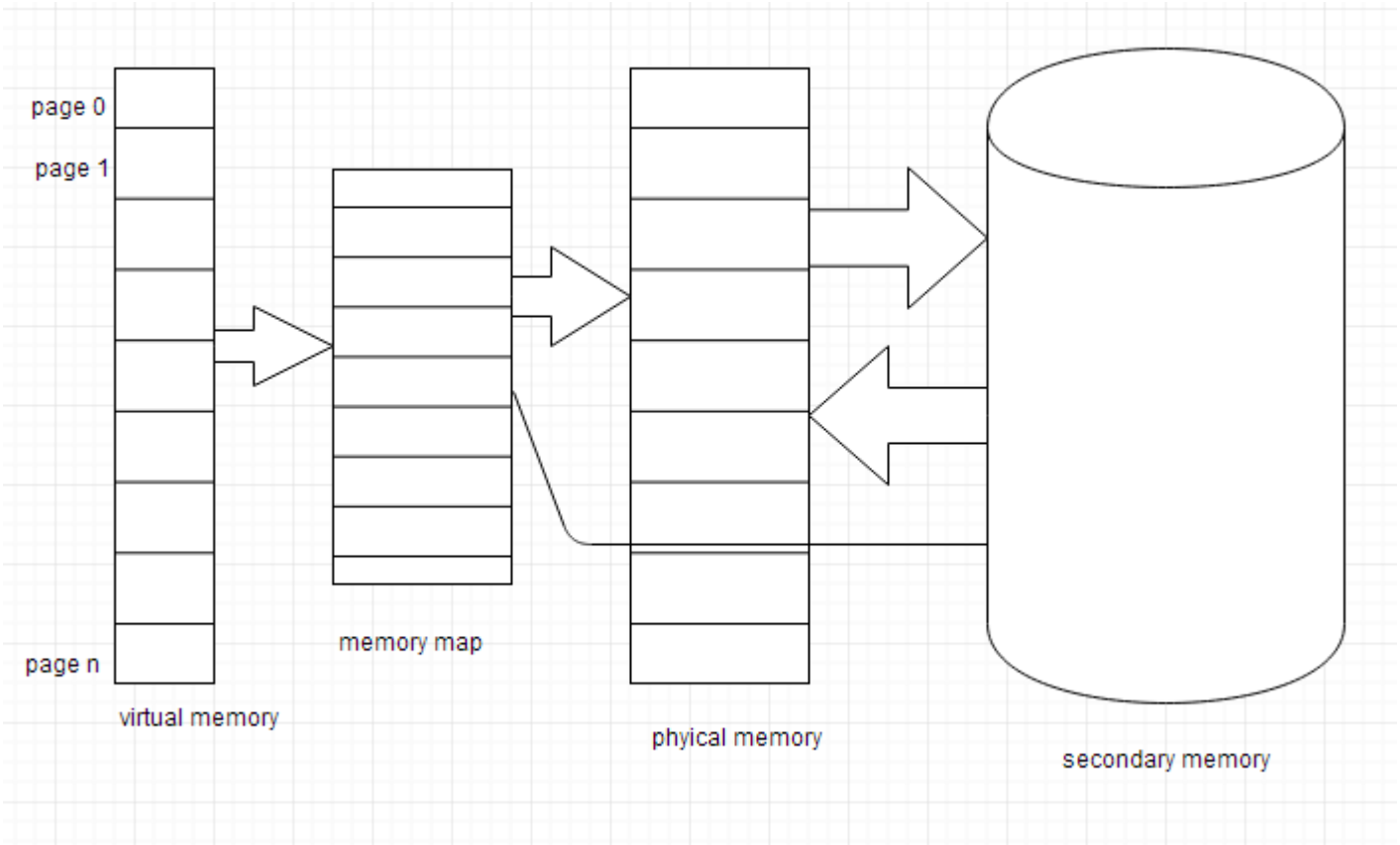
Data is continuously replaced with new data in the cache memory using replacement algorithms.
Following are the 2 replacement algorithms used:

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

Virtual Memory

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

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



Virtual memory is a valuable concept in computer architecture that allows you to run large, sophisticated programs on a computer even if it has a relatively small amount of RAM. A computer with virtual memory artfully juggles the conflicting demands of multiple programs within a fixed amount of physical memory. A PC that's low on memory can run the same programs as one with abundant RAM, although more slowly.

Physical vs Virtual Addresses

A computer accesses the contents of its RAM through a system of addresses, which are essentially numbers that locate each byte. Because the amount of memory varies from PC to PC, determining which software will work on a given computer becomes complicated. Virtual memory solves this problem by treating each computer as if it has a large amount of RAM and each program as if it uses the PC exclusively. The operating system, such as Microsoft Windows or Apple's OS X, creates a set of virtual addresses for each program. The OS translates virtual addresses into physical ones, dynamically fitting programs into RAM as it becomes available.

Paging

Virtual memory breaks programs into fixed-size blocks called pages. If a computer has abundant physical memory, the operating system loads all of a program's pages into RAM. If not, the OS fits as much as it can and runs the instructions in those pages. When the computer is done with those pages, it loads the rest of the program into RAM, possibly overwriting earlier pages. Because the operating system automatically manages these details, this frees the software developer to concentrate on program features and not worry about memory issues.

Multiprogramming

Virtual memory with paging lets a computer run many programs at the same time, almost regardless of available RAM. This benefit, called multiprogramming, is a key feature of modern PC operating systems, as they accommodate many utility programs such as printer drivers, network managers and virus scanners at the same time as your applications -- Web browsers, word processors, email and media players.

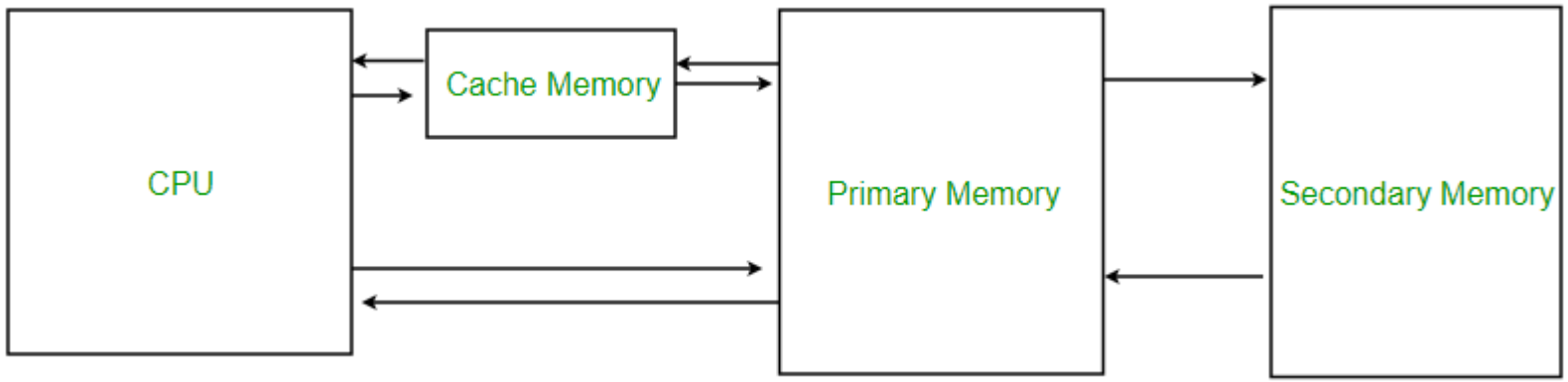
Paging File

With virtual memory, the computer writes program pages that have not been recently used to an area on the hard drive called a paging file. The file saves the data contained in the pages; if the program needs it again, the operating system reloads it when RAM becomes available. When many programs compete for RAM, the act of swapping pages to the file can slow a computer's processing speed, as it spends more time doing memory management chores and less time getting useful work done. Ideally, a computer will have enough RAM to handle the demands of many programs, minimizing the time the computer spends managing its pages.

Memory Protection

A computer without virtual memory can still run many programs at the same time, although one program might change, accidentally or deliberately, the data in another if its addresses point to the wrong program. Virtual memory prevents this situation because a program never "sees" its physical addresses. The virtual memory manager protects the data in one program from changes by another.

Cache Memory is a special very high-speed memory. It is used to speed up and synchronizing with high-speed CPU. Cache memory is costlier than main memory or disk memory but economical than CPU registers. Cache memory is an extremely fast memory type that acts as a buffer between RAM and the CPU. It holds frequently requested data and instructions so that they are immediately available to the CPU when needed.
Cache memory is used to reduce the average time to access data from the Main memory. The cache is a smaller and faster memory which stores copies of the data from frequently used main memory locations. There are various different independent caches in a CPU, which store instructions and data.



Levels of memory:

•**Level 1 or Register –**

It is a type of memory in which data is stored and accepted that are immediately stored in CPU.

•Most commonly used register is accumulator, Program counter, address register etc.

•**Level 2 or Cache memory –**

It is the fastest memory which has faster access time where data is temporarily stored for faster access.

•**Level 3 or Main Memory –**

It is memory on which computer works currently. It is small in size and once power is off data no

• longer stays in this memory.

•**Level 4 or Secondary Memory –**

It is external memory which is not as fast as main memory but data stays permanently in this memory.

Cache Performance:

When the processor needs to read or write a location in main memory,

it first checks for a corresponding entry in the cache.

•f the processor finds that the memory location is in the cache, a **cache hit** has occurred and data is read from cache
•If the processor **does not** find the memory location in the cache, a **cache miss** has occurred.

• For a cache miss, the cache allocates a new entry and copies in data from main memory, then the request is
•fulfilled from the contents of the cache.
The performance of cache memory is frequently measured in terms of a quantity called **Hit ratio.**
Hit ratio = hit / (hit + miss) = no. of hits/total accesses
We can improve Cache performance using higher cache block size,
higher associativity, reduce miss rate, reduce miss penalty, and reduce the time to hit in the cache.

1.Direct Mapping –

The simplest technique, known as direct mapping, maps each block of main memory into only one possible cache line. or

In Direct mapping, assigne each memory block to a specific line in the cache.

If a line is previously taken up by a memory block when a new block needs to be loaded, the old block is trashed.

An address space is split into two parts index field and a tag field. The cache is used to store the tag field whereas the rest is stored in the main memory. Direct mapping`s performance is directly proportional to the Hit ratio.

i = j modulo m where i=cache line number j= main memory block number m=number of lines in the cache

For purposes of cache access, each main memory address can be viewed as consisting of three fields.

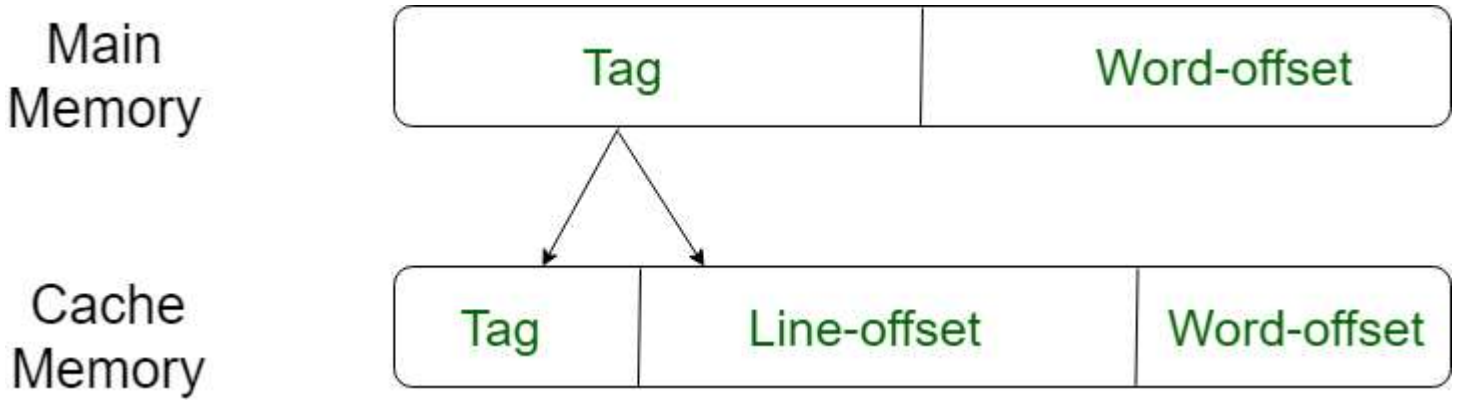
The least significant w bits identify a unique word or byte within a block of main memory.

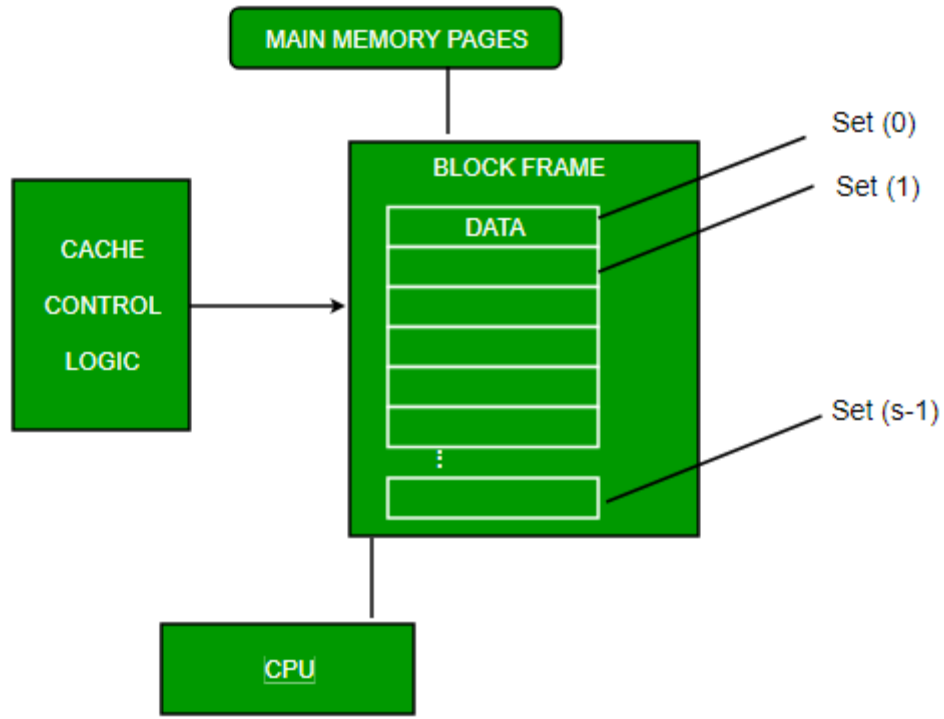
In most contemporary machines, the address is at the byte level.

The remaining s bits specify one of the 2^s blocks of main memory.

The cache logic interprets these s bits as a tag of s-r bits (most significant portion) and a line field of r bits.

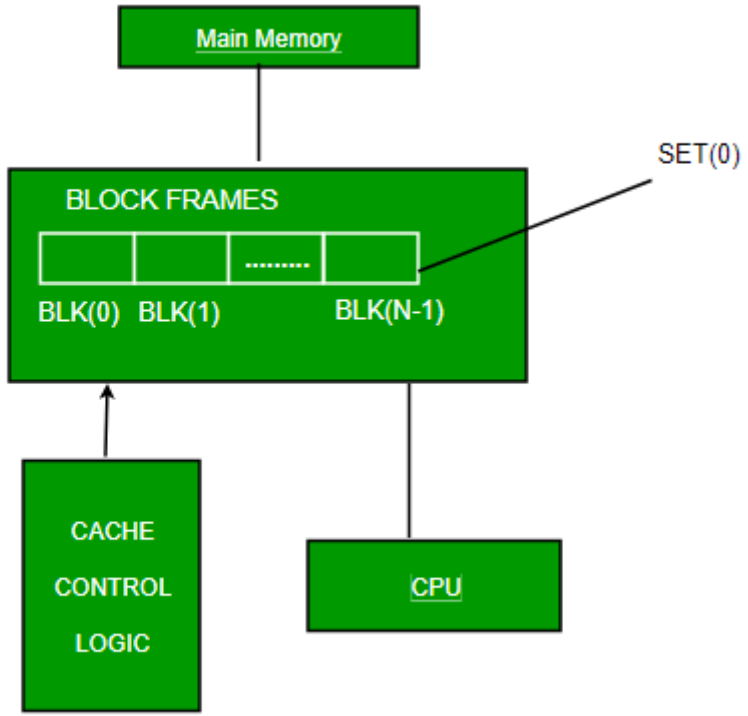
This latter field identifies one of the $m=2^r$ lines of the cache




MAIN MEMORY PAGES
BLOCK FRAME
DATA
Set (0)
Set (1)
Set (s-1)
CACHE
CONTROL
LOGIC
CPU

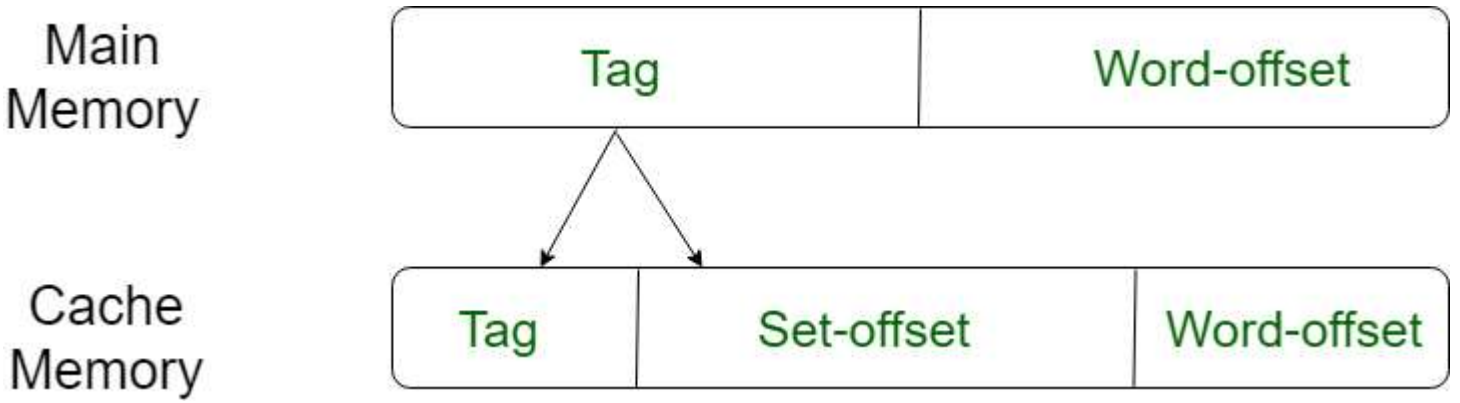
Associative Mapping –

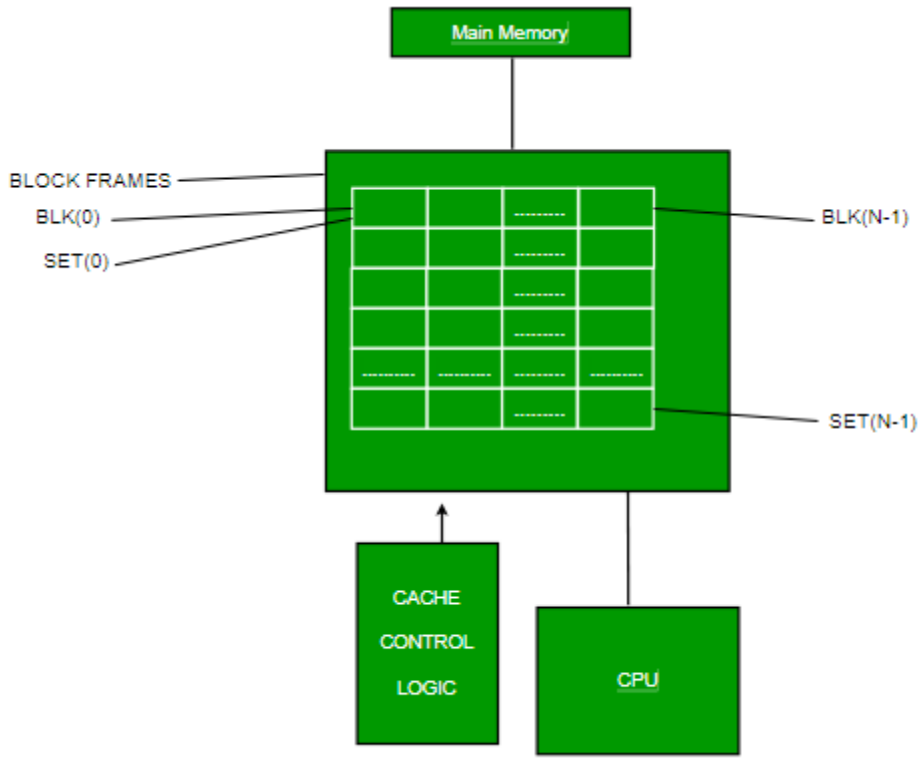
In this type of mapping, the associative memory is used to store content and addresses of the memory word. Any block can go into any line of the cache. This means that the word id bits are used to identify which word in the block is needed, but the tag becomes all of the remaining bits. This enables the placement of any word at any place in the cache memory. It is considered to be the fastest and the most flexible mapping form.



Set-associative Mapping –

This form of mapping is an enhanced form of direct mapping where the drawbacks of direct mapping are removed. Set associative addresses the problem of possible thrashing in the direct mapping method. It does this by saying that instead of having exactly one line that a block can map to in the cache, we will group a few lines together creating a ***set***. Then a block in memory can map to any one of the lines of a specific set..Set-associative mapping allows that each word that is present in the cache can have two or more words in the main memory for the same index address. Set associative cache mapping combines the best of direct and associative cache mapping techniques. In this case, the cache consists of a number of sets, each of which consists of a number of lines. The relationships are




Main Memory
BLOCK FRAMES
BLK(0)
SET(0)
BLK(N-1)
SET(N-1)
CACHE
CONTROL
LOGIC
CPU

Application of Cache Memory –

1.Usually, the cache memory can store a reasonable number of blocks at any given time, but this number is small compared to the total number of blocks in the main memory.

2.The correspondence between the main memory blocks and those in the cache is specified by a mapping function.

Types of Cache –

- **Primary Cache –**

A primary cache is always located on the processor chip. This cache is small and its access time is comparable to that of processor registers.

- **Secondary Cache –**

Secondary cache is placed between the primary cache and the rest of the memory. It is referred to as the level 2 (L2) cache. Often, the Level 2 cache is also housed on the processor chip.

Locality of reference –

Since size of cache memory is less as compared to main memory. So to check which part of main memory should be given priority and loaded in cache is decided based on locality of reference.

Types of Locality of reference

1.Spatial Locality of reference

This says that there is a chance that element will be present in the close proximity to the reference point and next time if again searched then more close proximity to the point of reference.

2.Temporal Locality of reference

In this Least recently used algorithm will be used. Whenever there is page fault occurs within a word will not only load word in main memory but complete page fault will be loaded because spatial locality of reference rule says that if you are referring any word next word will be referred in its register that’s why we load complete page table so the complete block will be loaded.