

# **Computer Organisation and Architecture**

**Course Code: CO206**

# Objective

To provide knowledge about the principles, concepts and applications of Computer Organization and Architecture.

# Course Description(Module-wise)

Module No	Title of Module	Number of Lectures
1	Introduction	8
2	Control Unit	8
3	Central Processing Unit	9
4	Input / Output Organisation	4
5	Modes of Data Transfer	5
6	Memory	8

# Recommended Reading Material

## Text Books:

1. Patterson, Computer Organization and Design, Elsevier Pub,2009
2. Morris Mano, Computer System Architecture, PHI
3. William Stallings, Computer Organization, PHI

## Reference Books:

1. Vravice, Hamacher&Zaky, Computer Organization, TMH
2. Tannenbaum, Structured Computer Organization, PHI

# Evaluation Criteria

Component	Description
Credits	4
Contact Hours	Theory: 3 Hours      Tutorial: 1 Hours
Relative Weightage	CWS: 25      MTE: 25      ETE: 50
Total	100

Component	Sub Components
Class Work Sessional(CWS)	Presentation, Seminar, Viva, Attendance
Mid Term Examination(MTE)	Innovative Project
End Term Examination(ETE)	CT1,CT2,CT3, Surprise Test1, Surprise Test 2

# Contents of Module 1

- Digital computer generation
- Computer types and classifications
- Functional units and their interconnections
- Bus architecture
- Types of buses and bus arbitration
- Register transfer language
- Data movement around registers
- Data movement from/to memory
- Arithmetic and logic micro-operations
- Concept of bus and timing in register transfer

# Preface

- Who is generally known as “Father of Digital Computers”?
  - Charles Babbage.
- Which is World’s first general purpose electronic digital computer?
  - ENIAC: Electronic Numerical Integrator and Computer(1946).
- What is a Computer?
  - It is an electronic device that takes some input, process it and gives some output.
- What is Computer Architecture?
  - It refers to those attributes that have a direct impact on the logical execution of a program.
- What is Computer Organization?
  - It refers to the operational units and their interconnections that realize the architectural specifications

# Difference between Computer Organisation & Architecture

Computer Architecture	Computer Organisation
<b>What does the system do?</b>	<b>How does the system do?</b>
<b>Deals with High level design issue</b>	<b>Deals with Low level design issue</b>
<b>Describes the functional behaviour</b>	<b>Describes the structural relationship</b>
<b>While designing a computing system, computer architecture is approached first</b>	<b>Computer organisation is approached after we have finalised the computer architecture</b>
<b>It involves instruction sets, addressing modes and data types</b>	<b>It involves the circuit design, signals, ALU, CPU and memory</b>

# Digital Computer Generation

<b>Generation</b>	<b>Approximate Dates</b>	<b>Technology</b>	<b>Typical Speed (operations per second)</b>
1	1946–1957	Vacuum tube	40,000
2	1958–1964	Transistor	200,000
3	1965–1971	Small and medium scale integration	1,000,000
4	1972–1977	Large scale integration	10,000,000
5	1978–1991	Very large scale integration	100,000,000
6	1991-	Ultra large scale integration	1,000,000,000

# First Generation-Vacuum Tubes (1946-1957)

- It had more than **1800 Vacuum tubes** and **1500 relays**
- Able to perform nearly **5000 additions or subtraction per second**
- It was a **decimal rather than a binary machine**
- It had **memory of 20 accumulators**, each capable of **storing a ten digit decimal number.**
- It had weight of **30 tons covering area of 15000 sq. ft** with power consumption of **140kW.**
- John Von Neumann introduced concept of stored program to design computer- **EDVAC(Electronic Discrete Variable Computer)**
- **Assembly language was used to prepare programs** and was translated into machine language for execution

## Second Generation-Transistors (1958-1964)

- Used transistors(semiconductor device which are smaller, cheaper and dissipates less heat)
- Greater speed, Larger memory capacity, smaller size than first generation
- Magnetic disks were used for secondary memory
- They had separate I/O processors having direct access to main memory
- It could handle both floating point and fixed point operations.
- Support higher level programming languages(Fortran)
- Provision of system software(compilers)
- Concept of multiprogramming was developed in systems

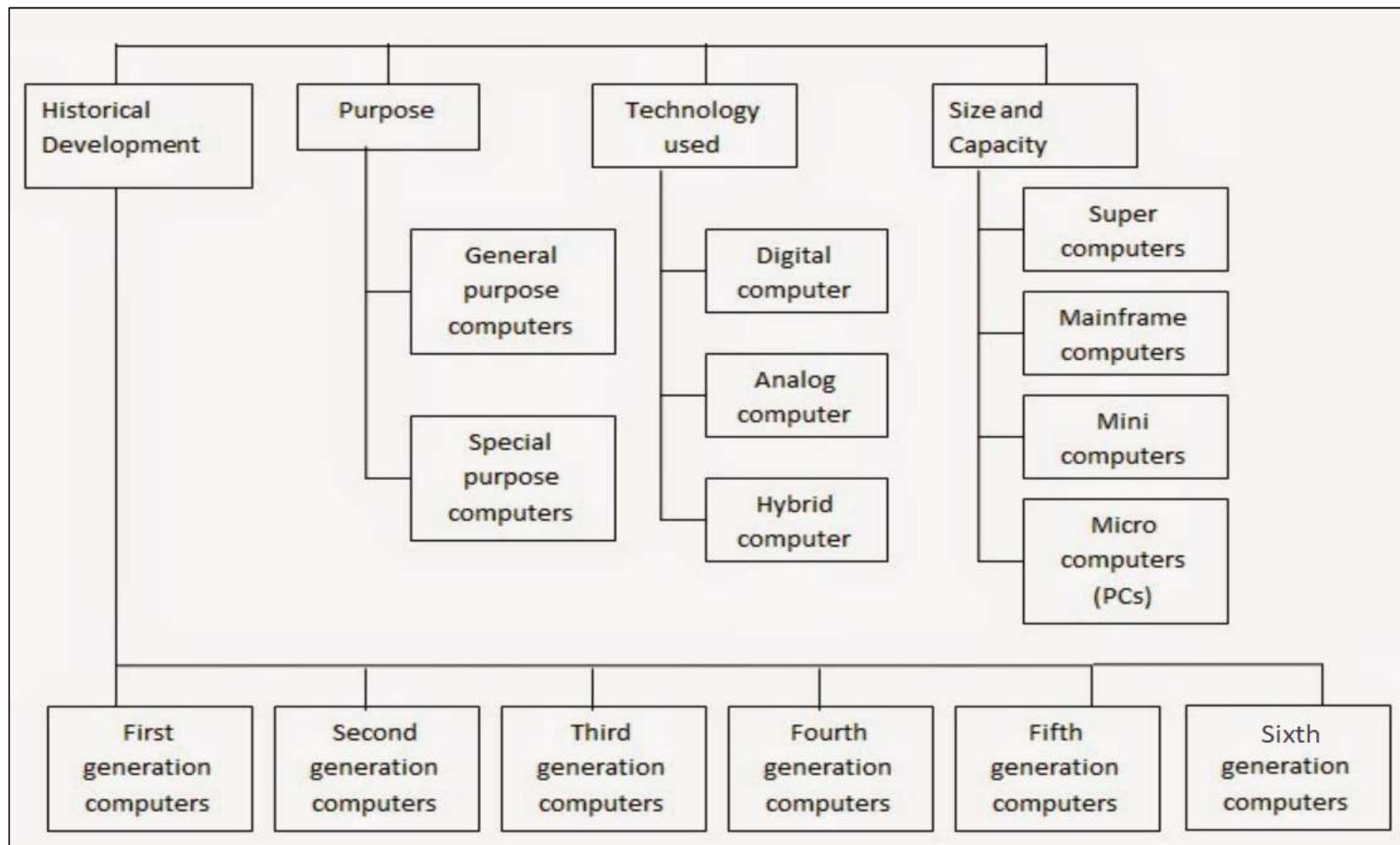
## Third Generation-Small and Medium Scale Integration (1965-1971)

- Used Integrated circuits(ICs)
- ICs enabled **lower cost, faster processors** and development of chips
- **Magnetic core** memories were **replaced by ICs memories**
- Various techniques introduced
  - Microprogramming
  - Parallel Processing
  - Sharing resources

# Later Generation

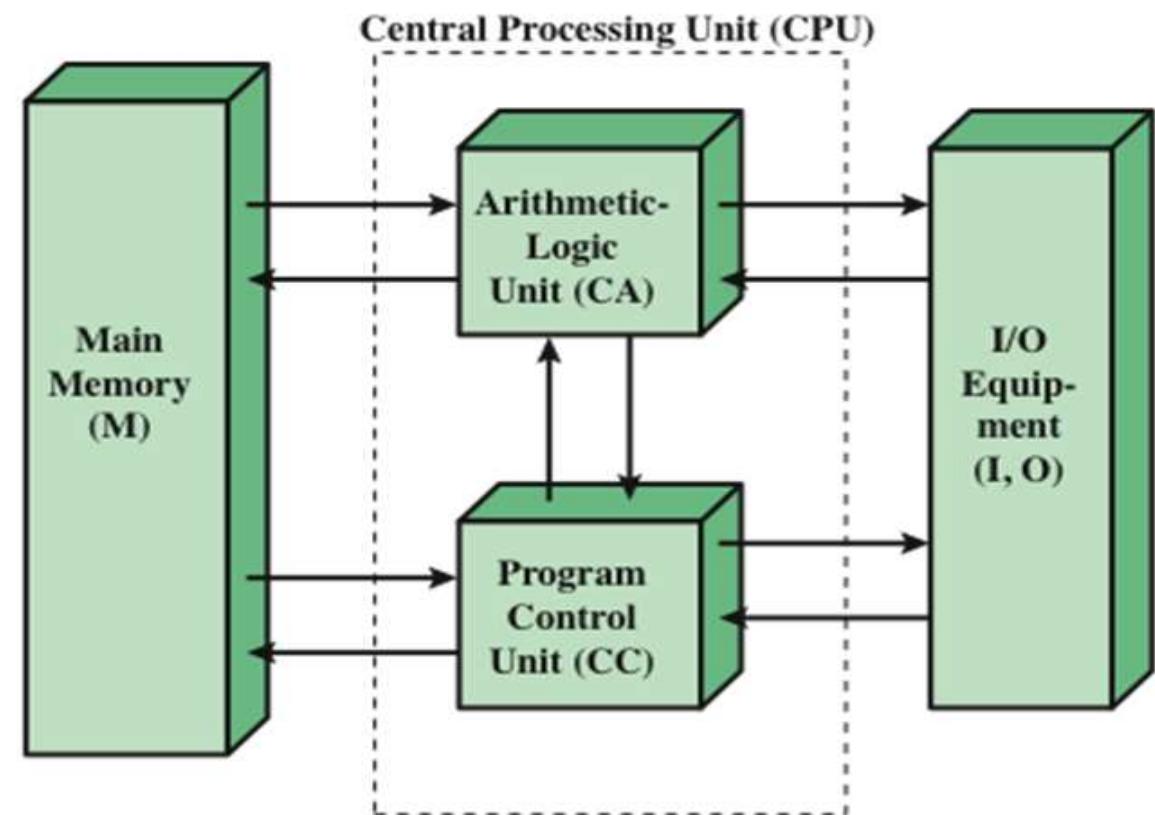
- Beyond the third generation there is less general agreement on defining generations of computers.
- **Large Scale Integration(1972-1977)**
  - More than 1000 components can be placed on a single integrated circuit chip
- **Very Large Scale Integration(1978-1991)**
  - More than 10,000 components per chip
- **Ultra Large Scale Integration(1991-..)**
  - More than one billion components

# Computer types and classification



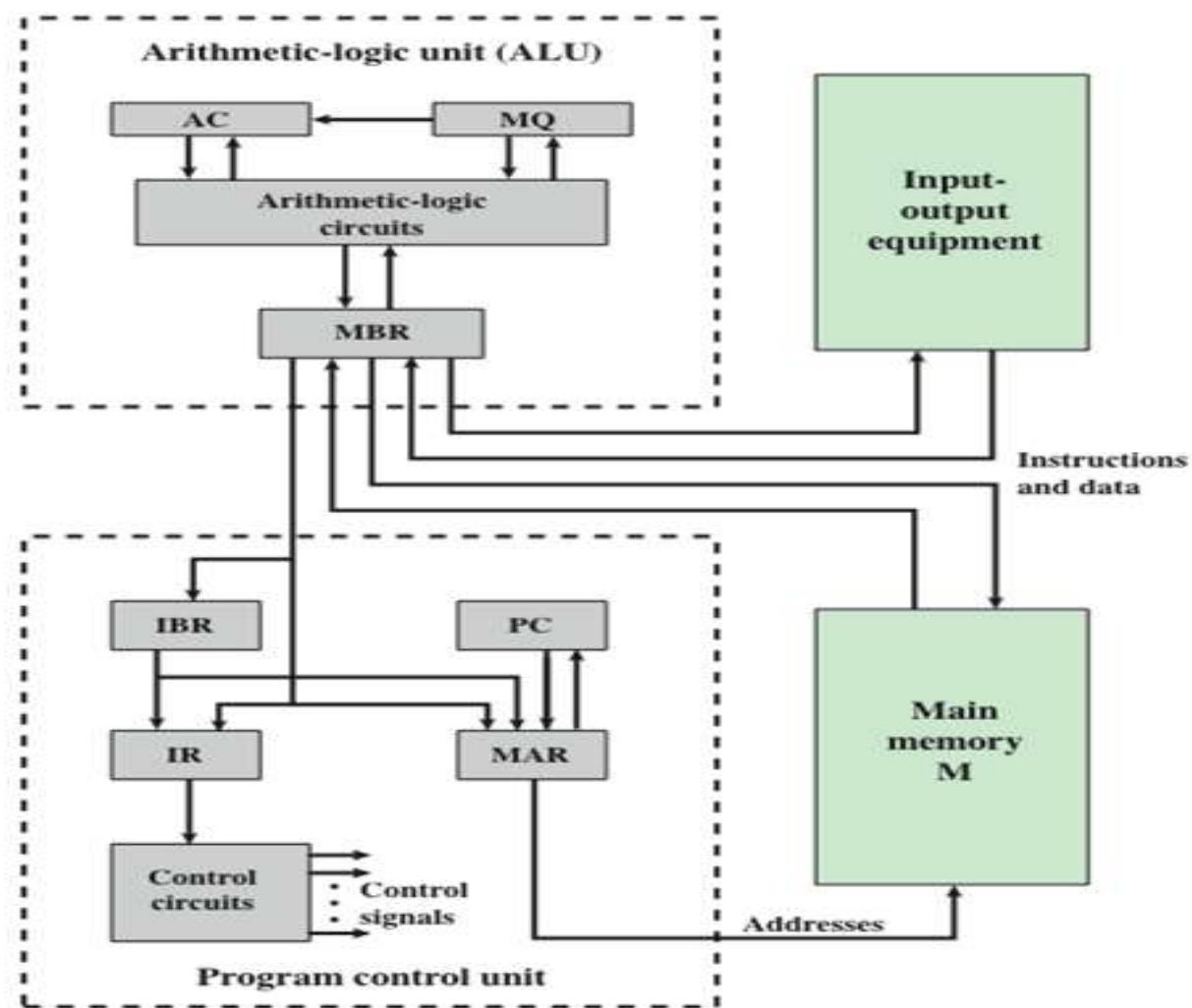
# Von Neumann Architecture

- In 1946, von Neumann and his colleagues began the design of a new stored program Computer
- Also referred to as the IAS computer, at the Princeton Institute for Advanced Studies.
- The IAS computer, although not completed until 1952, is the prototype of all subsequent general-purpose computers.



CA: Central Arithmetical  
CC: Control Center

# Internal Structure of Von Neumann Architecture



# Components in Von Neumann Architecture

## Accumulator (AC) and Multiplier quotient (MQ)

- Employed to temporarily hold operands and results of ALU operations

## Memory buffer register (MBR)

- Contains a word to be stored in memory or sent to the I/O unit
- It is used to receive a word from memory or from the I/O unit

## Instruction buffer register (IBR)

- Employed to temporarily hold the right-hand instruction from a word in memory

## Program counter (PC)

- Contains the address of the next instruction pair to be fetched from memory

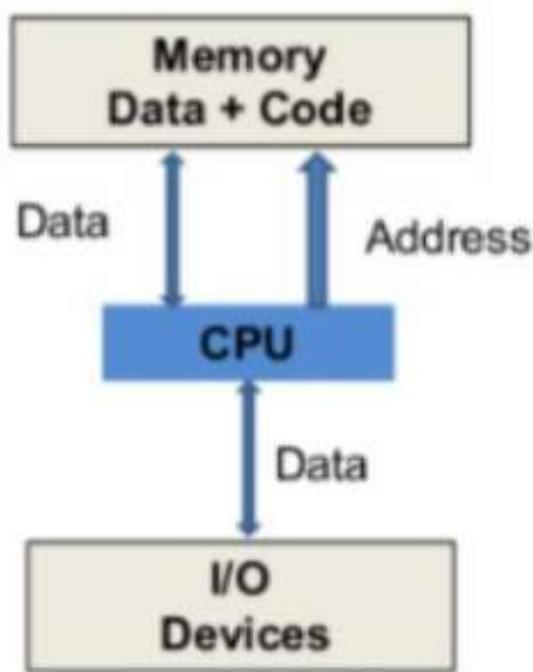
## Memory address register (MAR)

- Specifies the address in memory of the word to be written from or read into the MBR

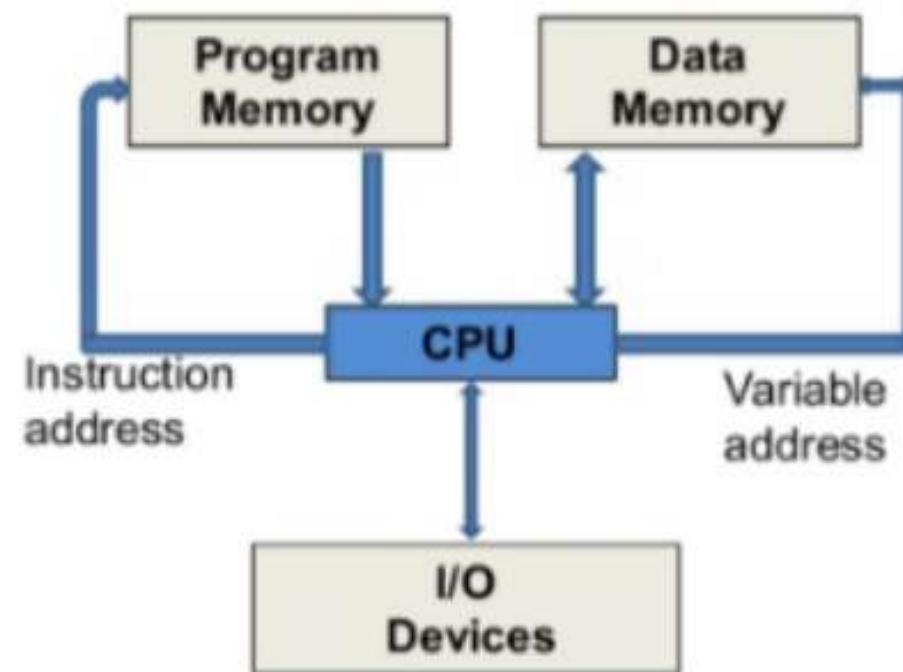
## Instruction register (IR)

- It is used to hold the instructions that is currently being executed

# Von Neumann Architecture Vs. Harvard Architecture



Von Neumann Architecture



Harvard Architecture

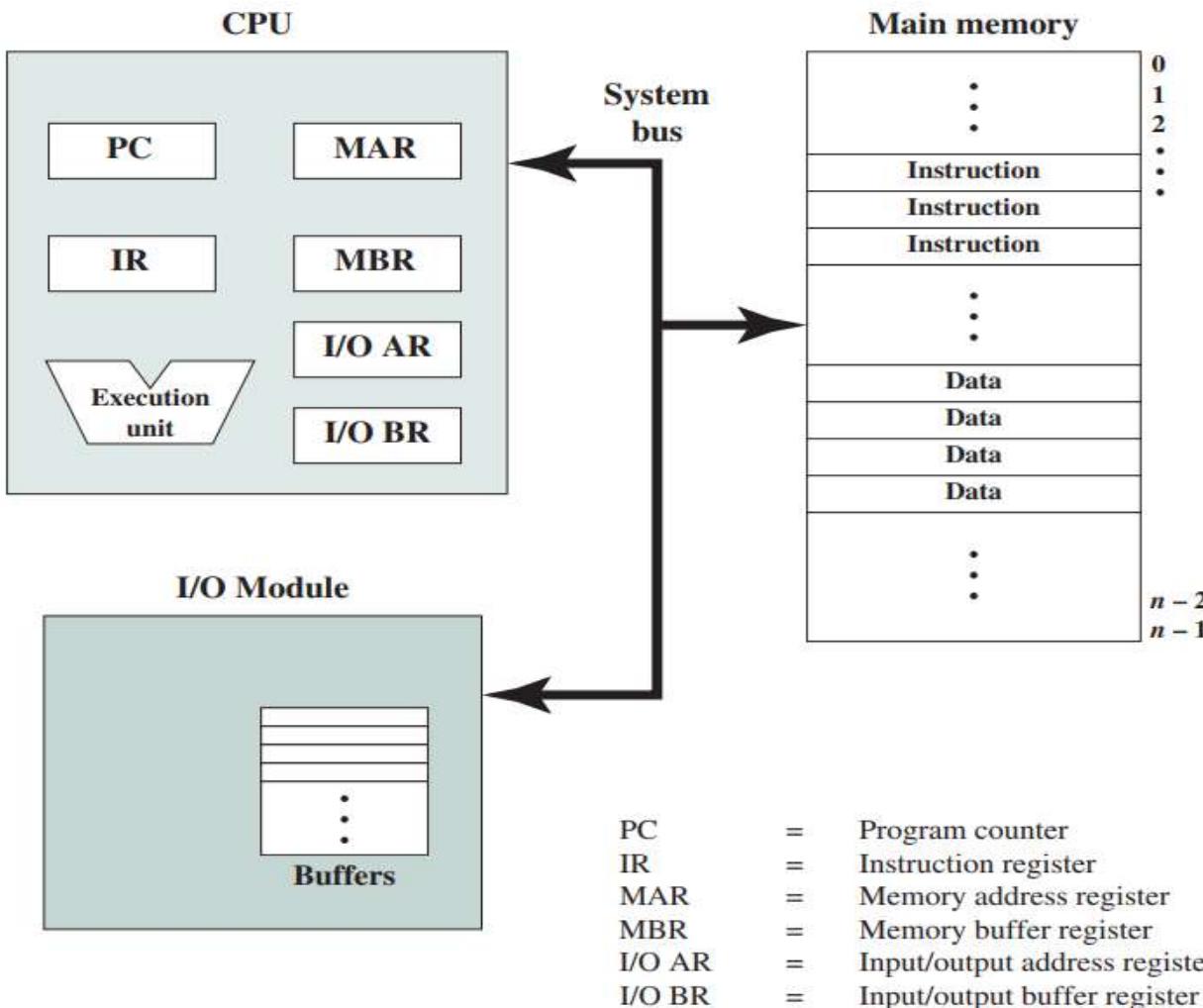
# Von Neumann Architecture Vs. Harvard Architecture

Von Neumann Architecture	Harvard Architecture
<b>Only one main memory</b>	<b>Two memory units i.e. one for data and another for program</b>
<b>System with one bus design is simpler</b>	<b>System with two bus is complex</b>
<b>Free memory space can be used for data or program</b>	<b>Free memory space in data cant be used for program or vice versa</b>
<b>Economical design</b>	<b>Costlier design</b>
<b>Commonly used in PCs, Laptops, workstations</b>	<b>Used primarily for small embedded systems and signal processing</b>

# Von Neumann Bottleneck/Memory Wall

- ❑ Because of the stored program architecture of the Von-Neumann Machine, the processor performance is tightly bound to the memory performance.
- ❑ We need to access memory at least once per cycle to read an instruction, the processor can only operate as fast as the memory.
- ❑ This is sometimes known as Von Neumann bottleneck or Memory wall.

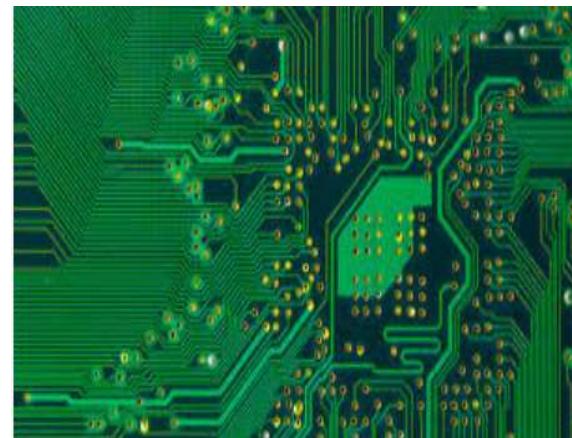
# Functional Units and their Interconnections



Source: *Computer Organisation and Architecture, Tenth Edition*, by William Stallings

# Bus

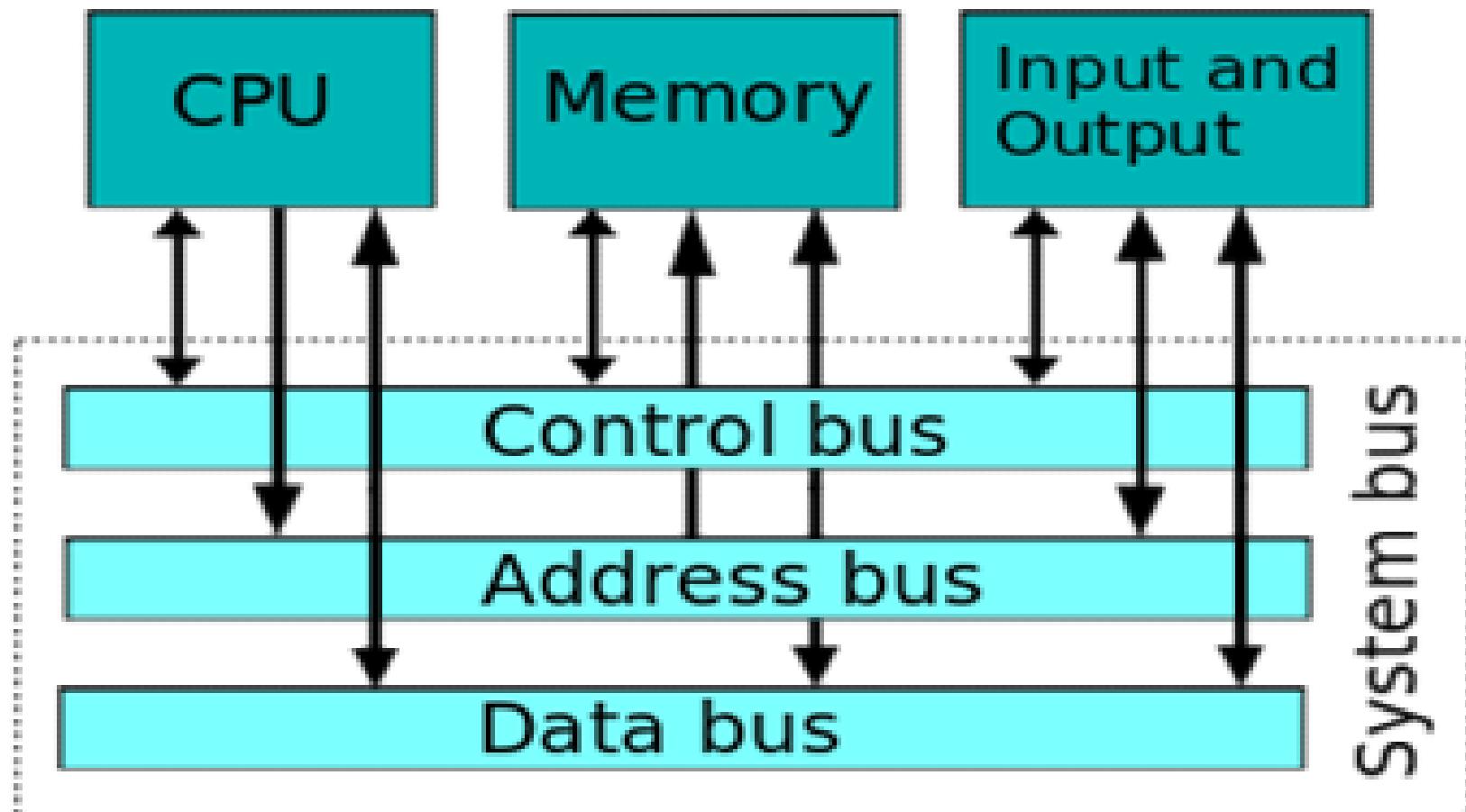
- ❑ A group of wires that connects two or more hardware devices for communication is called a BUS.
- ❑ It can be a cable or printed in circuit.
- ❑ It reduces the number of pathways between interconnecting hardware components
- ❑ A bus that connects the major components with CPU i.e. memory & I/O is called System Bus.
- ❑ The term “**width**” is used to refer the number of bits that a bus can transmit at once



# Some major functions of Bus in Computers

- ❑ Data transfer
- ❑ Addressing
- ❑ Timing/Syncronisation
- ❑ Power

# Bus Architecture



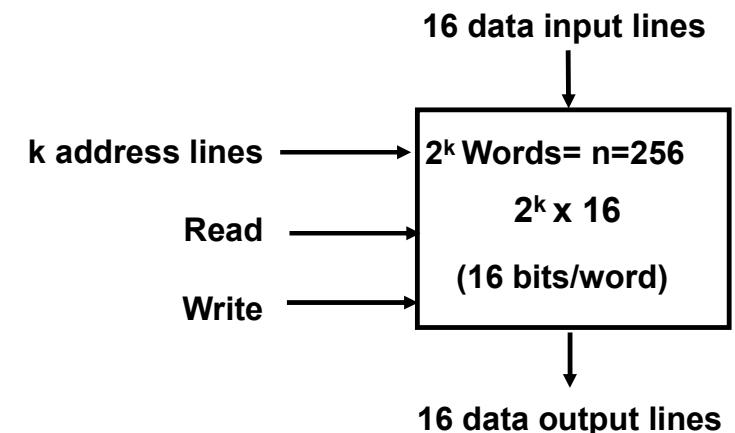
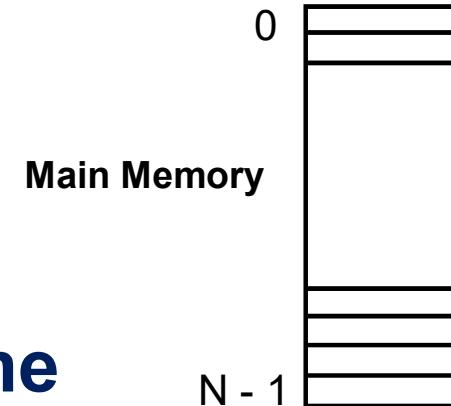
# Types of Bus(on the basis of contents)

- Data Bus
- Address Bus
- Control Bus

# Memory Components

## Main Memory

- Each word has a unique address
- Access to a word requires the same time independent of the location of the word



# Data Bus (Data Lines)

- ❑ Carries data throughout the system
- ❑ These are bi-directional lines
- ❑ Width is a key factor.
  - ❑ It determines the number of bytes that can be transferred in one cycle and hence the overall performance of the system.

# Address Bus (Address Lines)

- ❑ Designate source or destination of data on the bus
- ❑ It is an unidirectional bus.
- ❑ Width determines the maximum possible memory capacity of the system.
- ❑ Also used to address I/O ports
  - ❑ High order bits selects a particular module
  - ❑ Lower order bits select a memory location or I/O port within the memory

# Control Bus (Control Lines)

- ❑ Used to provide synchronization between CPU and other system components
- ❑ Control lines include
  - ❑ Memory read and Memory write
  - ❑ I/O read and I/O write
  - ❑ Transfer Acknowledgment
  - ❑ Bus request and bus grant
  - ❑ Interrupt request and interrupt acknowledgement
  - ❑ Clock
  - ❑ Reset

# Representation of Data in Memory

- ❑ If we need to store a data=11234567 in the memory then its representation can be done in two ways

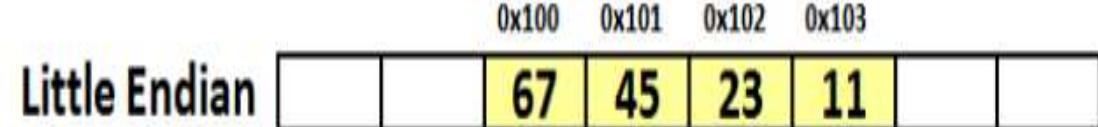
## ❑ Big Endian

- ❑ MSB will be stored first
- ❑ Intel, ARM processors

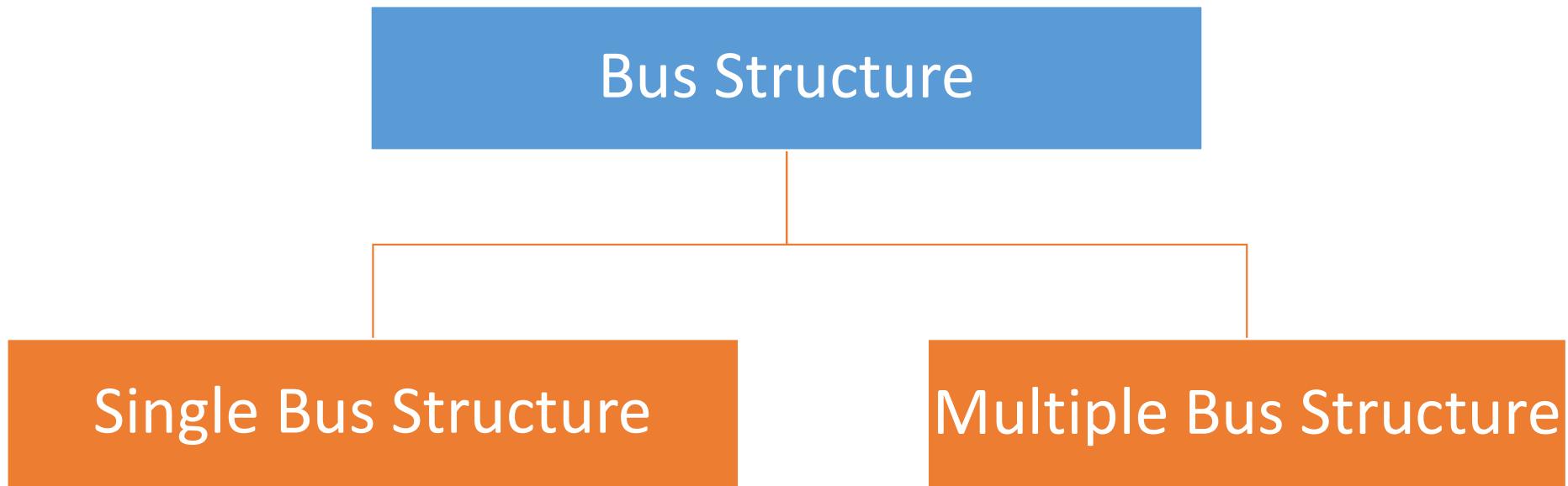
## ❑ Little Endian

- ❑ LSB will be stored first
- ❑ Motorola, PowerPC, SPARK

MSB(Most Significant Bits)      LSB(Least Significant Bits)  

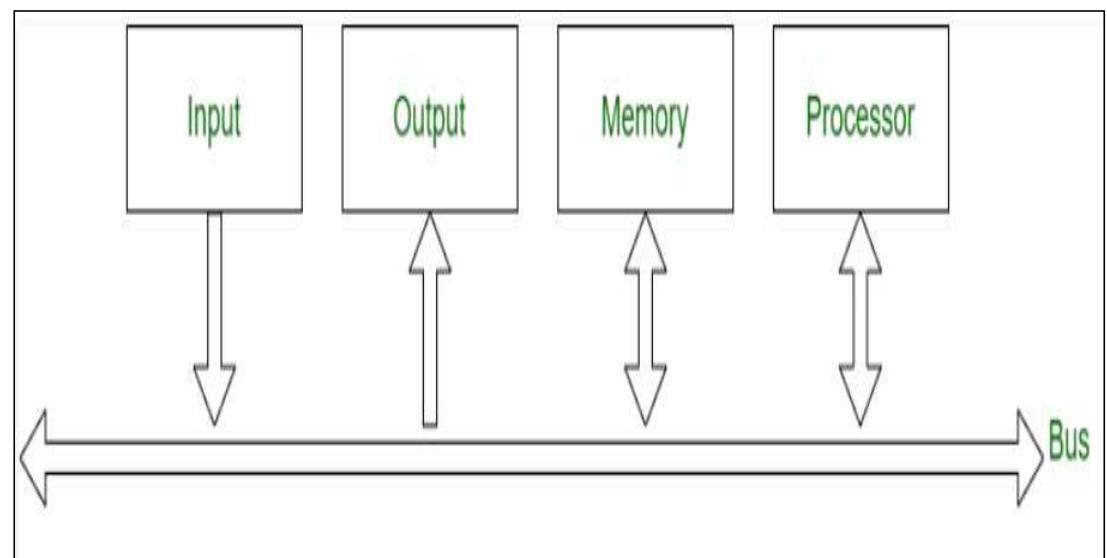
# Types of Bus (on the basis of number of bus)



# Single Bus Structure

## ❑ Pros and Cons

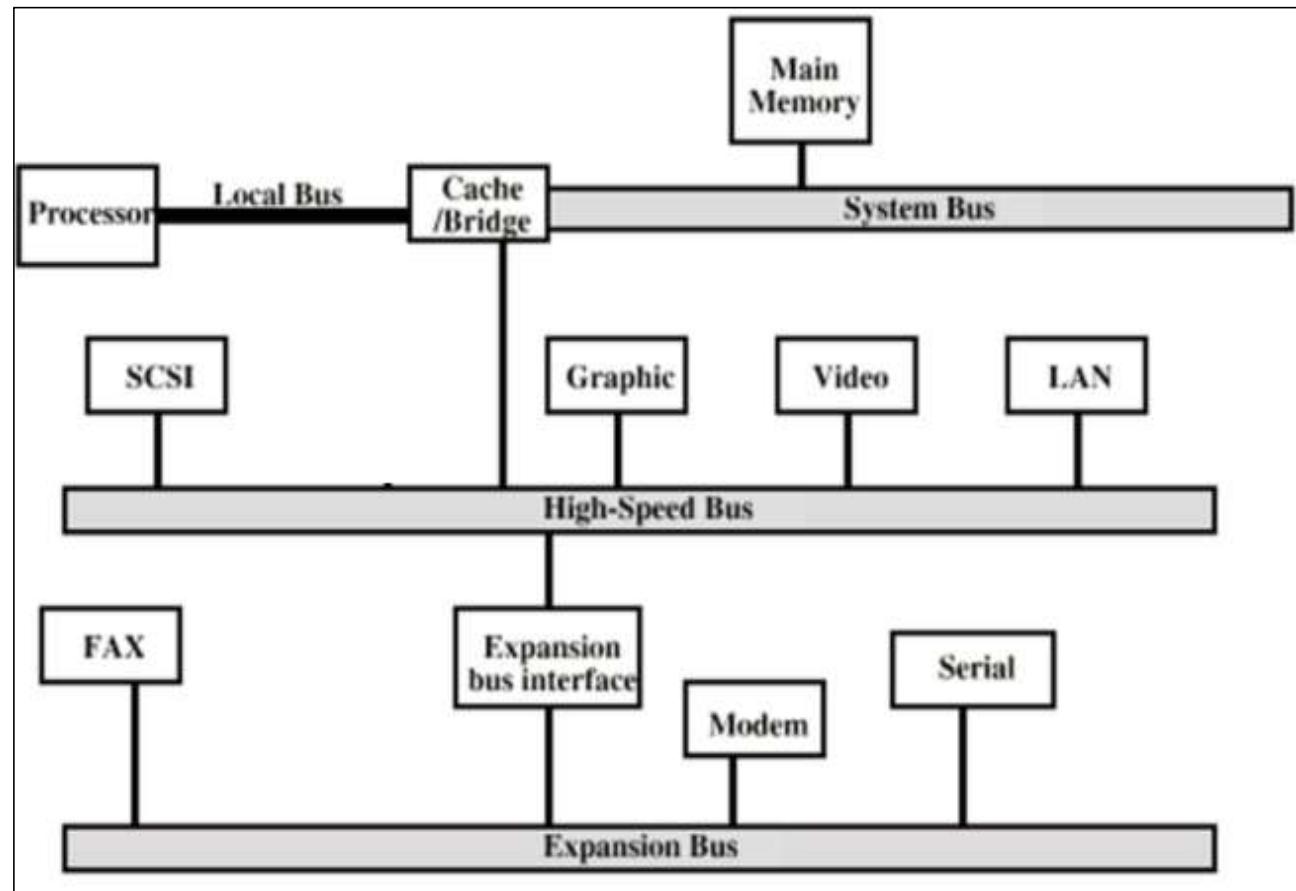
- ✓ Cheaper
- ✓ Easier design
- ✓ Low performance
- ✓ Limited Capacity
- ✓ Propagation delay



# Multiple Bus Structure

## ❑ Pros and Cons

- ✓ High performance
- ✓ Complex design
- ✓ Scalable
- ✓ Faster access
- ✓ Costlier



# Bus Arbitration

- ❑ The device which is allowed to initiate the data transfer on the bus at any given point of time is called the **BUS MASTER**.
- ❑ In computer system, there may be more than one bus master such as processor, DMA controller etc. They share the system bus.
- ❑ When one master relinquish control of the bus, another bus master can acquire the control of the Bus.
- ❑ **Bus Arbitration** refers to the transfer of control of bus from one master to another.

# Important terminology in Bus Arbitration

## ❑ Bus Master

- A controller who has currently access to the bus

## ❑ Bus Arbiter

- It decides who would become the next bus master

## ❑ Bus Request(BRQ)

- It is used by a module to request the control of the bus

## ❑ Bus Grant(BGT)

- It is signal which indicates that the access to the bus is granted

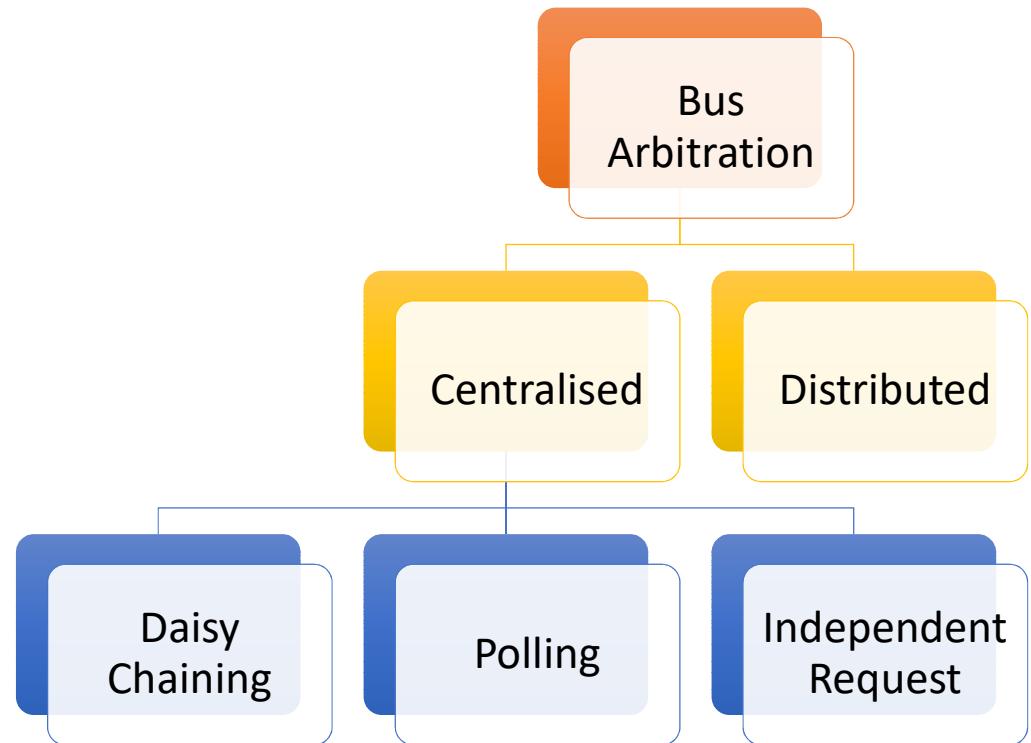
# Types of Bus Arbitration

## ❑ Centralised Arbitration

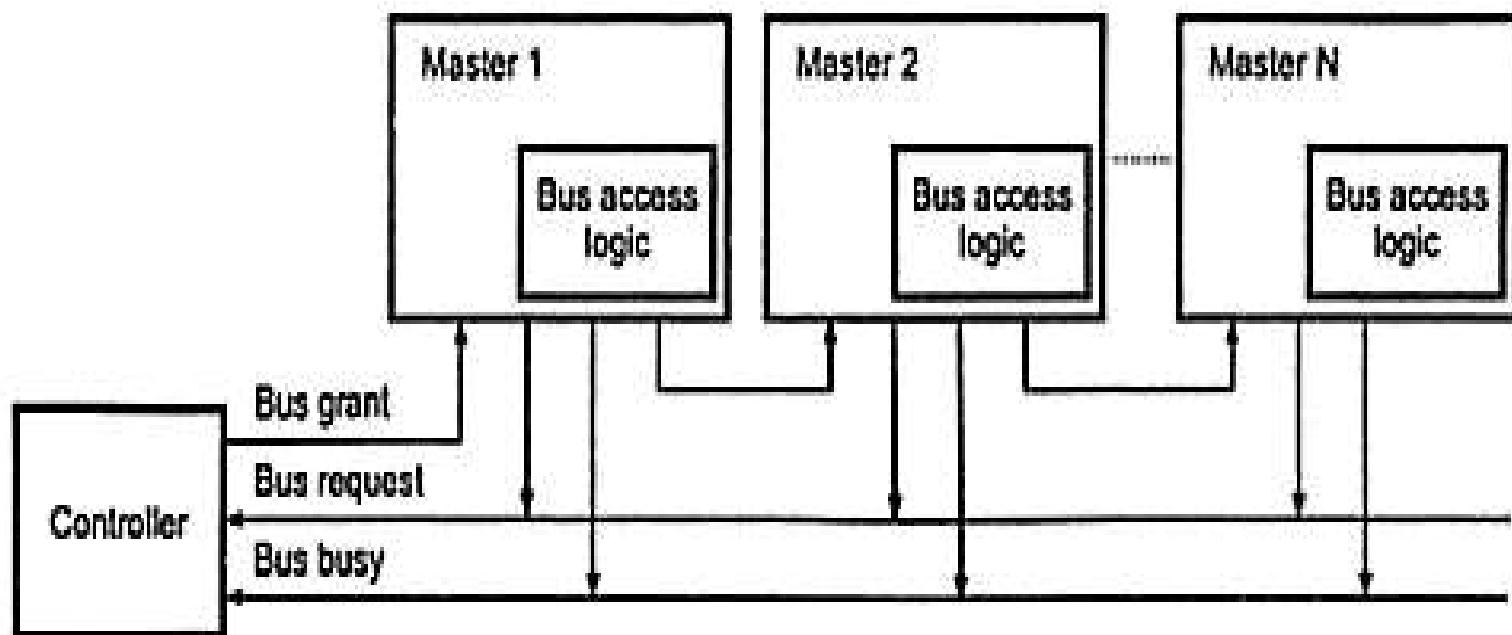
- Single hardware device controlling the bus access.
- May be part of CPU or separate

## ❑ Distributed Arbitration

- Any module may claim the bus
- Access control logic is on all modules
- Modules work together to control bus



# Daisy Chaining method



# Daisy Chaining method

- ❑ It is simple and cheaper method
- ❑ All masters make use of the same line for bus request
- ❑ In response to a bus request, the controller sends a bus grant signal if the bus is free.
- ❑ The bus grant signal propagates through each master until it reaches the device which has requested the access to the bus.
- ❑ The requesting master blocks the propagation of the bus grant signal, activates the busy line and gains control of the bus.
- ❑ Any other master requesting module will not receive the grant signal and hence cannot get bus access.

# Daisy Chaining method

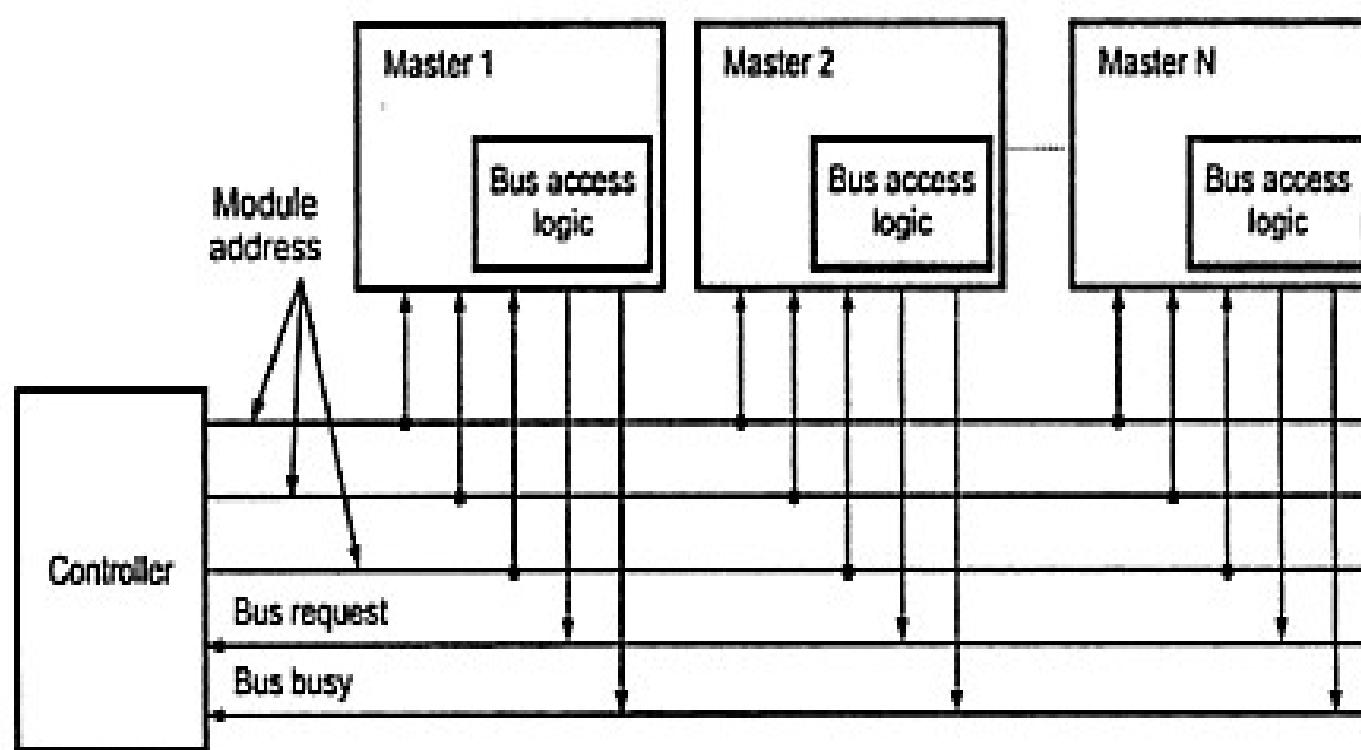
## □ Advantages

- It is simple and cheaper method
- It requires least number of lines
- Independent of the number of masters in the system

## □ Disadvantages

- Propagation delay is proportional to number of masters in the system
- Slow arbitration time
- Priority is based on the physical location
- Failure of any one master may cause the failure of whole system

# Polling or Rotating Priority Method



# Polling or Rotating Priority Method

- ❑ The controller is used to generate the addresses for the master.
- ❑ The number of address line required depends on the number of master connected in the system.
- ❑ For example, if there are 8 masters connected in the system, at least three address lines are required.
- ❑ In response to the bus request, controller generates a sequence of master address.
- ❑ When the requesting master recognizes its address, it activates the busy line and begins to use the bus.

# Polling or Rotating Priority Method

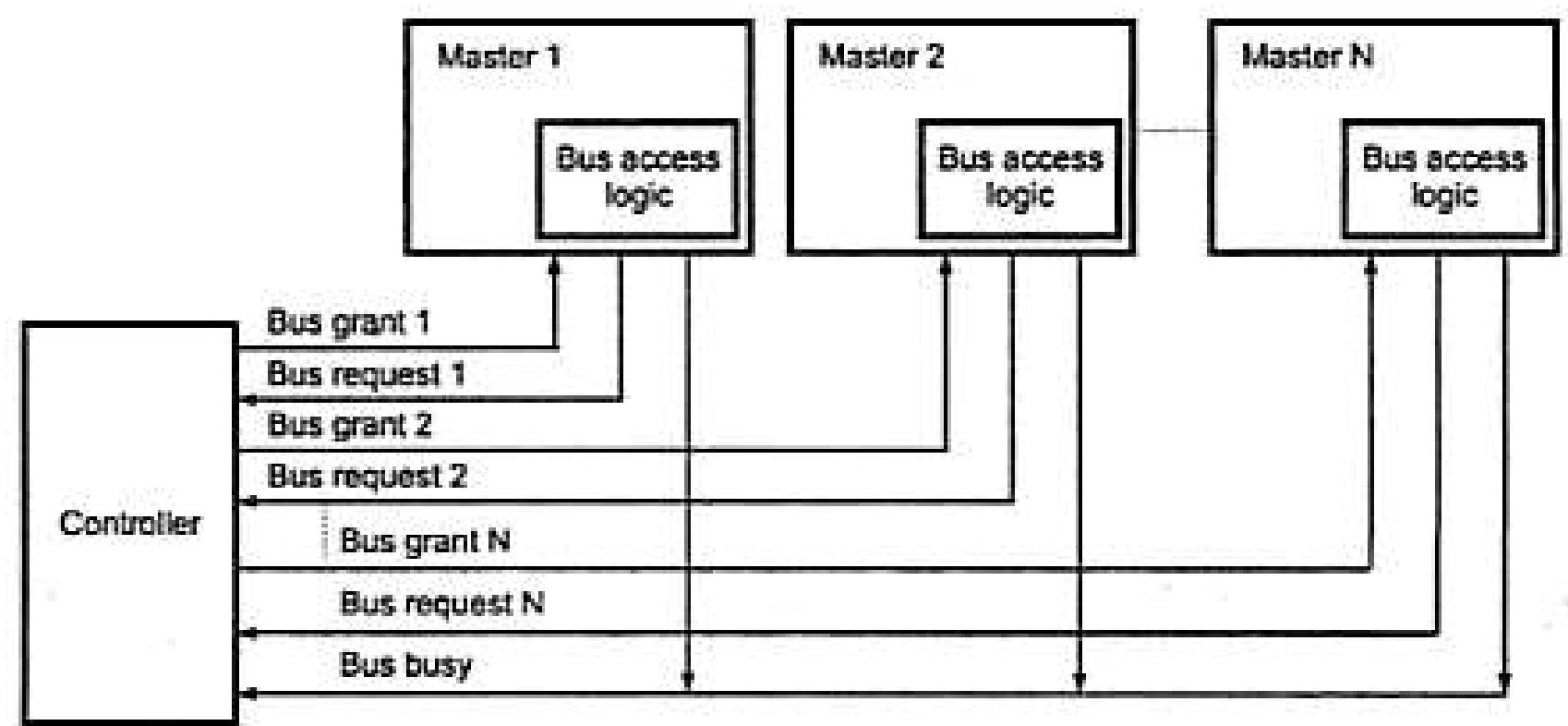
## □ Advantages

- The priority can be changed by altering the polling sequence stored in the controller
- Fast arbitration
- If one of the module fails, entire system does not fail

## □ Disadvantages

- Increasing the size will require more number of address lines

# Independent Request or Fixed Priority method



# Independent Request or Fixed Priority method

- ❑ In this scheme each master has a separate pair of bus request and bus grant lines
- ❑ Each pair has a priority assigned to it.
- ❑ The built in priority decoder within the controller selects the highest priority request and provides the corresponding bus grant signal

# Independent Request or Fixed Priority method

## □ Advantages

- Due to separate pairs of bus request and bus grant signals, **arbitration is fast and is independent of the masters in the system**
- **Very less chance of failure**

## □ Disadvantages

- **It requires more bus request and grant signals**
- **Costlier**
- **Chance of starvation**

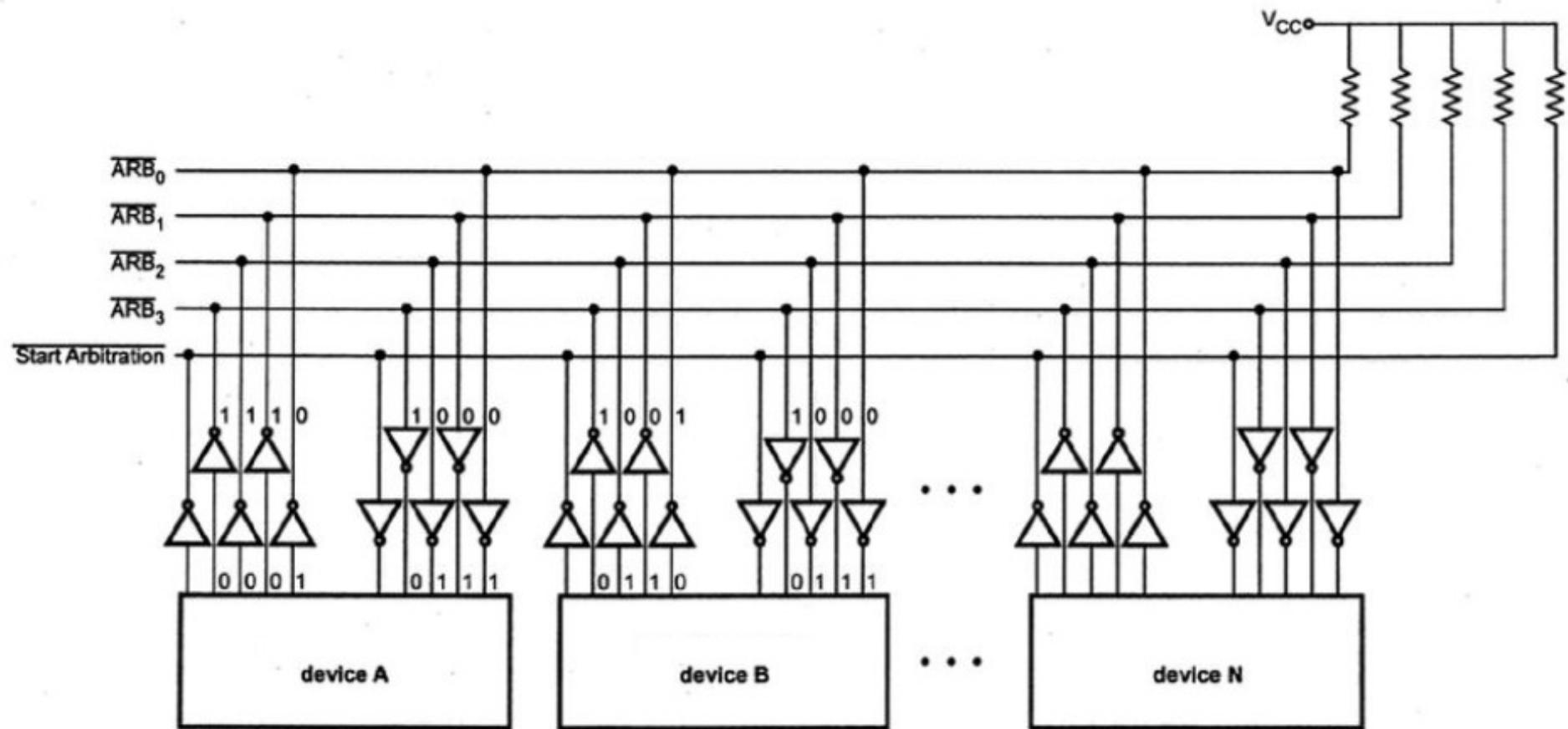
# Distributed Arbitration

- ❑ In distributed arbitration, all devices participate in the selection of the next bus master.
- ❑ In this scheme each device on the bus is assigned a 4-bit identification number.
- ❑ When one or more devices request for the control of bus, they assert the start-arbitration signal and place their 4-bit ID numbers on arbitration lines, ARB0 through ARB3.
- ❑ These four arbitration lines are all open-collector. Therefore, more than one device can place their 4-bit ID number to indicate that they need to control of bus.
- ❑ In this method, the device having highest ID number has highest priority.
- ❑ When two or more devices place their ID number on bus lines then it is necessary to identify the highest ID number from the status of bus line.
- ❑ The decentralized arbitration offers high reliability because operation of the bus is not dependent on any single device.

# Distributed Arbitration example

- ❑ Consider that two devices A and B, having ID number 1 (0001) and 6(0110), respectively are requesting the use of the bus.
- ❑ Device A puts the bit pattern 0001, and device B puts the bit pattern 0110. With this combination the status of bus-line will be 1000(for instance, XNOR of the bits); however because of inverter buffers, code seen by both devices is 0111.
- ❑ Each device compares the code formed on the arbitration line to its own ID, starting from the most significant bit. If there is a match, the output bit remains same but If it finds the difference at any bit position, it disables its drives by placing a 0 at that bit position as well as for all following lower-order bits
- ❑ In our example, device A detects a difference on line ARB2 and hence it disables its drives on line ARB2, ARB1 and ARB0. Thus, it's codes becomes 0000. Device B detects a difference on line ARB0 and hence it disables its drives on line ARB0. This causes the code on the arbitration lines to change to 0110. Device B is having highest number which means that device B has won the race.

# Distributed Arbitration example



# Digital Systems

- ❑ **Combinational and sequential circuits can be used to create simple digital systems**
- ❑ Simple **digital system** or module is frequently **characterized** in terms of
  - Set of registers and their functions
  - Set of Micro-operations
  - Control signals that initiate the sequence of micro-operations
- ❑ Such **digital system** or module are **interconnected with common data and control path** to form a **digital computer system.**

# Register and micro-operation

- ❑ A Register is a group of flip flops.
- ❑ An **elementary operation performed (in one clock cycle) on the data stored in one or more registers** is called **micro-operations**.
- ❑ There are **four types of micro-operations** in computer systems
  - Register Transfer micro-operations
  - Arithmetic micro-operations
  - Logic micro-operations
  - Shift micro-operations

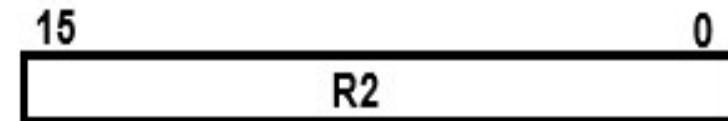
# Register Transfer Language (RTL)

- ❑ Rather than specifying a digital system in words, a specific notation is used which is called Register Transfer Language
- ❑ For any function of the computer, the register transfer language can be used to describe the sequence of micro-operations
- ❑ A Register Transfer Language is
  - A symbolic language
  - a convenient tool for describing the internal organization of digital computers
  - Can also be used to facilitate the design process of digital system

# Designation of Registers

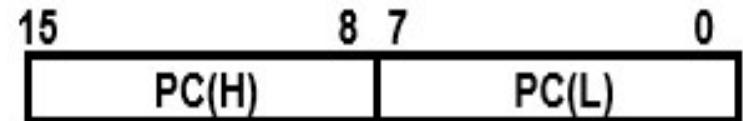
- Registers are **designated by capital letters, sometimes followed by numbers** (e.g., A, R13, IR)
- Often the names indicate function:
  - MAR - memory address register
  - PC- program counter
  - IR- instruction register
- Registers and their contents can be viewed and represented in *various ways*

Register



Numbering of bits

Showing individual bits



Subfields

# Registers Transfer

□ Copying the **contents of one register to another** is a register transfer

□ A register transfer is indicated as

$$R2 \leftarrow R1$$

- In this case the contents of register R1 are **loaded**(copied) into register R2
- A **simultaneous transfer** of all bits from the source R1 to the destination register R2, during one clock pulse
- Note that this is a **non-destructive**; i.e. the contents of R1 are not altered by loading(copying) them to R2

# Registers Transfer

- ❑ A register transfer such as

$R3 \leftarrow R5$

- ❑ Implies that the digital system has
  - the data lines from the source register (R5) to the destination register (R3)
  - Parallel load in the destination register (R3)
  - Control lines to perform the action

# Registers Transfer

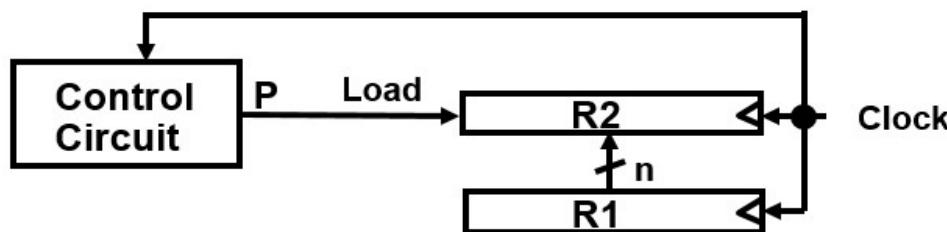
- Often actions need to be performed only if a certain condition is true
- Similar to an “if” statement in a programming language
- In digital systems, this is often done via a *control signal*, called a *control function*
  - If the signal is 1, the action takes place
- This is represented as:
  - $P: R2 \leftarrow R1$
  - Which means “if  $P = 1$ , then load the contents of register  $R1$  into register  $R2$ ”, i.e., if  $(P = 1)$  then  $(R2 \leftarrow R1)$

# Registers Transfer & Timing Diagram

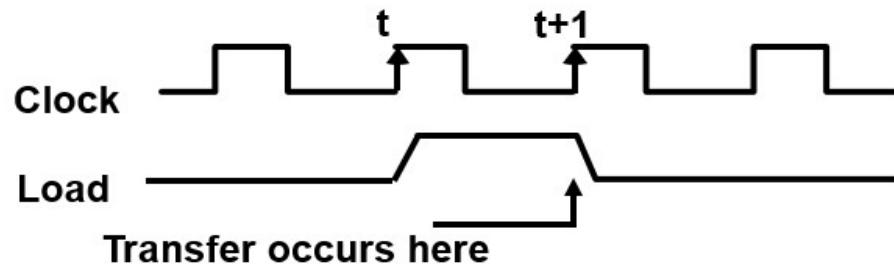
## □ Implementation of controlled transfer :

P:  $R_2 \leftarrow R_1$

## □ Block diagram



## □ Timing diagram



- The same clock controls the circuits that generate the control function and the destination register
- Registers are assumed to use *positive-edge-triggered flip-flops*

# Basic symbols for register transfer

Symbol	Description	Examples
Capital letters & numerals	Denotes a register	MAR, R2
Parentheses ()	Denotes a part of a register	R2(0-7), R2(L)
Arrow $\leftarrow$	Denotes transfer of information	R2 $\leftarrow$ R1
Colon :	Denotes termination of control function	P:
Comma ,	Separates two micro-operations	A $\leftarrow$ B, B $\leftarrow$ A

# Data movement around registers

- ❑ In a digital system with many registers, it is **impractical to have data and control lines to directly allow** each register to be loaded with the contents of every possible other registers
- ❑ To completely connect  $n$  registers  $\rightarrow n(n-1)$  lines
  - $O(n^2)$  cost
  - This is not a realistic approach to use in a large digital system
- ❑ Instead, take a different approach
- ❑ Have one centralized set of circuits for data transfer – the bus
- ❑ Have **control circuits to select which register is the source, and which is the destination**

# Data movement around registers using Bus

- Depending on whether the bus is to be mentioned explicitly or not, register transfer can be indicated as either

**R2 ← R1**

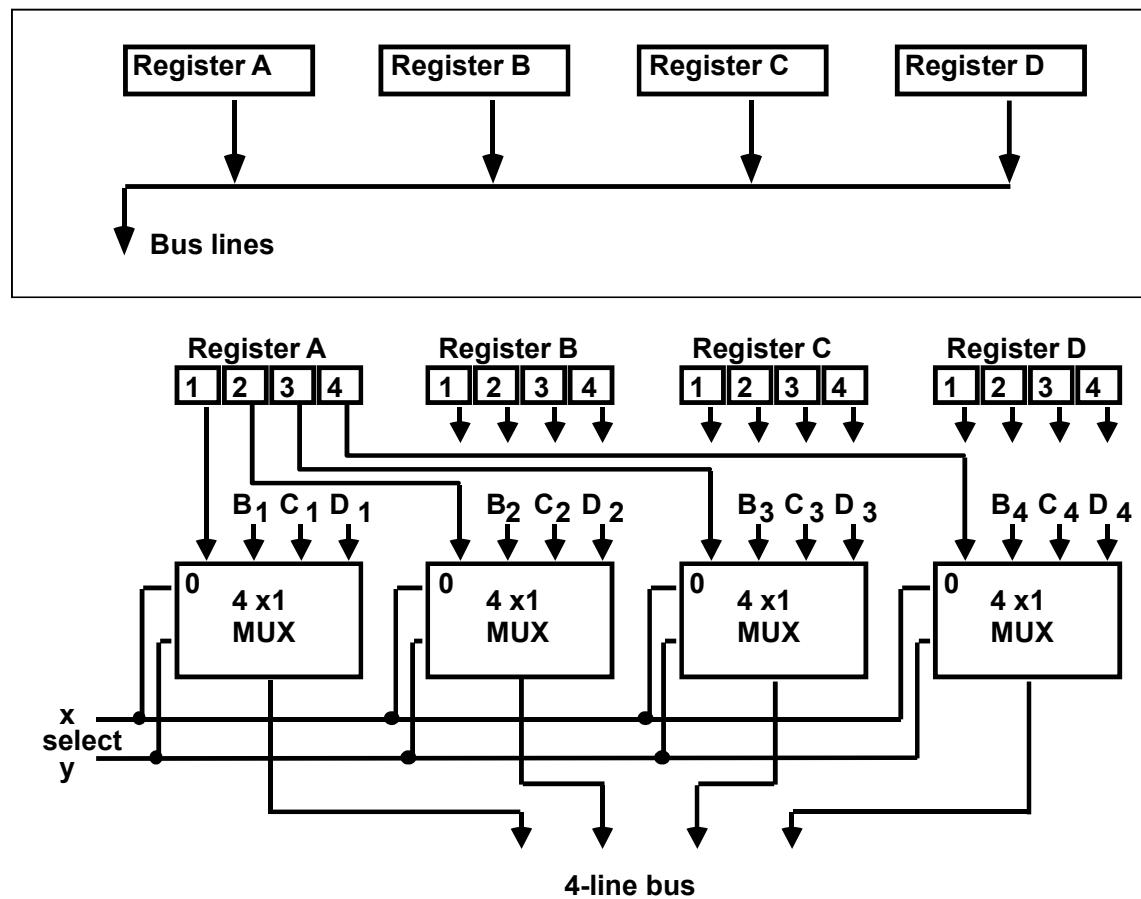
or

**BUS ← R1, R2 ← BUS**

- In the former case the bus is implicit, but in the latter, it is explicitly indicated

# Data movement from Register to Bus

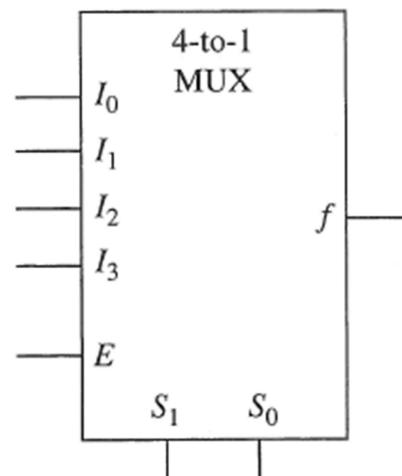
- From a register to bus:  $\text{BUS} \leftarrow R$



# What is Multiplexer (MUX)?

- ❑ Also called **data selectors**.
- ❑ Basic function: **select one of its  $2^n$  data input lines and place the corresponding information onto a single output line.**
- ❑  **$n$  input bits needed as selection to specify which input line is to be selected.**
- ❑ Place binary code for a desired data input line onto its  $n$  select input lines.

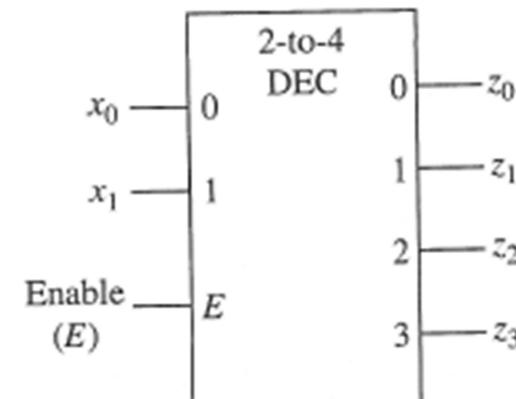
Select		Output
$S_1$	$S_0$	$f$
0	0	$I_0$
0	1	$I_1$
1	0	$I_2$
1	1	$I_3$



# What is Decoder?

- ❑ A decoder has
  - ❑ n inputs
  - ❑  $2^n$  outputs
- ❑ A decoder selects one of  $2^n$  outputs by decoding the binary value on the n inputs
- ❑ Exactly one output will be active for each combination of the inputs.

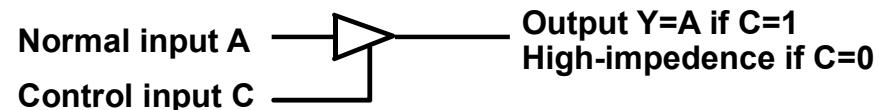
Inputs			Outputs			
$E$	$x_1$	$x_0$	$z_0$	$z_1$	$z_2$	$z_3$
0	$\times$	$\times$	0	0	0	0
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1



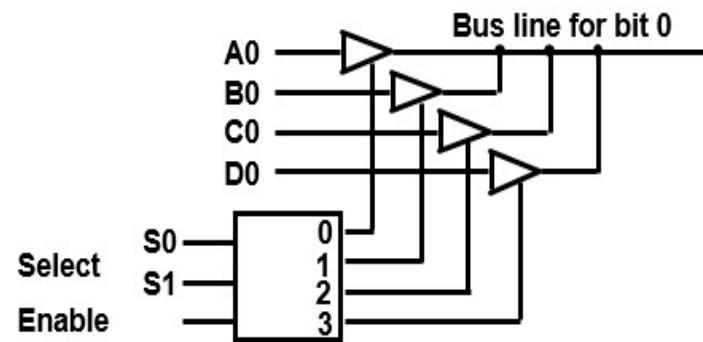
# Data movement from Register to Bus

- From a bus to register: Bus  $\leftarrow$  R

**Three-State Bus Buffers**

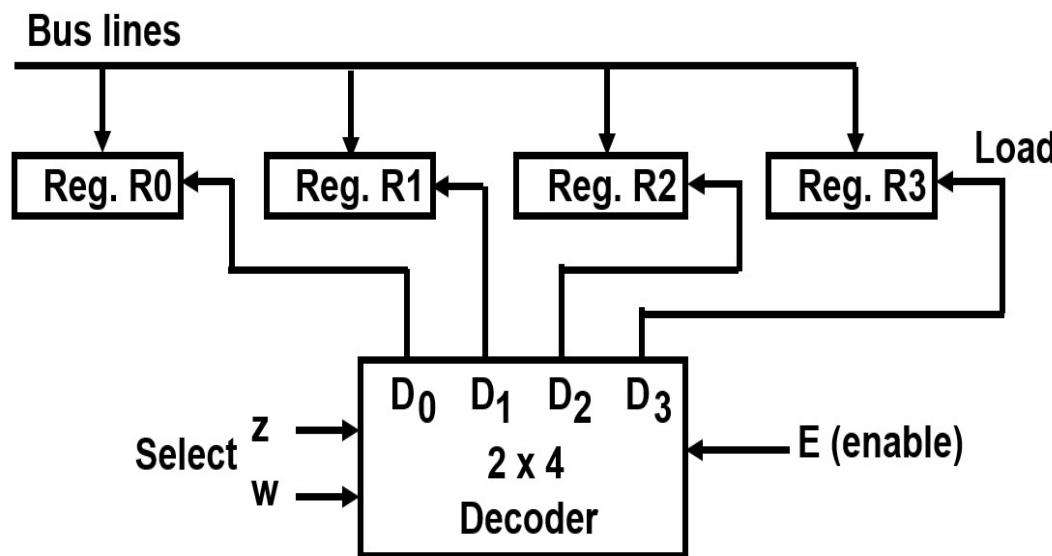


**Bus line with three-state buffers**



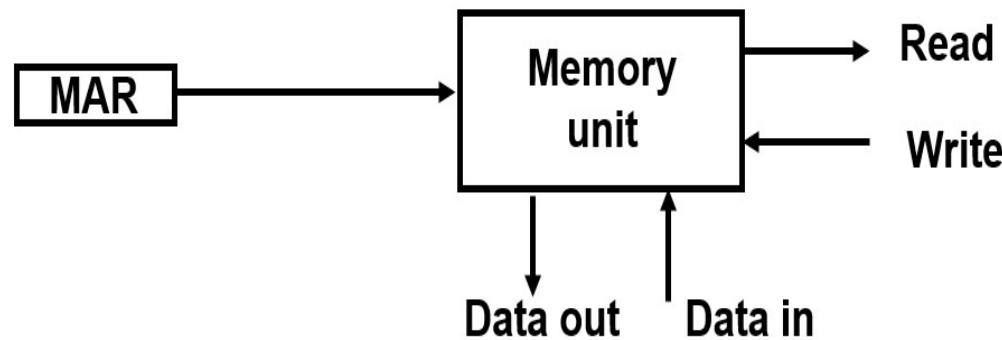
# Data movement from Bus to Register

- From a bus to register:  $R \leftarrow \text{BUS}$



# Data movement from/to Memory

- ❑ The memory is viewed as a device, M at the register level.
- ❑ Since it contains multiple locations, we must specify which address in memory we will be using
- ❑ This is done by indexing memory references
- ❑ Memory is usually accessed in computer systems by putting the desired address in a special register, the Memory Address Register (MAR, or AR)
- ❑ When memory is accessed, the contents of the MAR is sent to the memory unit's address lines



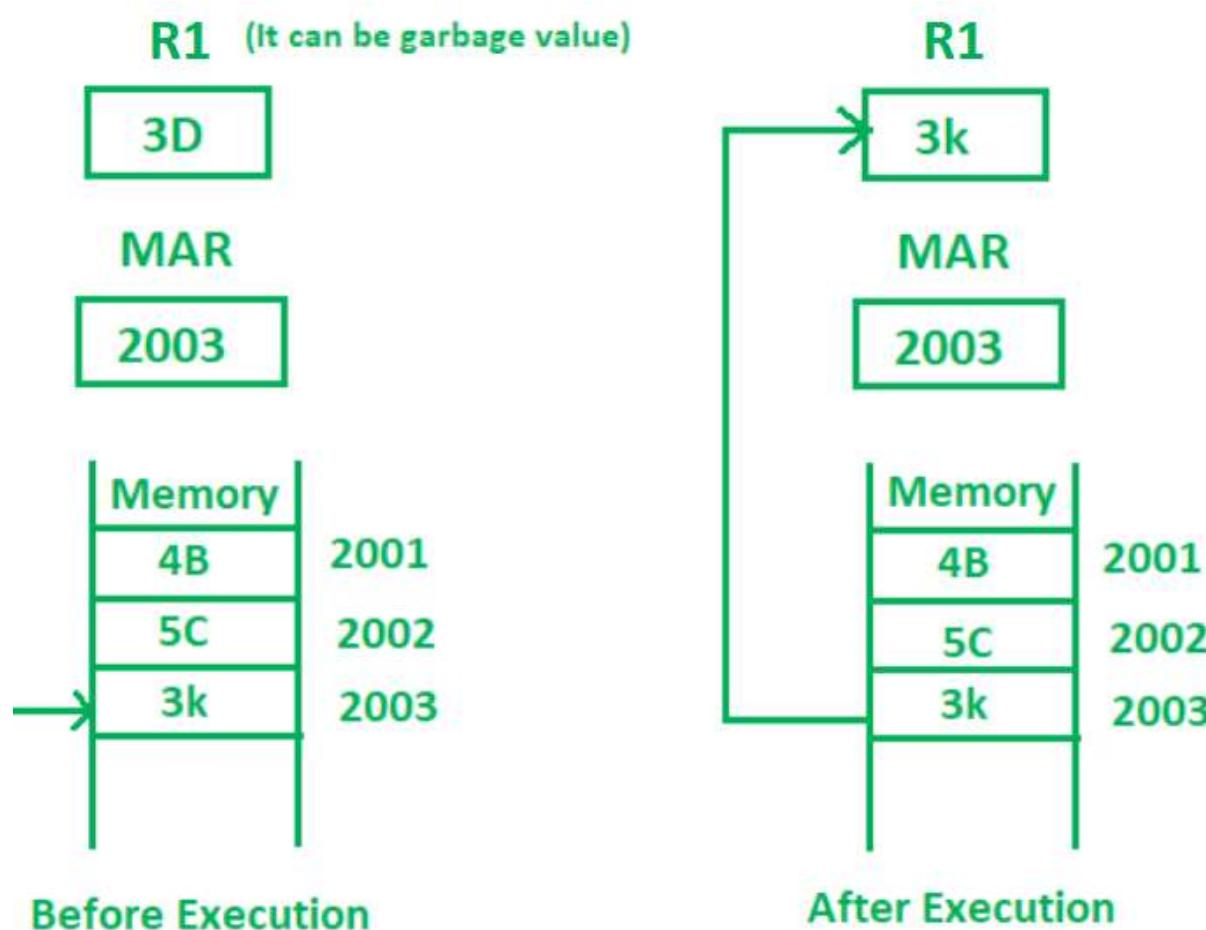
# Data Transfer from Memory to Register

- ❑ To read a value from a location in memory and load it into a register, the register transfer language notation looks like this:

**R1 ← M[MAR]**

- ❑ This causes the following to occur
  - The contents of the MAR is sent to the memory address lines
  - A Read= 1 signal is sent to the memory unit
  - The contents of the specified address are put on the memory's output data lines
  - The contents is sent over the bus to be loaded into register R1

# Data Transfer from Memory to Register



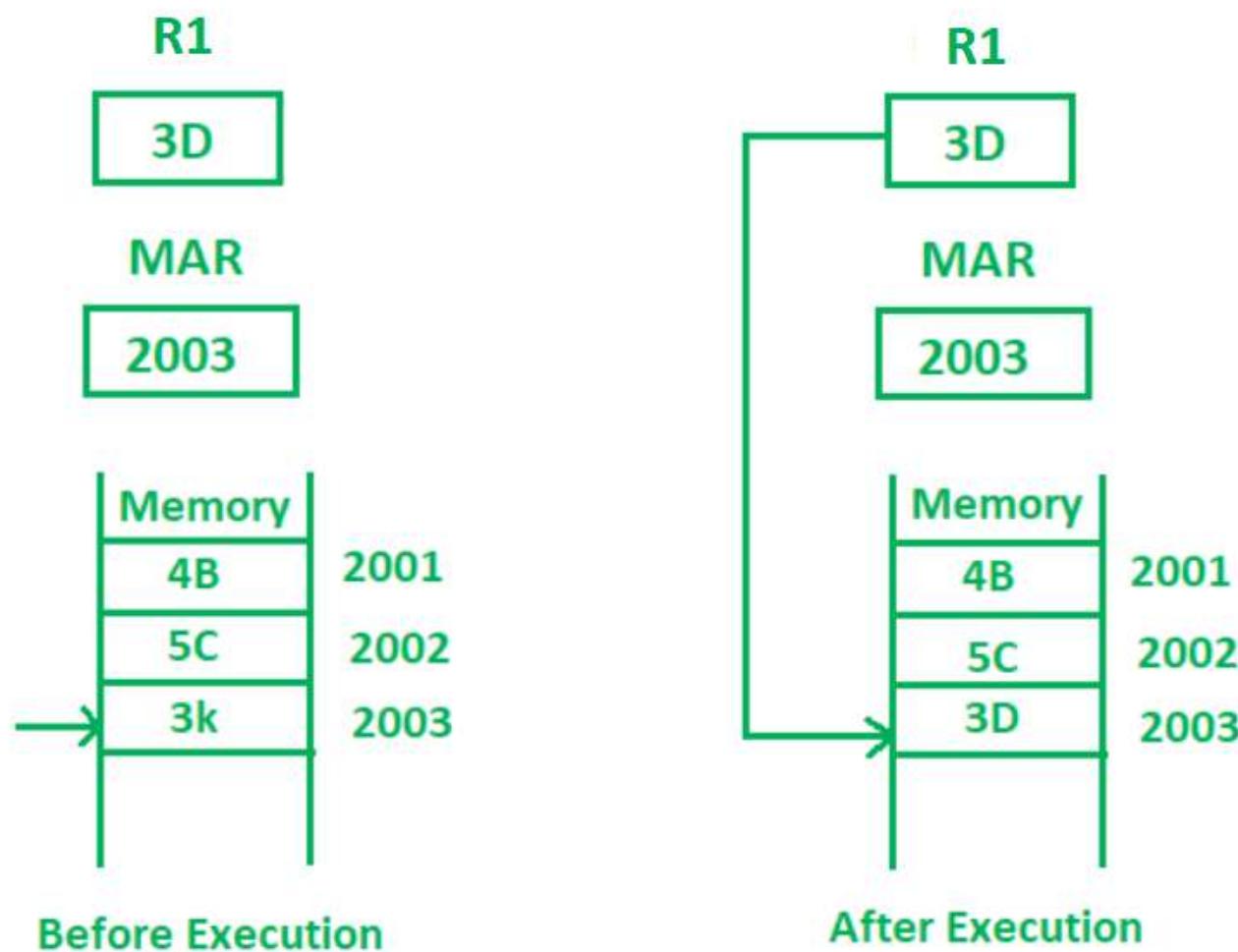
# Data Transfer from Register to Memory

- ❑ To write a value from a register to a location in memory, it is represented in register transfer language as:

**M[MAR] ← R1**

- ❑ This causes the following to occur
  - The contents of the MAR is sent to the memory address lines
  - A Write (= 1) gets sent to the memory unit
  - The values in register R1 is sent over the bus to the data input lines of the memory
  - The values get loaded into the specified address in the memory
  - decoder selects one of  $2^N$  outputs by decoding the binary value on the N inputs

# Data Transfer from Register to Memory



# Summary of Register Transfer micro-operations

$A \leftarrow B$

Transfer content of register B into register A

$A \leftarrow \text{constant}$

Transfer a binary constant into register A

$\text{ABUS} \leftarrow R1, R2 \leftarrow \text{ABUS}$  Transfer content of R1 into bus A and, at the same time, Transfer content of bus A into R2

MAR

Memory Address register

MDR

Memory Data register

$M[R]$

Memory word specified by register R

M

Equivalent to  $M[\text{MAR}]$

$\text{MDR} \leftarrow M$

Memory *read* operation: transfers content of memory word specified by MAR into MDR

$M \leftarrow \text{MDR}$

Memory *write* operation: transfers content of MDR into memory word specified by MAR

# Arithmetic Micro-Operations

□ The basic arithmetic micro-operations are

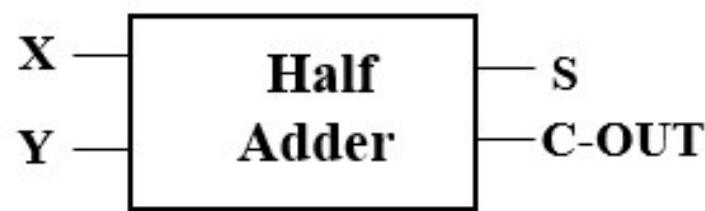
- Addition
- Subtraction
- Increment
- Decrement

□ The additional arithmetic micro-operations are

- Add with carry
- Subtract with borrow
- Transfer/Load

# Half Adder

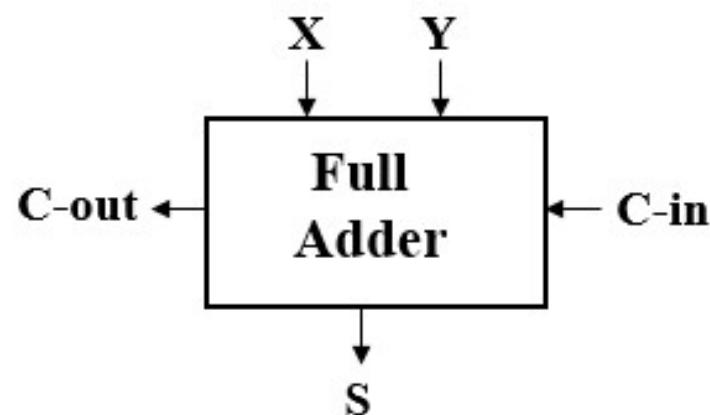
- Adding two single-bit binary values, X, Y and produces a sum bit S and a carry out C-out bit.



Inputs		Outputs	
X	Y	S	C-out
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

# Full Adder

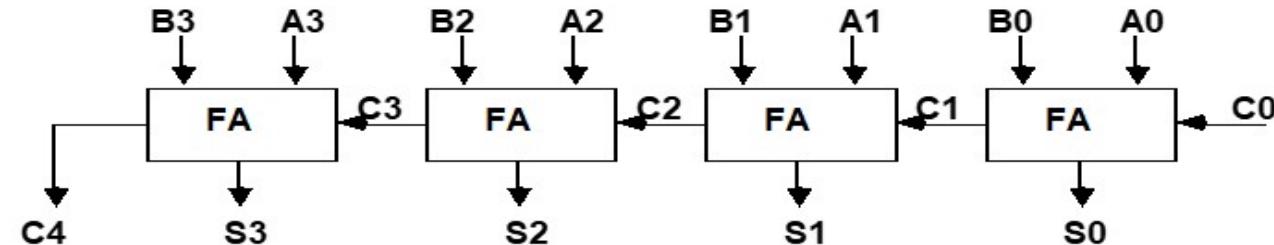
- Adding two single-bit binary values, X, Y along with a carry input bit C-in and produces a sum bit S and a carry out C-out bit.



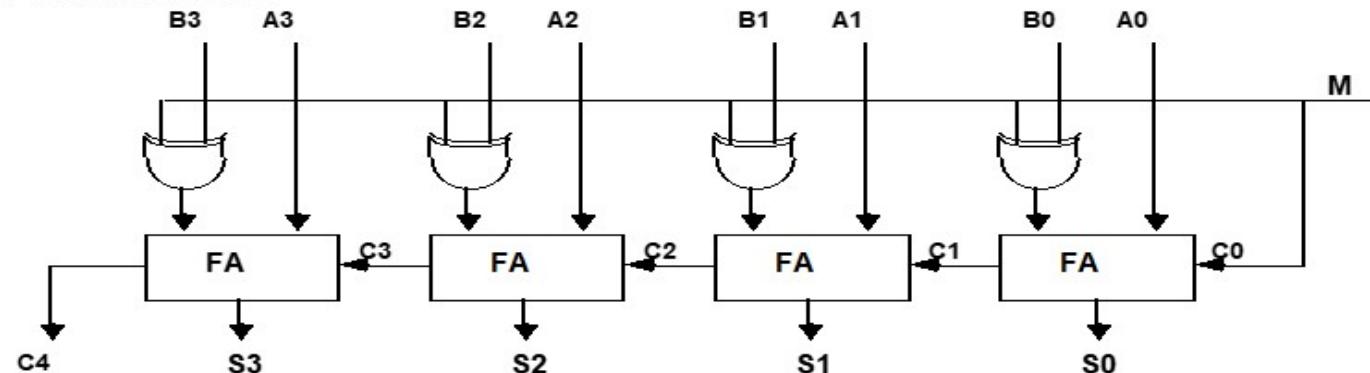
Inputs			Outputs	
X	Y	C-in	S	C-out
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

# Binary Adder/Subtractor/Incrementer

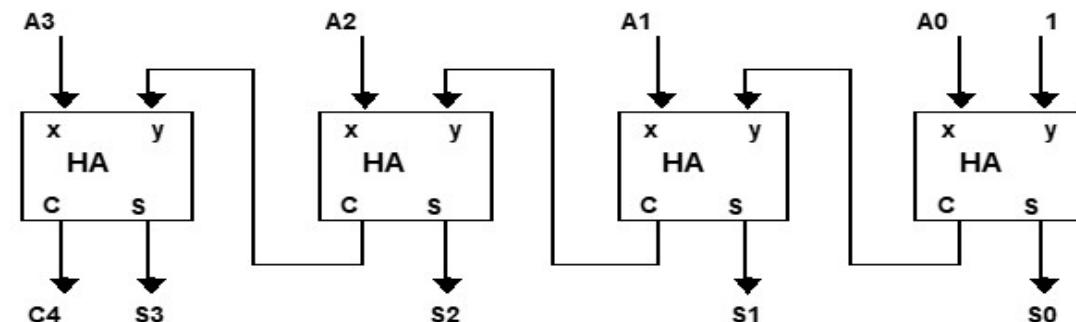
**Binary Adder**



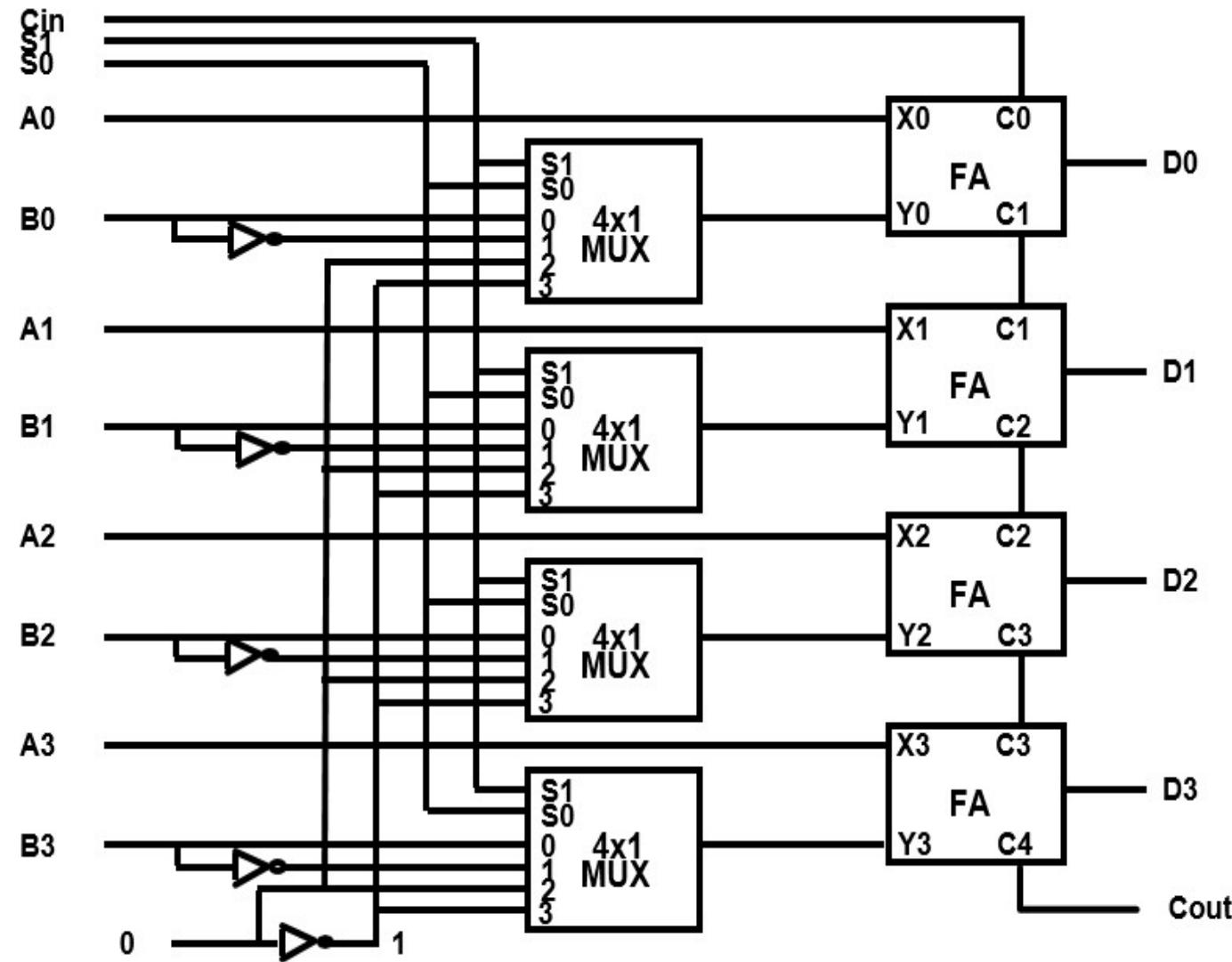
**Binary Adder-Subtractor**



**Binary Incrementer**



# Hardware for Binary Adder/Subtractor/Incrementer/Decrementer



# Truth Table for Binary Adder/Subtractor/Incrementer/Decrementer

S1	S0	Cin	Y	Output	Microoperation
0	0	0	B	$D = A + B$	Add
0	0	1	B	$D = A + B + 1$	Add with carry
0	1	0	B'	$D = A + B'$	Subtract with borrow
0	1	1	B'	$D = A + B' + 1$	Subtract
1	0	0	0	$D = A$	Transfer A
1	0	1	0	$D = A + 1$	Increment A
1	1	0	1	$D = A - 1$	Decrement A
1	1	1	1	$D = A$	Transfer A

# Summary of Arithmetic Micro-Operations

$R3 \leftarrow R1 + R2$

Contents of R1 plus R2 transferred to R3

$R2 \leftarrow R2'$

Complement the contents of R2

$R2 \leftarrow R2' + 1$

2's complement the content of R2

$R3 \leftarrow R1 + R2' + 1$

Subtraction

$R3 \leftarrow R1 - R2$

Contents of R1 minus R2 transferred to R3

$R1 \leftarrow R1 + 1$

Increment

$R1 \leftarrow R1 - 1$

Decrement

# Logic Micro-Operations

- Specify **binary operations** on the strings of bits in registers
- **Logic micro-operations** are **bit-wise operations**, i.e., they work on the individual bits of data
- Useful for bit manipulations on binary data
- Useful for making logical decisions based on the bit value
- There are, in principle, **16 different logic functions** that can be defined over two binary input variables

A	B	$F_0$	$F_1$	$F_2$	...	$F_{13}$	$F_{14}$	$F_{15}$
0	0	0	0	0	...	1	1	1
0	1	0	0	0	...	1	1	1
1	0	0	0	1	...	0	1	1
1	1	0	1	0	...	1	0	1

- However, most of the systems only implement four of them
  - AND ( $\wedge$ ), OR ( $\vee$ ), XOR ( $\oplus$ ), Complement/NOT
- The others can be created from combination of these four micro-operations

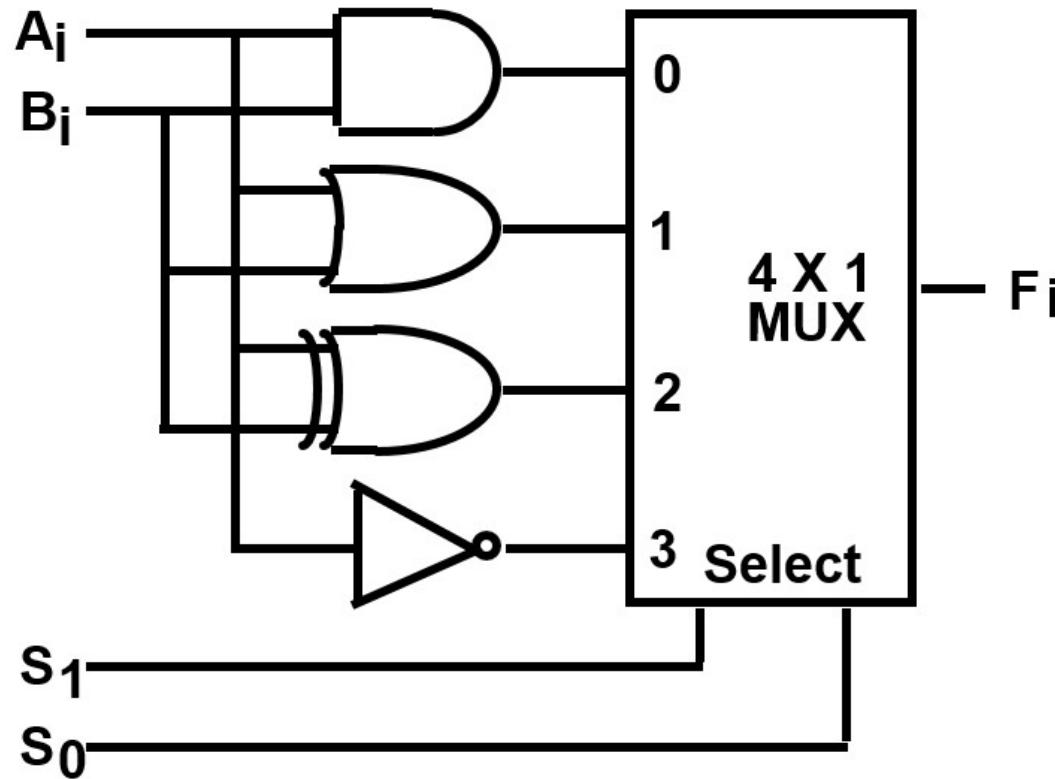
# Summary of Logic Micro-Operations

Truth tables for 16 functions of 2 variables and the corresponding 16 logic micro-operations

Note: Number of Logic Micro-operations  
 - n binary variables  $\rightarrow 2^{2^n}$  functions

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

# Hardware for Logic Micro-Operations



Truth table

S <sub>1</sub>	S <sub>0</sub>	Output	$\mu$ -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

# Applications of Logic Micro-Operations

- ❑ Logic micro-operations can be used to manipulate individual bits or a portions of a word in a register
- ❑ Consider the data in a register A. Another register, B will be used to modify the contents of A

➤ Selective-set	$A \leftarrow A \vee B$
➤ Selective-complement	$A \leftarrow A \oplus B$
➤ Selective-clear	$A \leftarrow A \wedge B'$
➤ Mask (Delete)	$A \leftarrow A \wedge B$
➤ Clear	$A \leftarrow A \oplus B$
➤ Insert	$A \leftarrow (A \wedge B) \vee C$

# Selective Set

- ❑ In a selective set operation, **the bit pattern in B is used to set certain bits in A**

$$\begin{array}{r} 1100 \\ 1010 \\ \hline 1110 \end{array} \quad \begin{array}{l} A_t \\ B \\ \hline A_{t+1} \end{array} \quad (A \leftarrow A \vee B)$$

- ❑ If a bit in B is set to 1, that same position in A gets set to 1, otherwise that bit in A keeps its previous value

# Selective Complement

- ❑ In a selective complement operation, the **bit pattern in B** is used to *complement* certain bits in A

$$\begin{array}{r} 1100 \quad A_t \\ 1010 \quad B \\ \hline 0110 \quad A_{t+1} \end{array} \quad (A \leftarrow A \oplus B)$$

- ❑ If a bit in B is set to 1, that same position in A gets complemented from its original value, otherwise it is unchanged

# Selective Clear

- ❑ In a selective clear operation, **the bit pattern in B is used to clear certain bits in A**

$$\begin{array}{r} 1100 \\ 1010 \\ \hline 0100 \end{array} \quad \begin{array}{l} A_t \\ B \\ \hline A_{t+1} \end{array} \quad (A \leftarrow A \wedge B')$$

- ❑ If a bit in B is set to 1, that same position in A gets set to 0, otherwise it is unchanged

# Mask Operation

- ❑ In mask operation, the bit pattern in B is used to clear certain bits in A

$$\begin{array}{r} 1100 \quad A_t \\ 1010 \quad B \\ \hline 1000 \quad A_{t+1} \end{array} \quad (A \leftarrow A \wedge B)$$

- ❑ If a bit in B is set to 0, that same position in A gets set to 0, otherwise it is unchanged

# Clear Operation

- ❑ In a clear operation, if the bits in the same position in A and B are the same, they are cleared in A, otherwise they are set in A

$$\begin{array}{r} 1100 \quad A_t \\ \hline 1010 \quad B \\ 0110 \quad A_{t+1} \quad (A \leftarrow A \oplus B) \end{array}$$

# Insert Operation

- An insert operation is used to introduce a specific bit pattern into A register, leaving the other bit positions unchanged
- This is done as
  - A mask operation to clear the desired bit positions, followed by
  - An OR operation to introduce the new bits into the desired positions
- Example: Suppose you wanted to introduce 1010 into the low order four bits of A:

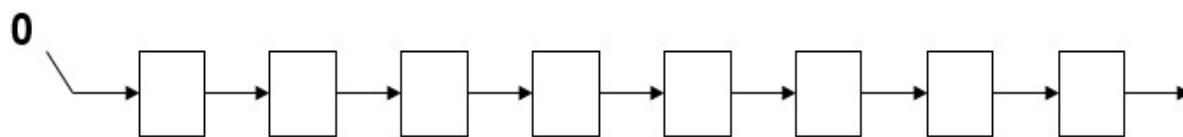
1101 1000 1011 0001	A (Original)
1101 1000 1011 1010	A (Desired)

1101 1000 1011 0001	<b>A (Original)</b>
1111 1111 1111 0000	<b>Mask</b>
<u>1101 1000 1011 0000</u>	<b><u>A (Intermediate)</u></b>
0000 0000 0000 <b>1010</b>	<b>OR Operation(new bits)</b>
<u>1101 1000 1011 <b>1010</b></u>	<b><u>A (Desired)</u></b>

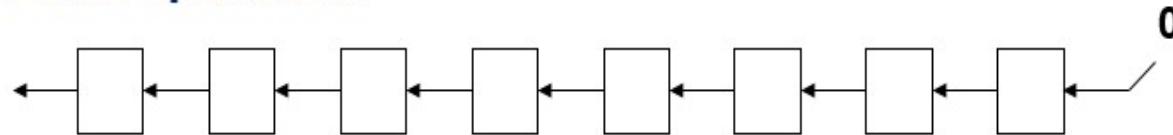
# Logical Shift

□ In a logical shift the serial input to the shift is a 0.

□ A right logical shift operation:



□ A left logical shift operation:



□ In a Register Transfer Language, the following notation is used

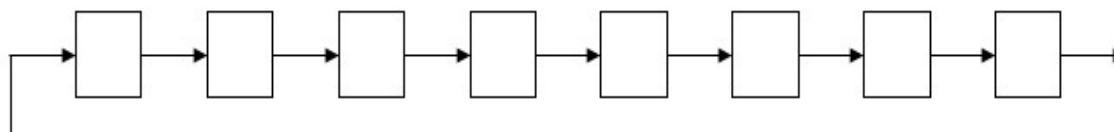
- *shl*      for a logical shift left
- *shr*      for a logical shift right
- Examples:

»  $R2 \leftarrow shr R2$

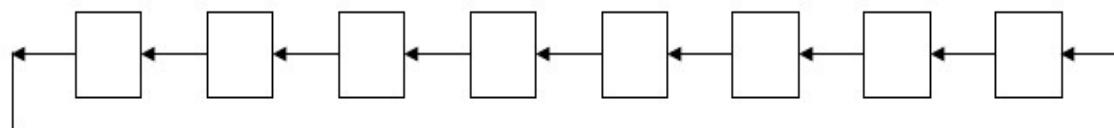
»  $R3 \leftarrow shl R3$

# Circular Shift

- In a circular shift the serial input is the bit that is shifted out of the other end of the register.
- A right circular shift operation:



- A left circular shift operation:



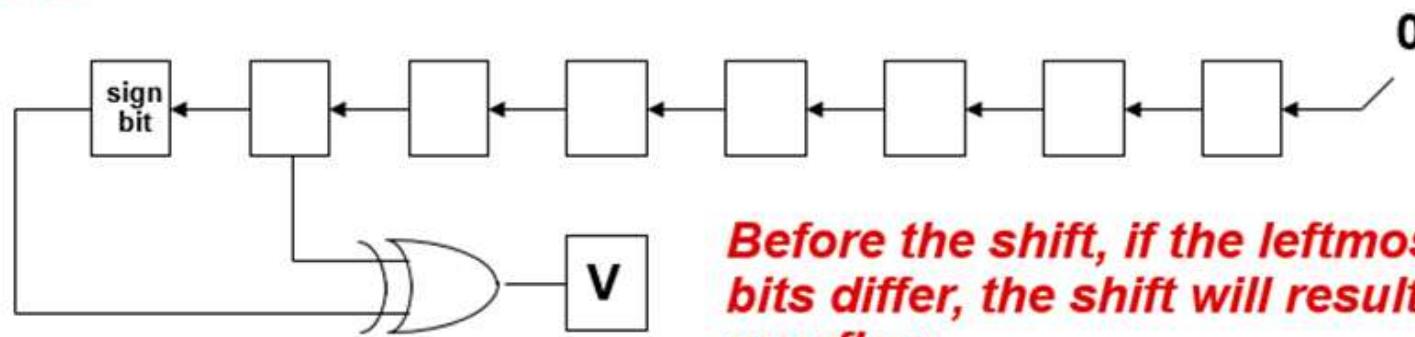
- In a RTL, the following notation is used

- *cil* for a circular shift left
- *cir* for a circular shift right
- Examples:

- »  $R2 \leftarrow cir R2$
  - »  $R3 \leftarrow cil R3$

# Arithmetic Shift

- A left arithmetic shift operation must be checked for the overflow



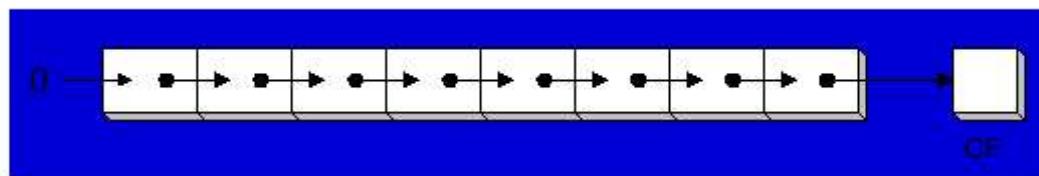
*Before the shift, if the leftmost two bits differ, the shift will result in an overflow*

- In a RTL, the following notation is used

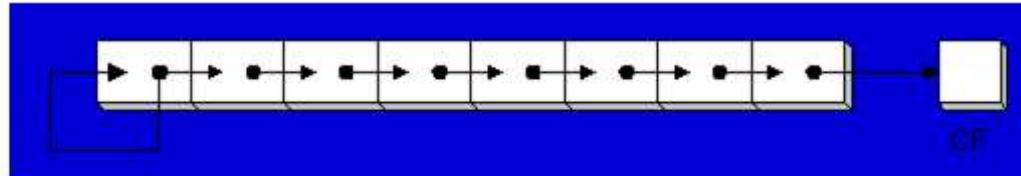
- `ashl` for an arithmetic shift left
- `ashr` for an arithmetic shift right
- Examples:
  - » `R2 ← ashr R2`
  - » `R3 ← ashl R3`

# Logical Vs. Arithmetic Shift

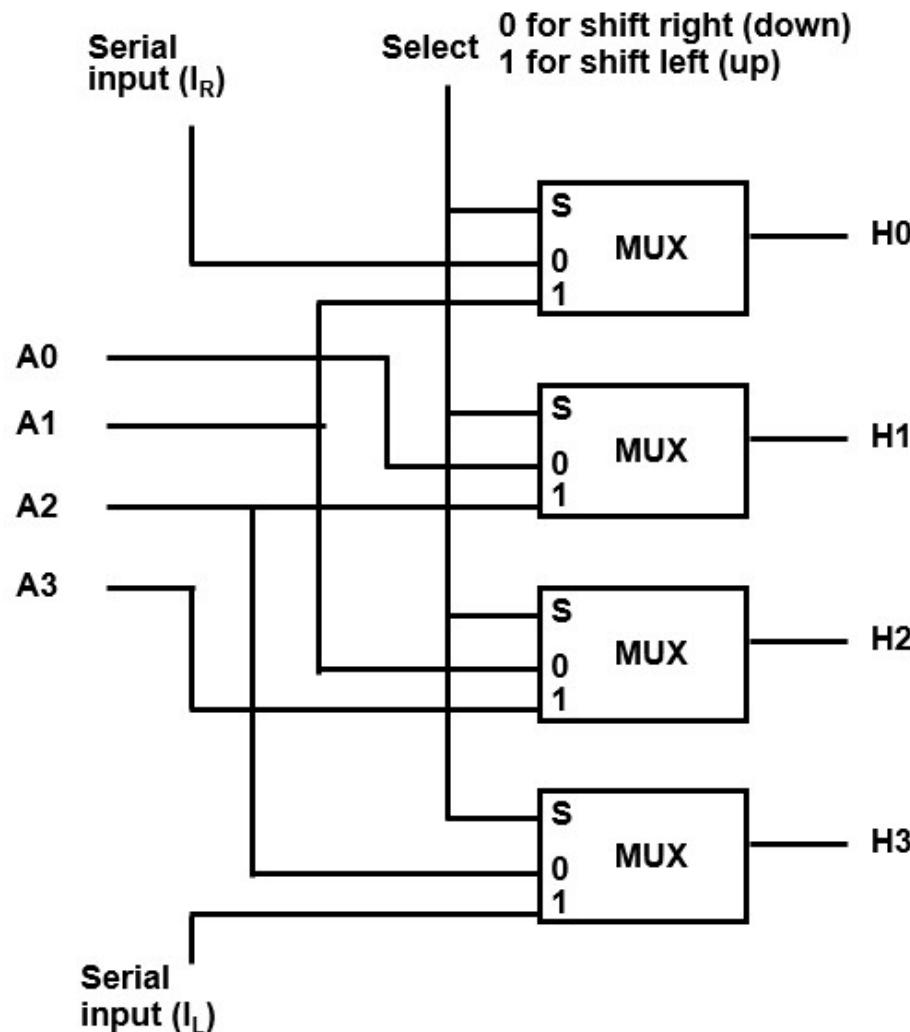
- ❑ A logical shift fills the newly created bit position with zero:



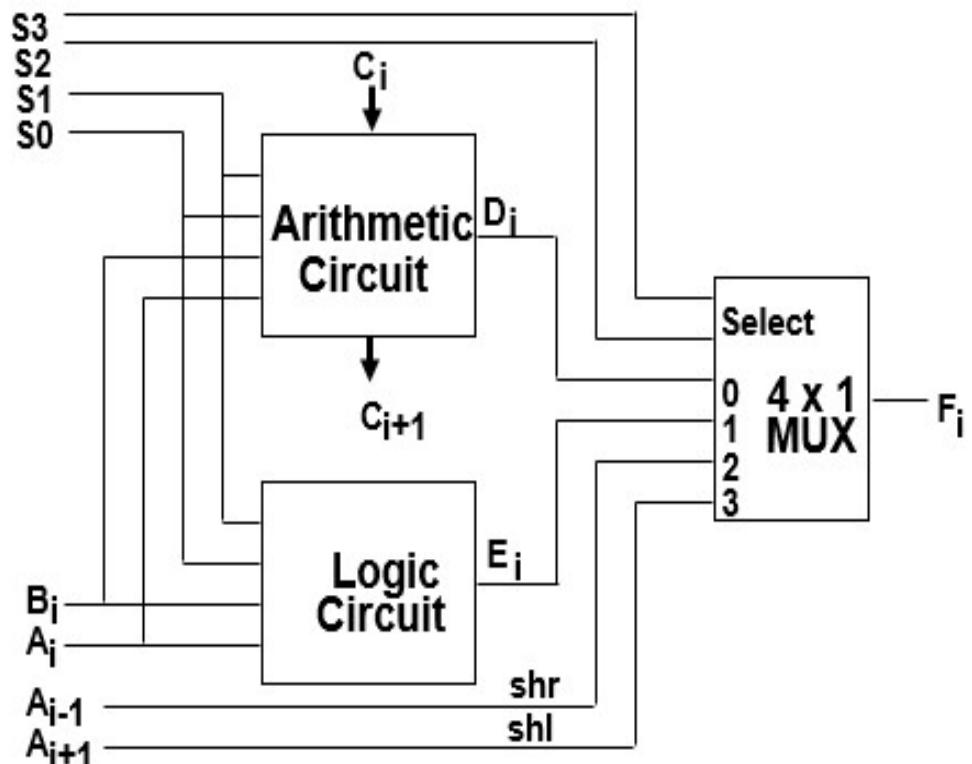
- ❑ An arithmetic shift fills the newly created bit position with a copy of the number's sign bit:



# Hardware for Shift Micro-Operation



# Hardware for Arithmetic –Logic Shift Micro-Operation



S3	S2	S1	S0	Cin	Operation	Function
0	0	0	0	0	F = A	Transfer A
0	0	0	0	1	F = A + 1	Increment A
0	0	0	1	0	F = A + B	Addition
0	0	0	1	1	F = A + B + 1	Add with carry
0	0	1	0	0	F = A + B'	Subtract with borrow
0	0	1	0	1	F = A + B' + 1	Subtraction
0	0	1	1	0	F = A - 1	Decrement A
0	0	1	1	1	F = A	Transfer A
0	1	0	0	X	F = A $\wedge$ B	AND
0	1	0	1	X	F = A $\vee$ B	OR
0	1	1	0	X	F = A $\oplus$ B	XOR
0	1	1	1	X	F = A'	Complement A
1	0	X	X	X	F = shr A	Shift right A into F
1	1	X	X	X	F = shl A	Shift left A into F