# COMPUTER ORGANIZATION AND ARCHITECTURE

Dr. Shreesh Kumar Shrivastava
Assistant Professor, Dept. Of ECE
GLA University, Mathura

## Computer Organization

- Computer Organization refers to the operational units and their interconnections.
- Computer Organization is concerned with the way the hardware components operate and the way they are connected together to form the computer system.
- Computer Organization explains how a computer works

- Deals with Low Level Design Issues .
- Organizational attributes include those hardware details transparent to the programmer, such as control signals, interfaces between computer and peripherals, and the memory technology used.
- Physical components like circuits with Adder , Signals, Peripheral etc

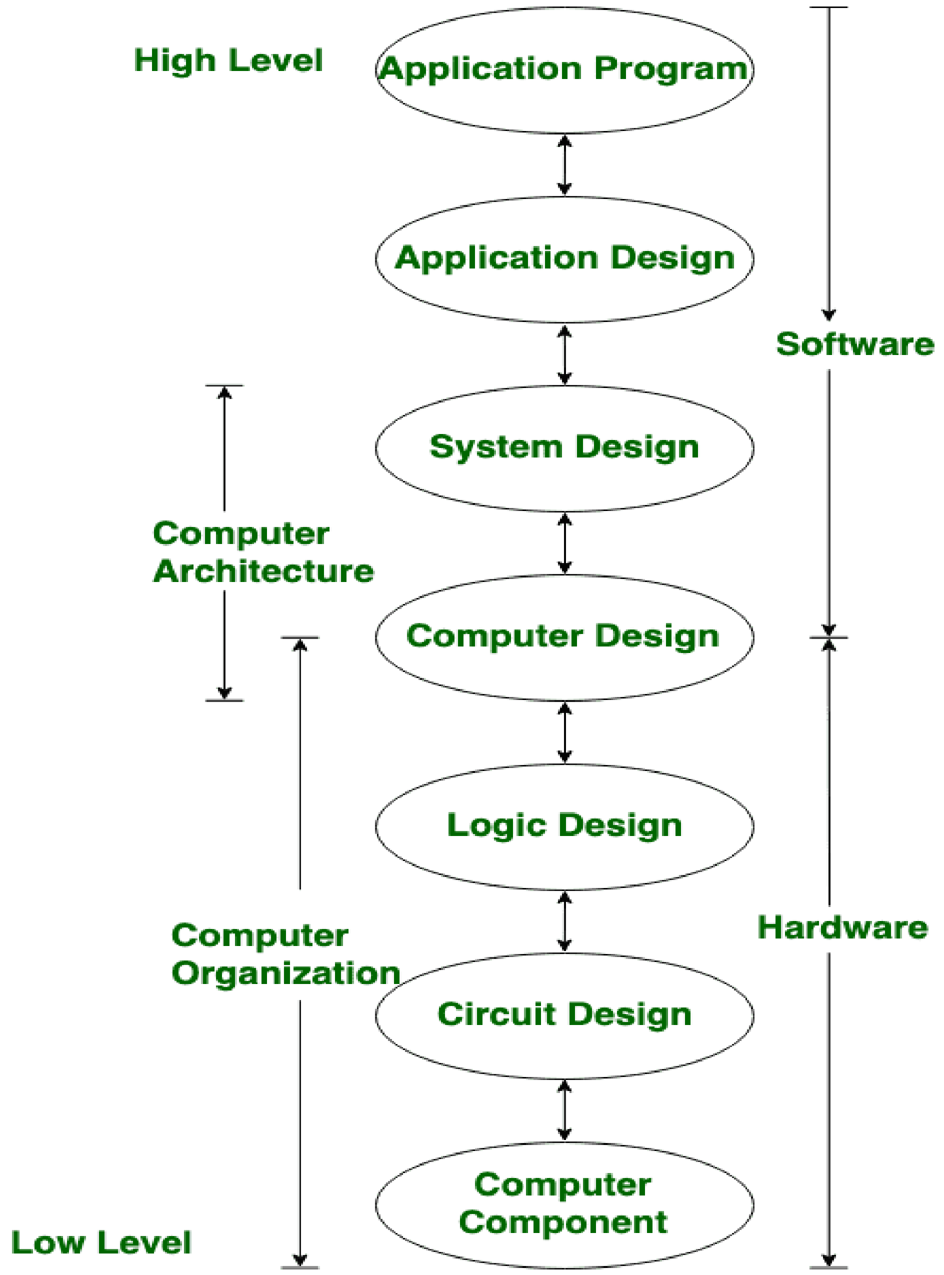
## Computer Architecture

- Computer Architecture is describing the capabilities and programming model of a computer but not a particular implementation .

- It includes the instruction formats, the instruction sets, and techniques for addressing memory.

- Architectural design of a computer system is concerned with the various functional modules, such as processors and memories, and structuring them together into computer system.

- Deals with High Level Design Issues.
- For designing a computer its architecture is fixed first.
- It describes ‘What  the system does’?
- It co-ordinate between hardware and software of the system
- Example Instruction Set, Addressing Modes, Data Type



Computer System

## There are two basic types of architecture

- **Von Neumann Architecture**
- **Harvard Architecture**

## Von Neumann Architecture (Stored Memory Program)

- In the Von Neumann Architecture, Program and data are stored in the same memory and managed by the same information handling subsystem.

- The Von Neumann Architecture has a single storage structure to hold both instructions and data.

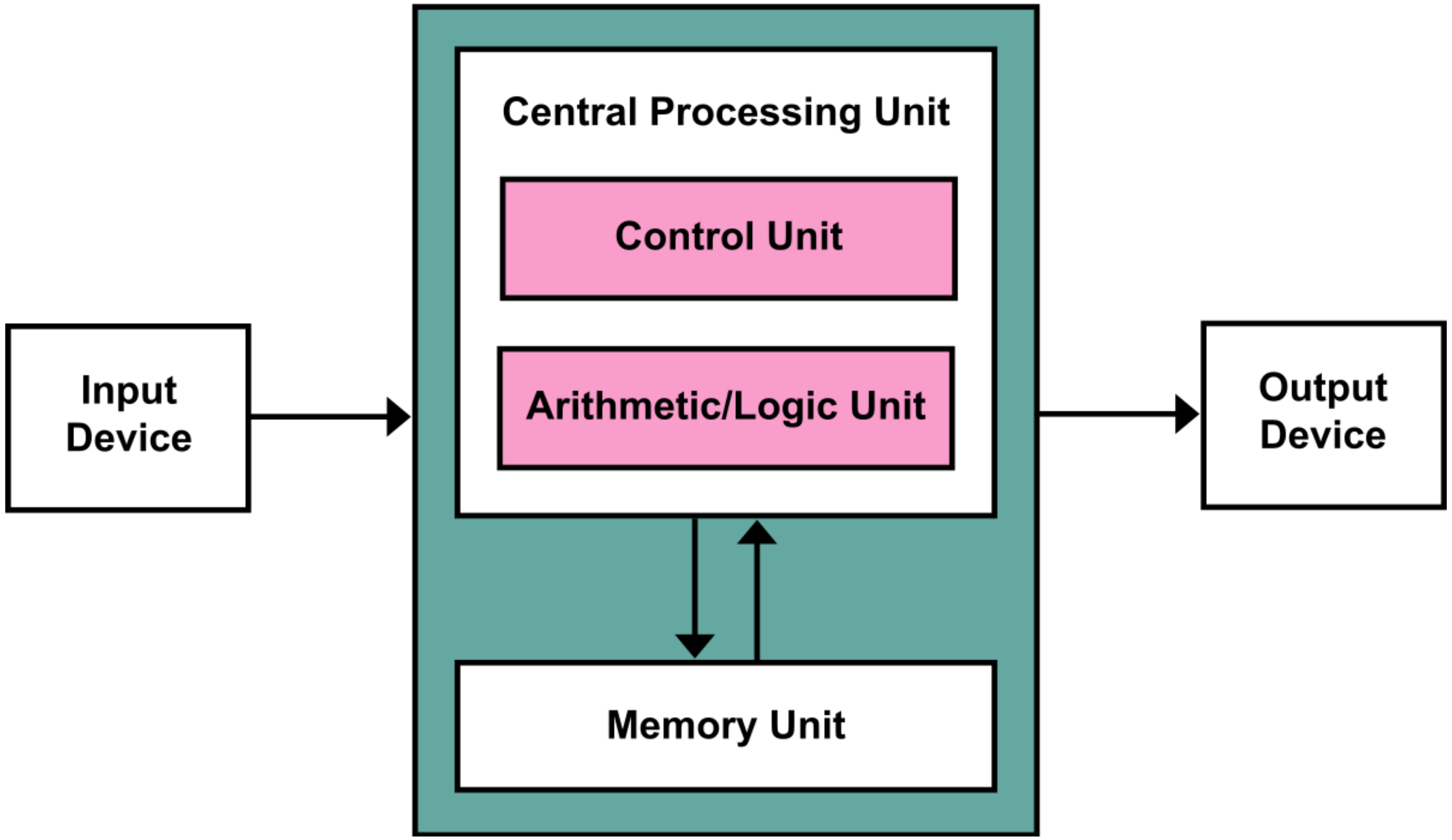
- The CPU can be either reading an instruction or reading / writing data from / to the memory because instructions and data use the same BUS system.

- According to Von Neumann Architecture structure of computer system as composed of the following components:-

- Control Unit
- Arithmetic and Logic Unit
- Main Memory
- Registers
- Inputs/Outputs Unit

## Von Neumann Architecture(Diagram)

V

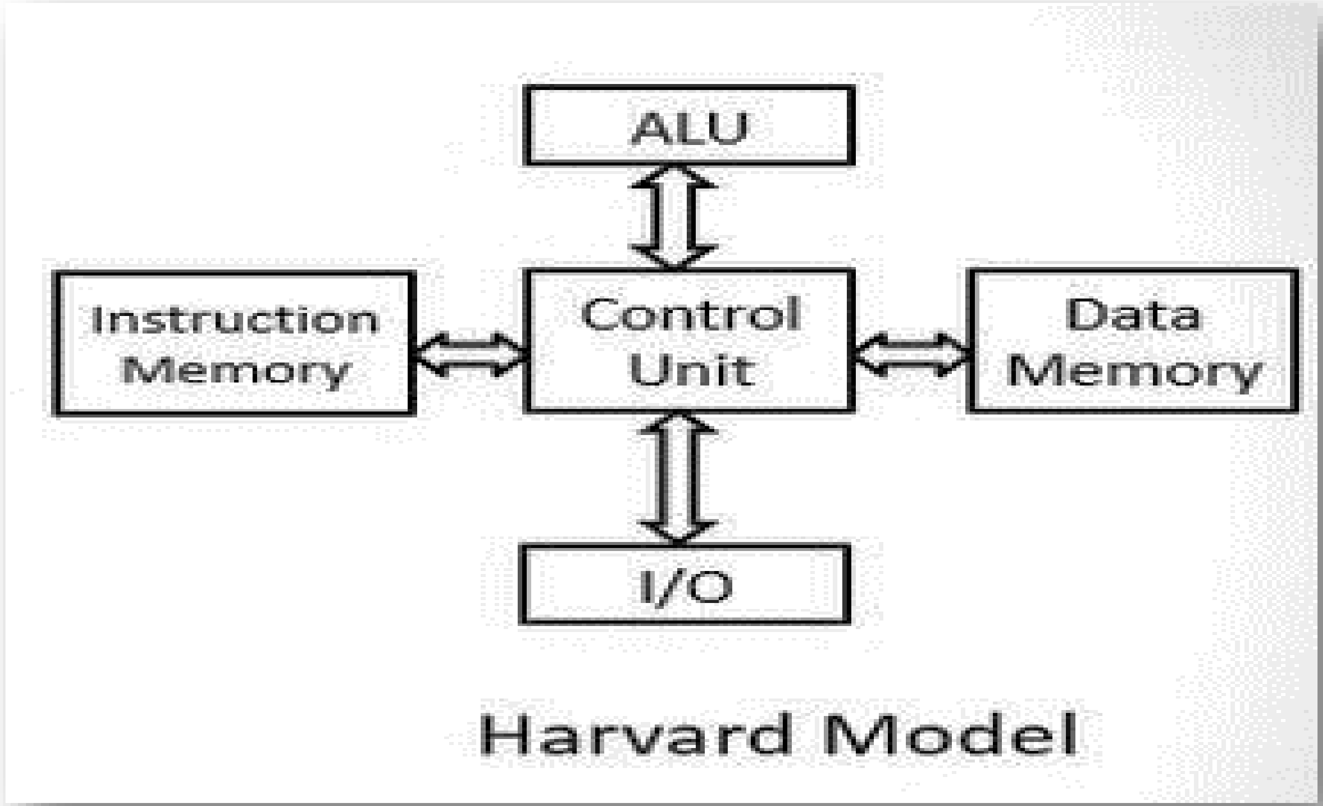


## Harvard Architecture

- The Harvard Architecture has a physically separated storage and signal pathways for instructions and data.

- In the Harvard Architectural Computer, the CPU can read both an instruction and data from memory at the same time, leading to double the memory bandwidth.

## HARDVARD ARCHITECTURE



- In the Harvard architecture, program and data are stored and handled by different subsystems.

- One example of Harvard Architecture is the early computer Mainframe systems where instructions are stored in one programming media such as punch cards and data are stored in the other programming media such as Tapes.

- A Microcontroller with Harvard Architecture can both read an instruction and perform a data memory access at the same time, even without a cache.

- A Harvard Architecture microcontroller can thus be faster for a given circuit complexity because instruction fetches and data access do not contend for a single memory pathway.

## Floating Point Number

- A floating-point number is a **Rational Number**, because it can be represented as one integer divided by another.
- Example: $1.45\times10^{3}$ is

$$(145/100)\times1000$$

or

145,000/100.

## Components of IEEE 754

- IEEE 754 is the most efficient in most cases. IEEE 754 has 3 basic components:

1. **Sign (+ve or –ve)**
2. **The Biased exponent**
3. **The Normalized Mantissa**

## Component 1: Sign (+ve or –ve)

- 0 represents a positive number
- 1 represents a negative number.

For example: 28.017 the sign is +ve. So it will be represent by 0.

-28.017 the sign is -ve. So it will be represent by 1.

## Component 2: The Biased exponent

- The exponent field needs to represent both positive and negative exponents. A bias is added to the actual exponent in order to get the stored exponent.

- The value of bias based of size of exponent component.

For example:

if size of exponent field is 8 bits then bias value is

$2^{size-1}-1 = 127$

if size of exponent field is 11 bits then bias value is

$2^{size-1}-1 = 1023$

## Component 3: The Normalized Mantissa

- The mantissa is part of a number in scientific notation or a floating-point number, consisting of its significant digits.

- A normalized mantissa is one with only one 1 to the left of the decimal like 1.mmmmmmmmmm….mmmmm

Example: 8.25

Binary Representation : 1000.01

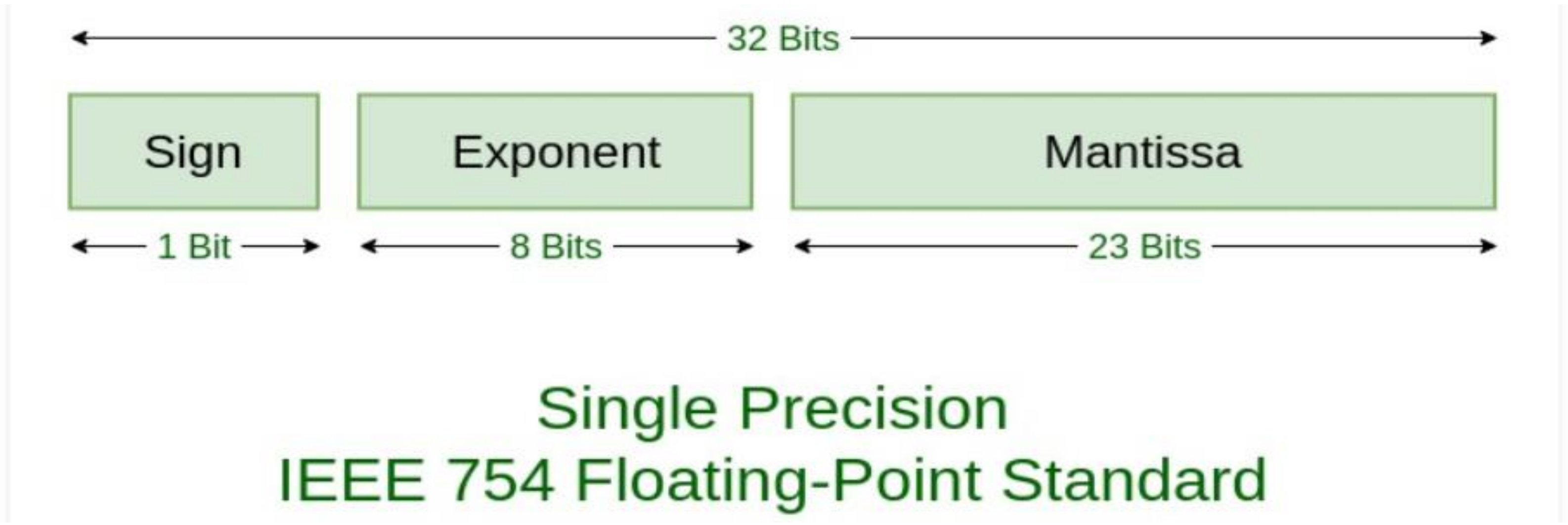
Normalized Mantissa : $1.00001 \times 2^{+3}$

## Representation of IEEE 754

There are two types of representation:

1. **Single precision**

   (+ve or –ve sign) 1.mantissa x $2^{\text{Exponent-127}}$



## Single Precision Example

**Question: Represent 85.125 in IEEE 32 bits Format.**

Binary of 85 = 1010101

Binary of 0.125 = 001

Floating Point Number 85.125 = 1010101.001 =1.010101001 x $2^{+6}$

So, Sign component: 0 because of +ve sign.

biased exponent: 127+6=133

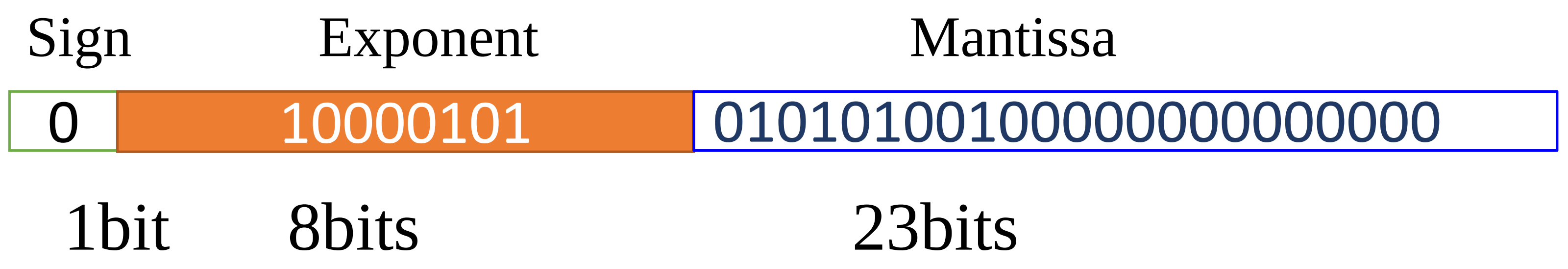
Binary of 133 = 10000101

Normalized mantissa = 010101001

we will add 0's to complete the 23 bits in mantissa

# The IEEE 754 Single precision is: =

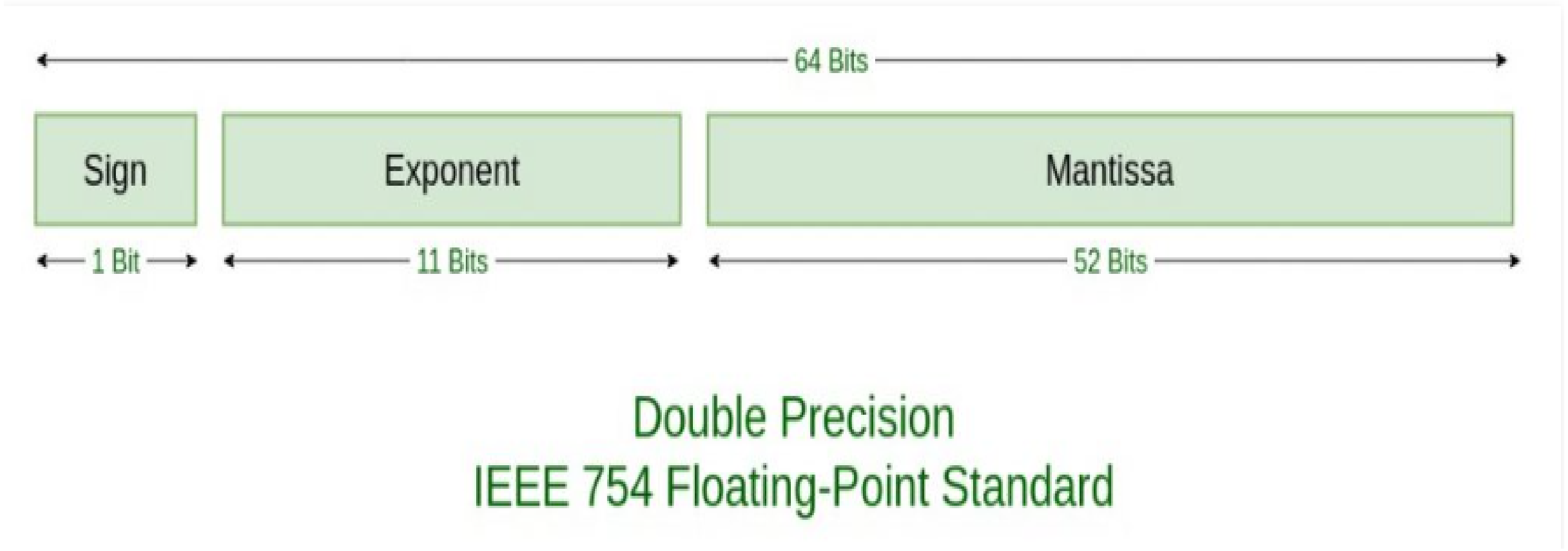
32bits : 0 10000101 01010100100000000000000



This can be written in hexadecimal form **42AA4000** in Computer.

## 2. Double precision

(+ve or –ve sign) 1.mantissa x $2^{\text{Exponent-1023}}$



# Double Precision Example

**Question:** Represent 85.125 in IEEE 64 bits Format.

Binary of 85 = 1010101

Binary of 0.125 = 001

Floating Point Number 85.125 = 1010101.001 =1.010101001 x $2^{+6}$

So, Sign component: 0 because of +ve sign.

Biased exponent: 1023+6=1029

Binary of 1029 = 10000000101

Normalized mantissa = 010101001

we will add 0's to complete the 52 bits in mantissa

## The IEEE Double precision is

64 bits :
010000000101
0101010010000000000000000000000000000000000000000000

Sign Exponent Mantissa

1bit 11bits 52bits

| | | |
|---|---|---|
| 0 | 10000000101 | 01010100100000000000000000000000000000000000000000000 |

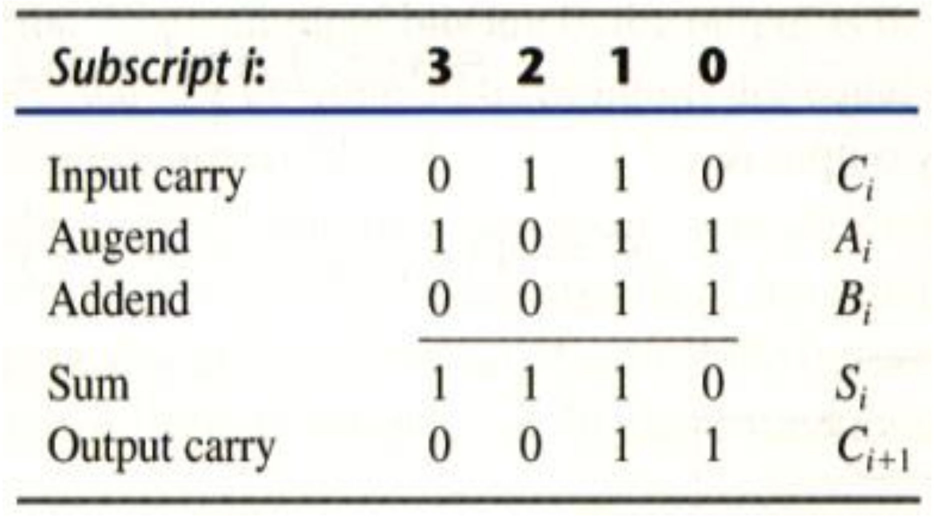
0000000000

This can be written in hexadecimal form **4055480000000000** in Computer.

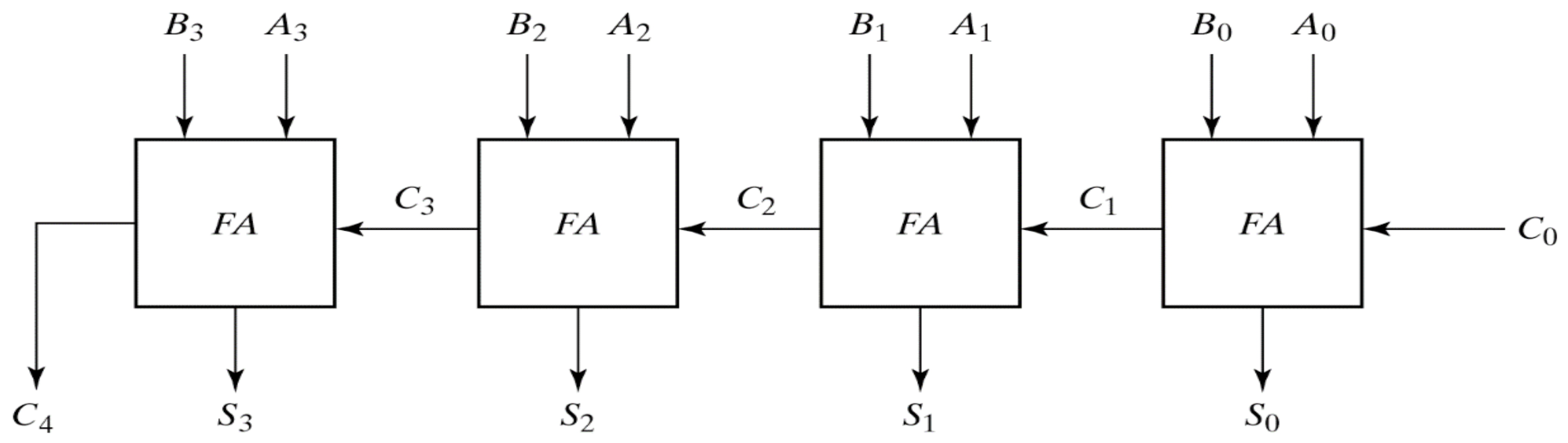
# Binary Adder/Subtractor
# and
# Carry Look-ahead Adder

## 4-bit Binary Adder

This is also called Ripple Carry Adder , because of the construction with full adders are connected in cascade.



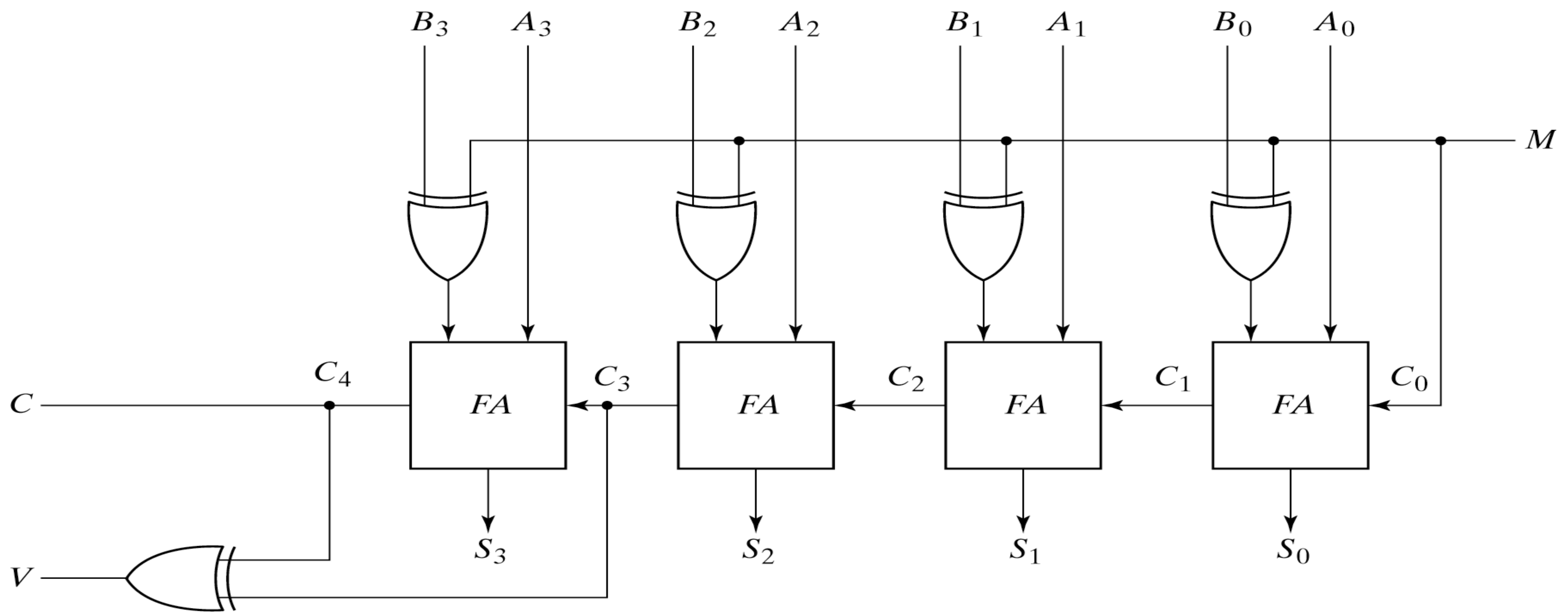
| *Subscript i:* | **3** | **2** | **1** | **0** | |
|---|---|---|---|---|---|
| Input carry | 0 | 1 | 1 | 0 | $C_i$ |
| Augend | 1 | 0 | 1 | 1 | $A_i$ |
| Addend | 0 | 0 | 1 | 1 | $B_i$ |
| Sum | 1 | 1 | 1 | 0 | $S_i$ |
| Output carry | 0 | 0 | 1 | 1 | $C_{i+1}$ |



**4-bit Binary Adder**

# 4-bit Binary Adder/Subtractor with overflow

**M = 1→subtractor ; M = 0→adder**



## Full adder using half adder

- Full-adder can also implemented with two half adders and one OR gate.

$$S = z \oplus (x \oplus y)$$
$$= z'(xy' + x'y) + z(xy' + x'y)'$$
$$= xy'z' + x'yz' + xyz + x'y'z$$
$$C = z(xy' + x'y) + xy = xy'z + x'yz + xy$$

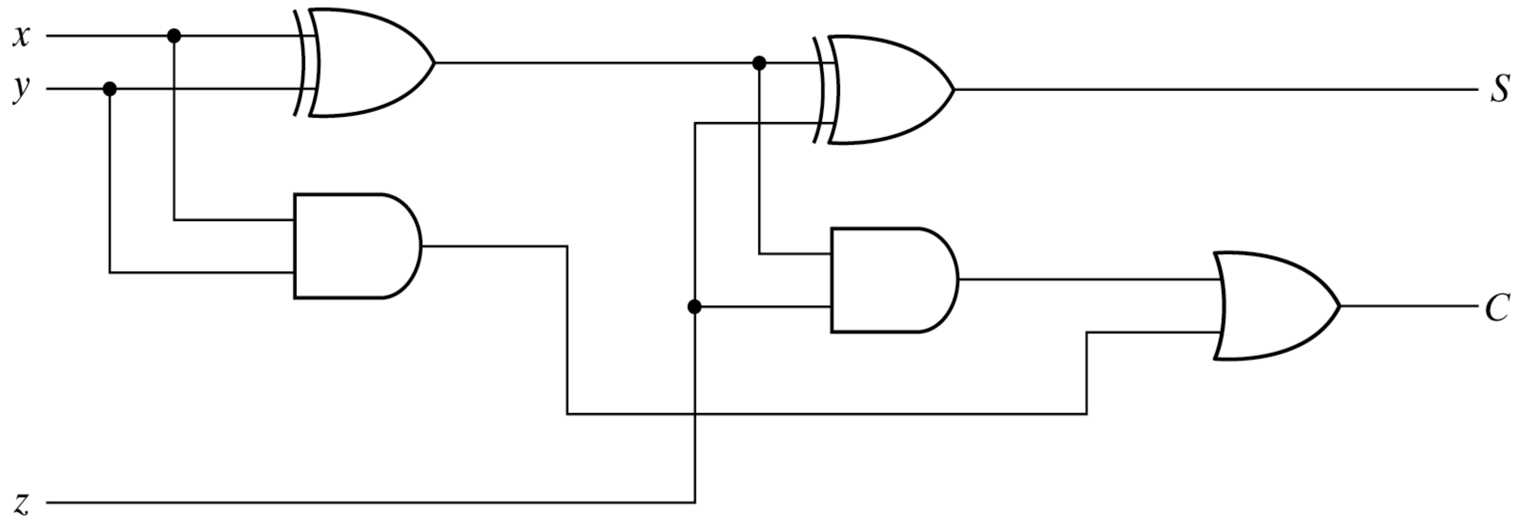


Fig. 4-8 Implementation of Full Adder with Two Half Adders and an OR Gate

# Boolean functions for Carry look ahead adder

$P_i = A_i \oplus B_i$ steady state value

$G_i = A_iB_i$ steady state value

Output sum and carry

$S_i = P_i \oplus C_i$

$C_{i+1} = G_i + P_iC_i$

$G_i$ : carry generate $P_i$ : carry propagate

$C_0$ = input carry

$C_1 = G_0 + P_0\boxed{C_0}$

$C_2 = G_1 + P_1C_1 = G_1 + P_1G_0 + P_1P_0\boxed{C_0}$

$C_3 = G_2 + P_2C_2 = G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0\boxed{C_0}$

- $C_3$ does not have to wait for $C_2$ and $C_1$ to propagate.

# Logic diagram of carry look-ahead generator

- $C_3$ is propagated at the same time as $C_2$ and $C_1$.

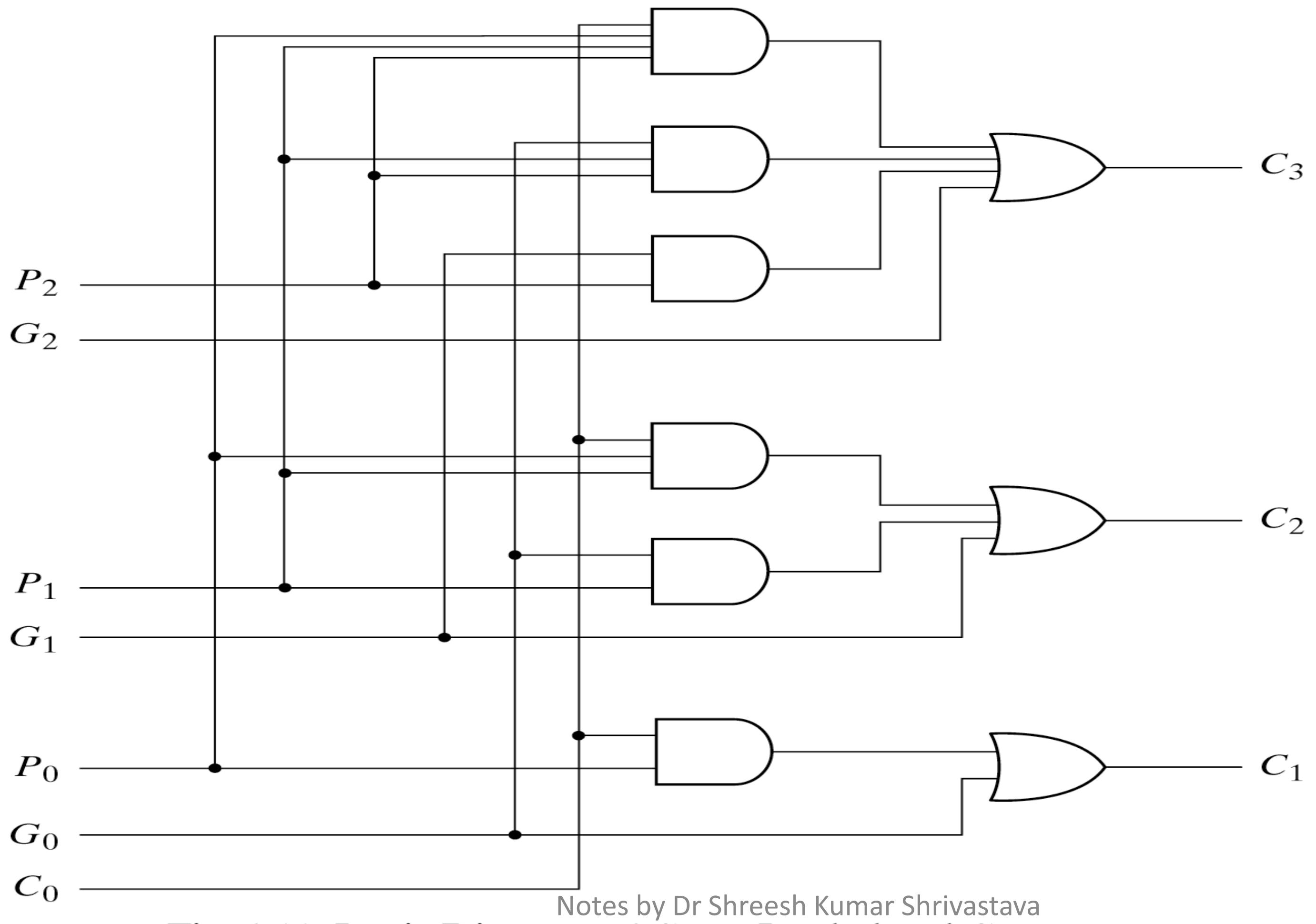


Fig. 4-11 Logic Diagram of Carry Lookahead Generator

## 4-bit adder with carry look ahead

- Delay time of n-bit CLAA = XOR + (AND + OR) + XOR

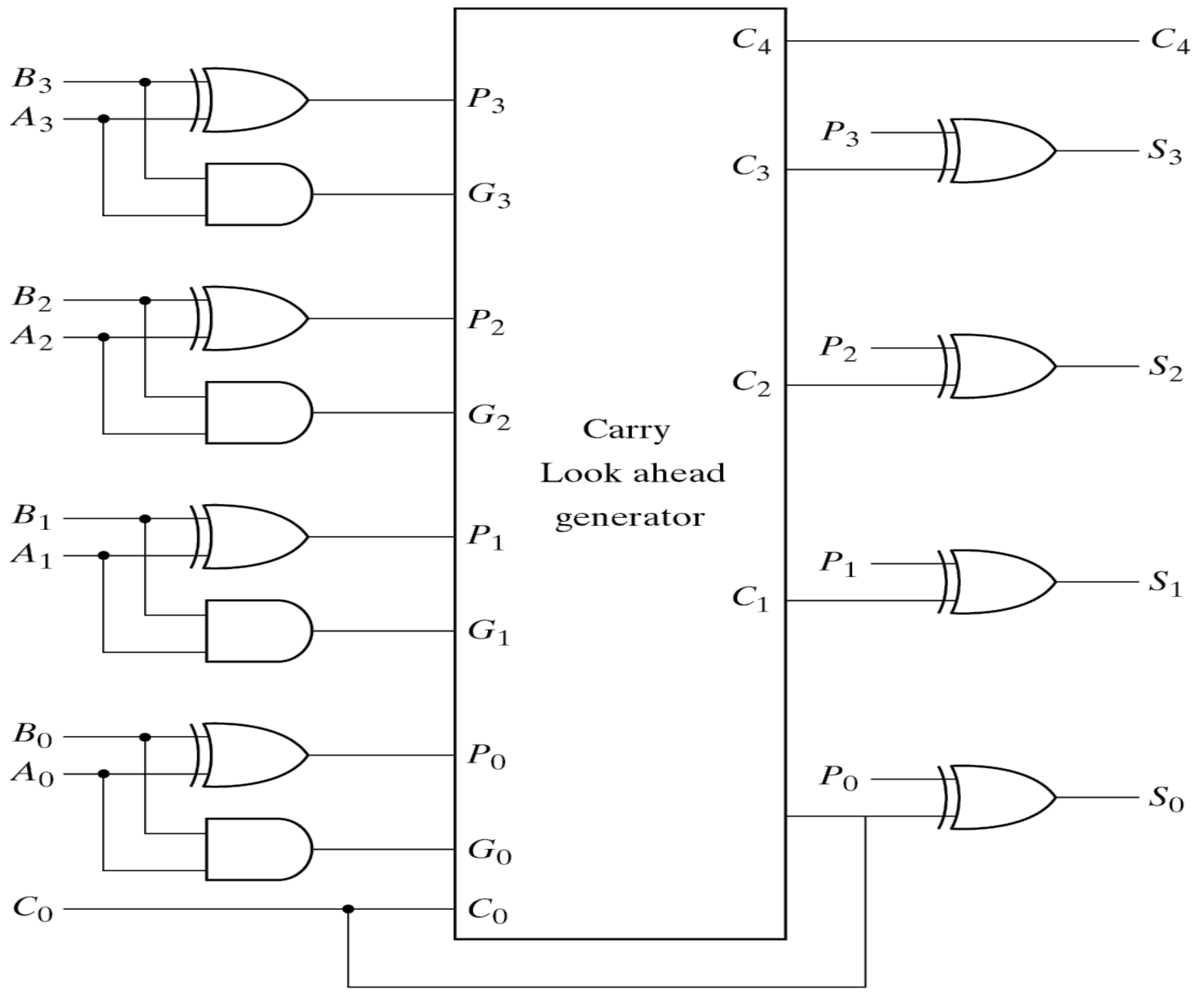


Fig. 4-12 4-Bit Adder with Carry Lookahead

## Computer Registers

## Computer Registers

- Register needed for:

1. Storing the instruction code after it is read from memory.
2. Processor register for manipulating the data.
3. Register for holding memory address.

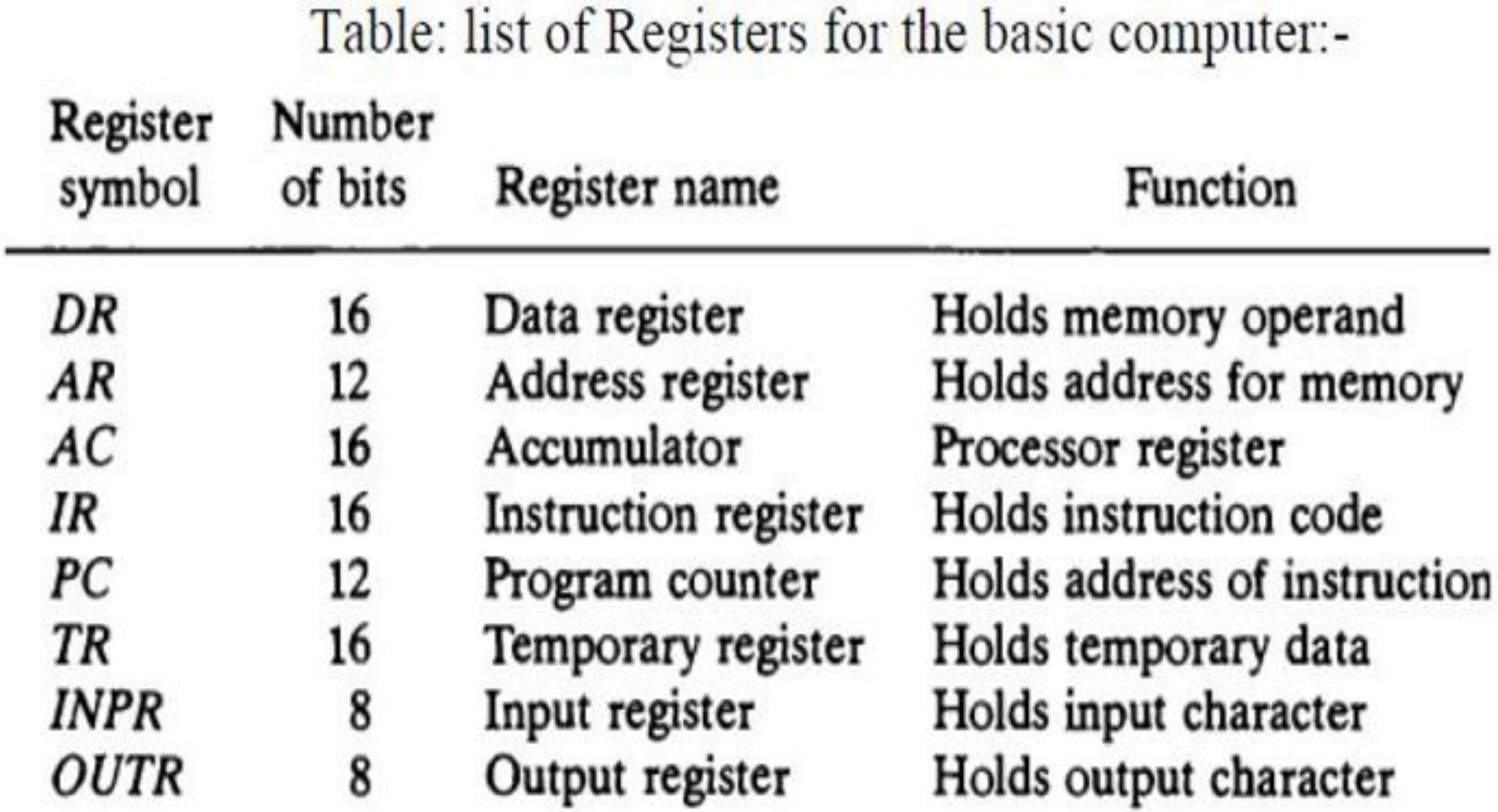
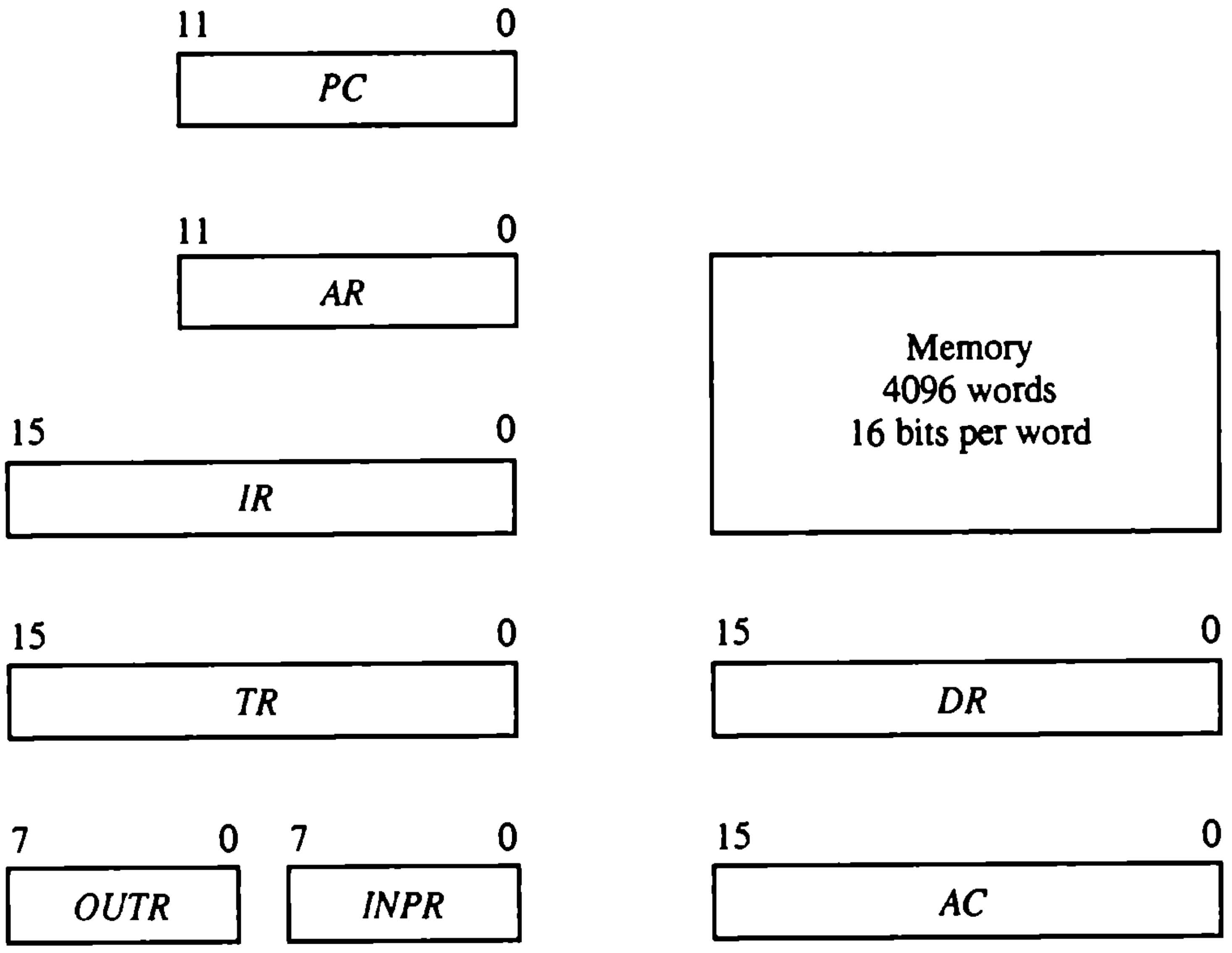


Table: list of Registers for the basic computer:-

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

## Computer Registers



## Registers

- Computer registers are designated by capital letters and numbers to denote the name of the register. i.e.

- **MAR -M**emory Address Register
- **PC** -Program Counter
- **IR - I**nstruction Register
- **R1** - Processor Register.
- **ACC**-Accumulator

- In this register, if the register is of n-bit then the numbering is started in sequence from 0 to n-1 starting from 0 in the right most position and increasing the number toward leftmost side.

- The representation of register in block diagram form the most common way to represent a register is by a rectangular box with the name of register inside.

(1)

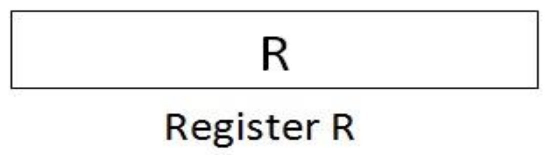


Fig. 4-1 a

(2)

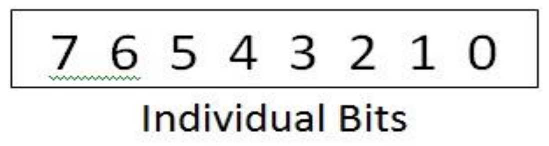
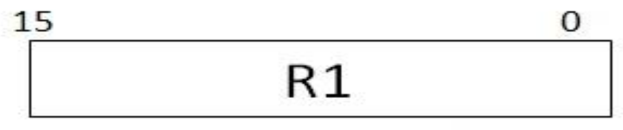


Fig. 4-1 b

The numbering of bits in 16 bit register can be marked on the top of the box as shown in the fig **4-1 c**.

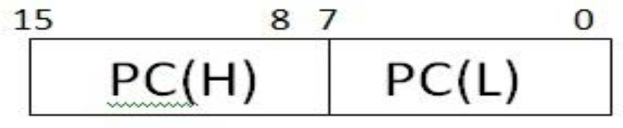
(3)



Numbering of Bits

**Fig. 4-1 c**

(4)



Two parts division of Register

**Fig. 4-1 d**

• A 16 bit register is divided into two parts in **4-1 d**. bits 0 through 7 are assigned the symbol L ( for the low bytes) and bits 8 through 15 are assigned the symbol H( for high byte).

## REGISTER TRANSFER LANGUAGE

- A Language in which the symbolic notation is used to describe the micro operation transfer among registers is called a register transfer language.

- ‘**Register Transfer**’ means a logic circuit that performs micro operations and transfer the result of the operation to the same or another register.

- A Register transfer language is a system for expressing in symbolic form the micro-operation sequences among the registers of a digital module.

- Information transfer from one register to another is designated in symbol form by means of a replacement operator in the statement.

$$\mathbf{R2 \leftarrow R1}$$

- Denotes the transfer of the content of register R1 into register R2. By definition the content of the source register R1 does not change after the transfer.

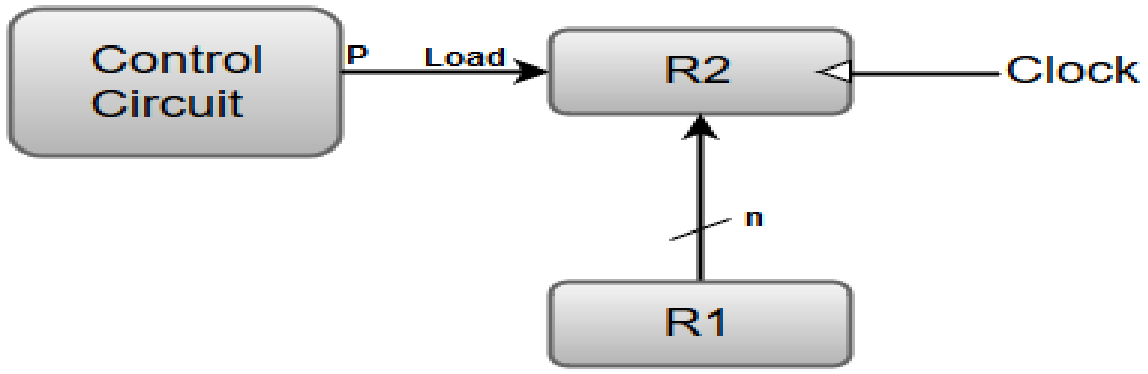
- Normally we want the transfer to occur only under a predetermined control condition. This can be shown by means of an If than statement

**If (P=1) than ( R2 🡐 R1)**

- Where p is a control function which is a Boolean variable that is equal to 1 or 0 the control function in the statement as follows

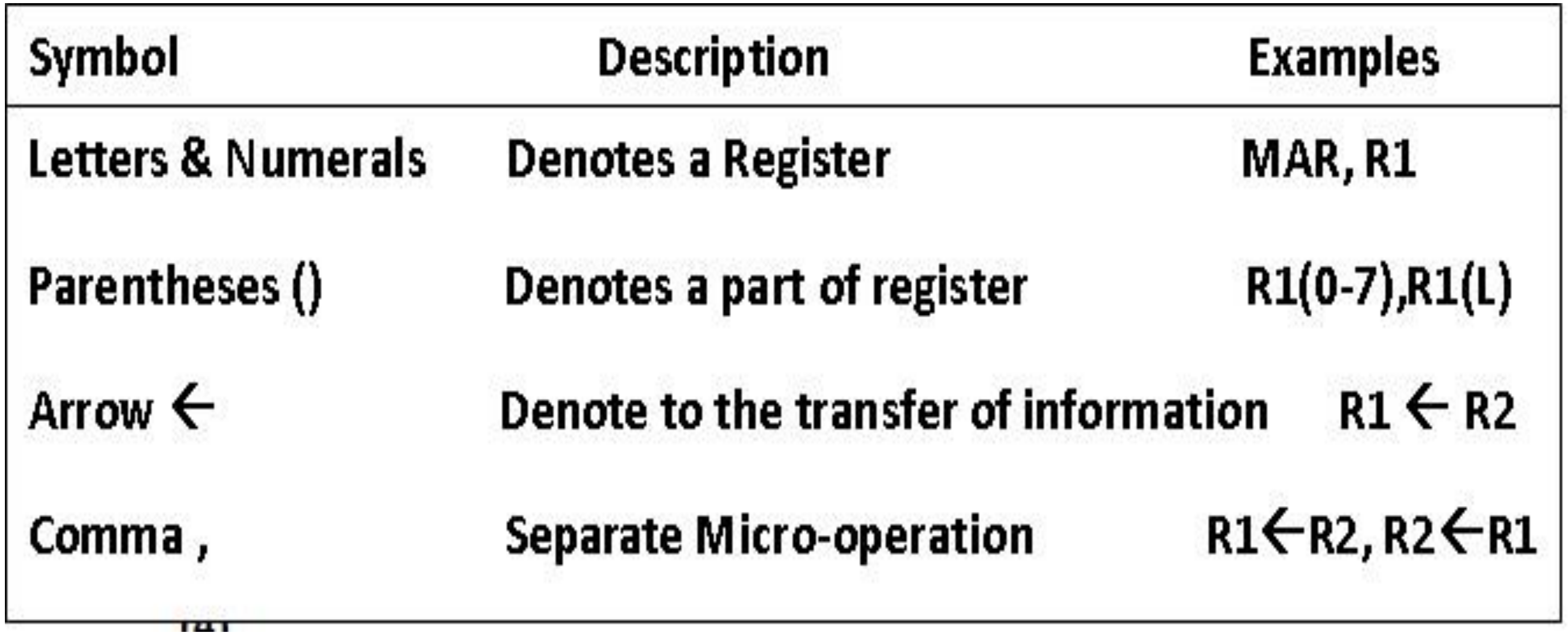
**P: R2 🡐 R1**

**Transfer from R1 to R2 when P = 1:**



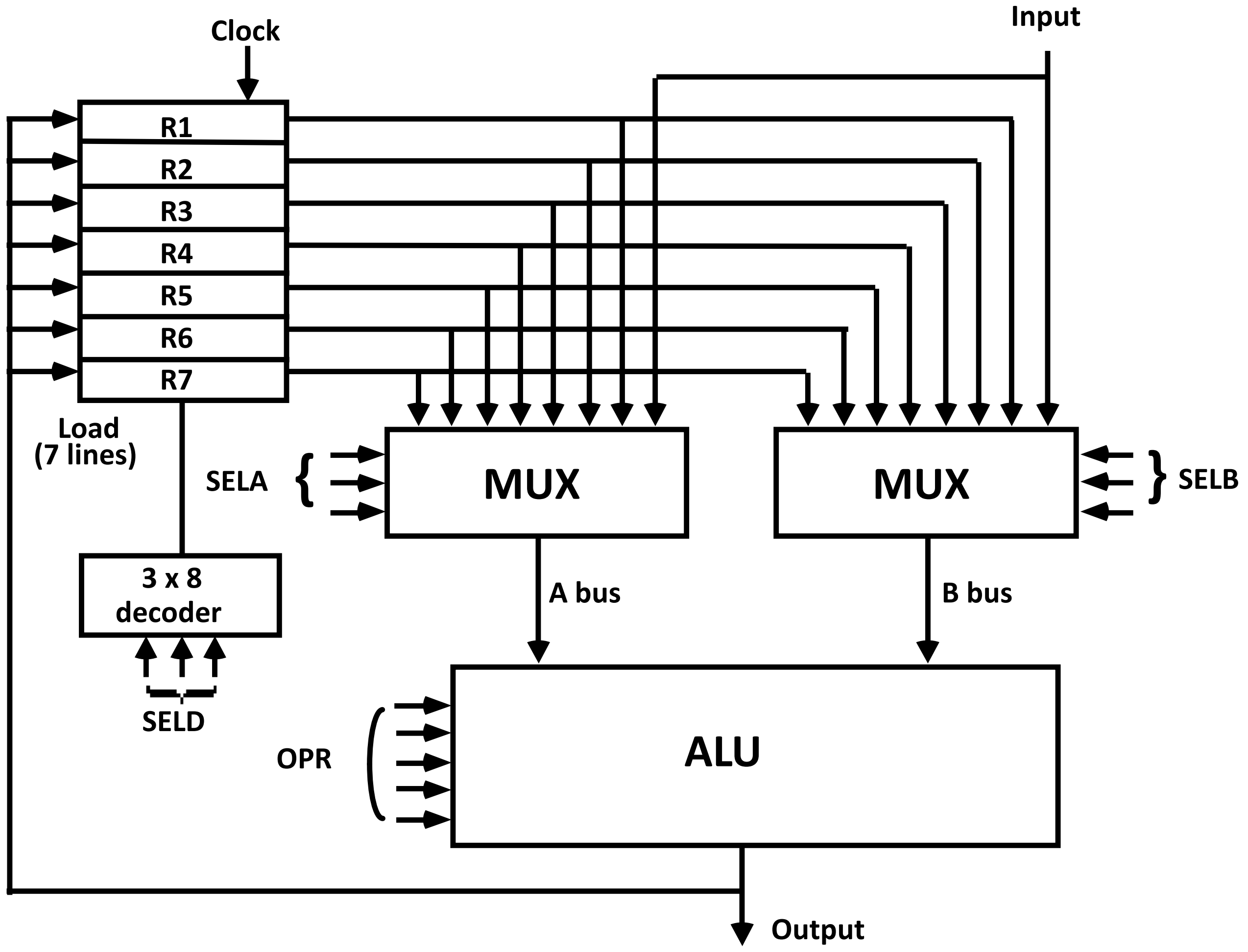
## Symbols of RTL :

Some symbols are used in Register Transfer Language to describe the expression.



| Symbol | Description | Examples |
|---|---|---|
| Letters & Numerals | Denotes a Register | MAR, R1 |
| Parentheses () | Denotes a part of register | R1(0-7),R1(L) |
| Arrow ← | Denote to the transfer of information | R1 ← R2 |
| Comma , | Separate Micro-operation | R1←R2, R2←R1 |

# GENERAL REGISTER ORGANIZATION



# OPERATION OF CONTROL UNIT

The control unit directs the information flow through ALU by:

- Selecting various *Components* in the system
- Selecting the *Function* of ALU

**Example: R1 <- R2 + R3**

[1] MUX A selector (SELA): BUS A ← R2
[2] MUX B selector (SELB): BUS B ← R3
[3] ALU operation selector (OPR): ALU to ADD
[4] Decoder destination selector (SELD): R1 ← Out Bus

Control Word

| 3 | 3 | 3 | 5 |
|---|---|---|---|
| SELA | SELB | SELD | OPR |

Encoding of register selection fields

| Binary Code | SELA | SELB | SELD |
|---|---|---|---|
| 000 | Input | Input | None |
| 001 | R1 | R1 | R1 |
| 010 | R2 | R2 | R2 |
| 011 | R3 | R3 | R3 |
| 100 | R4 | R4 | R4 |
| 101 | R5 | R5 | R5 |
| 110 | R6 | R6 | R6 |
| 111 | R7 | R7 | R7 |

## ALU CONTROL

### Encoding of ALU operations

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

### Examples of ALU Microoperations

| | Symbolic Designation | | | | |
|---|---|---|---|---|---|
| Microoperation | SELA | SELB | SELD | OPR | Control Word |
| $R1 \leftarrow R2 - R3$ | R2 | R3 | R1 | SUB | 010 011 001 00101 |
| $R4 \leftarrow R4 \vee R5$ | R4 | R5 | R4 | OR | 100 101 100 01010 |
| $R6 \leftarrow R6 + 1$ | R6 | - | R6 | INCA | 110 000 110 00001 |
| $R7 \leftarrow R1$ | R1 | - | R7 | TSFA | 001 000 111 00000 |
| Output $\leftarrow$ R2 | R2 | - | None | TSFA | 010 000 000 00000 |
| Output $\leftarrow$ Input | Input | - | None | TSFA | 000 000 000 00000 |
| $R4 \leftarrow$ shl R4 | R4 | - | R4 | SHLA | 100 000 100 11000 |
| $R5 \leftarrow 0$ | R5 | R5 | R5 | XOR | 101 101 101 01100 |

## BUS TRANSFER

- A BUS is the set of common lines (wires) in which each line is used for one bit of the register through which binary information is transferred at a time.

- A digital computer has many registers and path must be provided to transfer information from one register to another.

- The number of wires will be increased if separate lines are used between each register and all the other registers in the system.

- A more efficient scheme for transferring information between registers in a multiple register configuration in a common bus system.

- The bus structure contains of a set of common lines one for each bits of a register, through which binary information is transferred one at a time.

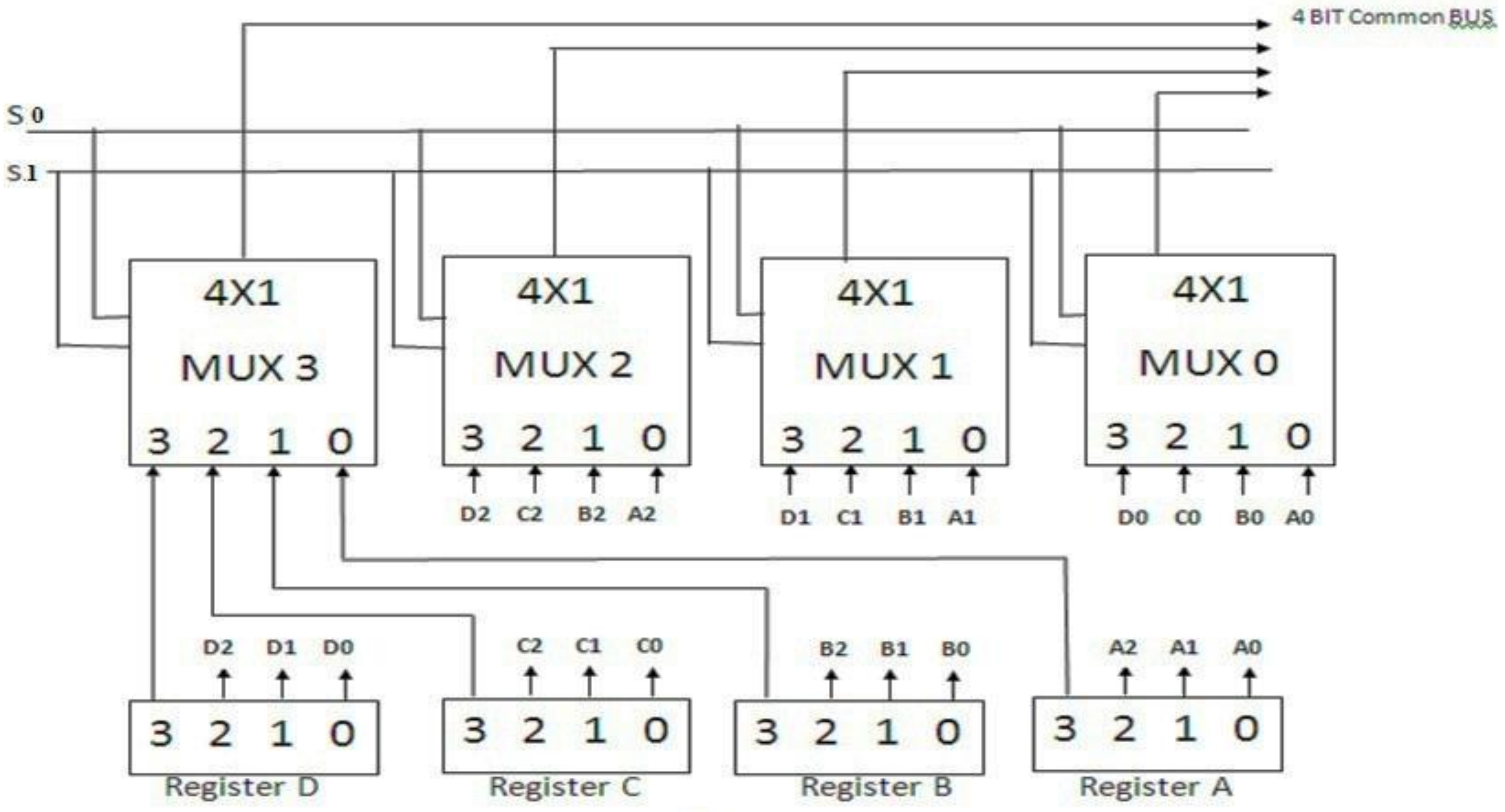

4 BIT Common BUS
S 0
S 1
4X1
MUX 3
3 2 1 0
4X1
MUX 2
3 2 1 0
D2 C2 B2 A2
4X1
MUX 1
3 2 1 0
D1 C1 B1 A1
4X1
MUX 0
3 2 1 0
D0 C0 B0 A0
D2 D1 D0
3 2 1 0
Register D
C2 C1 C0
3 2 1 0
Register C
B2 B1 B0
3 2 1 0
Register B
A2 A1 A0
3 2 1 0
Register A

fig 4-3

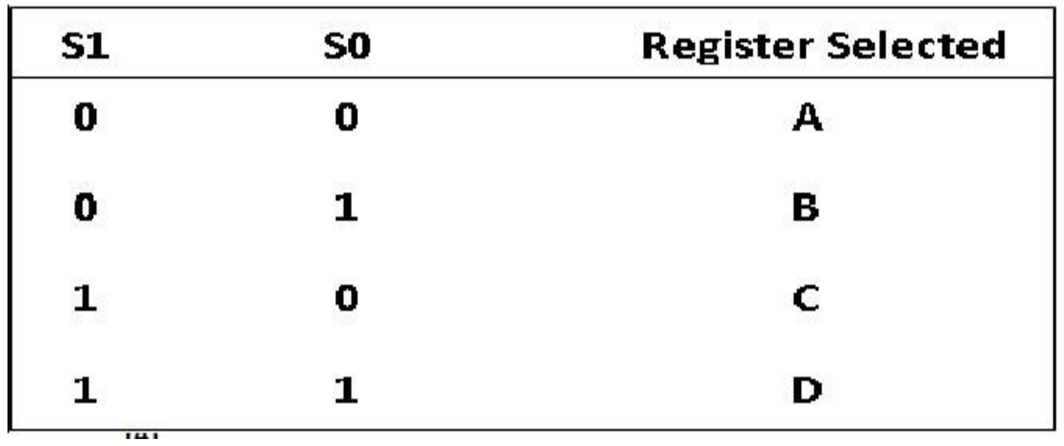
- The multiplexers select the source register whose binary information is then placed on the bus.

- Each register has four bits, numbered 0 through 3.

- The bus consists of four 4 x 1 multiplexers each having four data inputs, 0 through 3, and two selection inputs, S1 and S0.

- These two selection lines S1 and S0 are connected to the selection inputs of all four multiplexers.

▪The selection lines choose the four bits of one register and transfer them into the four- line common bus.



| S1 | S0 | Register Selected |
|---|---|---|
| 0 | 0 | A |
| 0 | 1 | B |
| 1 | 0 | C |
| 1 | 1 | D |

- A bus system will multiplex k registers of n bits each to produce an n-line common bus.

- The number of multiplexers needed to construct the bus is equal to n, the number of bits in each register.

- The size of each multiplexer must be K x 1 since it multiplexes K data lines.

  - Number of mux = Number of bits in the register.
  - Number of inputs in Mux = Number of registers.

## Question

- For a common bus for eight registers of 16 bits each requires how many mux?

- For a common bus system for eight registers what size of mux is required? And what number of selection lines are required?

## Question

A digital computer has a common bus system of 32 registers of 64 bits each. The bus is constructed with multiplexer.

a) How many selection inputs are there in each multiplexer?

b) What size of multiplexers are needed?

c) How many multiplexers are there in the bus?

Sol.

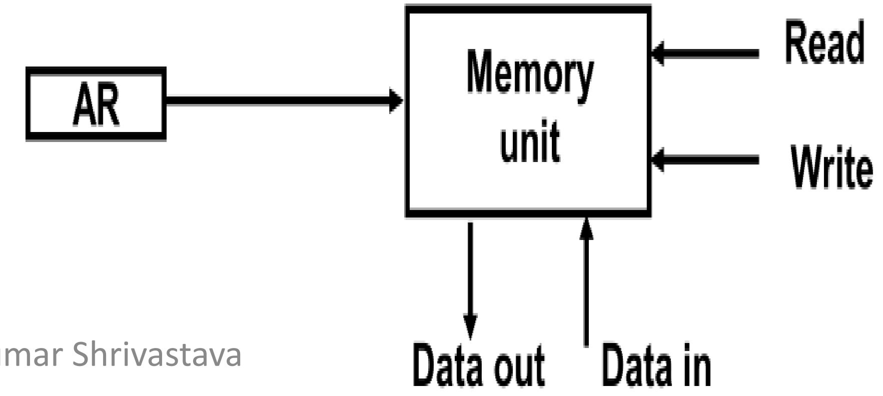
a) 5

b) 32 x 1

c) 64

## MEMORY TRANSFER

- **Read Operation:** The transfer of information from a memory word to the outside environment is called a read operation.
- **Write Operation:** The transfer of new information to be stored into the memory is called a write operation.
- A memory word will be symbolized by the letter M.
- This will be done by enclosing the address in square brackets following the letter M.



**Memory Read:** Consider a memory unit that receives the address from a register, called a register denoted by **AR.** The data are transferred to another register called data register denoted by **DR.**

The read operation can be stated as follows:-

**Read: DR $\leftarrow$ M[AR]**

This causes a transfer of information into DR from word M selected by the address in AR.

- **Memory Write:** The write operation transfers the content of a data register to a memory word M selected by the address. Consider that input data are in register R1 and address is in AR. The write operation can be stated symbolically as follows:

**Write: M[AR] $\leftarrow$ R1**

- This causes a transfer of information from R1 into the memory word M selected by address in AR.

## Micro-operations

- Digital systems are modular in nature, with modules containing registers, decoders, arithmetic elements, control logic, etc.

- These digital components are defined by the registers that they contain and the operations performed on their data. These operations are called micro operations.

- Micro operations are elementary operations performed on the information stored in one or more registers.

## Micro operations are classified into four categories:

- – **Register transfer micro operations** (data moves from register to register)
- – **Arithmetic micro operations** (perform arithmetic on data in registers)
- – **Logical micro operations** (perform bit manipulation on data in registers)
- – **Shift micro operations** (perform shift on data in registers)

## Arithmetic Microoperations

- Unlike register transfer microoperations, arithmetic microoperations change the information content.

- The basic arithmetic microoperations are:
  - addition
  - subtraction
  - increment
  - decrement

# Arithmetic Micro-operations

| **Symbolic Designation** | **Description** |
|---|---|
| $R3 \leftarrow R1+R2$ | Contents of R1 plus R2 transferred to R3 |
| $R3 \leftarrow R1-R2$ | Contents of R1 minus R2 transferred to R3 |
| $R2 \leftarrow R2'$ | Complement contents of R2 (1's comp.) |
| $R2 \leftarrow R2' + 1$ | 2's complement contents of R2 (negate) |
| $R3 \leftarrow R1+R2' + 1$ | R1 plus 2's comp. of R2 |
| $R1 \leftarrow R1 + 1$ | Increment content of R1 by 1 |
| $R1 \leftarrow R1 - 1$ | Decrement content of R1 by 1 |

## ARITHMETIC CIRCUIT

- Arithmetic circuit can be implemented in one composite arithmetic circuit. the basic component of an arithmetic circuit is parallel adder.

- We can implement 7 arithmetic micro-operations (add, add with carry, subtract, subtract with borrow, increment, decrement and transfer) with one circuit.

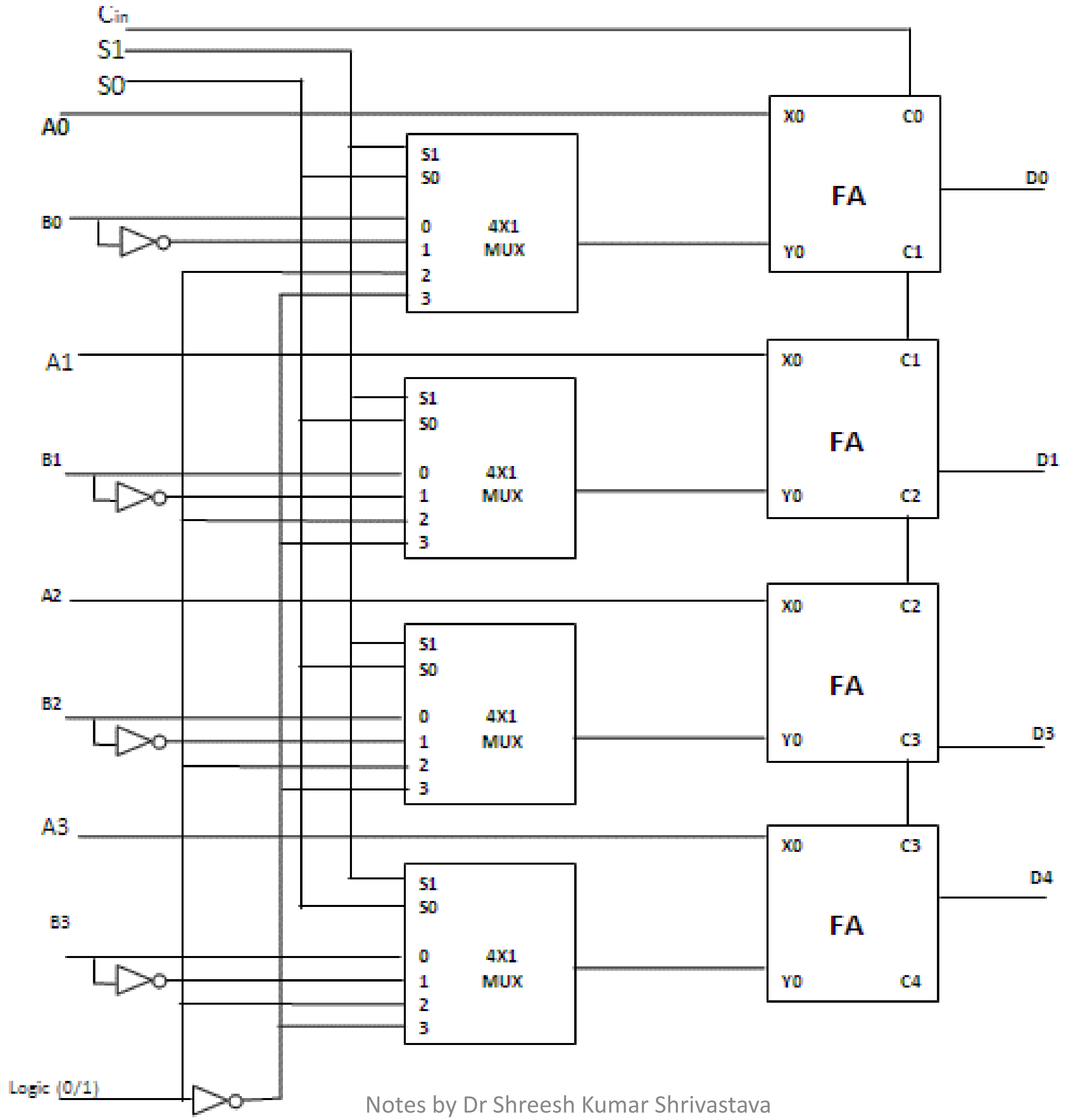
- Diagram of 4 bit parallel circuit is shown. This circuit is used for all arithmetic Micro operation in which 4 bit binary adder and 4 multiplexer are used.

- In each full adder , two inputs X & y are passed. The four A input is passed to the X input of full adder and four B input is passed to the Y input of each full adder through multiplexer.

- A and B are  4 bit numbers where A has $A_3A_2A_1A_0$ and B has $B_3B_2B_1B_0$.

- The four inputs of B has its normal and complement value which is input into 0, 1 position  input of each multiplexer .the input 0 and 1 is passed to the 2, 3 position of each multiplexer.

- A carry $C_{in}$ is passed to each full adder of 4 bit binary adder.


Cin
S1
S0
A0
B0
A1
B1
A2
B2
A3
B3
Logic (0/1)
S1
S0
0
1
2
3
4X1
MUX
X0
C0
FA
Y0
C1
C2
C3
C4
D0
D1
D3
D4

The four multiplexer uses two common selection lines s0, s1 to select the number of input value. In this arithmetic circuit the output of binary adder is calculated from the expression given below:-

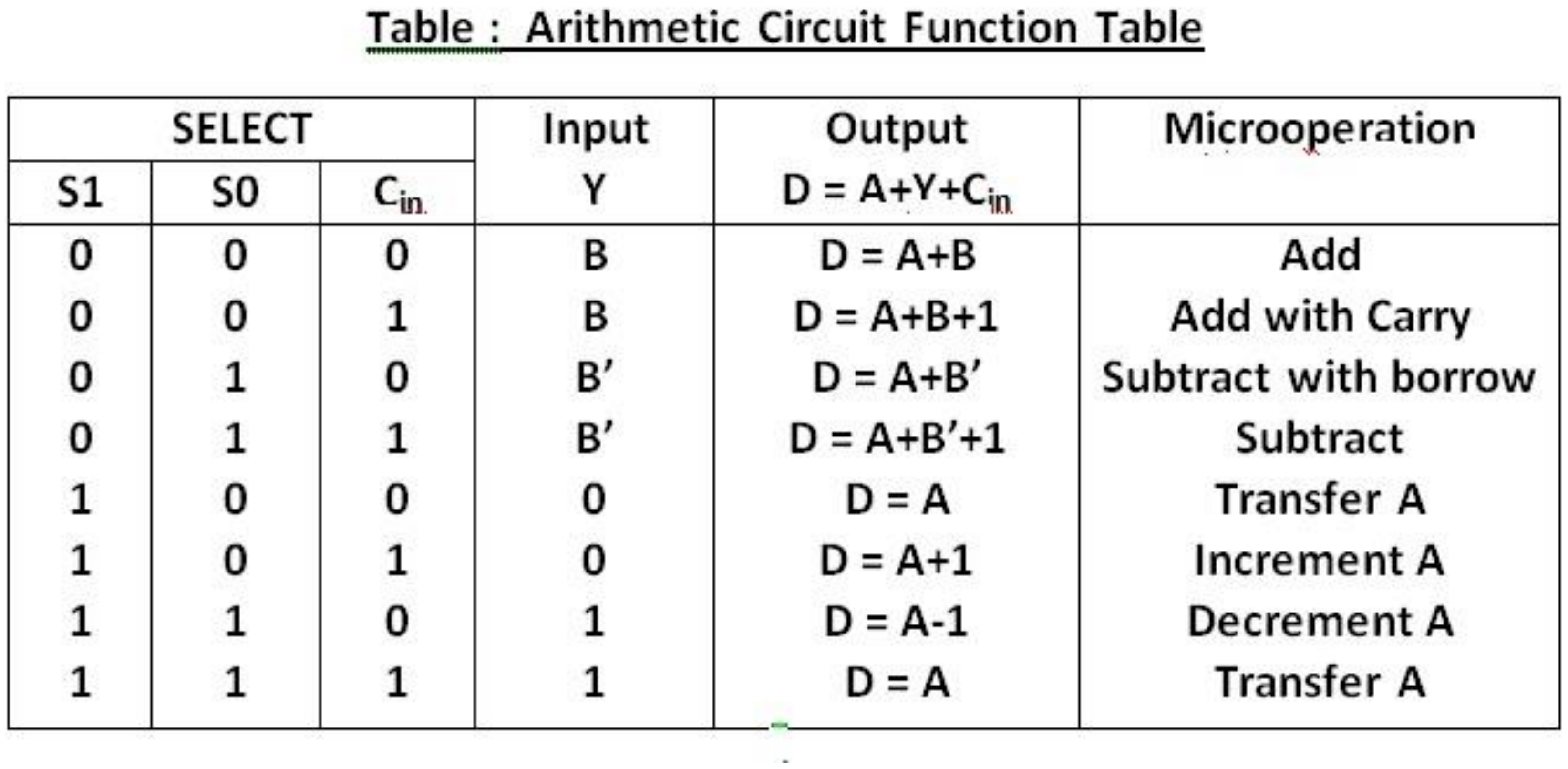


Table : Arithmetic Circuit Function Table

| SELECT | | | Input | Output | Microoperation |
|---|---|---|---|---|---|
| S1 | S0 | $C_{in}$ | Y | $D = A+Y+C_{in}$ | |
| 0 | 0 | 0 | B | D = A+B | Add |
| 0 | 0 | 1 | B | D = A+B+1 | Add with Carry |
| 0 | 1 | 0 | B’ | D = A+B’ | Subtract with borrow |
| 0 | 1 | 1 | B’ | D = A+B’+1 | Subtract |
| 1 | 0 | 0 | 0 | D = A | Transfer A |
| 1 | 0 | 1 | 0 | D = A+1 | Increment A |
| 1 | 1 | 0 | 1 | D = A-1 | Decrement A |
| 1 | 1 | 1 | 1 | D = A | Transfer A |

## Logic Microoperations

- Logic micro-operations are binary operations performed on corresponding bits of two bit strings.

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

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

- Special Symbols used for logic operations:

  $\wedge$ = AND  $\vee$ = OR  $\oplus$ = XOR

- 16 different logic operations with 2 binary variables.

- Truth tables for 16 functions of 2 variables and the corresponding 16 logic micro-operations are as follows:

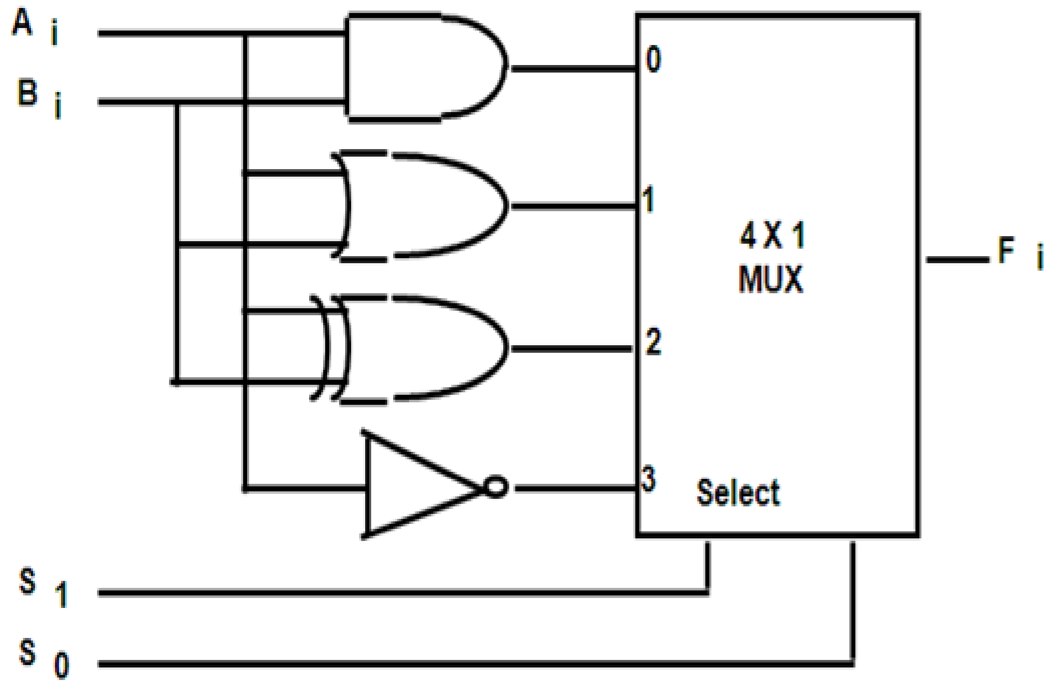
| x | 0 0 1 1 | *Boolean* | *Micro-* | *Name* |
|---|---|---|---|---|
| y | 0 1 0 1 | *Function* | *Operations* | |
| | 0 0 0 0 | $F0 = 0$ | $F \leftarrow 0$ | Clear |
| | 0 0 0 1 | $F1 = xy$ | $F \leftarrow A \wedge B$ | AND |
| | 0 0 1 0 | $F2 = xy'$ | $F \leftarrow A \wedge B'$ | |
| | 0 0 1 1 | $F3 = x$ | $F \leftarrow A$ | Transfer A |
| | 0 1 0 0 | $F4 = x'y$ | $F \leftarrow A' \wedge B$ | |
| | 0 1 0 1 | $F5 = y$ | $F \leftarrow B$ | Transfer B |
| | 0 1 1 0 | $F6 = x \oplus y$ | $F \leftarrow A \oplus B$ | Exclusive-OR |
| | 0 1 1 1 | $F7 = x + y$ | $F \leftarrow A \vee B$ | OR |
| | 1 0 0 0 | $F8 = (x + y)'$ | $F \leftarrow (A \vee B)'$ | NOR |
| | 1 0 0 1 | $F9 = (x \oplus y)'$ | $F \leftarrow (A \oplus B)'$ | Exclusive-NOR |
| | 1 0 1 0 | $F10 = y'$ | $F \leftarrow B'$ | Complement B |
| | 1 0 1 1 | $F11 = x + y'$ | $F \leftarrow A \vee B$ | |
| | 1 1 0 0 | $F12 = x'$ | $F \leftarrow A'$ | Complement A |
| | 1 1 0 1 | $F13 = x' + y$ | $F \leftarrow A' \vee B$ | |
| | 1 1 1 0 | $F14 = (xy)'$ | $F \leftarrow (A \wedge B)'$ | NAND |
| | 1 1 1 1 | $F15 = 1$ | $F \leftarrow$ all 1's | Set to all 1's |

## Draw a combinational circuit that perform four logical micro operations.

Function table

| $S_1$ | $S_0$ | Output | μ-operation |
|---|---|---|---|
| 0 | 0 | $F = A \wedge B$ | AND |
| 0 | 1 | $F = A \vee B$ | OR |
| 1 | 0 | $F = A \oplus B$ | XOR |
| 1 | 1 | $F = A'$ | Complement |

HARDWARE IMPLEMENTATION OF LOGIC MICROOPERATIONS



## Shift micro operations

- Shift micro operations are used for serial transfer of data.
- This operations shift contents of register in left or right direction.
- At the same time when the bits are shifted the last or flip-flop receives binary information from serial inputs.
- There are three types of shift operations:
    - Logical shift micro operation.
    - Circular shift micro operation
    - Arithmetic Shift micro operation

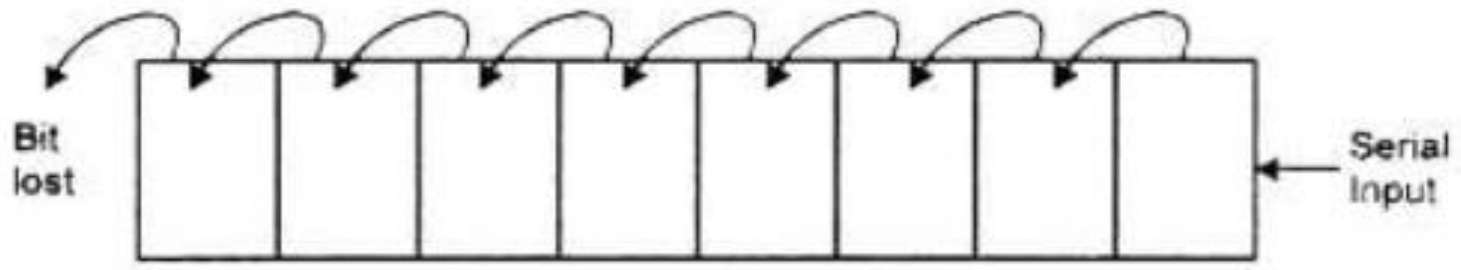
| Symbolic Designation | Description |
|---|---|
| $R \leftarrow$ shl R | Shift-left register R |
| $R \leftarrow$ shr R | Shift-right register R |
| $R \leftarrow$ cil R | Circular shift-left register R |
| $R \leftarrow$ cir R | Circular shift-right register R |
| $R \leftarrow$ ashl R | Arithmetic Shift-left register R |
| $R \leftarrow$ ashr R | Arithmetic Shift-right register R |

## Logical Shift

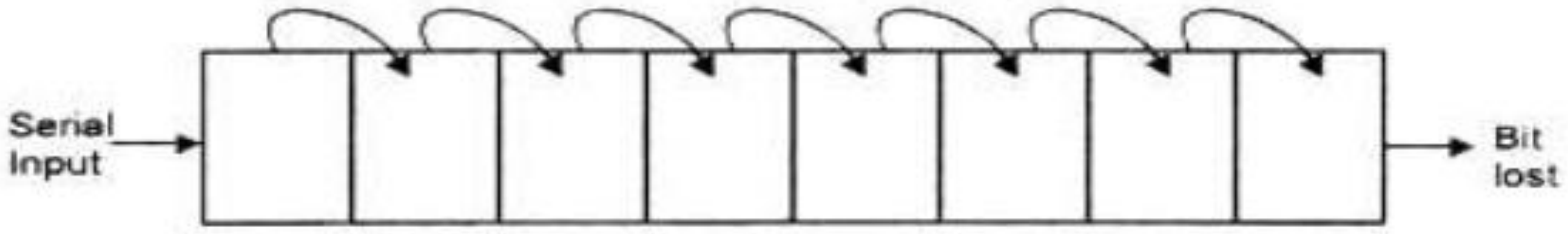
- The standard logical shifts shift all bits one bit position to the left or right and place a 0 in the leftmost or rightmost position, respectively.

- It can be defined in RTL by:
- $R \leftarrow shl\ R$ shift-left register R
- $R \leftarrow shr\ R$ shift-right register R

## example



The bits are shifted to words **left** by one bit position



The bits are shifted to words **right** by one bit position

## Circular shift

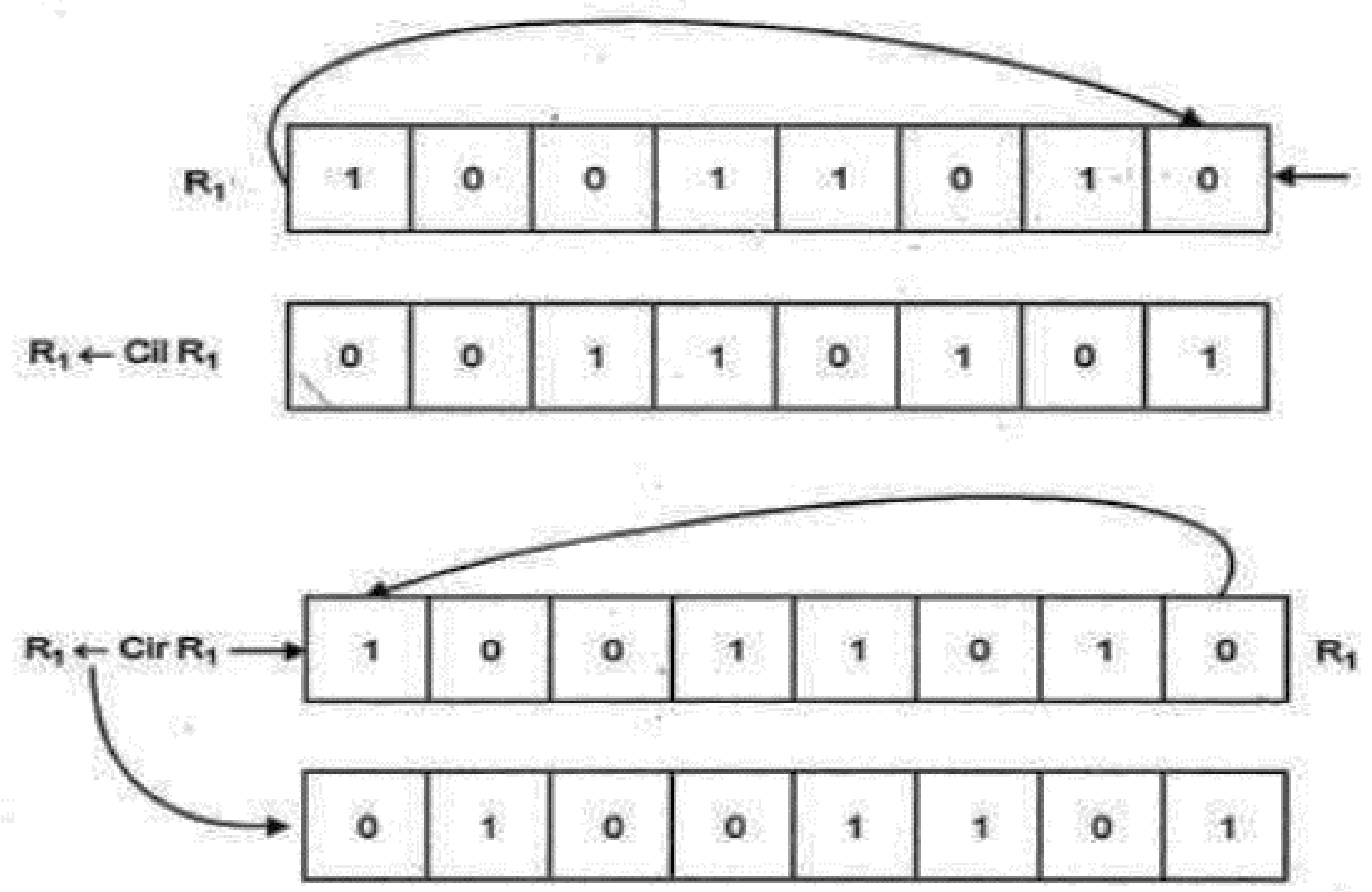
- The circular shift is also known as Rotate micro operation. It circulates the bit around both ends without any loss of information.

- It operate the same as a logical shifts except that instead of shifting a 0 into the end bit, the leftmost bit is shifted into the rightmost bit position on a CIL, and the rightmost bit is shifted into the leftmost bit position on a CIR.

- It can be defined in RTL by:

| | |
|---|---|
| $R \leftarrow$ cil l | circular shift-left register R |
| $R \leftarrow$ cir r | circular shift-right register R |

# Circular shift



## Arithmetic shift

- The arithmetic shift shifts a signed binary number to the left or right
- To the left is multiplying by 2, to the right is dividing by 2.
- Arithmetic shifts must leave the sign bit unchanged.
- The left bit hold the sign bit and others hold number.
- The sign bit 0 for positive and 1 for negative.
- It can be defined in RTL by:

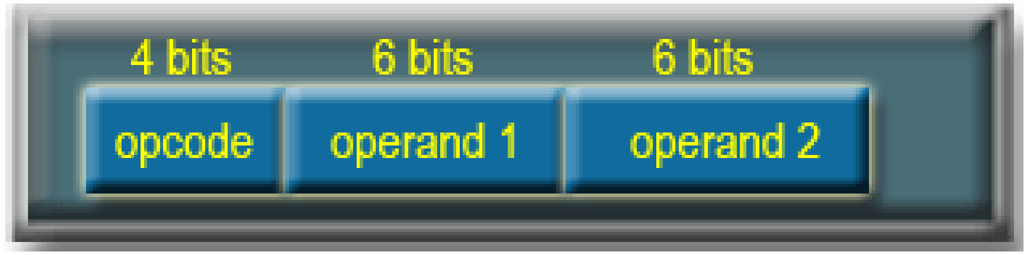
| | |
|---|---|
| R ← ashl R | arithmetic shift-left register R |
| R1 ← ashr R1 | arithmetic shift-right register R |

## Instruction Set

## Instruction Representation

Within the computer, each instruction is represented by a sequence of bits.

The instruction format is highly machine specific and it mainly depends on the machine architecture.



It is assume that it is a 16-bit CPU. 4 bits are used to provide the operation code. So, we may have to 16 ($2^4 = 16$) different set of instructions.

With each instruction, there are two operands. To specify each operands, 6 bits are used. It is possible to provide 64 ( $2^6 = 64$ ) different operands for each operand reference.

The elements of an instruction are as follows:

## Operation Code:

Specifies the operation to be performed (e.g., add, move etc.). The operation is specified by a binary code, know as the operation code or opcode.

## Source operand reference:

The operation may involve one or more source operands; that is, operands that are inputs for the operation.

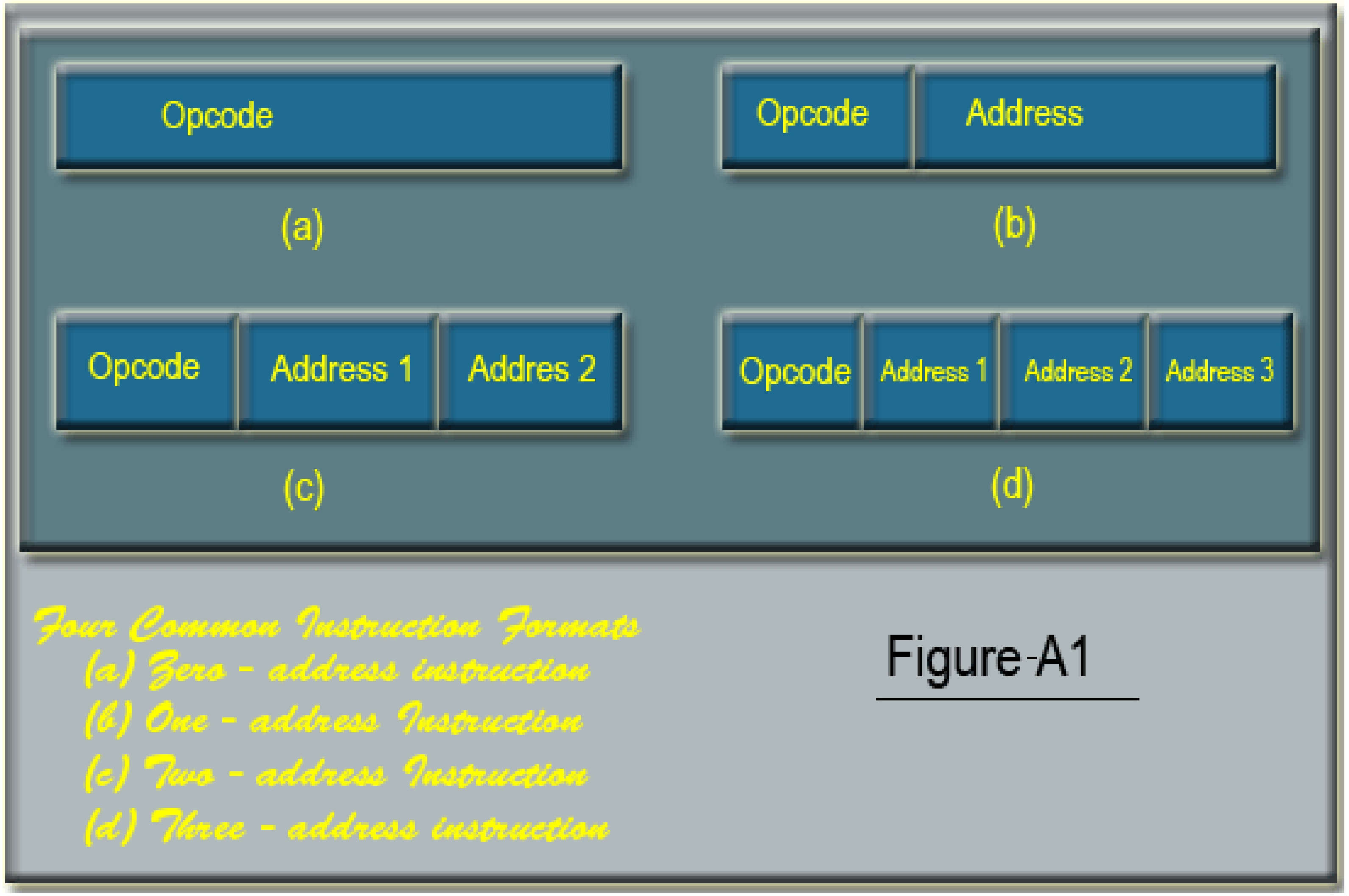
## Result operand reference:

The operation may produce a result.

## Next instruction reference:

This tells the CPU where to fetch the next instruction after the execution of this instruction is complete.

## Instruction Format:


Opcode
(a)
Opcode
Address
(b)
Opcode
Address 1
Addres 2
(c)
Opcode
Address 1
Address 2
Address 3
(d)
Four Common Instruction Formats
(a) Zero - address instruction
(b) One - address Instruction
(c) Two - address Instruction
(d) Three - address instruction
Figure-A1

## Instruction formats

- Three Address Instructions
- Two Address Instructions
- One Address Instructions
- Zero Address Instructions

- **Three Address Instructions:**

  - Computers with three-address instruction formats can use each address field to specify either a processor register or a memory operand.

  - The program in assembly language that evaluates X = (A + B) * (C + D) is shown below.

```
• ADD R1, A, B       R1 <= M [A] + M [B]
• ADD R2, C, D       R2 <= M[C] + M [D]
• MUL X, R1, R2      M[X] <= R1 * R2
```

## 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 |

## One Address Instructions:

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

| | | |
|---|---|---|
| **LOAD** | A | AC <= M [A] |
| ADD | B | AC <= AC + M [B] |
| **STORE** | T | M [T] <= AC |
| LOAD | C | AC <= M[C] |
| ADD | D | AC <= AC + M [D] |
| MUL | T | AC <= AC * M [T] |
| STORE | X | M [X] <= AC |

## Zero Address Instructions:

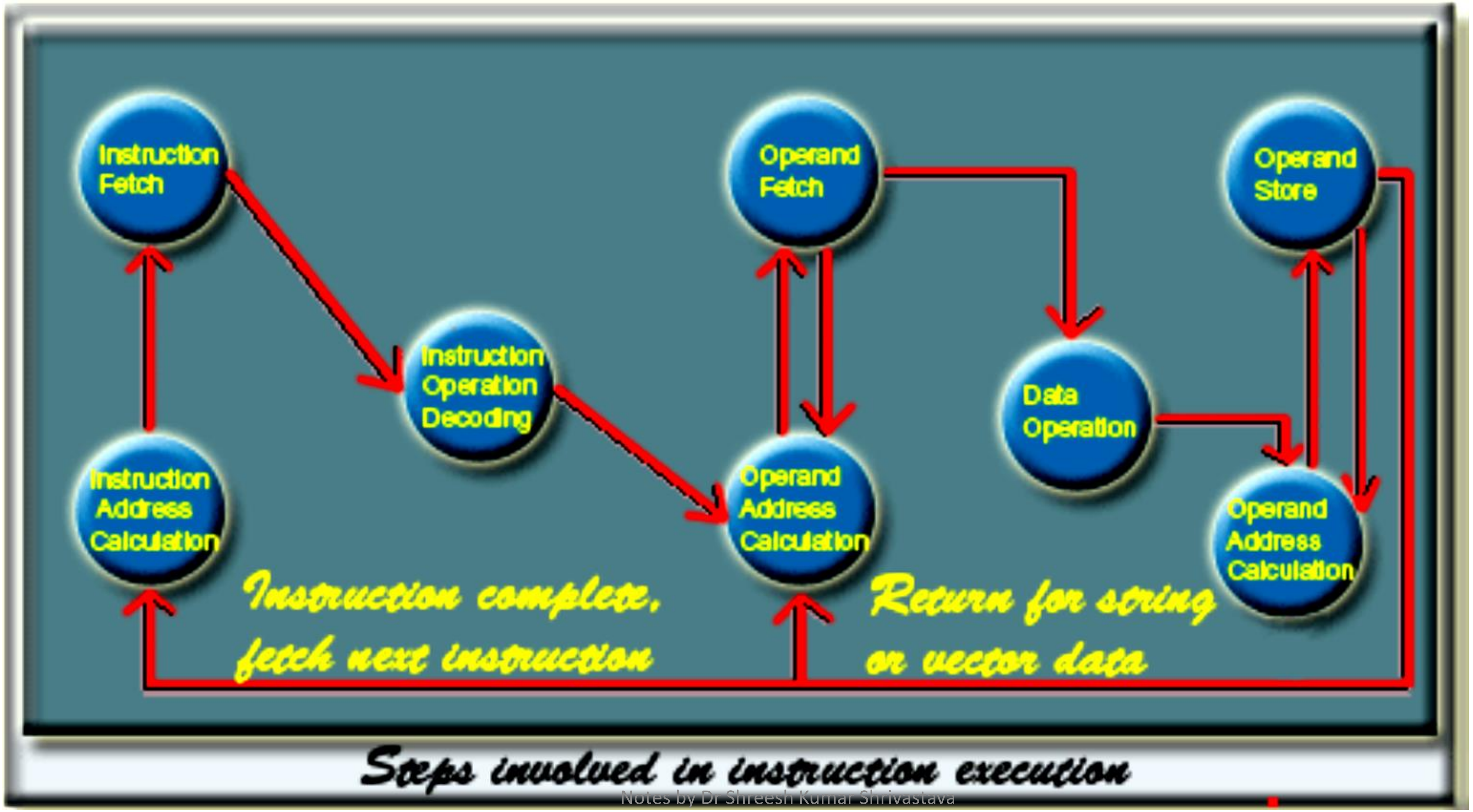
- A stack-organized computer does not use an address field for the instructions ADD and MUL.

- The PUSH and POP instructions, however, need an address field to specify the operand that communicates with the stack.

- The program to evaluate X = (A + B) * (C + D) will be written for a stack-organized computer.

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

## Steps involved in instruction execution


Instruction Fetch
Instruction Operation Decoding
Operand Fetch
Operand Store
Data Operation
Instruction Address Calculation
Operand Address Calculation
Operand Address Calculation
Instruction complete, fetch next instruction
Return for string or vector data
Steps involved in instruction execution

Op-codes are represented by abbreviations, called *mnemonics*, that indicate the operations.

Common examples include:

| | |
|---|---|
| ADD | : Add |
| SUB | : Subtract |
| MULT | : Multiply |
| DIV | : Division |
| LOAD | : Load data from memory to CPU |
| STORE | : Store data to memory from CPU. |

## Types of Operations

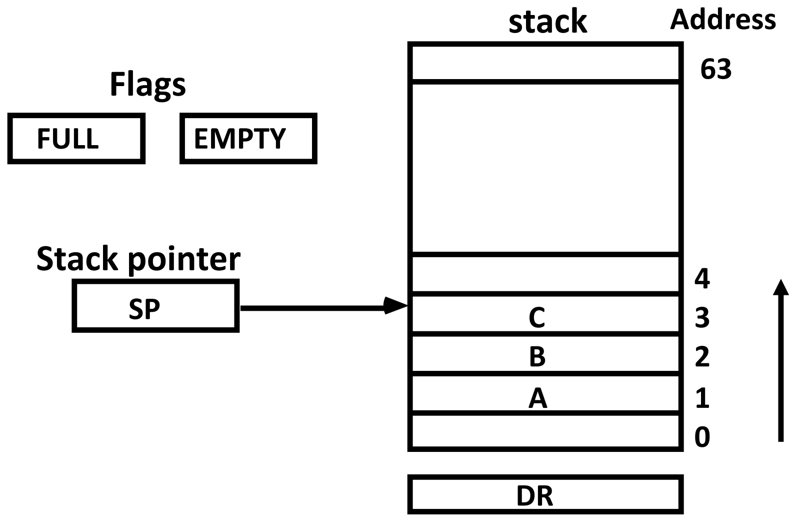
- Data Transfer
- Arithmetic
- Logical
- Conversion
- Input Output [ I/O ]
- Transfer Control

# STACK ORGANIZATION

Stack

- Efficient for arithmetic expression evaluation
- Storage which can be accessed in LIFO
- Pointer: SP
- Only PUSH and POP operations are applicable

Register Stack



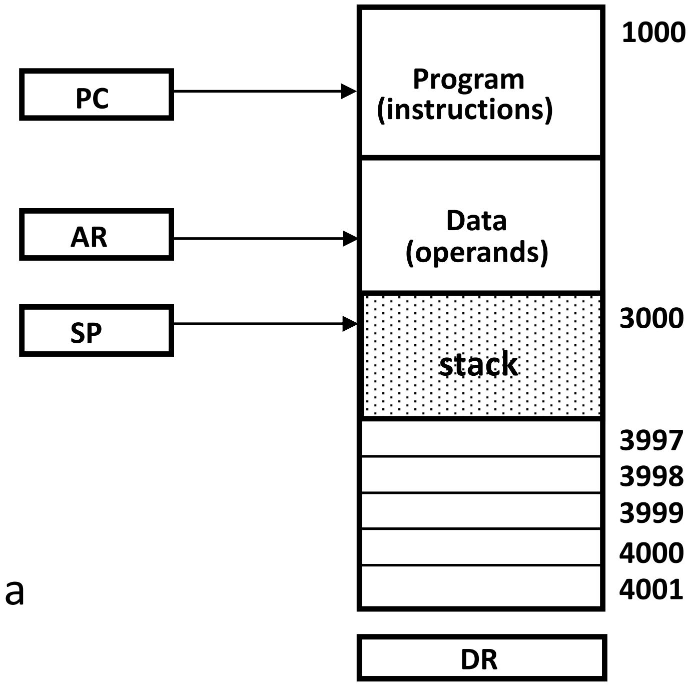
Push, Pop operations

**/* Initially, SP = 0, EMPTY = 1, FULL = 0 */**

| <u>**PUSH**</u> | <u>**POP**</u> |
|---|---|
| **SP ← SP + 1** | **DR ← M[SP]** |
| **M[SP] ← DR** | **SP ← SP - 1** |
| **If (SP = 0) then (FULL ← 1)** | **If (SP = 0) then (EMPTY ← 1)** |
| **EMPTY ← 0** | **FULL ← 0** |

# MEMORY STACK ORGANIZATION

Memory with Program, Data,
and Stack Segments



- A portion of memory is used as a stack with a processor register as a stack pointer

- PUSH: $SP \leftarrow SP - 1$
  $M[SP] \leftarrow DR$
- POP: $DR \leftarrow M[SP]$
  $SP \leftarrow SP + 1$

## ReversePolish Notation(RPN)

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

| | |
|---|---|
| • **A + B** | **Infix notation** |
| • **+ A B** | **Prefix or Polish notation** |
| • **A B +** | **Postfix or reverse Polish notation** |

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

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

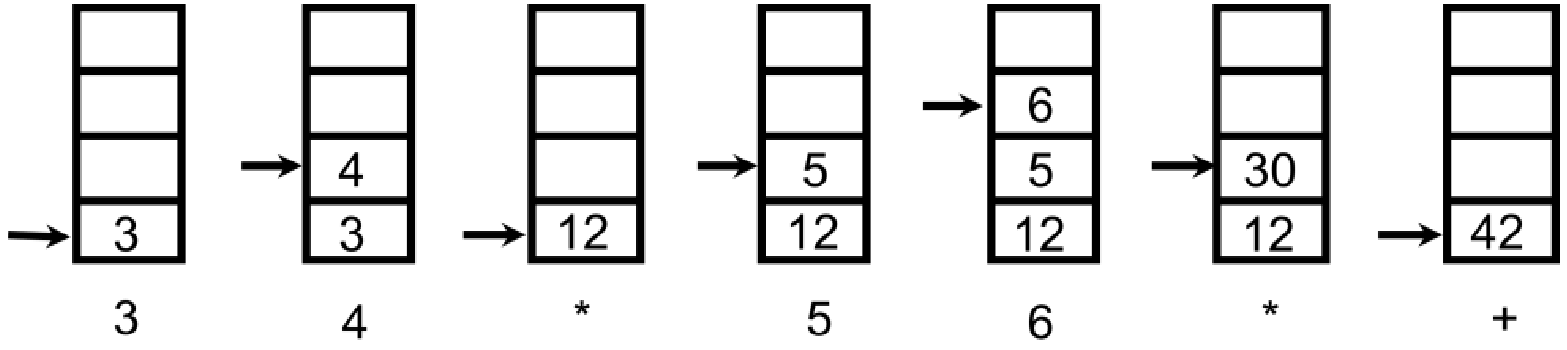
A B * C D * +

  - The conversion from infix notation to reverse Polish notation must take into consideration the operational hierarchy adopted for infix notation.

  - This hierarchy dictates that we first perform all arithmetic inside inner parentheses, then inside outer parentheses, and do multiplication and division operations before addition and subtraction operations.

- ***Evaluation of Arithmetic Expressions***
  - Any arithmetic expression can be expressed in parenthesis-free Polish notation, including reverse Polish notation

(3 * 4) + (5 * 6) = 3 4 * 5 6 * +



# Addressing Mode

**Ques.** How is the address of an operand specified, and how are the *bits* of an instruction organized to define the *operand addresses* and operation of that instruction.

The most common addressing techniques are:

- **Immediate**
- **Direct**
- **Indirect**
- **Register**
- **Register Indirect**
- **Displacement**
- **Stack**

## Immediate Addressing:

The simplest form of addressing is immediate addressing, in which the operand is actually present in the

**Instruction: Opcode Operand**
**Operand = data**

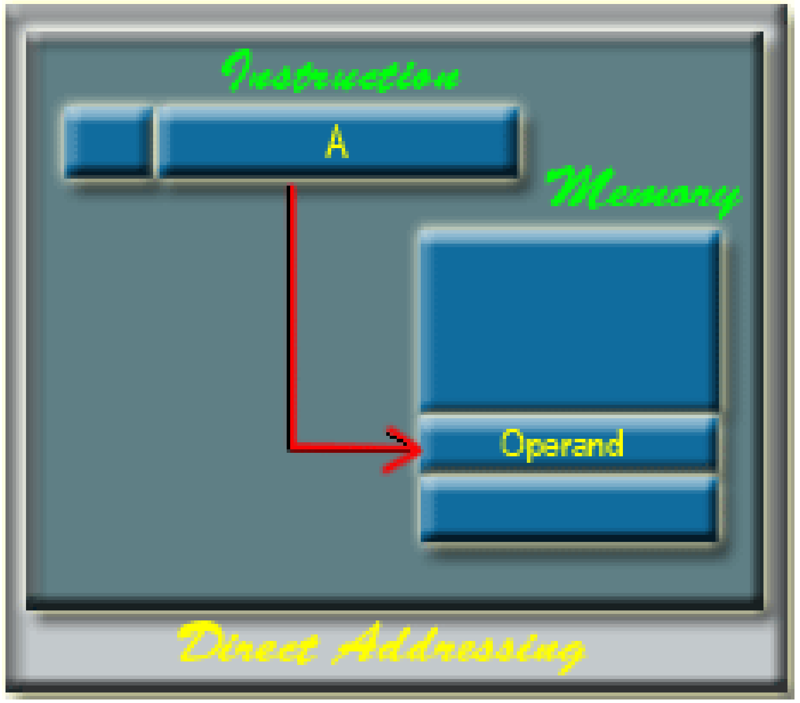
The advantage of immediate addressing is that no memory reference other than the instruction fetch is required to obtain the operand.

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

## Direct Addressing:

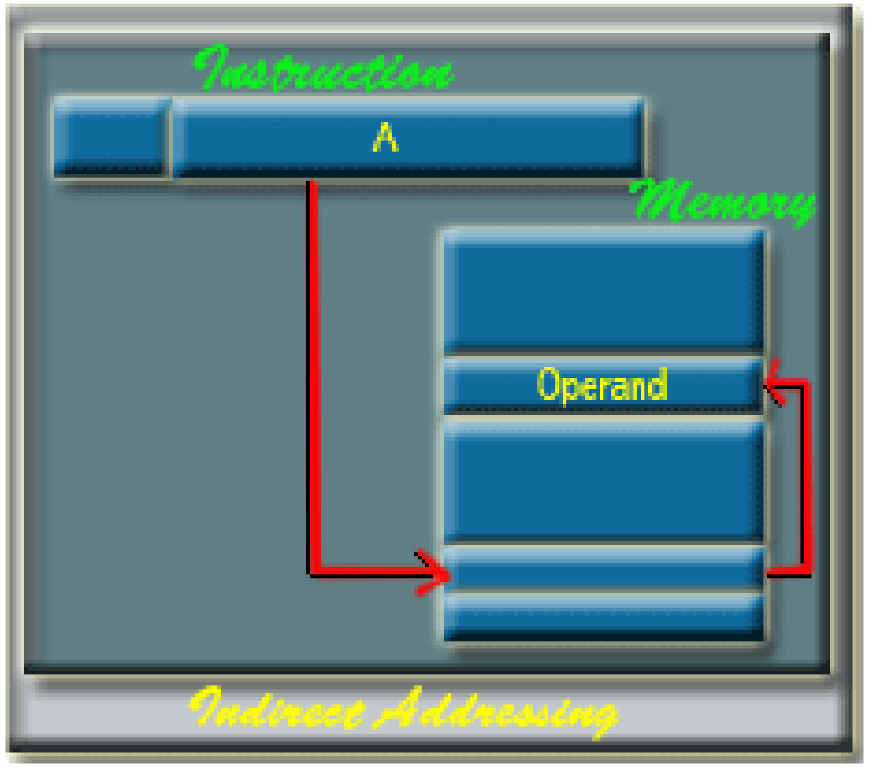
A very simple form of addressing is direct addressing, in which the address field contains the effective address of the operand:

It requires only one memory reference and no special calculation.



## Indirect Addressing:

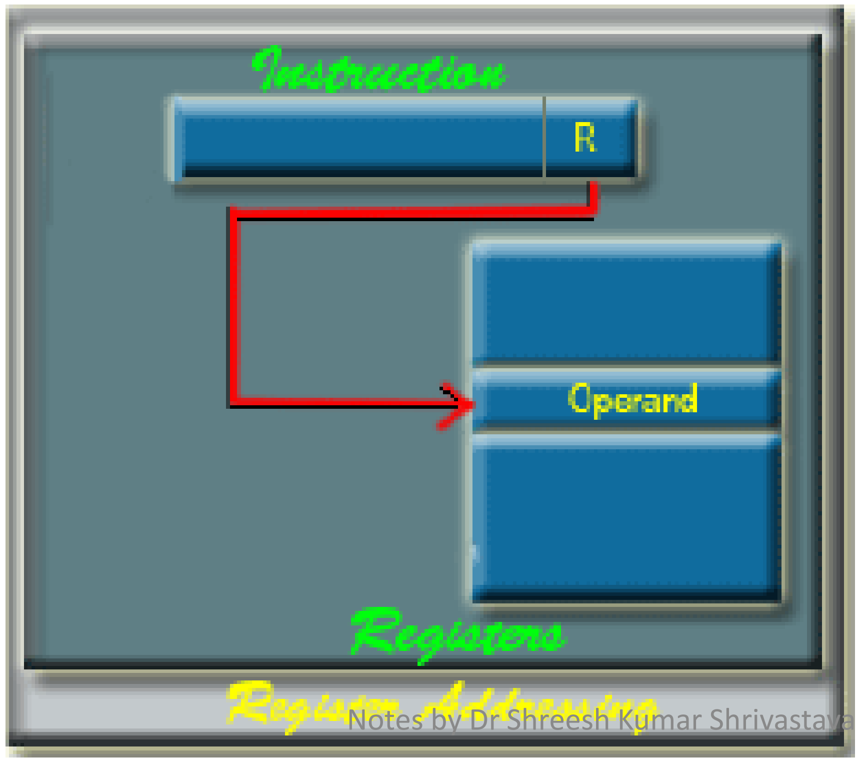
In indirect addressing, the address field refer to the address of a word in memory, which in turn contains a full-length address of the operand. This is know as indirect addressing:



## Register Addressing:

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

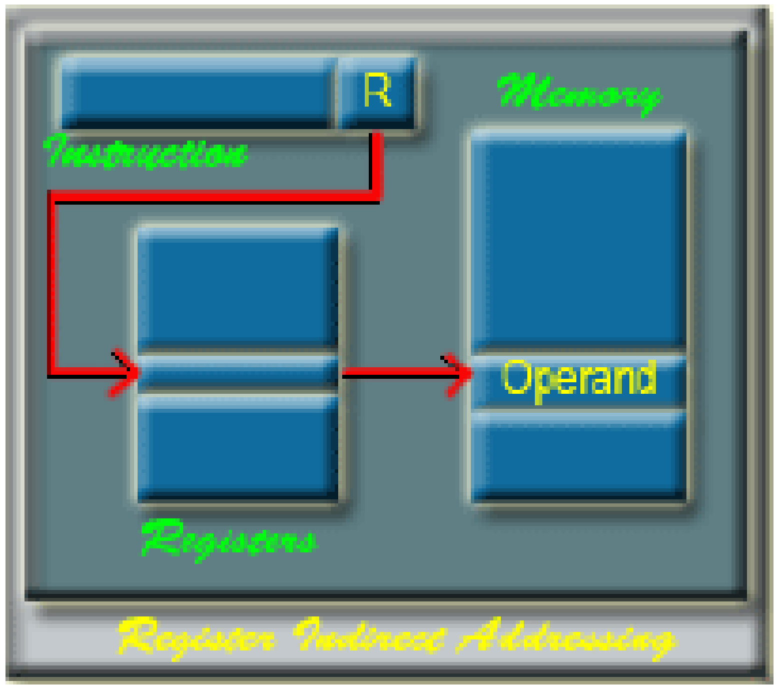
The advantages of register addressing are that only a small address field is needed in the instruction and no memory reference is required.



## Register Indirect Addressing:

Register indirect addressing is similar to indirect addressing, except that the address field refers to a register instead of a memory location.

It requires only one memory reference and no special calculation.

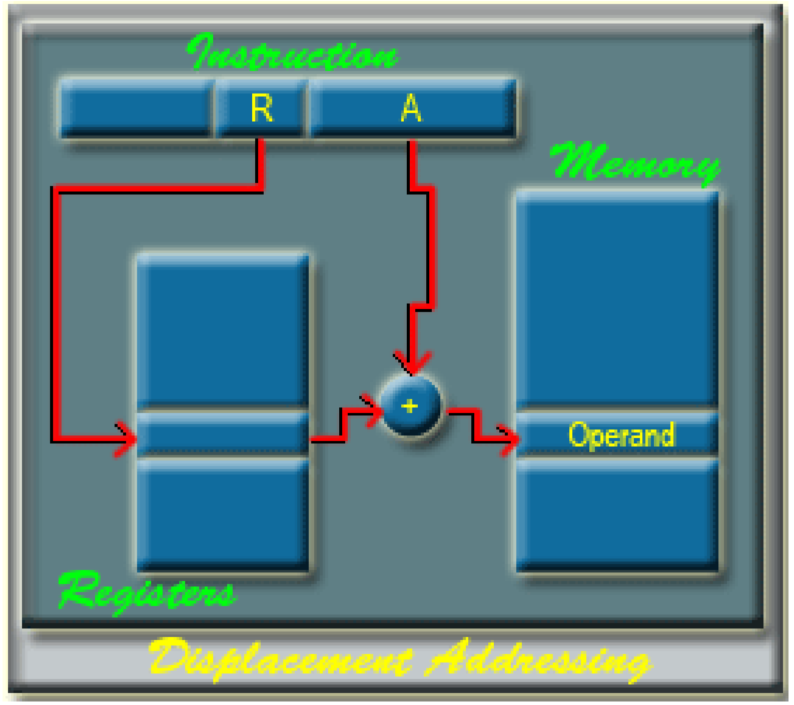


## Displacement Addressing:

A very powerful mode of addressing combines the capabilities of direct addressing and register indirect addressing, which is broadly categorized as displacement addressing:

$$= A + (R)$$

The value contained in one address field (value = A) is used directly. The other address field, refers to a register whose contents are added to A to produce the effective address.



Three of the most common use of displacement addressing are:

- **Relative addressing**
- **Base-register addressing**
- **Indexing**

**Relative Addressing:**

For relative addressing, the implicitly referenced register is the program counter (PC). That is, the current instruction address is added to the address field to produce the actual address. Thus, the effective address is a displacement relative to the address of the instruction.

**Base-Register Addressing:**

The reference register contains a memory address, and the address field contains a displacement from that address.

**Indexing:**

- The address field references a main memory address, and the reference register contains a positive displacement from that address.
- In this case also the register reference is sometimes explicit and sometimes implicit.
- Generally index register are used for iterative tasks, it is typical that there is a need to increment or decrement the index register after each reference to it.

In some common operation, many system will automatically do this as part of the same instruction cycle.

This is known as *auto-indexing*. We may get two types of auto-indexing:

- auto-incrementing
- auto-decrementing

Auto-indexing using increment can be depicted as follows:

$$EA = A + (R)$$

$$R = (R) + 1$$

Auto-indexing using decrement can be depicted as follows:

$$EA = A + (R)$$

$$R = (R) - 1$$

## Stack Addressing:

- A stack is a linear array or list of locations. It is sometimes referred to as a push down list or last-in-first-out queue.
- A stack is a reserved block of locations.
- Stack is associated a pointer whose value is the address of the top of the stack.
- The stack pointer is maintained in a register. Thus, references to stack locations in memory are in fact register indirect addresses.
- The stack mode of addressing is a form of implied addressing. The machine instructions need not include a memory reference but implicitly operate on the top of the stack.

Thank you