CSE211

Computer Organization and Design

Lecture : 3 **Tutorial: 1** **Practical: 0** **Credit: 4**

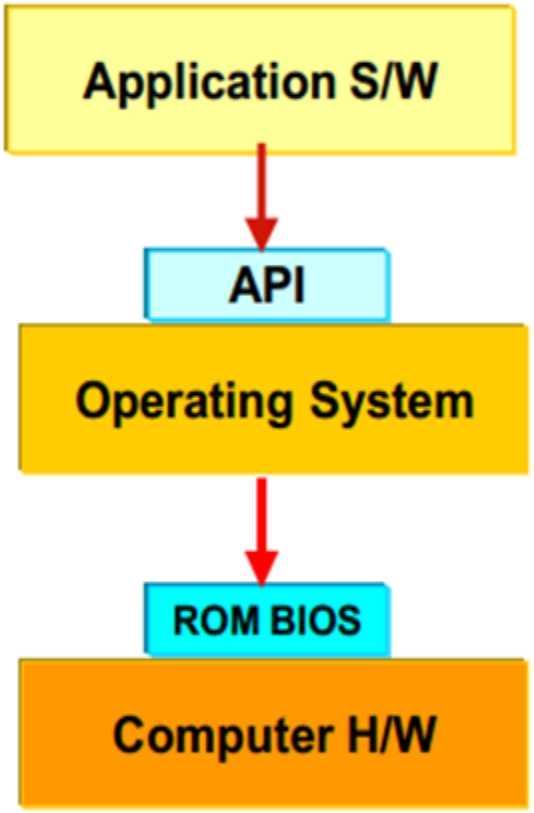
Unit 1 : Basics of Digital Electronics

- Introduction
- Logic Gates
- Flip Flops
- Decoder
- Encoder
- Multiplexers
- Demultiplexer

1-1 Digital Computers

- Digital – A limited number of discrete value
- Bit – A Binary Digit
- Program – A Sequence of instructions

- Computer = H/W + S/W
- Program(S/W)
  - A sequence of instruction
  - S/W = Program + Data
    - The data that are manipulated by the program constitute the data base
  - Application S/W
    - DB, word processor, Spread Sheet
  - System S/W
    - OS, Firmware, Compiler, Device Driver



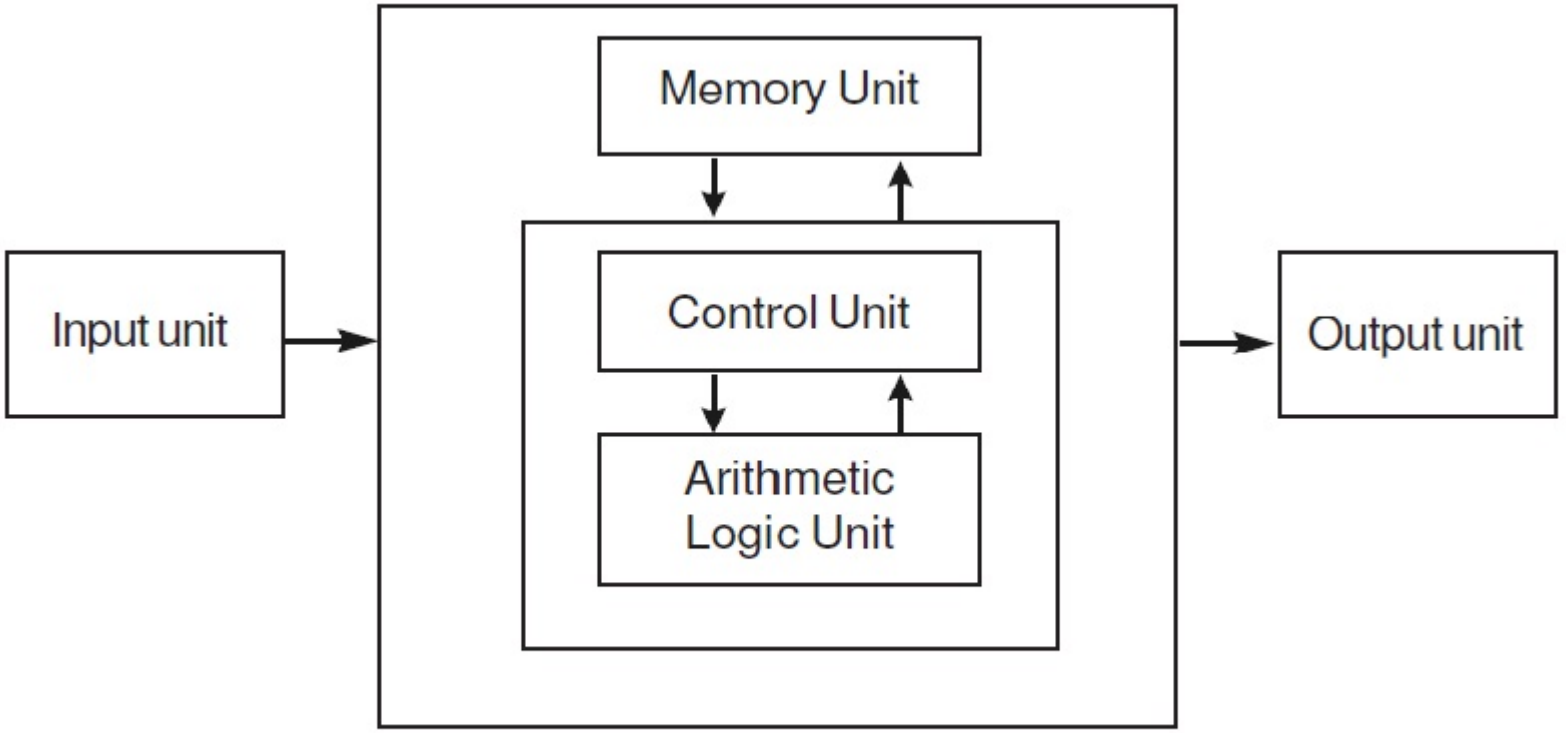
1-1 Digital Computers

- 3 different point of view(Computer Hardware)
  - Computer Organization
    - H/W components operation/connection
  - Computer Design
    - H/W Design/Implementation
  - Computer Architecture
    - Structure and behavior of the computer as seen by the user
    - Information format, Instruction set, memory addressing, CPU, I/O, Memory
- ISA(Instruction Set Architecture)
  - the attributes of a system as seen by the programmer, i.e., the conceptual structure and functional behavior, as distinct from the organization of the data flows and controls, the logic design, and the physical implementation.

    - Amdahl, Blaaw, and Brooks(1964)

- **Computer Organization :** It is concerned with the way hardware components operate and the way they are connected together to form a computer system.
- **Computer Architecture :** It is concerned with the structure and behaviour of the computer as seen by the user. It includes the information, formats, the instruction set, and techniques for addressing memory.
- **Computer Design :** It is concerned with the hardware design of the computer (the determination of what hardware should be used and how the parts should be connected.)

Basic Organization of a Computer



Logic Gates

- A logic gate is a building block of a digital circuit.
- At any given moment, every terminal is in one of the two binary conditions ***false*** **(high) or** ***true*** **(low)**.
- False represents **0**, and true represents **1**.
- Depending on the type of logic gate being used and the combination of inputs, the binary output will differ.
- A logic gate can be thought of like a light switch, wherein one position the output is off—0, and in another, it is on —1.
- Logic gates are commonly used in integrated circuits (IC).

LOGIC GATES

- Digital electronics relies on the actions of just seven types of logic gates, called **AND, OR, NAND (Not AND), NOR (Not OR), XOR (Exclusive OR) XNOR (Exclusive NOR) and NOT.**

Applications of logic Gates

- In manufacturing more complex devices e.g. binary counters etc.
- In decision making regarding automatic control of machines and different industrial processes.
- In calculators and computers.
- In digital processing of communications.

AND GATE

The AND gate produces the AND logic function, that is, the output is 1 if input A and input B are both equal to 1; otherwise the output is 0.

- Symbol

- A.B, A^B
- Truth table

INPUT		OUTPUT
A	B	Q
0	0	0
0	1	0
1	0	0
1	1	1

OR GATE

The OR gate produces the inclusive-OR function; that is, the output is 1 if input A or input B or both inputs are 1; otherwise, the output is 0.

The algebraic symbol of the OR function is +, similar to arithmetic **addition**.

OR gates may have more than two inputs, and by definition, the output is 1 if any input is 1

- Symbol.

- A+B , A V B
- TRUTH TABLE

INPUT		OUTPUT
A	B	Q
0	0	0
0	1	1
1	0	1
1	1	1

NAND GATE

The NAND function is the complement of the AND function, as indicated by the graphic symbol, which consists of an AND graphic symbol followed by a small circle.

The designation NAND is derived from the abbreviation of NOT-AND.

- Symbol

$$\overline{A \cdot B} \text{ or } A \uparrow B$$

INPUT		OUTPUT
A	B	Q
0	0	1
0	1	1
1	0	1
1	1	0

NOR GATE

The NOR gate is the complement of the OR gate and uses an OR graphic symbol followed by a small circle.

$\overline{A + B}$ or $A \downarrow B$

INPUT		OUTPUT
A	B	Q
0	0	1
0	1	0
1	0	0
1	1	0

XOR GATE

The exclusive-OR gate has a graphic symbol similar to the OR gate except for the additional curved line on the input side.

The output of the gate is 1 if any input is 1 but excludes the combination when both inputs are 1. It is similar to an odd function; that is, its output is 1 if an odd number of inputs are 1.

$A \oplus B$ or $A \veebar B$

INPUT		OUTPUT
A	B	Q
0	0	0
0	1	1
1	0	1
1	1	0

XNOR GATE

The exclusive-NOR is the complement of the exclusive-OR, as indicated by the small circle in the graphic symbol.

The output of this gate is 1 only if both the inputs are equal to 1 or both inputs are equal to 0.

$\overline{A \oplus B}$ or $A \odot B$

INPUT		OUTPUT
A	B	Q
0	0	1
0	1	0
1	0	0
1	1	1

NOT GATE , Buffer

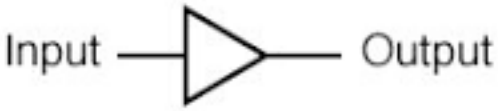
Output comparison of 1-input logic gates.

INPUT	OUTPUT	
A	Buffer	Inverter
0	0	1
1	1	0

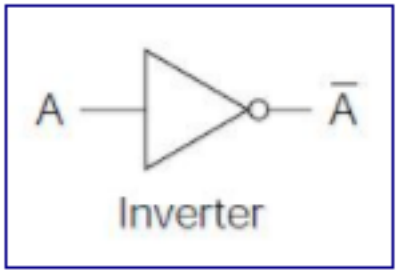
The inverter circuit inverts the logic sense of a binary signal. It produces the NOT, or complement, function.

The algebraic symbol used for the logic complement is either a prime or a bar over the variable symbol.

"Buffer" gate



Input	Output
0	0
1	1



Input	Output
0	1
1	0

Output comparison of 2-input logic gates

INPUT		OUTPUT					
A	B	AND	NAND	OR	NOR	XOR	XNOR
0	0	0	1	0	1	0	1
0	1	0	1	1	0	1	0
1	0	0	1	1	0	1	0
1	1	1	0	1	0	0	1

Question

- The output X of X-OR gate is high when

1. A=1 , B=1
2. A=0 , B=1
3. A=0 , B=0
4. A=1 , B=0
5. 2^{nd} and 4^{th} both
6. None of these

Q.. The output of a logic gate is 1 when all the input are at logic 0 as shown below:

The gate is either _____
a) A NAND or an EX-OR
b) An OR or an EX-NOR
c) An AND or an EX-OR
d) A NOR or an EX-NOR

INPUT		OUTPUT
A	B	C
0	0	1
0	1	0
1	0	0
1	1	0

INPUT		OUTPUT
A	B	C
0	0	1
0	1	0
1	0	0
1	1	1

Integrated Circuits

An IC is a small silicon semiconductors crystal called chip containing the electronic components for digital gates.

- Various gates are interconnected inside chip to form required circuit.
- Chip is mounted in ceramic/plastic container connected to external pin

Small scale Integration (SSI) : less than 10 gates

Medium Scale Integration(MSI) : between 10 to 200 gates (decoders, adders, registers)

Large Scale Integration(LSI) : between 200 and few thousands gates (Processors, Memory Chips)

Very Large Scale Integration (VLSI) : Thousands of gate within single package (Large Memory Arrays, Complex Microcomputer Chips)

Digital Electronics

Digital Electronics is divided into two circuits :-

- **Combinational circuits**
- **Sequential circuits**

Combinational Circuits

- No feedback paths
- No memory
- connected arrangement of logic gates with set of inputs and outputs.
- Output is independent of time and depends only on the present input.
- Used for arithmetic as well as boolean operations .
- don't have clock , don't require triggering .
- Examples are half Adder, Full adder, Encoder, Decoder, Multiplexer, De-multiplexer.

Sequential Circuits

- Feedback paths exist
- Memory present
- Time dependent
- 2 Types- Synchronous and Asynchronous
- Synchronisation is achieved with help of device called clock.
- Output depends not only on present input but also on the past output.
- Examples are Flip-flops, registers, counters.

Sequential Circuits

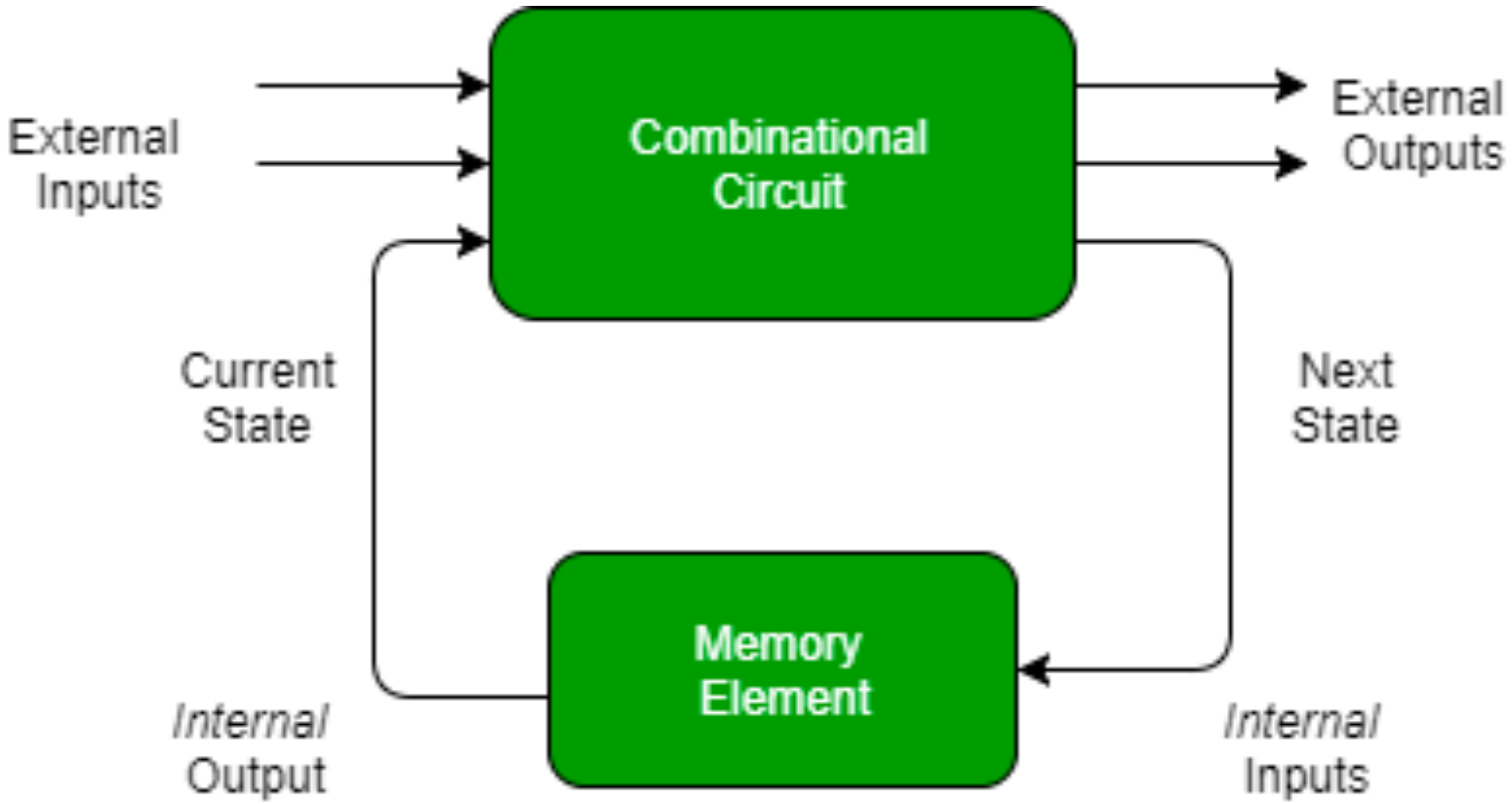
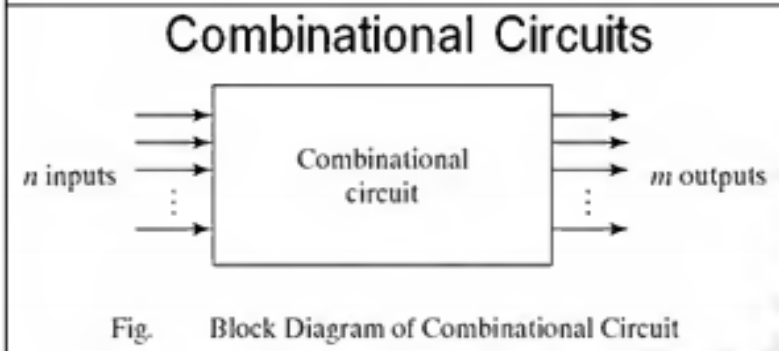
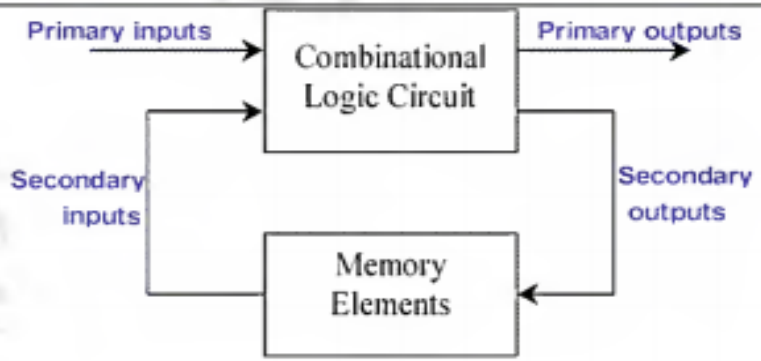


Figure: Sequential Circuit

Combinational Circuits	Sequential Circuits
1. The circuit whose output at any instant depends only on the input present at that instant only is known as combinationational circuit.	1. The circuit whose output at any instant depends not only on the input present but also on the past output a is known as sequential circuit
2. This type of circuit has no memory unit.	2. This type of circuit has memory unit for store past output.
3. Examples of combinational circuits are half adder, full adder, magnitude comparator, multiplexer, demultiplexer e.t.c.	3. Examples of sequential circuits are Flip flop, register, counter e.t.c.
4. Faster in Speed	Slower compared to Combinational Circuit
 Fig. Block Diagram of Combinational Circuit	

Question

- Sequential circuits have which of the following signal?
- A—Inputs
- B—Outputs
- C—Feedback
- D—All of above

Question

- Combinational circuits don’t have which of the following signal?
- A—Inputs
- B—Outputs
- C—Feedback
- D—All of above

FLIP FLOPS

Flip-flop is a circuit that maintains a state until directed by input to change the state.

A basic flip-flop can be constructed using four-NAND or four-NOR gates.

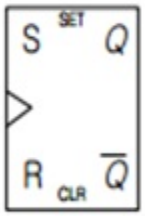
Types of flip-flops:

1. SR Flip Flop
2. JK Flip Flop
3. D Flip Flop
4. T Flip Flop

1-6 Flip-Flops

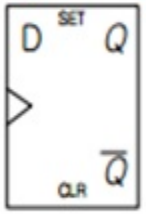
Combinational Circuit = Gate
Sequential Circuit = Gate + F/F

- Flip-Flop
  - The *storage elements* employed in clocked *sequential circuit*
  - A binary cell capable of storing one bit of information
- SR*(Set/Reset)* F/F



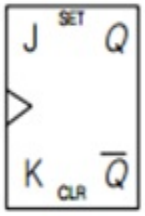
S	R	Q(t+1)	
0	0	Q(t)	no change
0	1	0	clear to 0
1	0	1	set to 1
1	1	?	Indeterminate

- D*(Data)* F/F



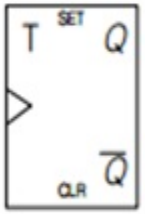
D	Q(t+1)	
0	0	clear to 0
1	1	set to 1

- "no change" condition
  1) Disable Clock
  2) Feedback output into input
- JK*(Jack/King)* F/F



J	K	Q(t+1)	
0	0	Q(t)	no change
0	1	0	clear to 0
1	0	1	set to 1
1	1	Q(t)'	Complement

  - JK F/F is a refinement of the SR F/F
  - The indeterminate condition of the SR type is defined in complement
- T*(Toggle)* F/F



T	Q(t+1)	
0	Q(t)	no change
1	Q'(t)	Complement

  - T=1(J=K=1), T=0(J=K=0)

A flip-flop can store:——

- one bit of data
- two bits of data
- three bits of data
- any number of bits of data

The race around condition occurs in a J-K flip-flop when:——-

- both inputs are 0
- both inputs are 1
- the inputs are complementary
- any one of the above input combinations is present

CSE211

Computer Organization and Design

Lecture : 3 **Tutorial: 1** **Practical: 0** **Credit: 4**

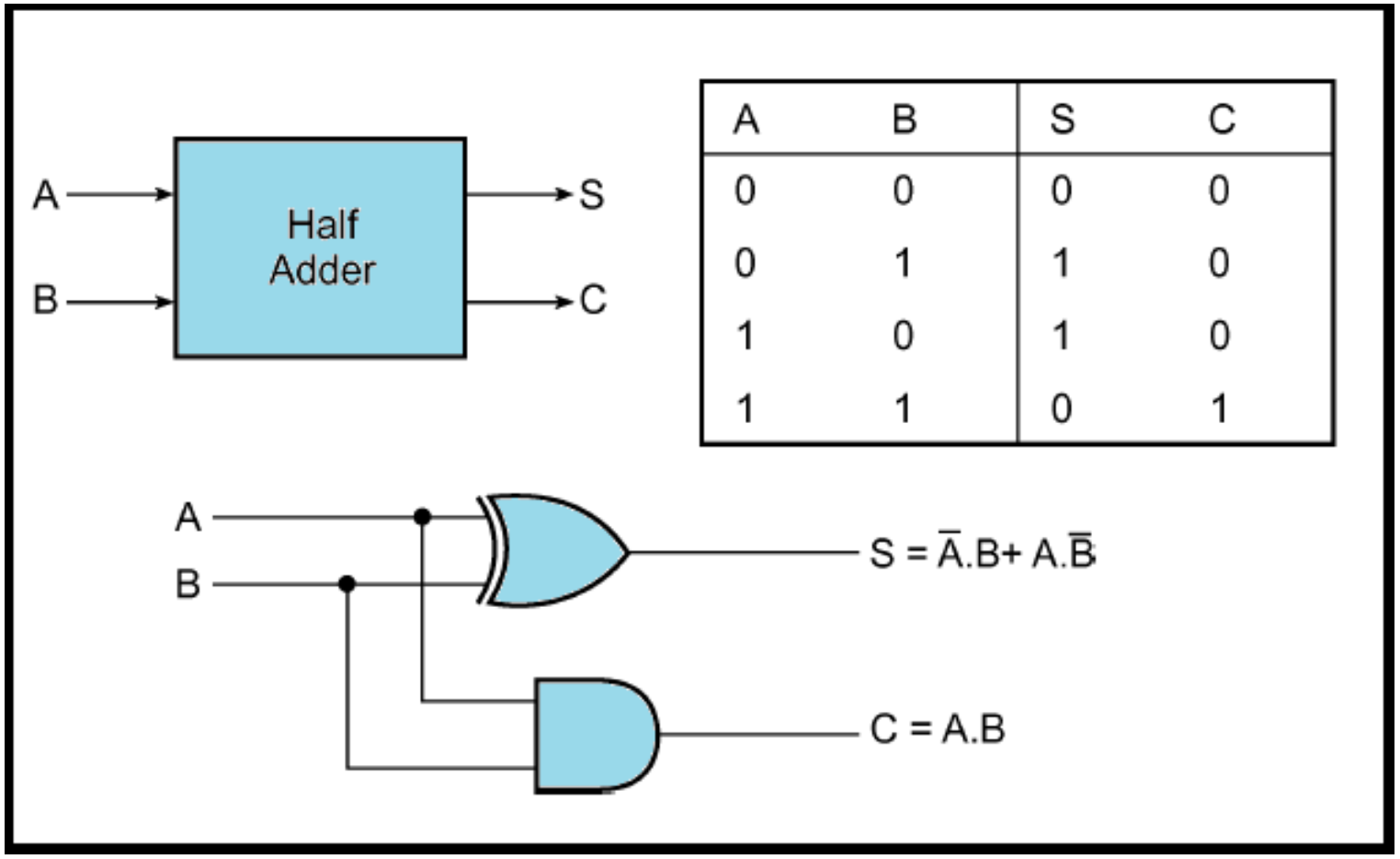
Unit 1 : Basics of Digital Electronics

- Introduction
- Logic Gates
- Flip Flops
- Decoder
- Encoder
- Multiplexers
- Demultiplexer
- Registers

Half Adder

- A combinational circuit that performs the arithmetic addition of two bits is called a half-adder.
- Two input variables used.
- The output variables are Sum and Carry.
- The variable S represents the least significant bit of the sum.
- The C output is 0 unless both the inputs are 1.

Half Adder

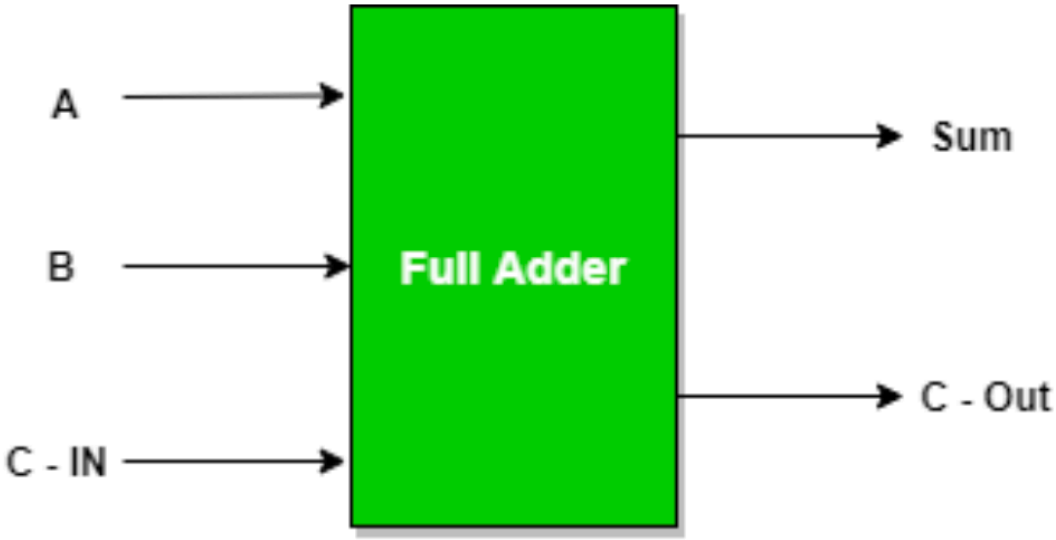


A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Full Adder

- A combinational circuit that performs the arithmetic addition of three bits is called a full-adder.
- Two half-adders are needed to implement a full-adder.
- Three input variables used.
- The output variables are Sum and Carry.
- The variable S represents the least significant bit of the sum.
- The binary variable C gives the output carry.

Full Adder



Full Adder Truth Table:

Inputs			Outputs	
A	B	C – IN	Sum	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-DECODER

The name **"Decoder"** means to translate or decode coded information from one format into another, so a digital decoder transforms a set of digital input signals into an equivalent decimal code at its output.

In **digital electronics**, a binary **decoder** is a combinational **logic** circuit that converts binary information from the **n coded inputs to a maximum of 2^n unique outputs.**

They are used in a wide variety of applications,

- data demultiplexing,
- seven segment displays, and
- memory address **decoding**

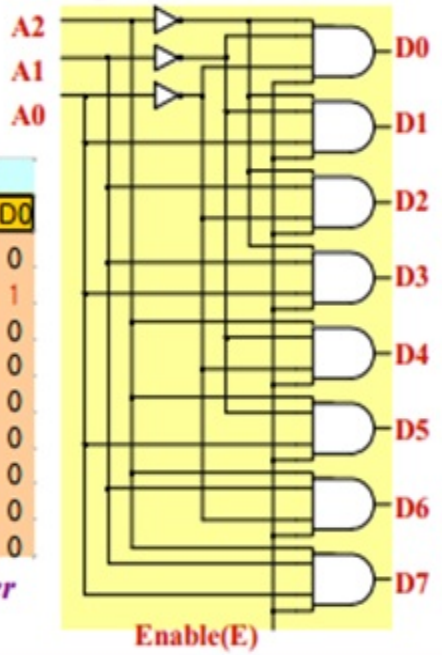
2-2 Decoder/Encoder

- Decoder
  - A combinational circuit that converts binary information from the ***n*** coded inputs to a maximum of 2^n unique outputs
  - ***n*-to-*m*** line decoder = ***n x m*** decoder
    - ***n*** inputs, ***m*** outputs
  - If the n-bit coded information has unused bit combinations, the decoder may have less than 2^n outputs
    - $m \leq 2^n$
- 3-to-8 Decoder
  - A Binary-to-octal conversion
  - Logic Diagram : *Fig. 2-1*
  - Truth Table : *Tab. 2-1*
  - Commercial decoders include one or more Enable Input(E)

Enable	Inputs			Outputs							
E	A2	A1	A0	D7	D6	D5	D4	D3	D2	D1	D0
0	x	x	x	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	1
1	0	0	1	0	0	0	0	0	0	1	0
1	0	1	0	0	0	0	0	0	1	0	0
1	0	1	1	0	0	0	0	1	0	0	0
1	1	0	0	0	0	0	1	0	0	0	0
1	1	0	1	0	0	1	0	0	0	0	0
1	1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	1	0	0	0	0	0	0	0

Tab. 2-1 Truth table for 3-to-8 Decoder

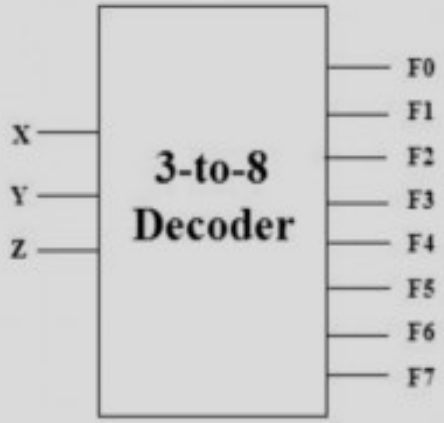
Fig. 2-1 3-to-8 Decoder

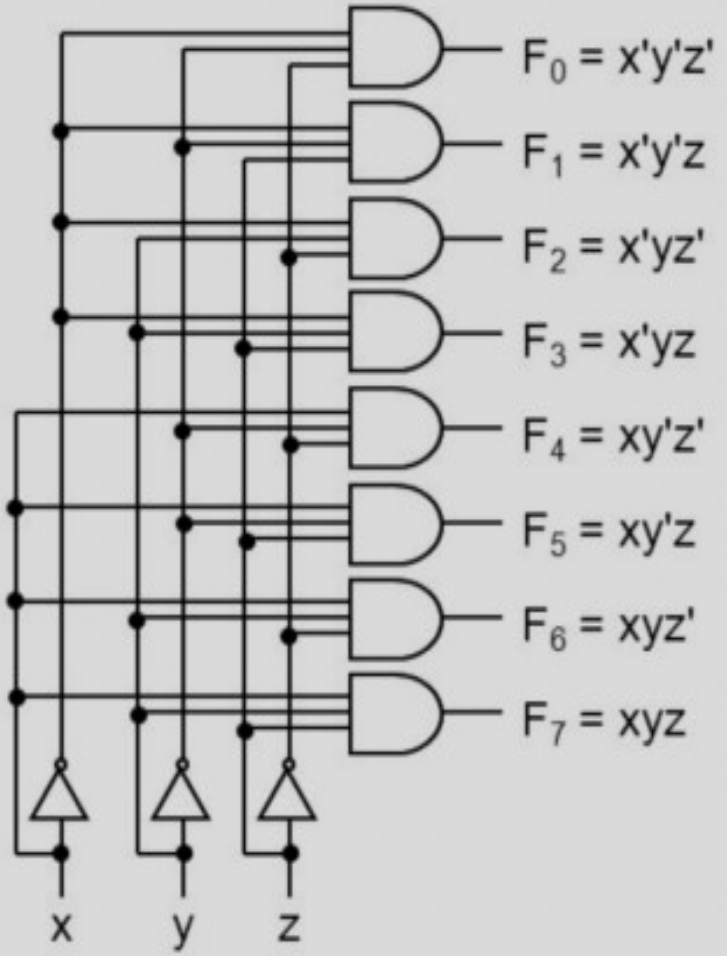


3-to-8 Binary Decoder

Truth Table:

x	y	z	F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1





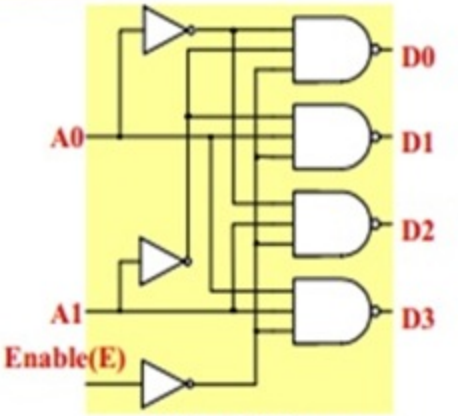
2-2 Decoder/Encoder

- NAND Gate Decoder
  - Constructed with NAND instead of AND gates
  - Logic Diagram/Truth Table : *Fig. 2-2*

* Active Low Output
* Fig. 2-1 3-to-8 Decoder ≒ Active High Output

Enable	Input		Output			
E	A1	A0	D0	D1	D2	D3
0	0	0	0	1	1	1
0	0	1	1	0	1	1
0	1	0	1	1	0	1
0	1	1	1	1	1	0
1	x	x	1	1	1	1

(a) Truth Table



(b) Logic Diagram

Fig. 2-2 2-to-4 Decoder with NAND gates

- Decoder Expansion
  - Constructed decoder : *Fig. 2-3*
  - 3 X 8 Decoder constructed with two 2 X 4 Decoder
- Encoder
  - Inverse Operation of a decoder
  - 2^n input, n output
  - Truth Table : *Tab. 2-2*
    - 3 OR Gates Implementation
      - » A0 = D1 + D3 + D5 + D7
      - » A1 = D2 + D3 + D6 + D7
      - » A2 = D4 + D5 + D6 + D7

Tab. 2-2 Truth Table for Encoder

Inputs								Outputs		
D7	D6	D5	D4	D3	D2	D1	D0	A2	A1	A0
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	1	1	0
1	0	0	0	0	0	0	0	1	1	1

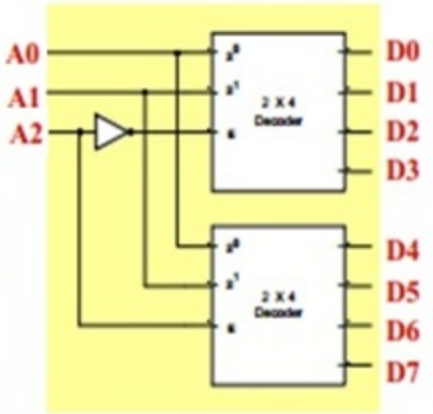


Fig. 2-3 A 3-to-8 Decoder constructed with two with 2-to-4 Decoder

LOVELY PROFESSIONAL UNIVERSITY

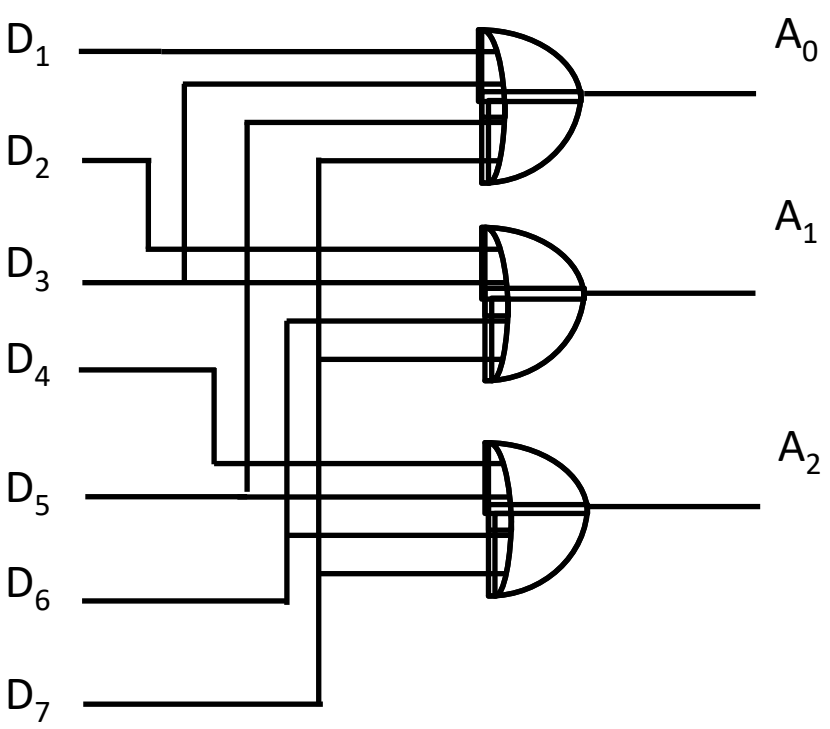
Octal to Binary Encoder

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	A_2	A_1	A_0
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	1	1	0
1	0	0	0	0	0	0	0	1	1	1

$A_0 = D_1 + D_3 + D_5 + D_7$

$A_1 = D_2 + D_3 + D_6 + D_7$

$A_2 = D_4 + D_5 + D_6 + D_7$



If we record any music in any recorder, such types of process is called

_____

a) Multiplexing
b) Encoding
c) Decoding
d) Demultiplexing

Q—A decoder converts n inputs to _____ outputs.

a) n
b) n^2
c) 2^n
d) n^n

Q—-Which of the following represents a number of output lines for a decoder with 4 input lines?

a) 15
b) 16
c) 17
d) 18

2-3 Multiplexers

- Multiplexer(Mux)
  - A combinational circuit that receives binary information from one of 2^n input data lines and directs it to a single output line
  - A 2^n -to 1 multiplexer has *2^n input data lines* and *n input selection lines*(Data Selector)
  - 4-to-1 multiplexer Diagram : ***Fig. 2-4***
  - 4-to-1 multiplexer Function Table : ***Tab. 2-3***

Tab. 2-3 Function Table for 4-to-1 line Multiplexter

Select		Output
S1	S0	Y
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

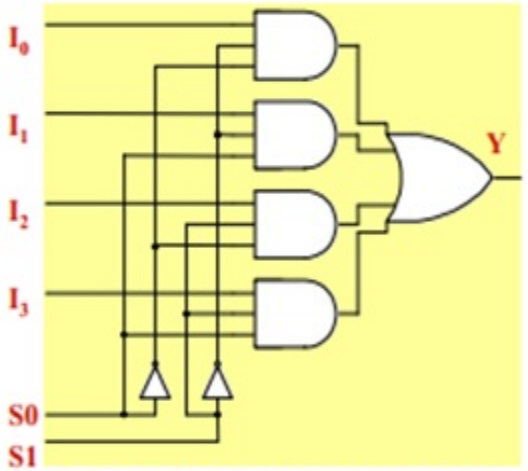


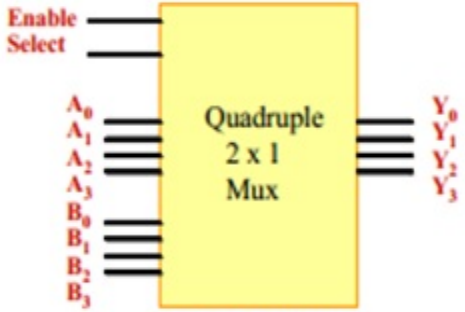
Fig. 2-4 4-to-1 Line Multiplexer

- Quadruple 2-to-1 Multiplexer
  - Quadruple 2-to-1 Multiplexer : ***Fig. 2-5***

Fig. 2-5 Quadruple 2-to-1 line Multiplexter

Select		Output
E	S	Y
0	0	All 0's
1	0	A
1	1	B

(a) Function Table



(b) Block Diagram

Applications of multiplexer

- Data Routing
- Parallel to Serial Conversion
- Logic Function Generation
- As a data selector device
- used in communication systems to increase the efficiency of the system.
- used in telephone networks for integration of several audio signals on a single transmission line

.

2-3 Multiplexers

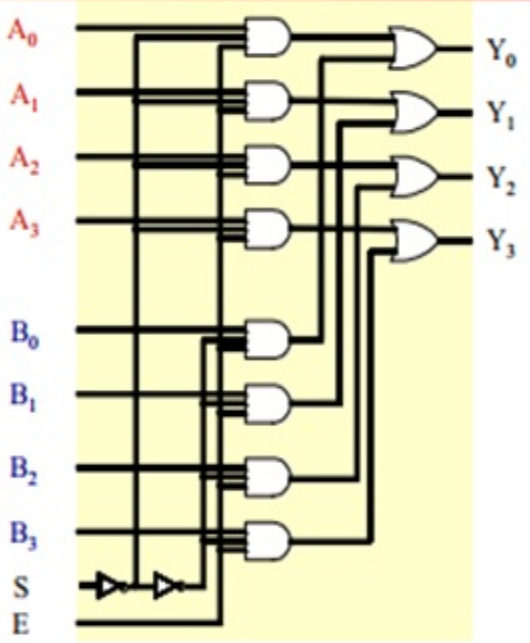
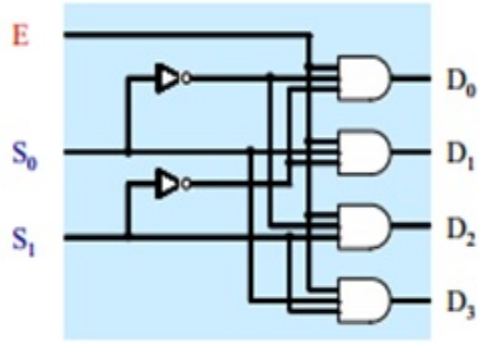


Fig A. Combinational logic diagram with four 2×1 multiplexer



E → Data input
S_0, S_1 → Select Data

Fig B. Demultiplexer

A **Demultiplexer,** sometimes abbreviated **DMUX** is a circuit that has one input and more than one output. It is used when a circuit wishes to send a signal to one of many devices

Use of Demultiplexers:——

- With the help of Demultiplexer, the output of the Arithmetic Logic Unit is stored in several registers.
- Demultiplexers are used in decoder circuits and Boolean function generators.
- Demultiplexer is used to connect a single source to multiple destinations.

2-4 Registers

- Register
  - A group of flip-flops with each flip-flop capable of storing one bit of information
  - An n-bit register has a group of n flip-flops and is capable of storing any binary information of n bits
  - The simplest register consists only of flip-flops, with no external gate : *Fig. 2-6*
  - A clock input C will load all four inputs in parallel
    - The clock must be *inhibited* if the content of the register must be left unchanged
- Register with Parallel Load
  - A 4-bit register with a load control input : *Fig. 2-7*
  - The clock inputs receive clock pulses at all times
  - The buffer gate in the clock input will increase "fan-out"
  - Load Input
    - 1 : Four input transfer
    - 0 : Input inhibited, Feedback from output to input(**no change**)

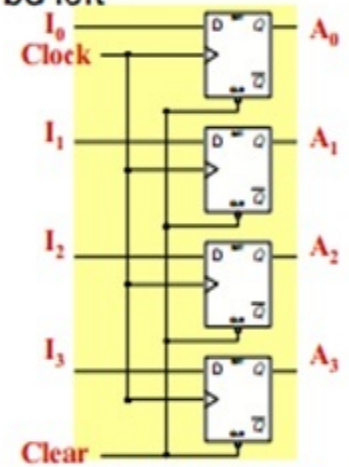


Fig. 2-6 4-bit register

2-4 Registers

- When the load input is 1 , the data in the four inputs are transferred into the register with the next positive transition of a clock pulse

- When the load input is 0, the data inputs are inhibited and the D-output of flip flop are connected to their inputs.

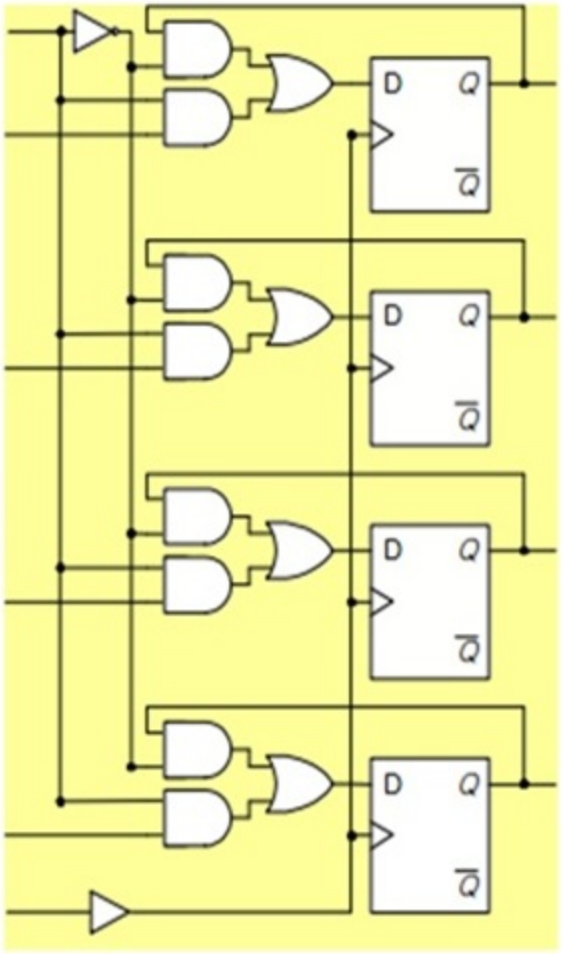
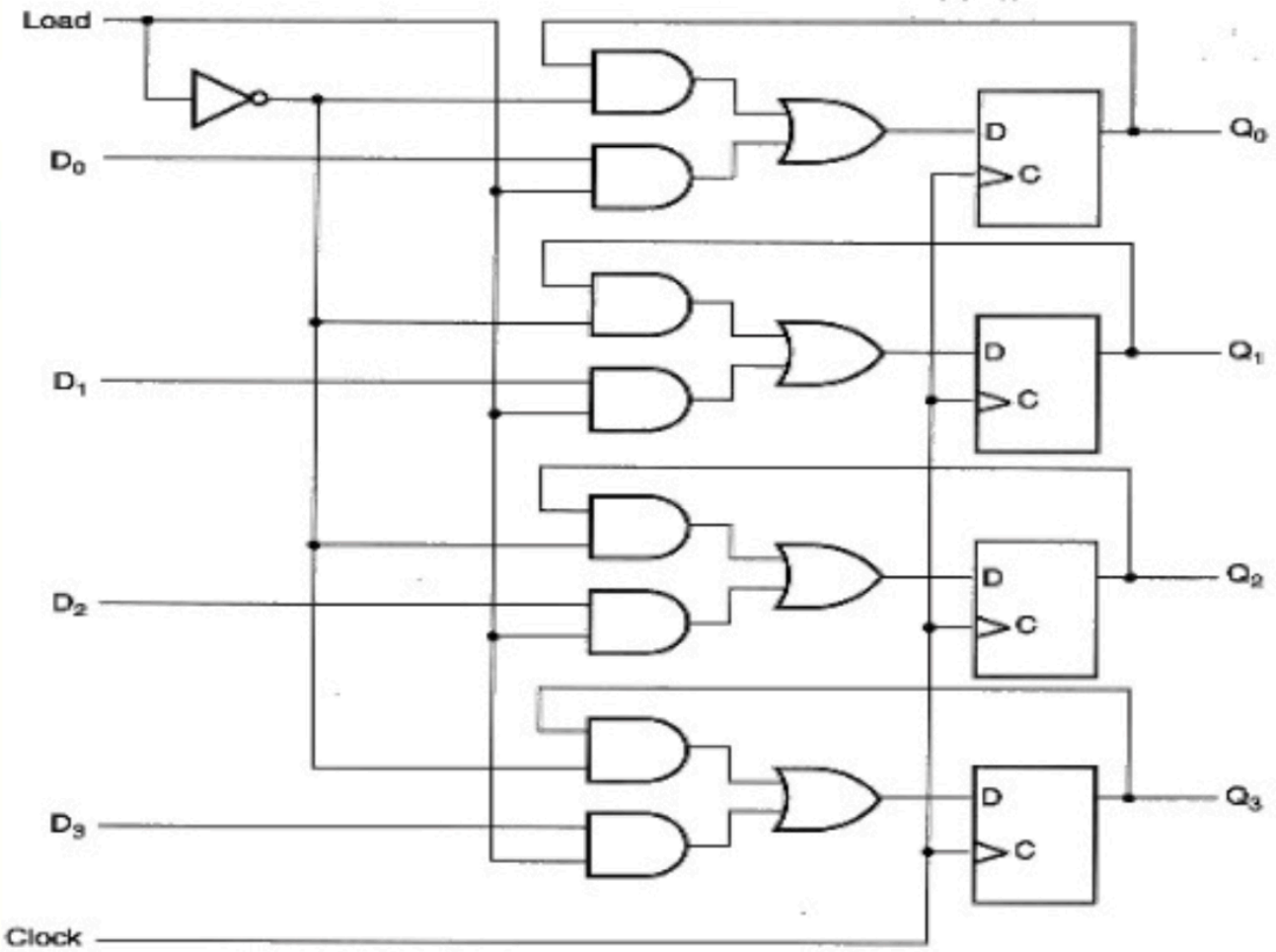


Fig. 2-7 4-bit register with parallel load



4- bit register with parallel load

2-5 Shift Registers

- Shift Register
  - A register capable of shifting its binary information in one or both directions
  - The logical configuration of a shift register consists of a chain of flip-flops in cascade
  - The simplest possible shift register uses only flip-flops : *Fig. 2-8*
  - The ***serial input*** determines what goes into the leftmost position during the shift
  - The ***serial output*** is taken from the output of the rightmost flip-flop

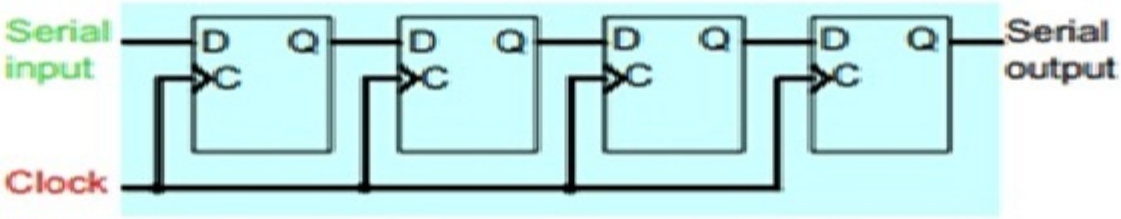


Fig. 2-8 4-bit shift register

2-5 Shift Registers

- Bidirectional Shift Register with Parallel Load
  - A register capable of shifting in *one direction only* is called a ***unidirectional shift register***
  - A register that can shift in *both directions* is called a ***bidirectional shift register***
  - The most general shift register has all the capabilities listed below:
    - An input clock pulse to synchronize all operations
    - A shift-right /left (serial output/input)
    - A parallel load, n parallel output lines
    - The register unchanged even though clock pulses are applied continuously
  - 4-bit bidirectional shift register with parallel load : *Fig. 2-9*
    - 4 X 1 Mux = 4 , D F/F = 4

Tab. 2-4 Function Table for Register of Fig. 2-9

Mode		Operation
S1	S0	
0	0	No chage
0	1	Shift right(down)
1	0	shift left(up)
1	1	Parallel load

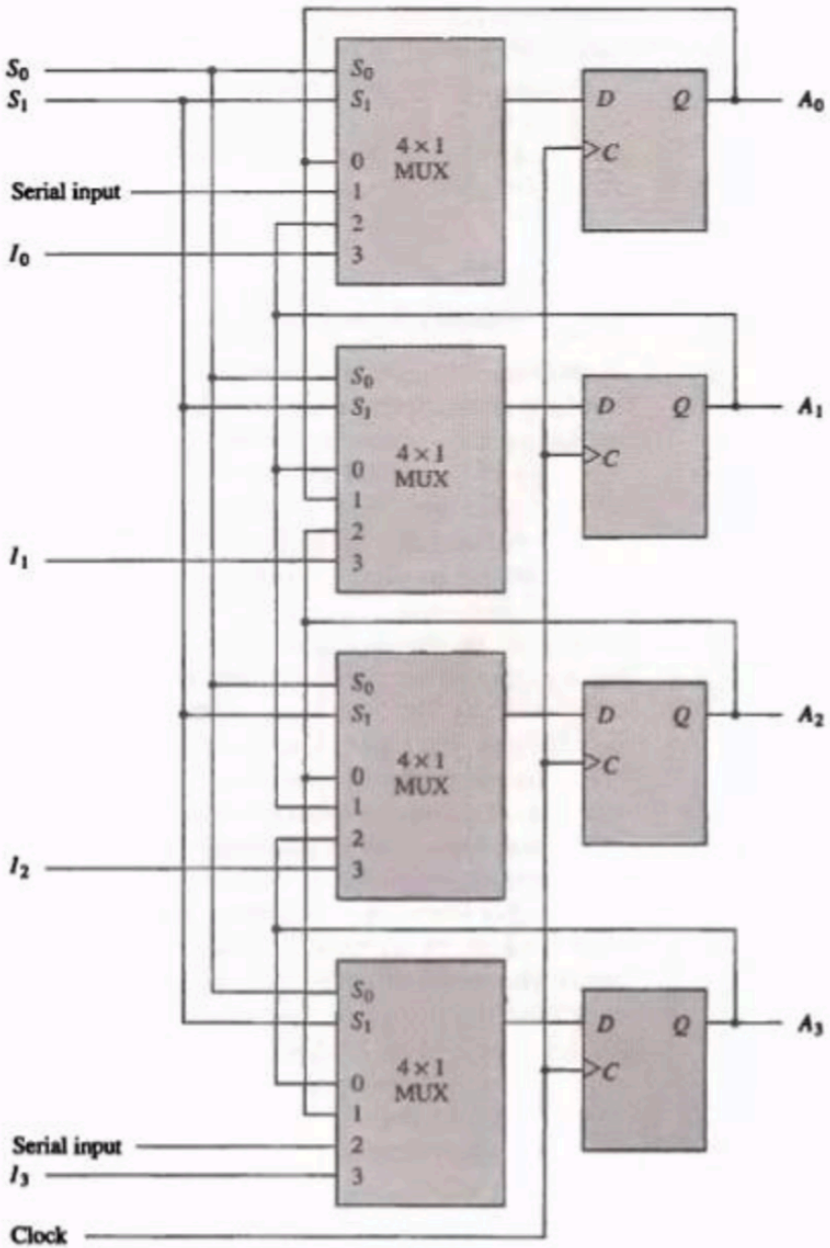
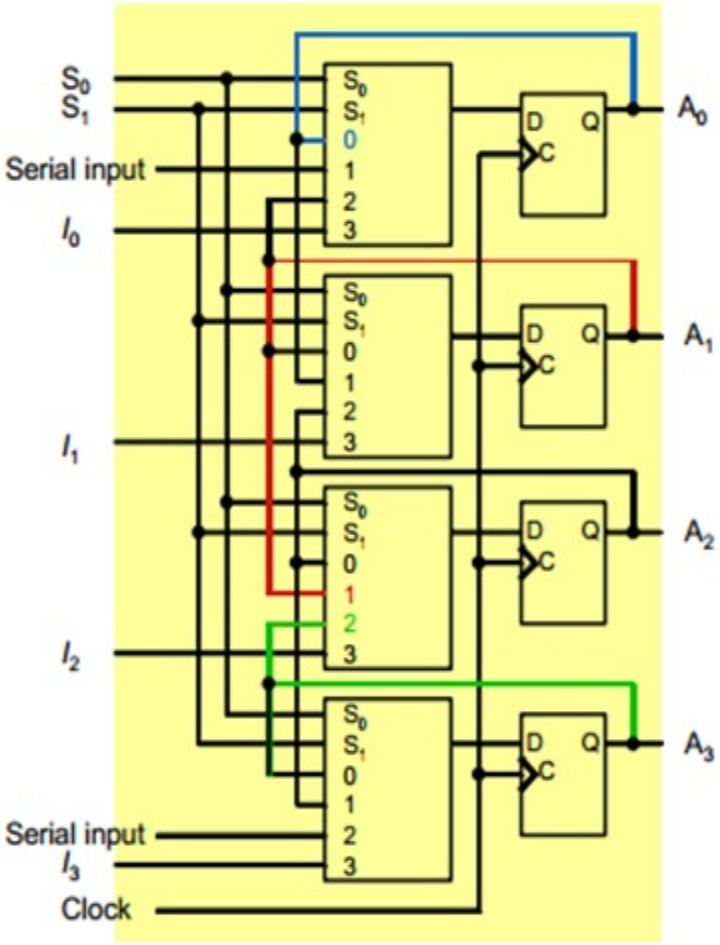


Figure 2-9 Bidirectional shift register with parallel load.

2-5 Shift Registers



$S_1S_0 = 00$: $A_i \rightarrow A_i$ (No change)
$S_1S_0 = 01$: $A_{i-1} \rightarrow A_i$ (Shift)
$S_1S_0 = 10$: $A_{i+1} \rightarrow A_i$ (Shift)
$S_1S_0 = 11$: Parallel load

- Shift Register

  Interface digital systems situated remotely from each other

Fig. 2-9 Bidirectional shift register

Registers capable of shifting in one direction is _____

a) Universal shift register
b) Unidirectional shift register
c) Unipolar shift register
d) Unique shift register

ANSWER——B

Overview

- **Register Transfer Language**
- **Register Transfer**
- Bus and Memory Transfers
- Logic Micro-operations
- Shift Micro-operations
- Arithmetic Logic Shift Unit

Register Transfer Language

- **Combinational and sequential circuits can be used to create simple digital systems.**
- **These are the low-level building blocks of a digital computer.**
- **Simple digital systems are frequently characterised in terms of**
  - **the registers they contain, and**
  - **the operations that are performed on data stored in them**
- **The operations executed on the data in registers are called micro-operations e.g. shift, count, clear and load**

Register Transfer Language

- **Rather than specifying a digital system in words, a specific notation is used, Register Transfer Language**
- **The symbolic notation used to describe the micro operation transfer among register is called a register transfer language**
- **For any function of the computer, the register transfer language can be used to describe the (sequence of) micro-operations**
- **Register transfer language**
  - **A symbolic language**
  - **A convenient tool for describing the internal organization of digital computers in concise/precise manner.**

Register Transfer Language

- **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***
  - **A register can be viewed as a single entity:**

Following are some commonly used registers:

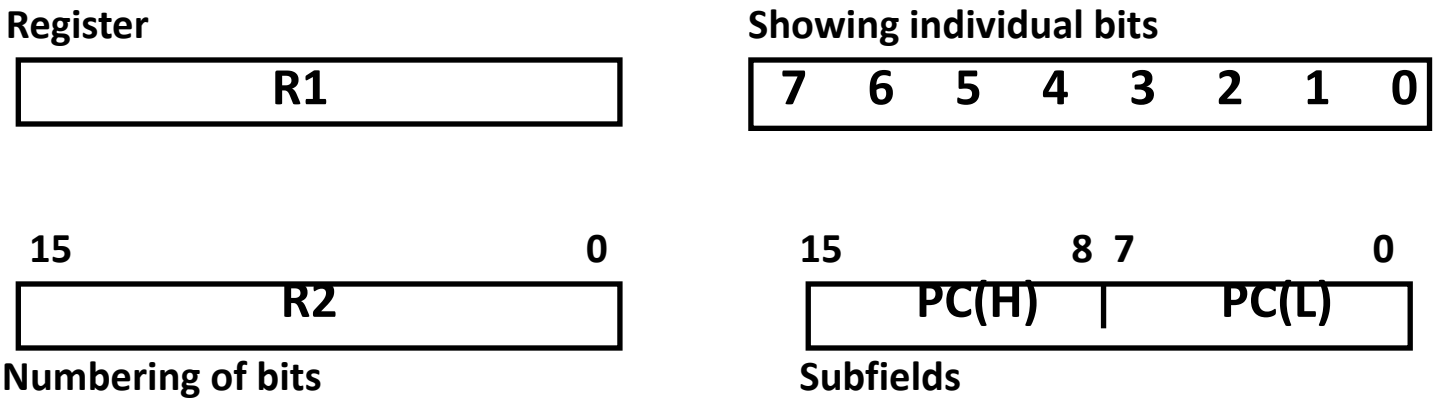
1. **Accumulator**:used to store data taken out from memory.
2. **General Purpose Registers**: This is used to store data intermediate results during program execution.
3. **Special Purpose Registers**: Users do not access these registers. These registers are for Computer system,
   - **MAR:** Memory Address Register:- holds the address for memory unit.
   - **MBR:** Memory Buffer Register stores instruction and data received from the memory and sent from the memory.
   - **PC:** Program Counter points to the next instruction to be executed.
   - **IR:** Instruction Register holds the instruction to be executed.

Register Transfer Language

- **Designation of a register**

  - a register
  - portion of a register
  - a bit of a register

- **Common ways of drawing the block diagram of a register**



Register Transfer Language

- **Copying the contents of one register to another is a register transfer**
- **A register transfer is indicated as**

R2 ← R1

- ➢ **In this case the contents of register R1 are copied (loaded) 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 copying (loading) them to R2**

Control Functions

- **Often actions need to only occur if a certain condition is true**
- **This is 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 ← R1

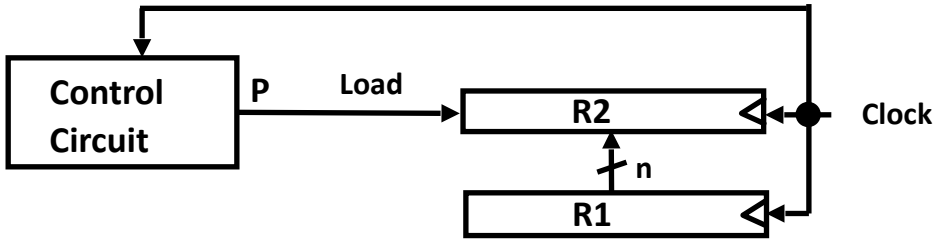
Which means "if P = 1, then load the contents of register R1 into register R2", i.e., if (P = 1) then (R2 ← R1)

Hardware Implementation of Controlled Transfers

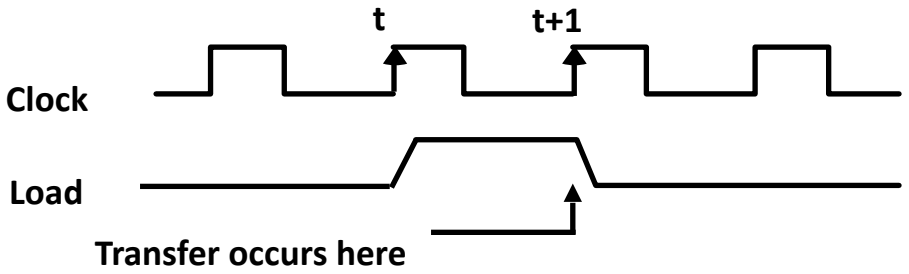
Implementation of controlled transfer

P: R2 ← R1

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 in Register Transfer

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

Micro operation is shown as:

A——R1->R2

B——R1<-R2

C——Both

D——None

Connecting Registers - Bus Transfer

- 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

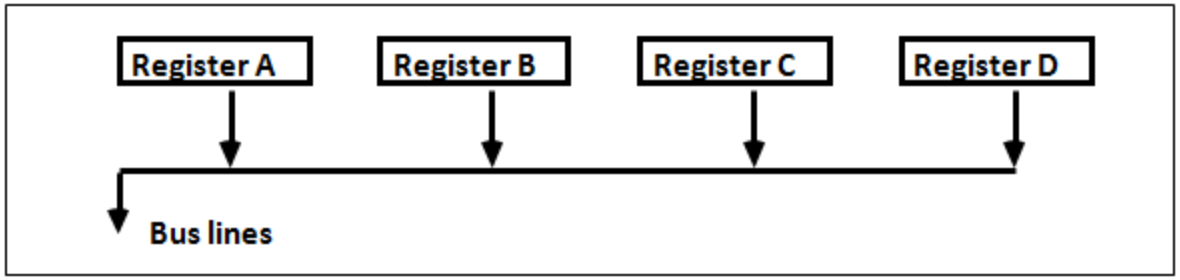
- To completely connect n registers → n(n-1) lines
- $O(n^2)$ cost
  - This is not a realistic approach to use in a large digital system

- Instead, take a different approach
- Have one centralized set of circuits for data transfer – the bus

- BUS STRUCTURE CONSISTS OF SET OF COMMON LINES, ONE FOR EACH BIT OF A REGISTER THROUGH WHICH BINARY INFORMATION IS TRANSFERRED ONE AT A TIME
- Have control circuits to select which register is the source, and which is the destination

Connecting Registers - Bus Transfer

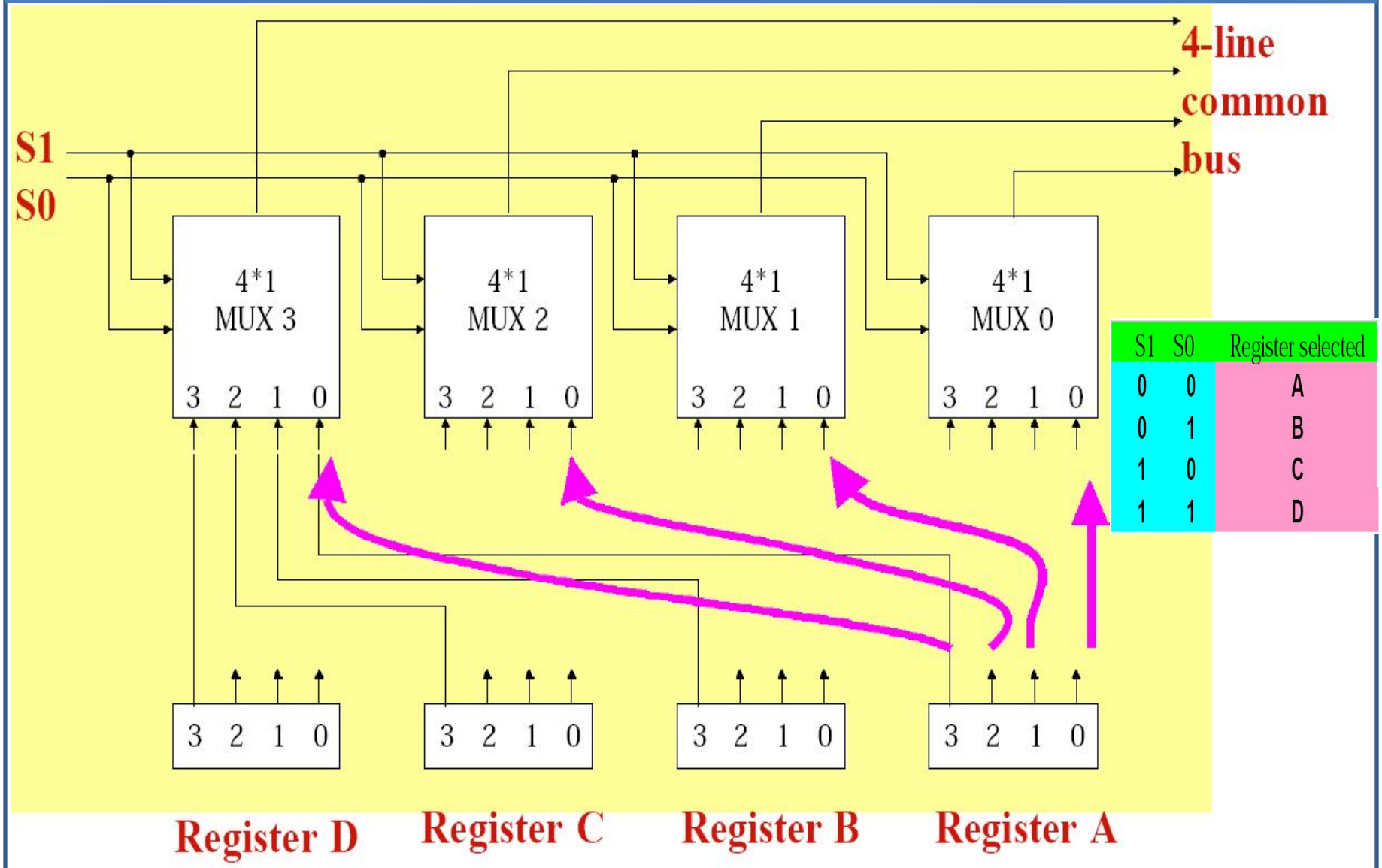
From a register to bus: BUS ← R



- One way of constructing common bus system is with multiplexers
- Multiplexer selects the source register whose binary information is kept on the bus.

- Construction of bus system for 4 register (Next Fig)
  - 4 bit register X 4
  - four 4X1 multiplexer
  - Bus selection S0, S1

Connecting Registers - Bus Transfer



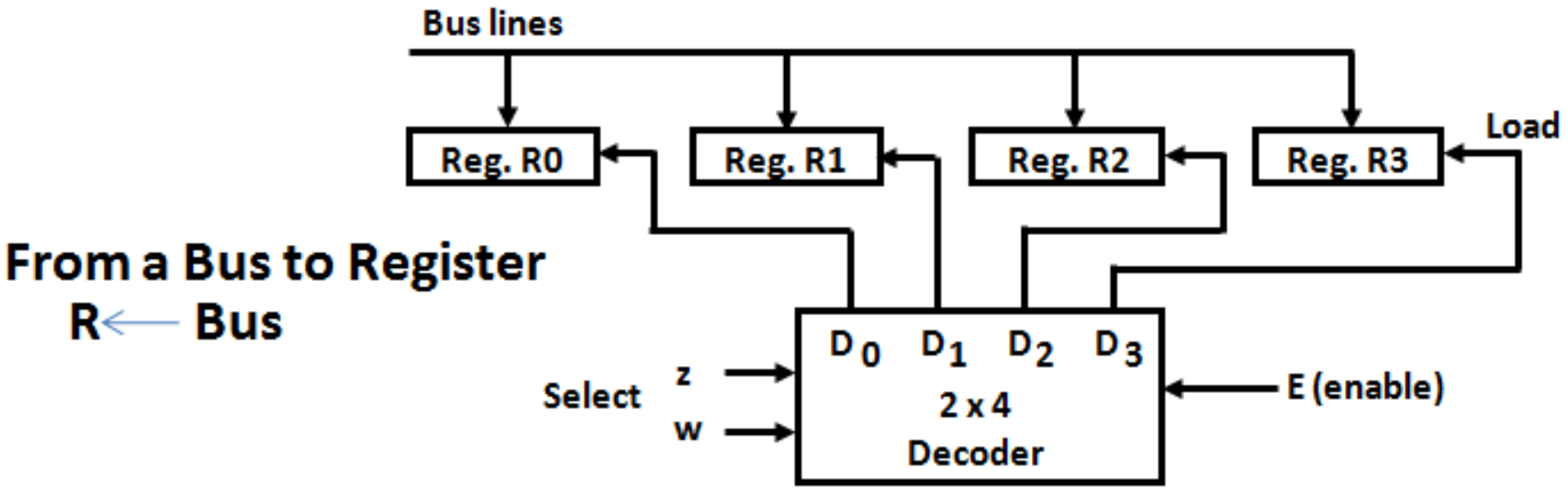
S1	S0	Register selected
0	0	A
0	1	B
1	0	C
1	1	D

Connecting Registers - Bus Transfer

- For a bus system to multiplex k registers of n bits each
  - No. of multiplexer = n = No. of bits
  - Size of each multiplexer = k x 1, k data lines in each MUX

- **Construction of bus system for 8 register with 16 bits**
  - **16 bit register X 8**
  - **Sixteen 8X1 multiplexer**
  - **Bus selection S0, S1, S2**

Connecting Registers - Bus Transfer



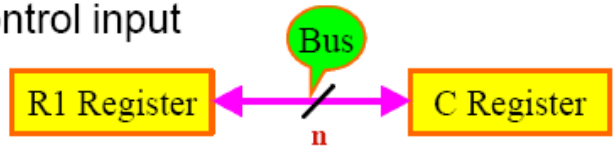
- **If we Construct a bus system for 64 register with 128 bits, how many MUX required ?**

- **A—- 188**
- **B—-128**
- **C—-16**
- **D—-64**

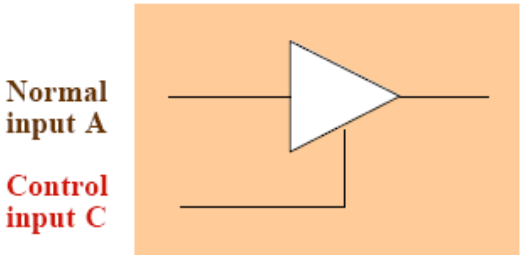
Connecting Registers - Bus Transfer

- Bus Transfer
  - The content of register C is placed on the bus, and the content of the bus is loaded into register R1 by activating its load control input

$$\left.\begin{array}{l} Bus \leftarrow C,\ R1 \leftarrow Bus \\ R1 \leftarrow C \end{array}\right\} =$$



- Three-State Bus Buffers
  - A bus system can be constructed with ***three-state gates*** *instead of* ***multiplexers***
  - Tri-State : 0, 1, High-impedance(***Open circuit***)
  - Buffer
    - » A device designed to be inserted between other devices to match impedance, to prevent mixed interactions, and ***to supply additional drive or relay capability***
    - » Buffer types are classified as inverting or noninverting
  - Tri-state buffer gate : Fig. 4-4
    - » When control input =1 : The output is enabled(output Y = input A)
    - » When control input =0 : The output is disabled(output Y = high-impedance)

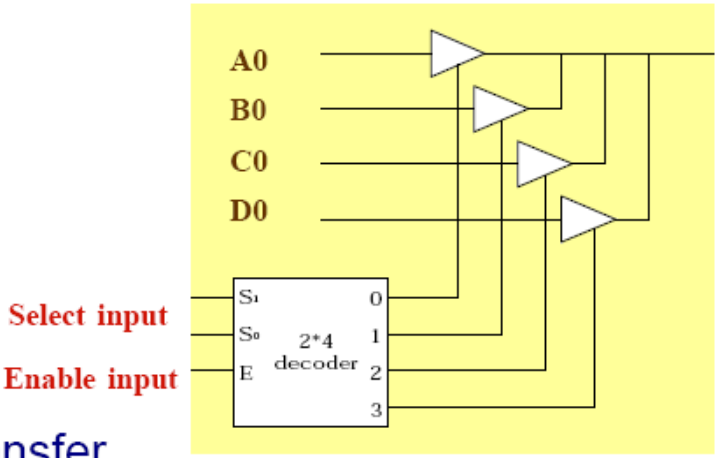


If C=1, Output Y = A

If C=0, Output = High-impedance

Connecting Registers - Bus Transfer

- The construction of a bus system with tri-state buffer : ***Fig.***
  - The outputs of four buffer are connected together to form a single bus line(Tri-state buffer
  - No more than one buffer may be in the active state at any given time(2 X 4 Decoder
  - To construct a common bus for 4 register with 4 bit : Fig.



AR: Address Reg.
DR: Data Reg.
M : Memory Word(Data)

$READ: \quad DR \leftarrow M[AR]$

$WRITE: \quad M[AR] \leftarrow R1$

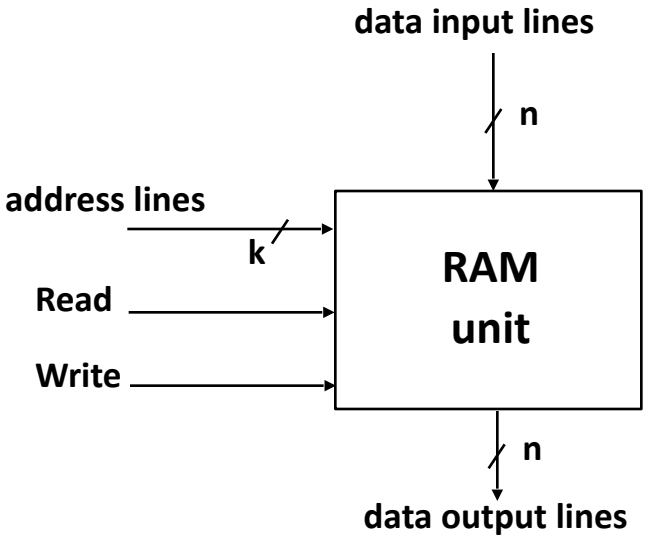
- Memory Transfer
  - Memory read : A transfer information into DR from the memory word M selected by the address in AR
  - Memory Write : A transfer information from R1 into the memory word M selected by the address in AR

What will be the output of TRI-STATE BUS BUFFER if input is 0 ?

A—-the "1" (H level) state
B—-the "0" (L level) state
C— the high impedance state

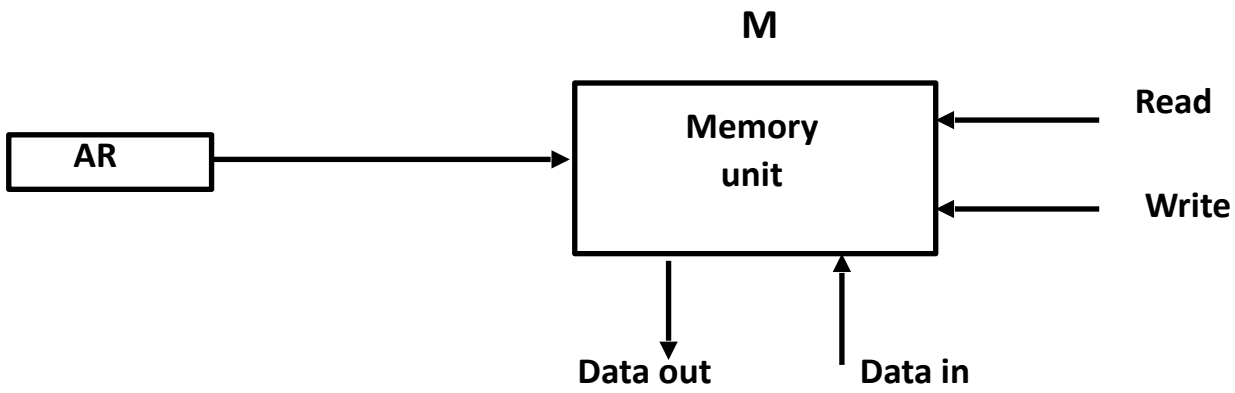
Memory - RAM

- Memory (RAM) can be thought as a sequential circuits containing some number of registers
- Memory stores binary information in groups of bits called words
- These registers hold the *words* of memory
- Each of the r registers is indicated by an *address*
- These addresses range from 0 to r-1
- Each register (word) can hold n bits of data
- Assume the RAM contains $r = 2^k$ words. It needs the following
  1. n data input lines
  2. n data output lines
  3. k address lines
  4. A Read control line
  5. A Write control line



Memory Transfer

Memory is usually accessed in computer systems by putting the desired address in a special register, the Memory Address Register (MAR, or AR)



Memory Read

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

$$R1 \leftarrow M[MAR]$$

- **This causes the following to occur**
  1. **The contents of the MAR get sent to the memory address lines**
  2. **A Read (= 1) gets sent to the memory unit**
  3. **The contents of the specified address are put on the memory's output data lines**
  4. **These get sent over the bus to be loaded into register R1**

Memory Write

- **To write a value from a register to a location in memory looks like this in register transfer language:**

$$M[MAR] \leftarrow R1$$

- **This causes the following to occur**
  1. **The contents of the MAR get sent to the memory address lines**
  2. **A Write (= 1) gets sent to the memory unit**
  3. **The values in register R1 get sent over the bus to the data input lines of the memory**
  4. **The values get loaded into the specified address in the memory**

MICROOPERATIONS

Computer system micro-operations are of four types:

- **Register transfer micro-operations**
- **Arithmetic micro-operations**
- **Logic micro-operations**
- **Shift micro-operations**

SUMMARY OF R. TRANSFER MICROOPERATIONS

A ← B	1.Transfer content of reg. B into reg. A
A ← constant	3.Transfer a binary constant into reg. A
ABUS ← R1, R2 ← ABUS	4.Transfer content of R1 into bus A and, at the same time, transfer content of bus A into R2
AR	5.Address register
DR	6.Data register
M[R]	7.Memory word specified by reg. R
M	8.Equivalent to M[AR]
DR ← M	9.Memory *read* operation: transfers content of memory word specified by AR into DR
M ← DR	10.Memory *write* operation: transfers content of DR into memory word specified by AR

Arithmetic MICROOPERATIONS

- The basic arithmetic microoperations are
  - Addition
  - Subtraction
  - Increment
  - Decrement

- The additional arithmetic microoperations are
  - Add with carry
  - Subtract with borrow
  - Transfer/Load
  - etc. …

Summary of Typical Arithmetic Micro-Operations

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

Overview

- Register Transfer Language
- Register Transfer
- Bus and Memory Transfers
- **Arithmetic Micro-operations**
- Logic Micro-operations
- Shift Micro-operations
- Arithmetic Logic Shift Unit

MICROOPERATIONS

Computer system microoperations are of four types:

- **Register transfer microoperations**
- **Arithmetic microoperations**
- **Logic microoperations**
- **Shift microoperations**

Arithmetic MICROOPERATIONS

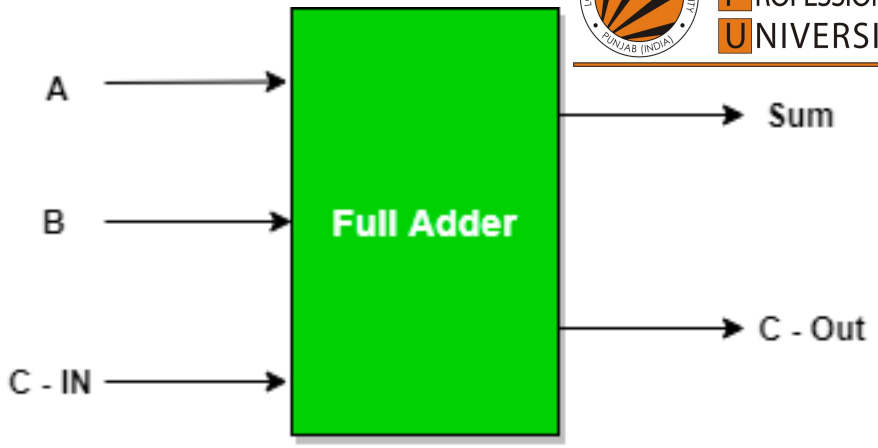
- The basic arithmetic microoperations are
  - Addition
  - Subtraction
  - Increment
  - Decrement

- The additional arithmetic microoperations are
  - Add with carry
  - Subtract with borrow
  - Transfer/Load
  - etc. …

Summary of Typical Arithmetic Micro-Operations

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

Full Adder is the adder which adds three inputs and produces two outputs. The first two inputs are A and B and the third input is an input carry as C-IN. The output carry is designated as C-OUT and the normal output is designated as S which is SUM.



Inputs			Outputs	
A	B	C – IN	Sum	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

- 4-bit Binary Adder : ***Fig. 4-6***
  - Full adder = 2-bits sum + previous carry
  - Binary adder = the arithmetic sum of two binary numbers of any length
  - c_0(input carry), c_4(output carry)

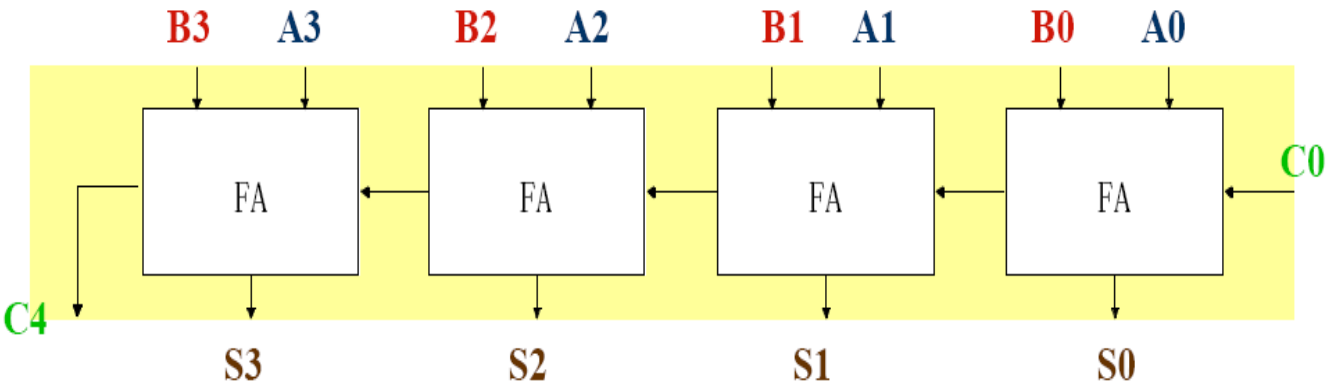
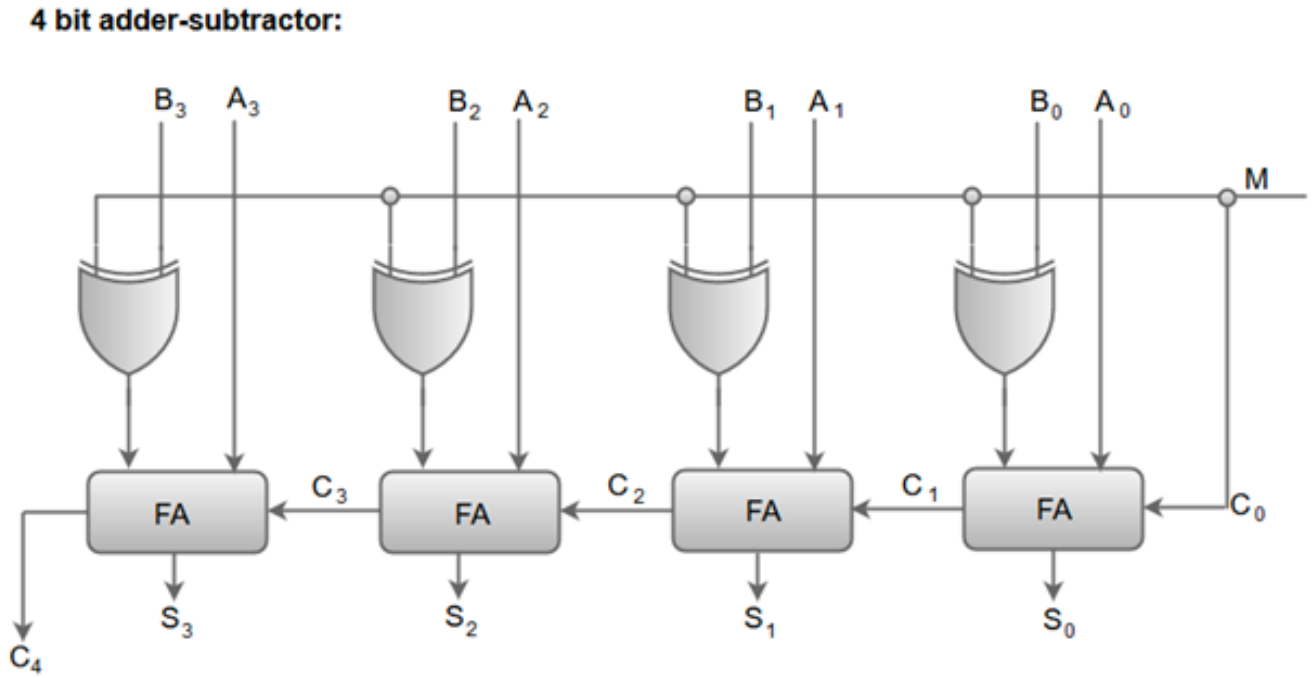


Figure 4-6. 4-bit binary adder

Binary Adder-Subtractor

Binary Adder-Subtractor

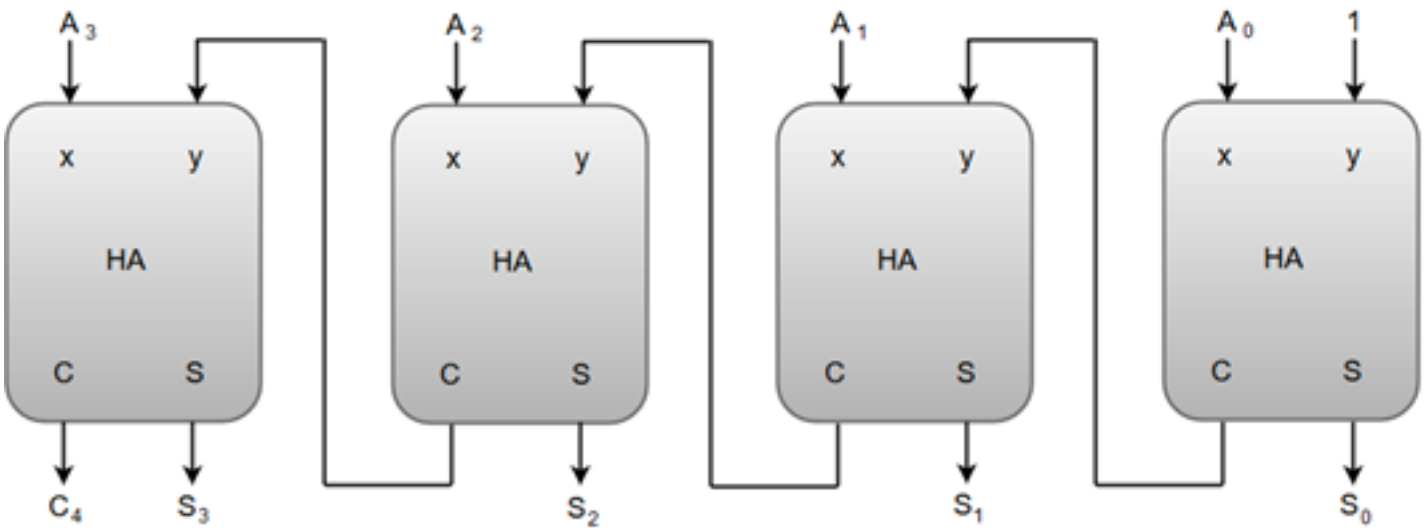


- Mode input M controls the operation
  - M=0 ---- adder
  - M=1 ---- subtractor

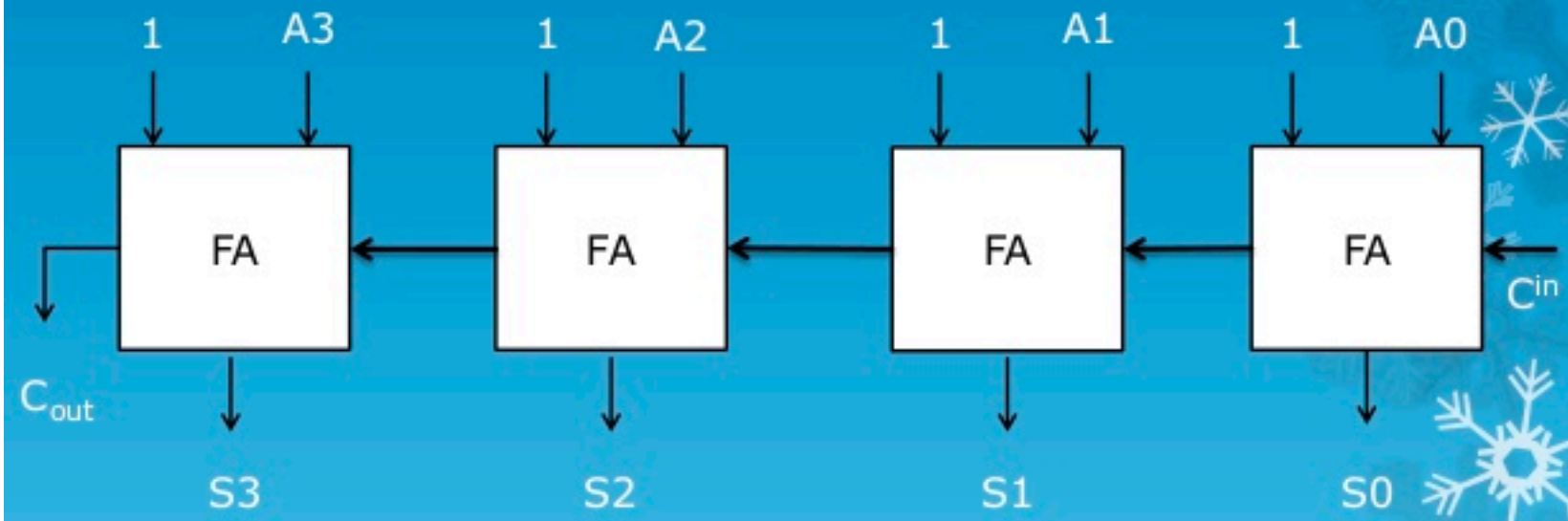
Binary Incrementer

Binary Incrementer

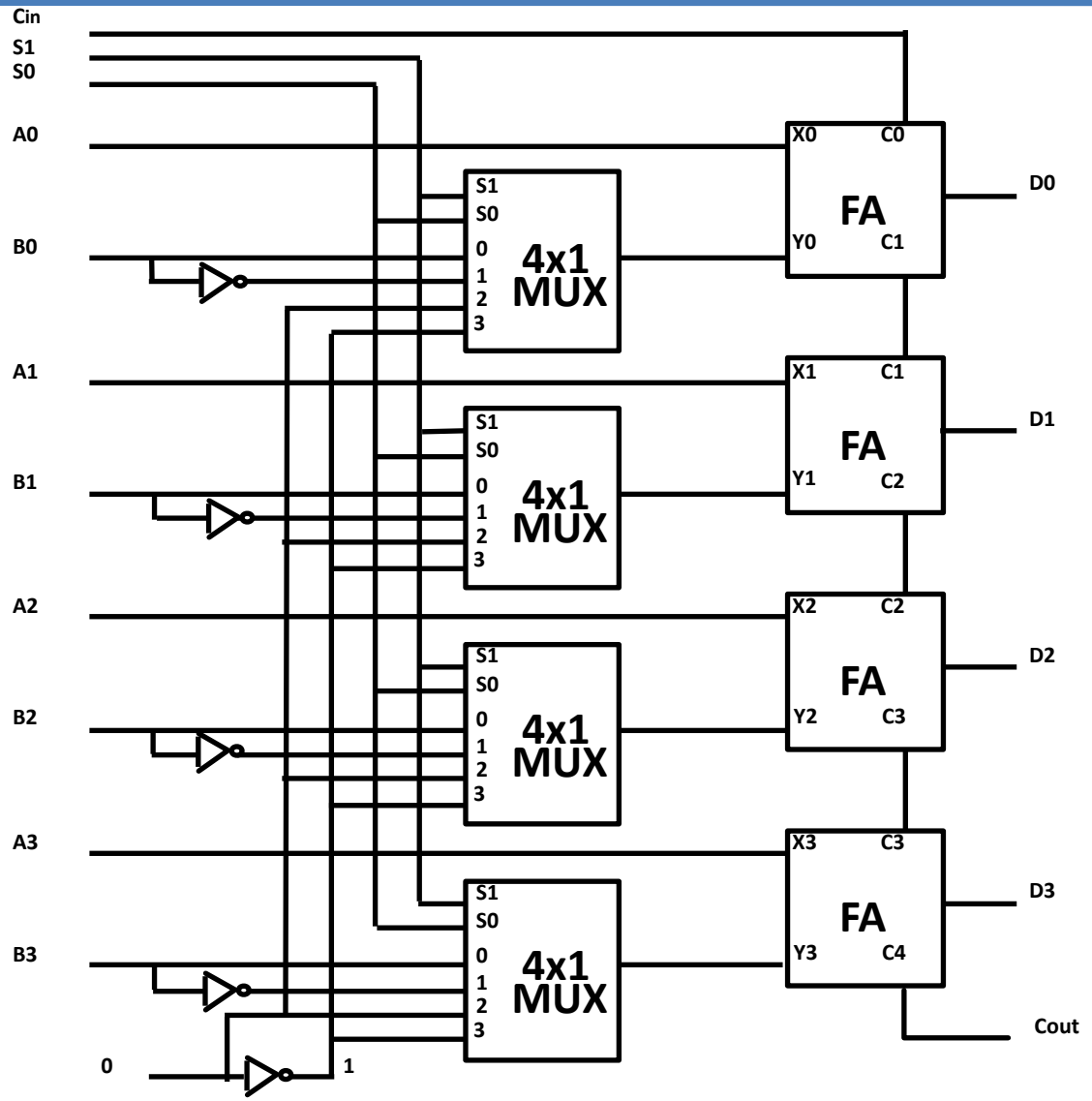
4-bit binary incrementer:



Binary Decrement Using Full Adder (4-bit)



Arithmetic Circuits



Select			Input	Output
S1	S0	C_{in}	Y	D=A+Y+C_{in}
0	0	0	B	D=A+B
0	0	1	B	D=A+B+1
0	1	0	B'	D=A+B'
0	1	1	B'	D=A+B'+1
1	0	0	0	D=A
1	0	1	0	D=A+1
1	1	0	1	D=A-1
1	1	1	1	D=A

TABLE Arithmetic Circuit Function Table

Select			Input	Output	
S_1	S_0	C_{in}	Y	$D = A + Y + C_{in}$	Microoperation
0	0	0	B	$D = A + B$	Add
0	0	1	B	$D = A + B + 1$	Add with carry
0	1	0	$\overline{B}$	$D = A + \overline{B}$	Subtract with borrow
0	1	1	$\overline{B}$	$D = A + \overline{B} + 1$	Subtract
1	0	0	0	$D = A$	Transfer A
1	0	1	0	$D = A + 1$	Increment A
1	1	0	1	$D = A - 1$	Decrement A
1	1	1	1	$D = A$	Transfer A

Logic Micro operations

- Logic microoperation
  - Logic microoperations consider ***each bit of the register separately*** and treat them as binary variables
    - » *exam)*

$P: R1 \leftarrow R1 \oplus R2$

```
  1010  Content of R1
+ 1100  Content of R2
  0110  Content of R1 after P=1
```

  - Special Symbols
    - » Special symbols will be adopted for the logic microoperations ***OR(∨), AND(∧),*** and ***complement(a bar on top)***, to distinguish them from the corresponding symbols used to express Boolean functions
    - » *exam)*

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

Logic OR

Arithmetic ADD

- List of Logic Microoperation
  - Truth Table for 16 functions for 2 variables : ***Tab. 4-5***
  - 16 Logic Microoperation : ***Tab. 4-6***
- Hardware Implementation
  - 16 microoperation → Use only 4(AND, OR, XOR, Complement)
  - One stage of logic circuit

∵ All other Operation can be derived

Logic Microoperations

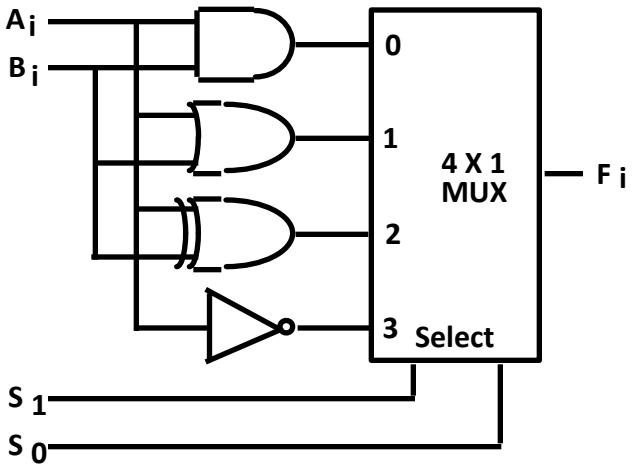
X	Y	F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}	F_{11}	F_{12}	F_{13}	F_{14}	F_{15}
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

TABLE 4-5. Truth Table for 16 Functions of Two Variables

Boolean function	Microoperation	Name
$F_0 = 0$	$F \leftarrow 0$	Clear
$F_1 = xy$	$F \leftarrow A \wedge B$	AND
$F_2 = xy'$	$F \leftarrow A \wedge \overline{B}$	
$F_3 = x$	$F \leftarrow A$	Transfer A
$F_4 = x'y$	$F \leftarrow \overline{A} \wedge B$	
$F_5 = y$	$F \leftarrow B$	Transfer B
$F_6 = x \oplus y$	$F \leftarrow A \oplus B$	Ex-OR
$F_7 = x+y$	$F \leftarrow A \vee B$	OR
$F_8 = (x+y)'$	$F \leftarrow \overline{A \vee B}$	NOR
$F_9 = (x \oplus y)'$	$F \leftarrow \overline{A \oplus B}$	Ex-NOR
$F_{10} = y'$	$F \leftarrow \overline{B}$	Compl-B
$F_{11} = x+y'$	$F \leftarrow A \vee \overline{B}$	
$F_{12} = x'$	$F \leftarrow \overline{A}$	Compl-A
$F_{13} = x'+y$	$F \leftarrow \overline{A} \vee B$	
$F_{14} = (xy)'$	$F \leftarrow \overline{A \wedge B}$	NAND
$F_{15} = 1$	$F \leftarrow$ all 1's	set to all 1's

TABLE 4-6. Sixteen Logic Microoperations

Hardware Implementation



Function table

S_1 S_0	Output	μ-operation
0 0	F = A ∧ B	AND
0 1	F = A ∨ B	OR
1 0	F = A ⊕ B	XOR
1 1	F = A'	Complement

Applications of Logic Microoperation

- **Logic microoperations can be used to manipulate individual bits or a portions of a word in a register**

- **Consider the data in a register A. In another register, B, is bit data that will be used to modify the contents of A**

Selective-set	**$A \leftarrow A + B$**
Selective-complement	**$A \leftarrow A \oplus B$**
Selective-clear	**$A \leftarrow A \bullet B'$**
Mask (Delete)	**$A \leftarrow A \bullet B$**
Clear	**$A \leftarrow A \oplus B$**
Insert	**$A \leftarrow (A \bullet B) + C$**
Compare	**$A \leftarrow A \oplus B$**

Applications of Logic Microoperation

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

1 1 0 0 A_t

1 0 1 0 B

1 1 1 0 A_{t+1} (A ← A + 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

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

1 1 0 0 A_t

1 0 1 0 B

0 1 1 0 A_{t+1} (A ← A ⊕ 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

Applications of Logic Microoperation

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

$$1\ 1\ 0\ 0 \quad A_t$$
$$1\ 0\ 1\ 0 \quad B$$
$$0\ 1\ 0\ 0 \quad A_{t+1} \qquad (A \leftarrow A \cdot B')$$

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

4. In a mask operation, the bit pattern in B is used to *clear* certain bits in A

$$1\ 1\ 0\ 0 \quad A_t$$
$$1\ 0\ 1\ 0 \quad B$$
$$1\ 0\ 0\ 0 \quad A_{t+1} \qquad (A \leftarrow A \cdot B)$$

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

Applications of Logic Microoperation

5. 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

1 1 0 0 A_t

1 0 1 0 B

0 1 1 0 A_{t+1} $(A \leftarrow A \oplus B)$

Applications of Logic Microoperation

6. **An insert operation is used to introduce a specific bit pattern into A register, leaving the other bit positions unchanged**

   This is done as

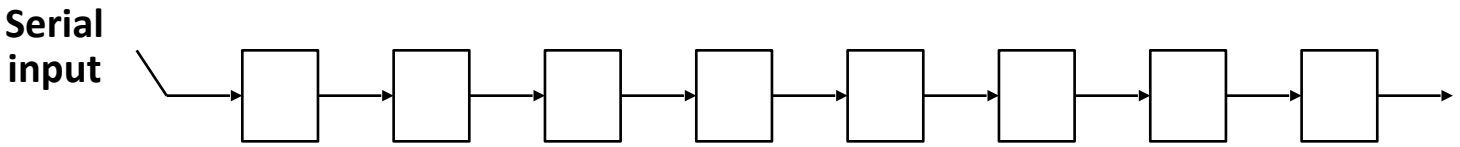
   - **A mask operation to clear the desired bit positions, followed by**
   - **An OR operation to introduce the new bits into the desired positions**
   - **Example**
     - **Suppose you wanted to introduce 1010 into the low order four bits of A:**
     - **1101 1000 1011 0001 A (Original)**
       1101 1000 1011 1010 A (Desired)

     - `1101 1000 1011 0001  A (Original)`
       `1111 1111 1111 0000  Mask`
       `1101 1000 1011 0000  A (Intermediate)`
       `0000 0000 0000 1010  Added bits`
       `1101 1000 1011 1010  A (Desired)`

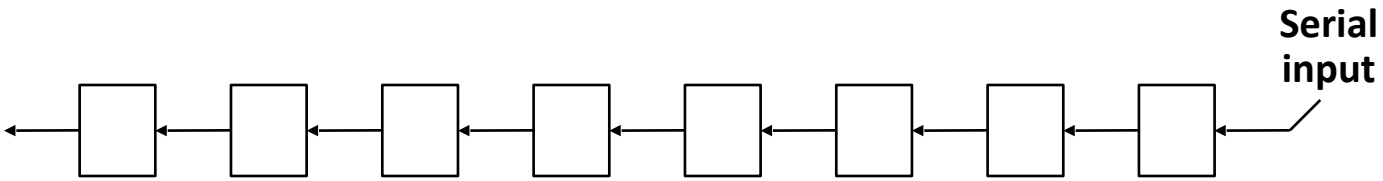
Shift Microoperations

- **There are three types of shifts**
  - ***Logical shift***
  - ***Circular shift***
  - ***Arithmetic shift***
- **What differentiates them is the information that goes into the serial input**

- **A right shift operation**

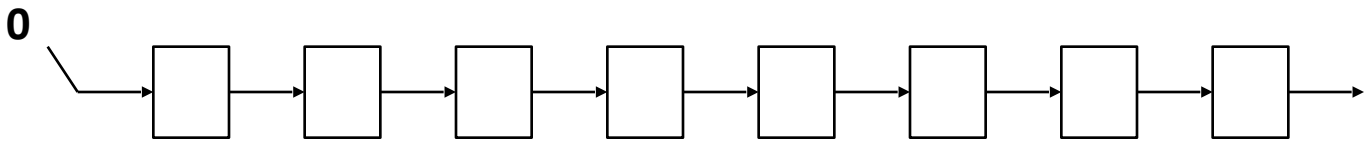


- **A left shift operation**

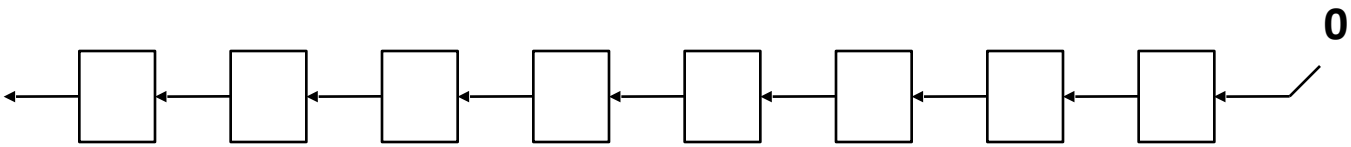


Logical Shift

- **In a logical shift the serial input to the shift is a 0.**
- **A right logical shift operation:**



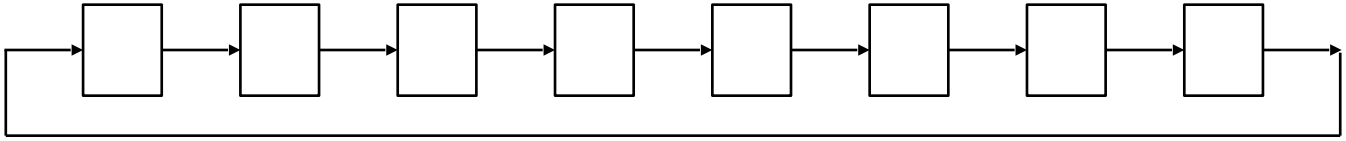
- **A left logical shift operation:**



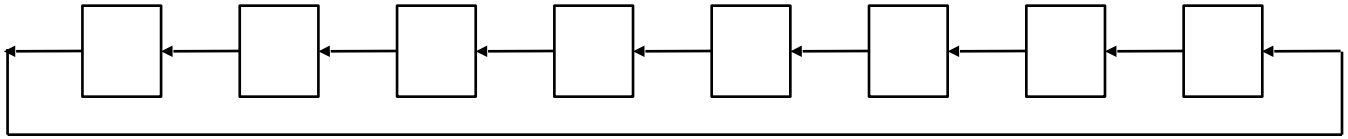
- **In a Register Transfer Language, the following notation is used**
  - ***shl*** **for a logical shift left**
  - ***shr*** **for a logical shift right**
  - **Examples:**
    - **R2 ← *shr* R2**
    - **R3 ← *shl* R3**

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:**

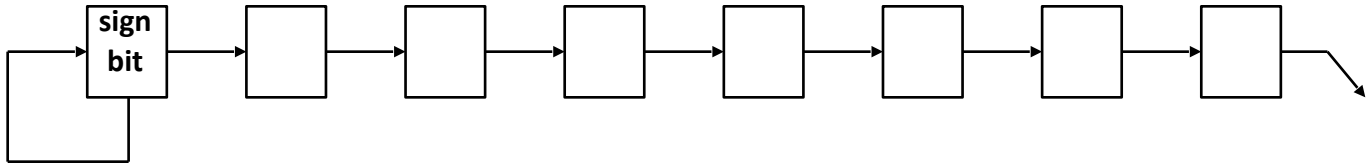


- **In a RTL, the following notation is used**
  - ***cil*** **for a circular shift left**
  - ***cir*** **for a circular shift right**
  - **Examples:**
    - **R2 ← *cir* R2**
    - **R3 ← *cil* R3**

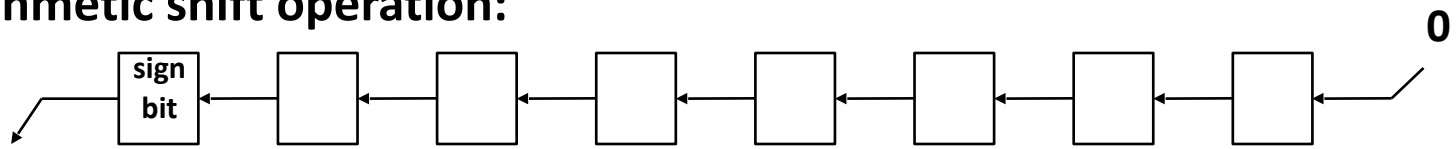
Arithmetic Shift

- An arithmetic shift is meant for signed binary numbers (integer)
- An arithmetic left shift multiplies a signed number by two
- An arithmetic right shift divides a signed number by two
- Sign bit : 0 for positive and 1 for negative
- The main distinction of an arithmetic shift is that it must keep the sign of the number the same as it performs the multiplication or division

- A right arithmetic shift operation:

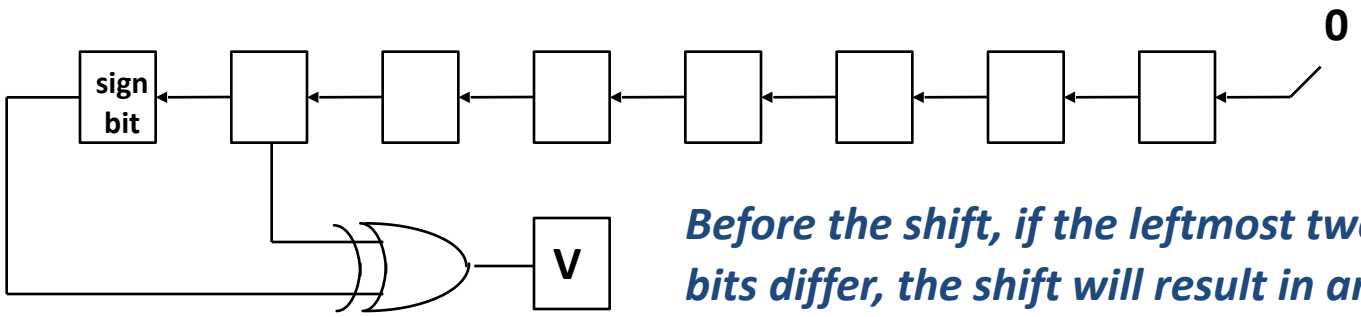


- A left arithmetic shift operation:



Arithmetic Shift

- **An left arithmetic shift operation must be checked for the overflow**

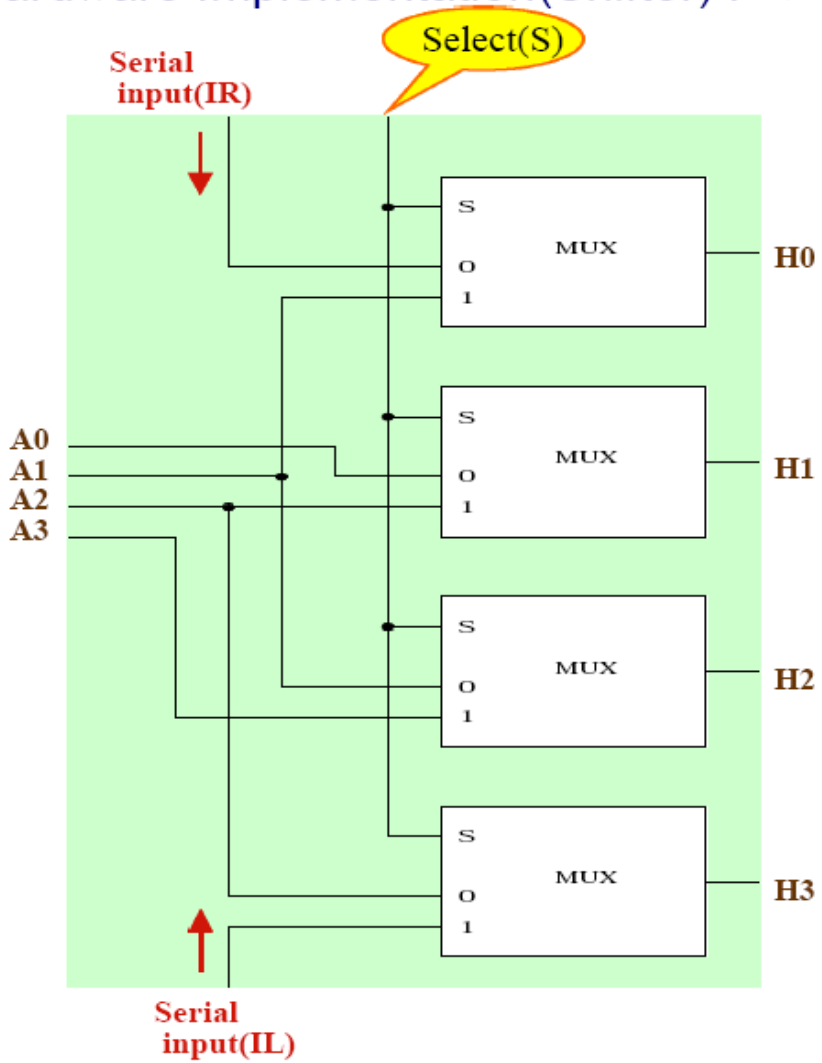


Before the shift, if the leftmost two bits differ, the shift will result in an overflow

- **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**

Hardware Implementation of Shift Microop

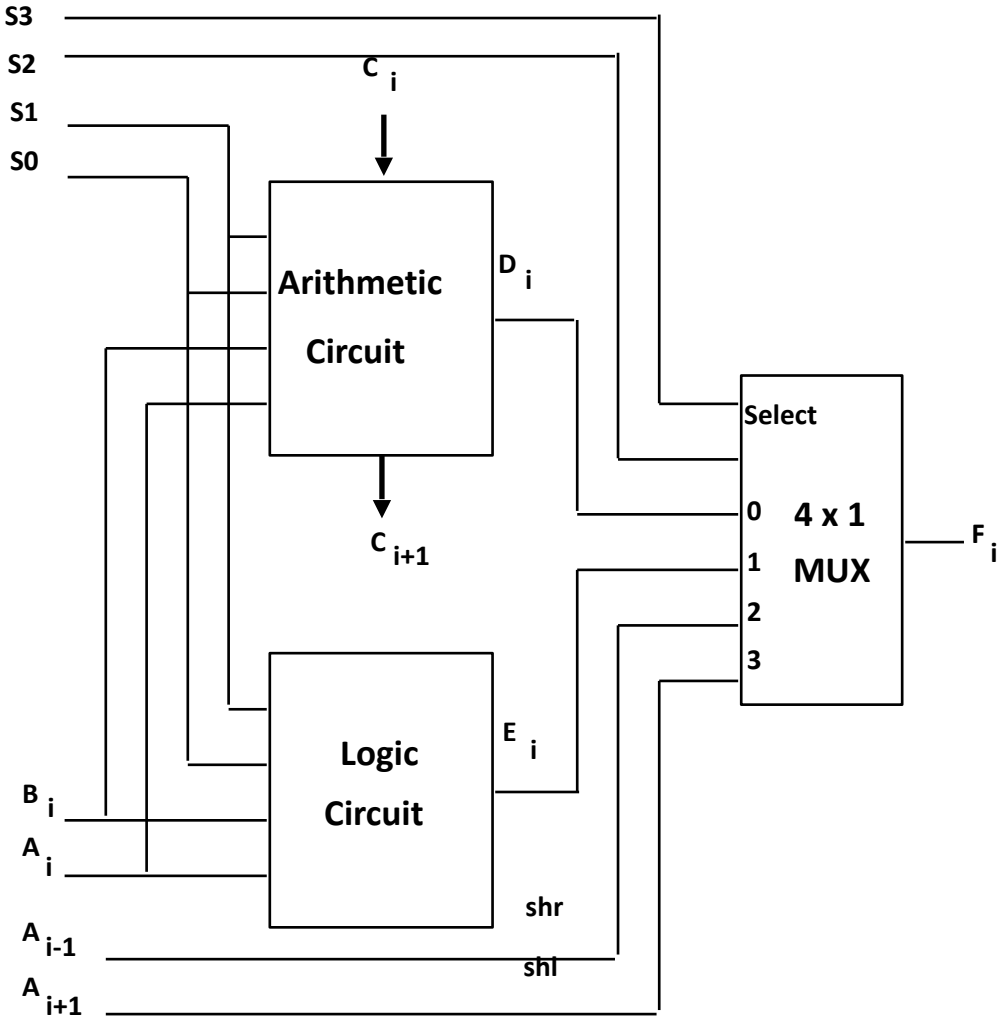
- Hardware Implementation(Shifter) :



Function Table

Select	output			
S	H0	H1	H2	H3
0	IR	A0	A1	A2
1	A1	A2	A3	IL

Arithmetic Logic and Shift Unit



s2	s3	Operation
0	0	Arithmetic
0	1	Logical
1	0	Shr
1	1	shl

TABLE 4-8 Function Table for Arithmetic Logic Shift Unit

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

Overview

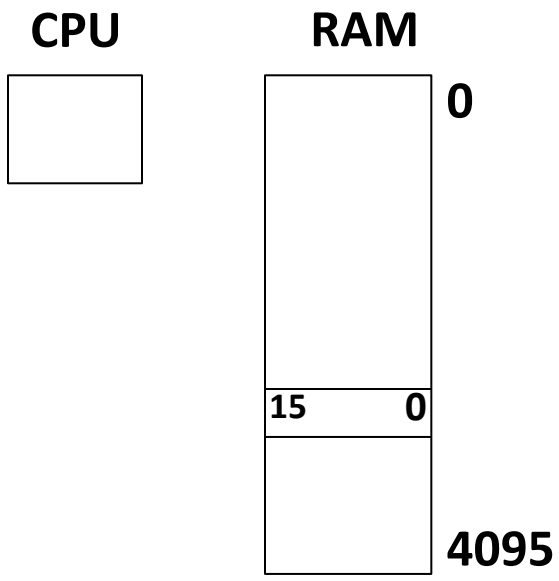
- **Instruction Codes**
- **Computer Registers**
- **Common bus System**
- Computer Instructions
- Timing and Control
- Instruction Cycle
- Memory Reference Instructions
- Input-Output and Interrupt
- Complete Computer Description

Introduction

- **Every different processor type has its own design (different registers, buses, microoperations, machine instructions, etc)**
- **Modern processor is a very complex device**
- **It contains**
  - **Many registers**
  - **Multiple arithmetic units, for both integer and floating point calculations**
  - **The ability to pipeline several consecutive instructions to speed execution**
  - **Etc.**

Basic Computer

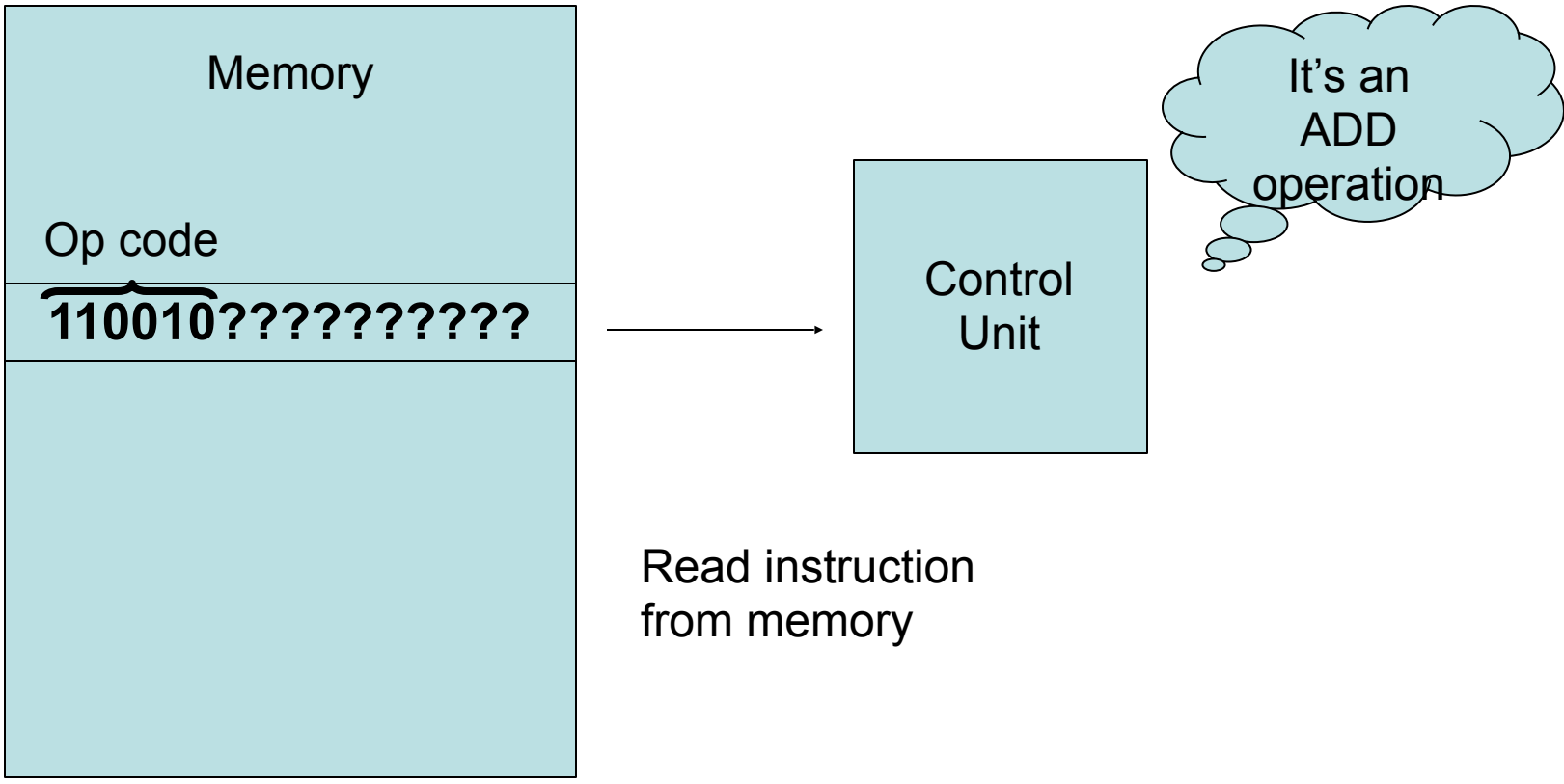
- **The Basic Computer has two components, a processor and memory**
- **The memory has 4096 words in it**
  - **4096 = 2^{12}, so it takes 12 bits(address) to select a word in memory**
- **Each word is 16 bits long**



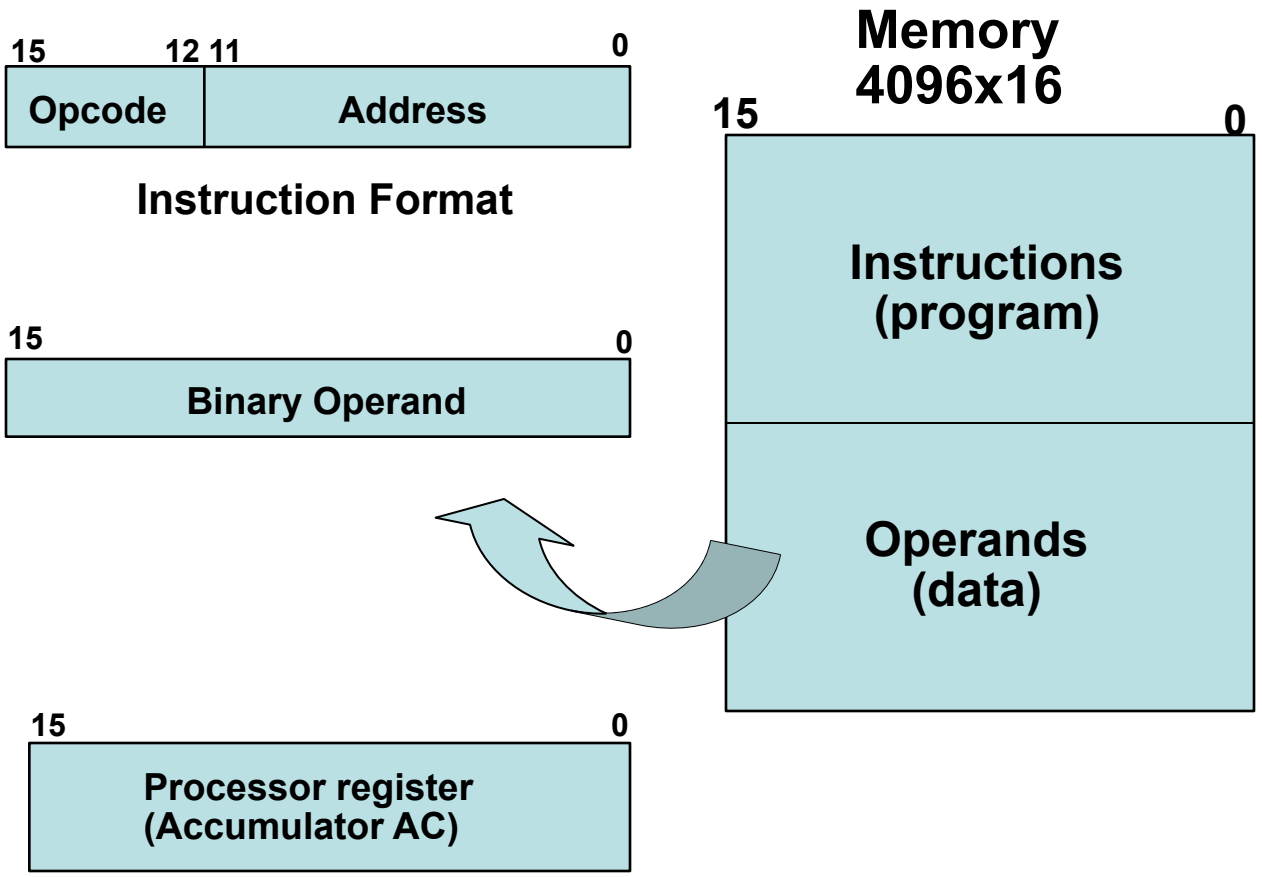
Instruction

- **Program**
  - **A sequence of (machine) instructions**
- **(Machine) Instruction**
  - **A group of bits that tell the computer to *perform a specific operation* (a sequence of micro-operation)**
- **The instructions of a program, along with any needed data are stored in memory**
- **The CPU reads the next instruction from memory**
- **It is placed in an Instruction Register (IR)**
- **Control circuitry in control unit then translates the instruction into the sequence of micro-operations necessary to implement it.**

Instruction Codes



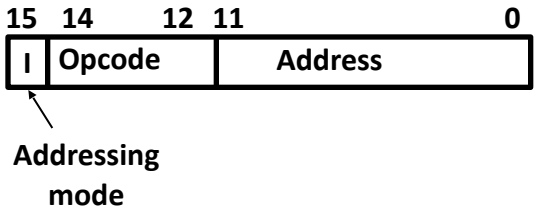
Instruction Codes
Stored Program Organization



Instruction Format

- A computer instruction is often divided into two parts
  - An opcode (Operation Code) that specifies the operation for that instruction
  - An address that specifies the registers and/or locations in memory to use for that operation
- In the Basic Computer, since the memory contains 4096 (= 2^{12}) words, we needs 12 bit to specify which memory address this instruction will use
- In the Basic Computer, bit 15th of the instruction specifies the addressing mode (0: direct addressing, 1: indirect addressing)
- Since the memory words, and hence the instructions, are 16 bits long, that leaves 3 bits for the instruction's opcode

Instruction Format

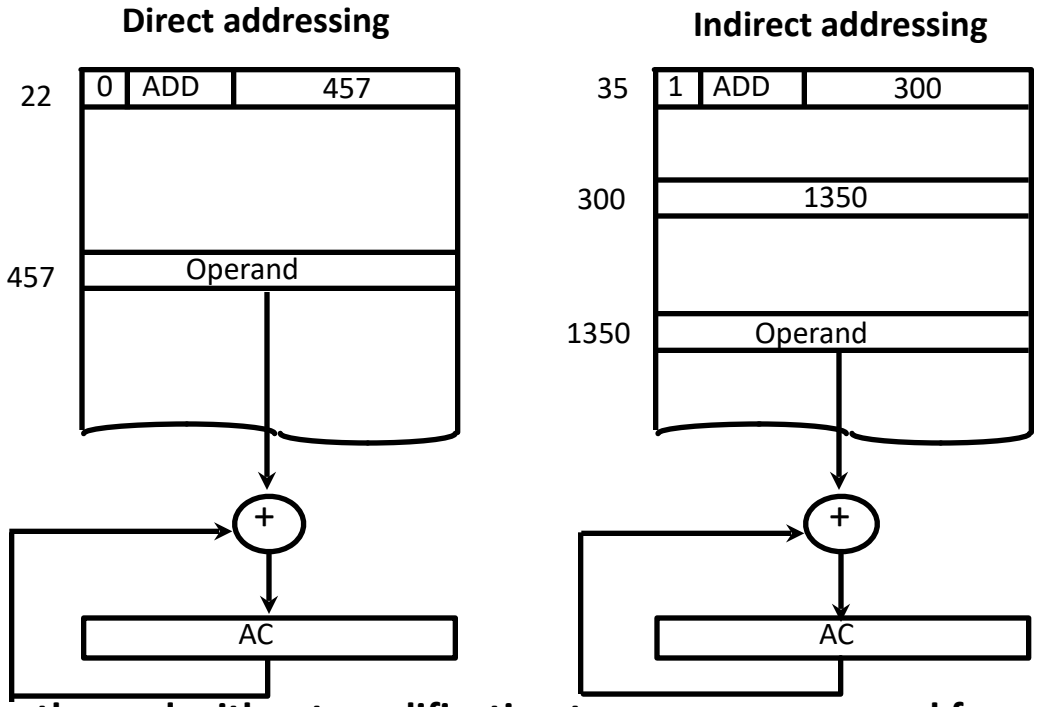


- **Q—-In the Basic Computer, if the memory contains 1024 words, How many bits specify memory address ?**

- **A-12**
- **B-10**
- **C-16**
- **D-24**

Addressing Mode

- **The address field of an instruction can represent either**
  - **Direct address: the address in memory of the data to use (the address of the operand), or**
  - **Indirect address: the address in memory of the address in memory of the data to use**



- **Effective Address (EA)**
  - **The address, that can be directly used without modification to access an operand for a computation-type instruction, or as the target address for a branch-type instruction**

Q—-The addressing mode, where you directly specify the operand value is

_____

a) Immediate
b) Direct
c) Definite
d) Relative

Processor Register

- **A processor has many registers to hold instructions, addresses, data, etc**
- **The processor has a register, the Program Counter (PC) that holds the memory address of the next instruction to get**

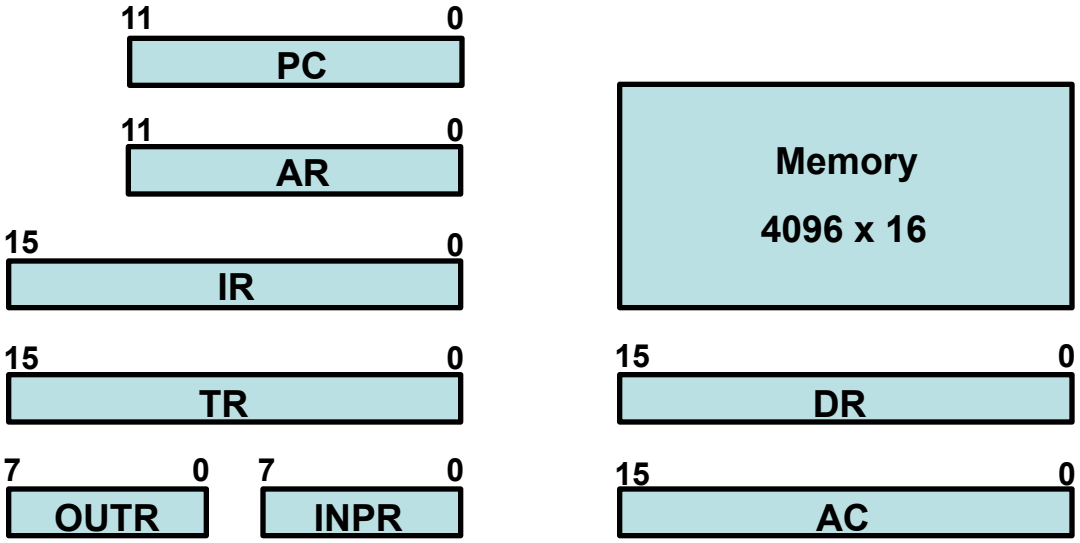
  Since the memory in the Basic Computer only has 4096 locations, the PC only needs 12 bits
- **In a direct or indirect addressing, the processor needs to keep track of what locations in memory it is addressing: The Address Register (AR) is used for this**

  The AR is a 12 bit register in the Basic Computer
- **When an operand is found, using either direct or indirect addressing, it is placed in the Data Register (DR). The processor then uses this value as data for its operation**
- **The Basic Computer has a single general purpose register – the Accumulator (AC)**

Processor Register

- **The significance of a general purpose register is that it can be referred to in instructions**
  - **e.g. load AC with the contents of a specific memory location; store the contents of AC into a specified memory location**
- **Often a processor will need a scratch register to store intermediate results or other temporary data; in the Basic Computer this is the Temporary Register (TR)**
- **The Basic Computer uses a very simple model of input/output (I/O) operations**
  - **Input devices are considered to send 8 bits of character data to the processor**
  - **The processor can send 8 bits of character data to output devices**
- **The Input Register (INPR) holds an 8 bit character gotten from an input device**
- **The Output Register (OUTR) holds an 8 bit character to be send to an output device**

Registers in the Basic Computer

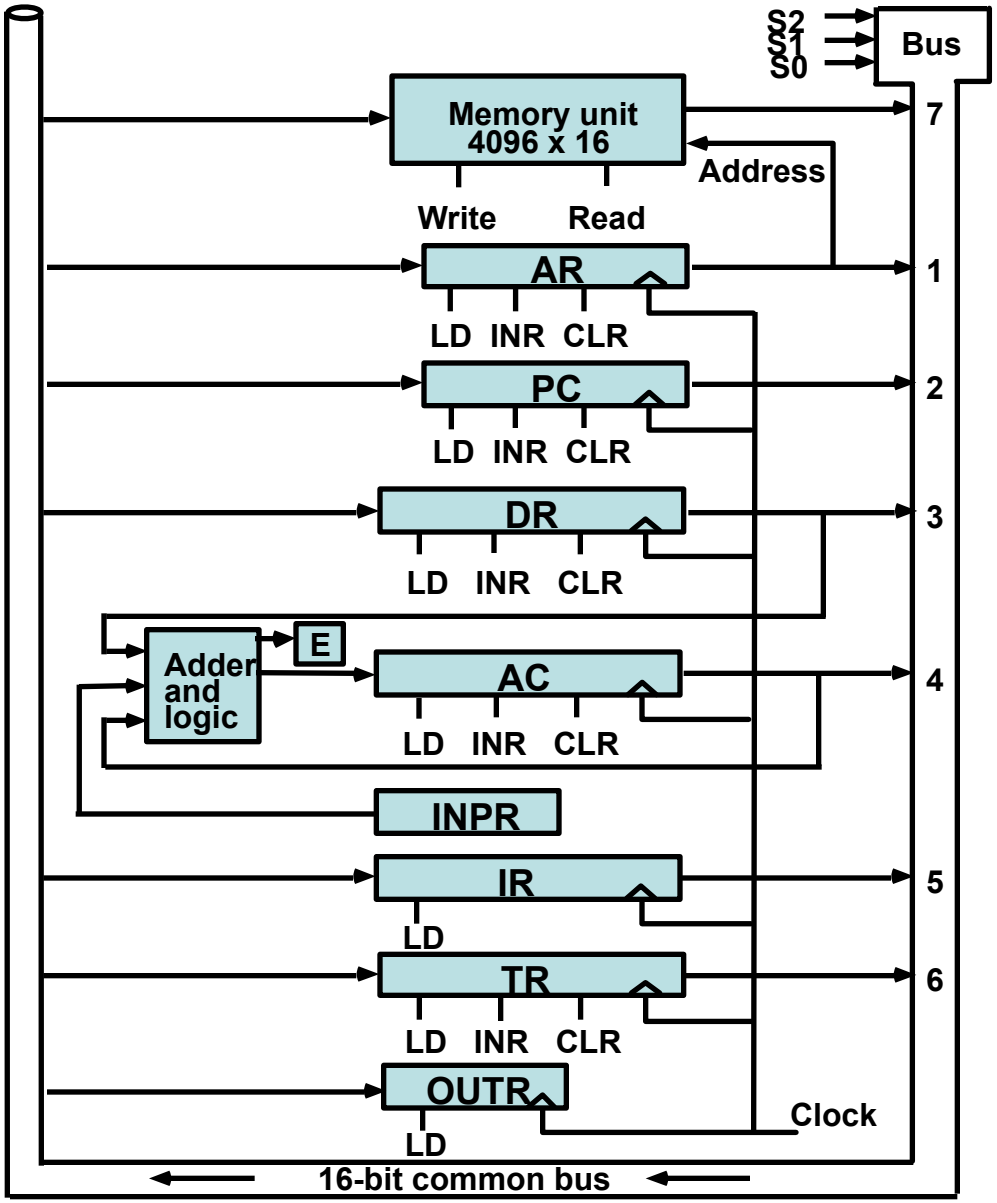


List of BC Registers

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

Common Bus System

- **The registers in the Basic Computer are connected using a bus**
- **This gives a savings in circuitry over complete connections between registers**



Computer Registers
Common Bus System

Common Bus System

- Three control lines, S_2, S_1, and S_0 control which register the bus selects as its input

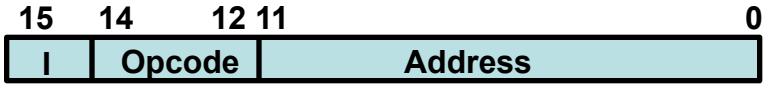
S_2 S_1 S_0	Register
0 0 0	x
0 0 1	AR
0 1 0	PC
0 1 1	DR
1 0 0	AC
1 0 1	IR
1 1 0	TR
1 1 1	Memory

- Either one of the registers will have its load signal activated, or the memory will have its read signal activated
  - Will determine where the data from the bus gets loaded
- The 12-bit registers, AR and PC, have 0's loaded onto the bus in the high order 4 bit positions
- When the 8-bit register OUTR is loaded from the bus, the data comes from the low order 8 bits on the bus

5-3 Computer Instructions

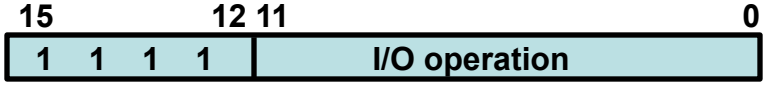
Basic Computer Instruction code format

Memory-Reference Instructions (OP-code = 000 ~ 110)



Register-Reference Instructions (OP-code = 111, I = 0)

Input-Output Instructions (OP-code =111, I = 1)



Instruction Set Completeness

A computer should have a set of instructions so that the user can construct machine language programs to evaluate any function that is known to be computable.

Instruction Types

- **Functional Instructions**
  - **Arithmetic, logic, and shift instructions**
  - **ADD, CMA, INC, CIR, CIL, AND, CLA**
- **Transfer Instructions**
  - **Data transfers between the main memory and the processor registers**
  - **LDA, STA**
- **Control Instructions**
  - **Program sequencing and control**
  - **BUN, BSA, ISZ**
- **Input/output Instructions**
  - **Input and output**
  - **INP, OUT**

BASIC COMPUTER INSTRUCTIONS

Symbol	Hex Code I = 0	Hex Code I = 1	Description
AND	0xxx	8xxx	AND memory word to AC
ADD	1xxx	9xxx	Add memory word to AC
LDA	2xxx	Axxx	Load AC from memory
STA	3xxx	Bxxx	Store content of AC into 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 instr. if AC is positive
SNA	7008		Skip next instr. if AC is negative
SZA	7004		Skip next instr. if AC is zero
SZE	7002		Skip next instr. if E is zero
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

Control Unit

- **Control unit (CU) of a processor translates from machine instructions to the control signals for the microoperations that implement them**

- **Control units are implemented in one of two ways**

  Hardwired Control

  CU is made up of sequential and combinational circuits to generate the control signals

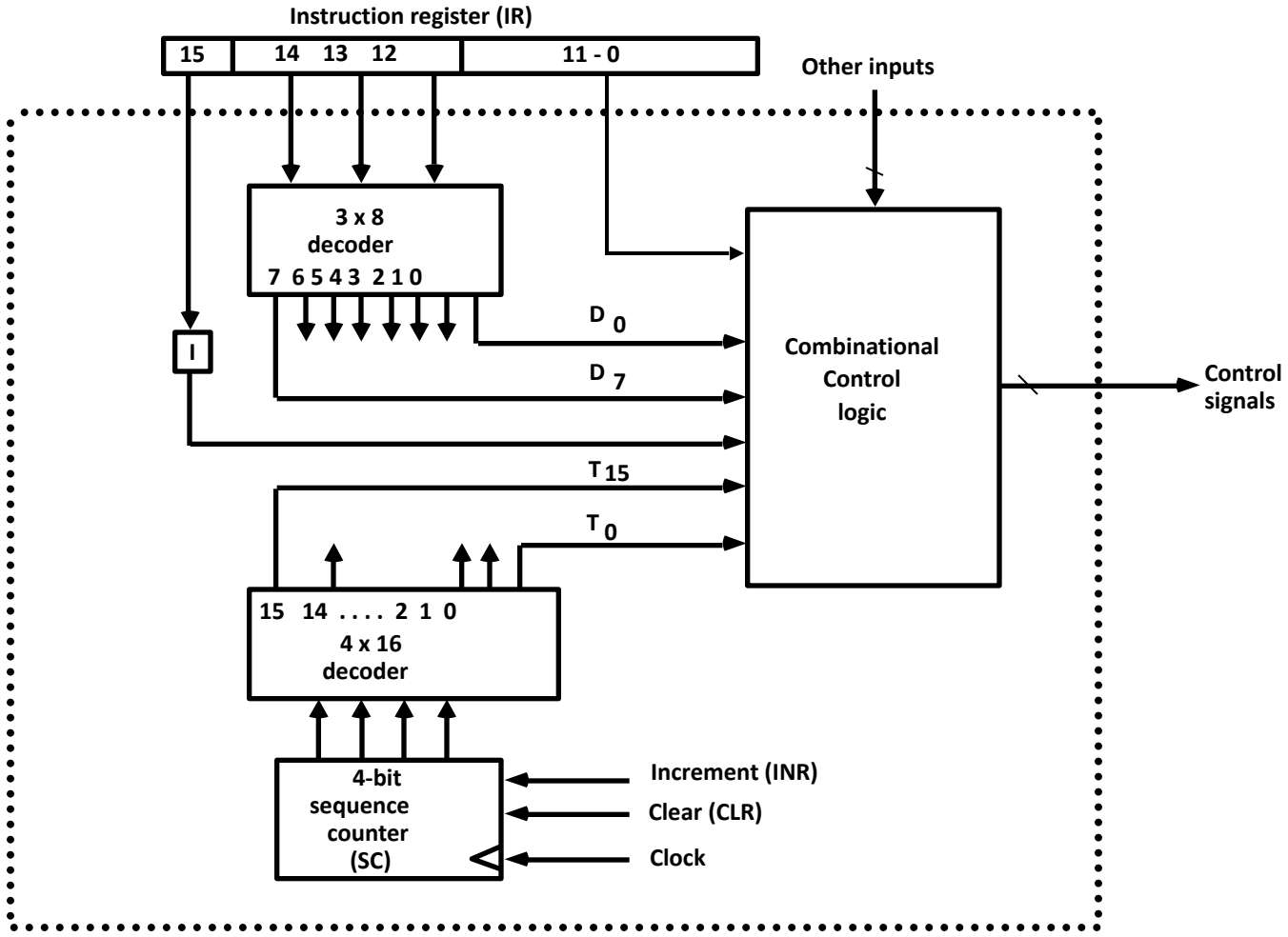
  Microprogrammed Control

  A control memory on the processor contains microprograms that activate the necessary control signals

- **We will consider a hardwired implementation of the control unit for the Basic Computer**

Timing and Control

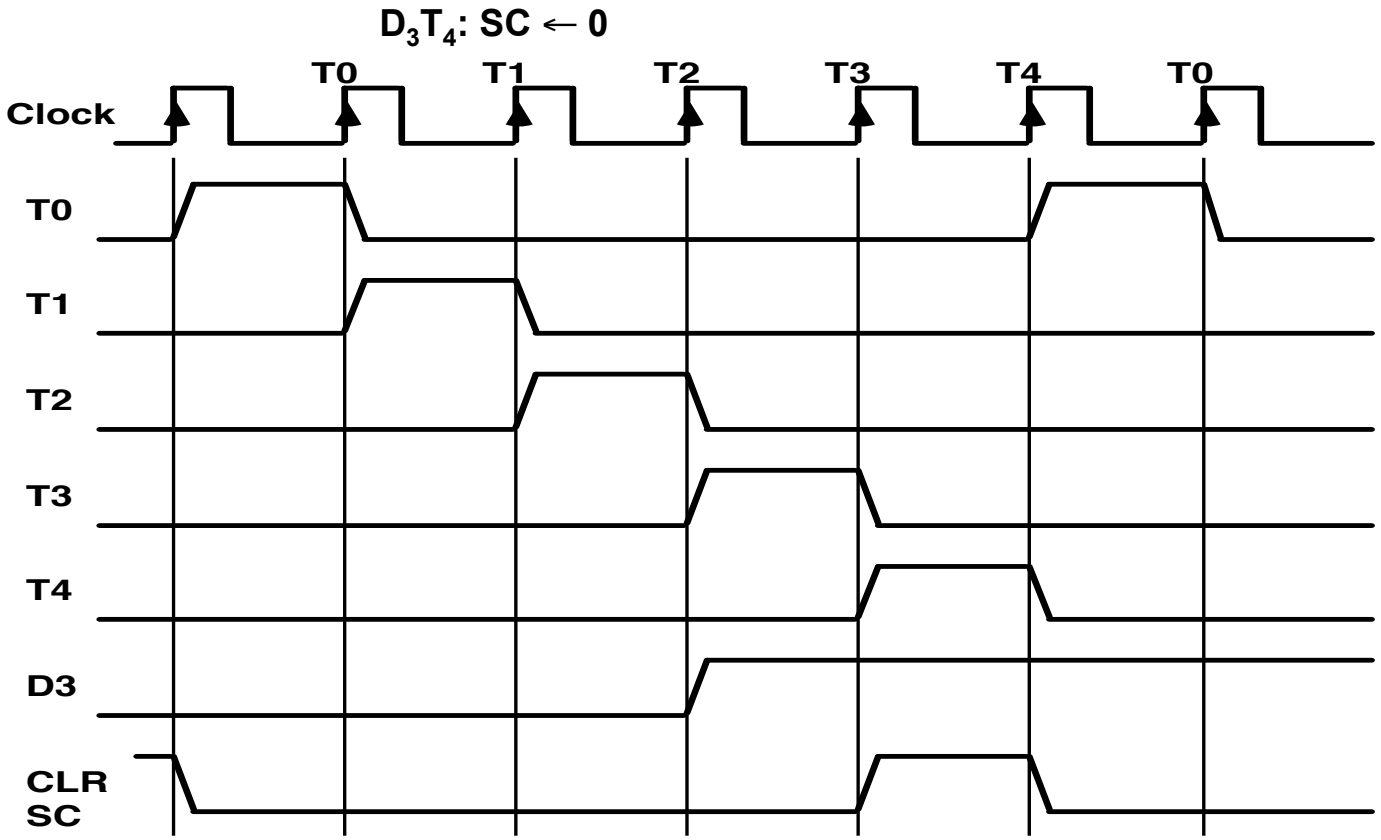
Control unit of Basic Computer



Timing Signals

- Generated by 4-bit sequence counter and 4×16 decoder
- The SC can be incremented or cleared.

- Example: $T_0, T_1, T_2, T_3, T_4, T_0, T_1, \ldots$
  Assume: At time T_4, SC is cleared to 0 if decoder output D3 is active.



Instruction Cycle

- **In Basic Computer, a machine instruction is executed in the following cycle:**
  1. **Fetch an instruction from memory**
  2. **Decode the instruction**
  3. **Read the effective address from memory if the instruction has an indirect address**
  4. **Execute the instruction**

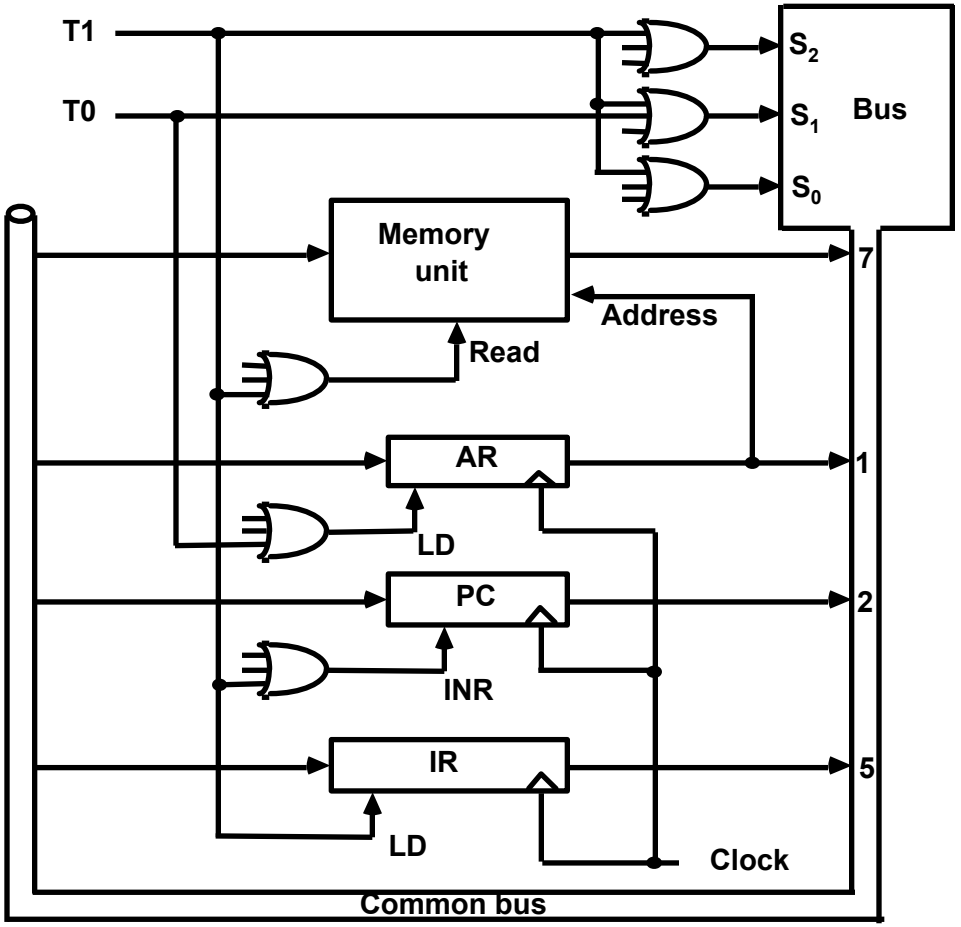
- **After an instruction is executed, the cycle starts again at step 1, for the next instruction**

***Note*: Every different processor has its own (different) instruction cycle**

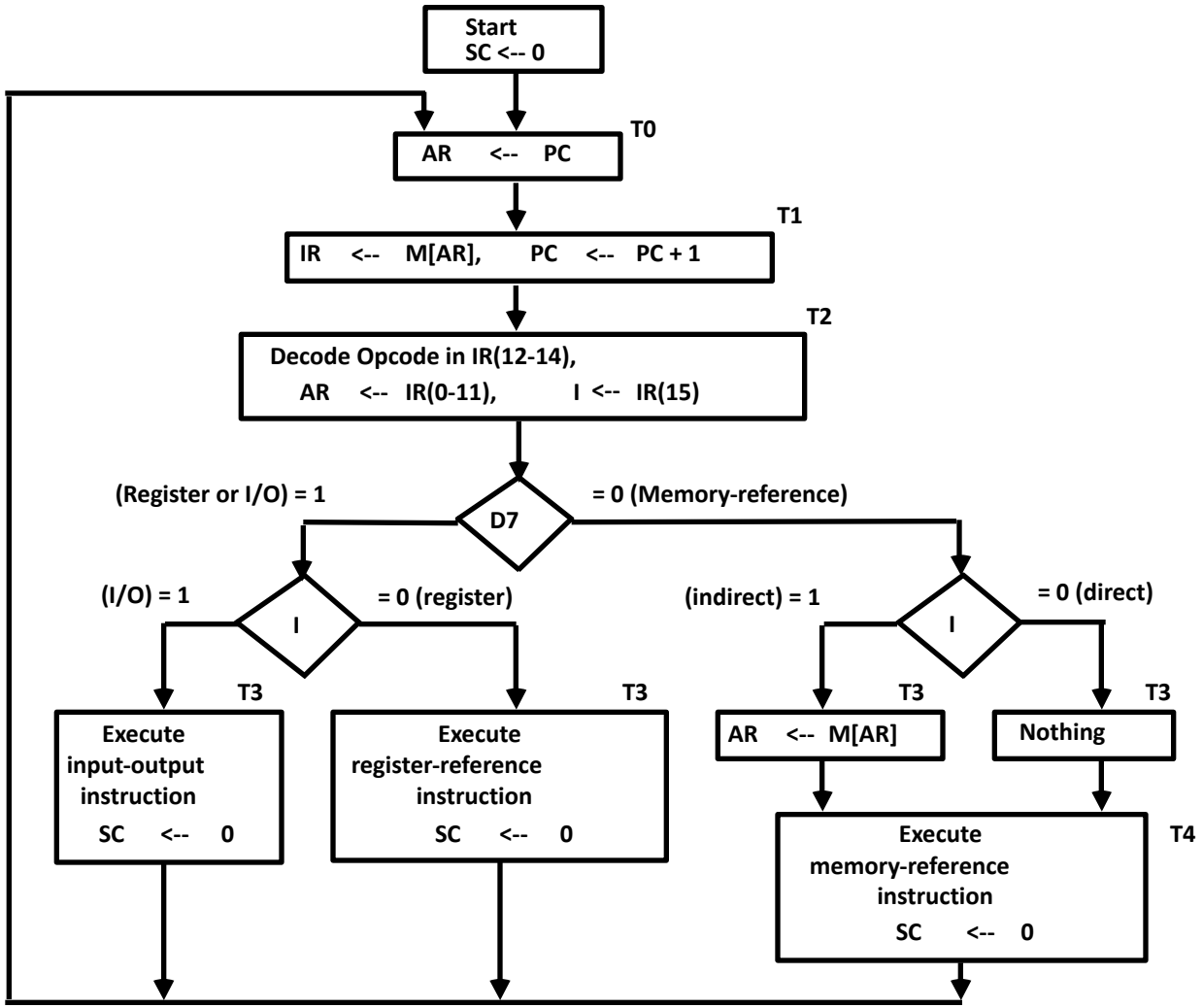
Fetch and Decode

Fetch and Decode

T0: AR ← PC ($S_0S_1S_2$=010, T0=1)
T1: IR ← M [AR], PC ← PC + 1 (S0S1S2=111, T1=1)
T2: D0, . . . , D7 ← Decode IR(12-14), AR ← IR(0-11), I ← IR(15)



Flow Chart (Instruction Cycle)



Determining Type of Instruction

- **D'_7IT_3: AR ← M[AR]**
- **$D'_7I'T_3$: Nothing**
- **$D_7I'T_3$: Execute a register-reference instr.**
- **D_7IT_3: Execute an input-output instr.**

Register Reference Instruction

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)

Memory Reference Instructions

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

- The effective address of the instruction is in AR and was placed there during timing signal T_2 when I = 0, or during timing signal T_3 when I = 1
- Memory cycle is assumed to be short enough to complete in a CPU cycle
- The execution of MR instruction starts with T_4

AND to AC

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

ADD to AC

D_1T_4:	$DR \leftarrow M[AR]$	Read operand
D_1T_5:	$AC \leftarrow AC + DR, E \leftarrow C_{out}, SC \leftarrow 0$	Add to AC and store carry in E

Memory Reference Instructions

LDA: Load to AC

D_2T_4: DR ← M[AR]

D_2T_5: AC ← DR, SC ← 0

STA: Store AC

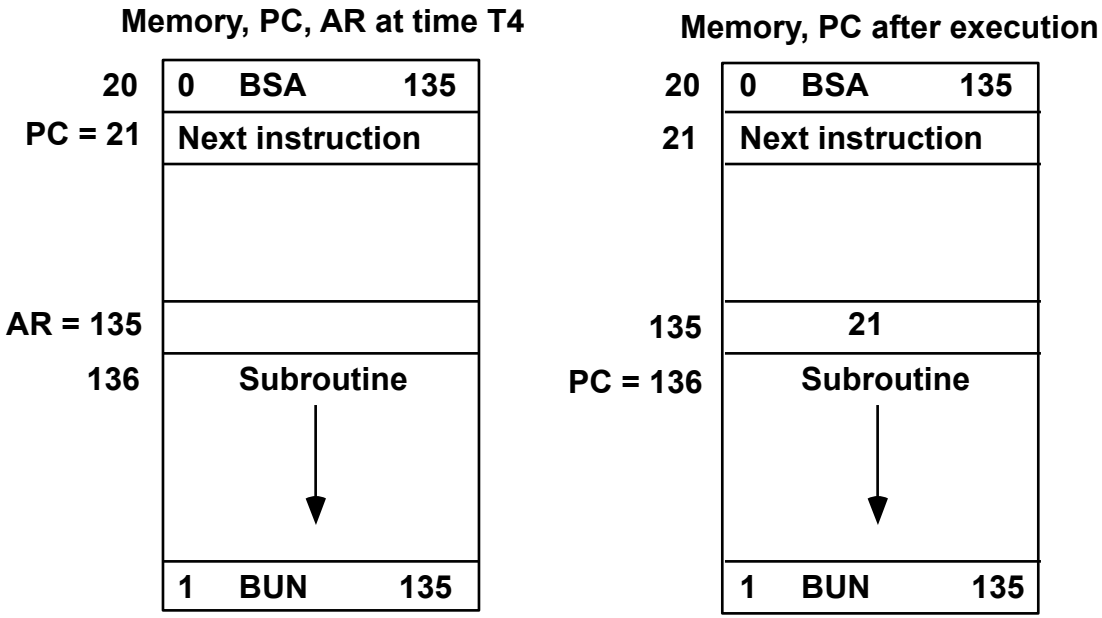
D_3T_4: M[AR] ← AC, SC ← 0

BUN: Branch Unconditionally

D_4T_4: PC ← AR, SC ← 0

BSA: Branch and Save Return Address

M[AR] ← PC, PC ← AR + 1



Memory Reference Instructions

BSA:

D_5T_4: M[AR] ← PC, AR ← AR + 1

D_5T_5: PC ← AR, SC ← 0

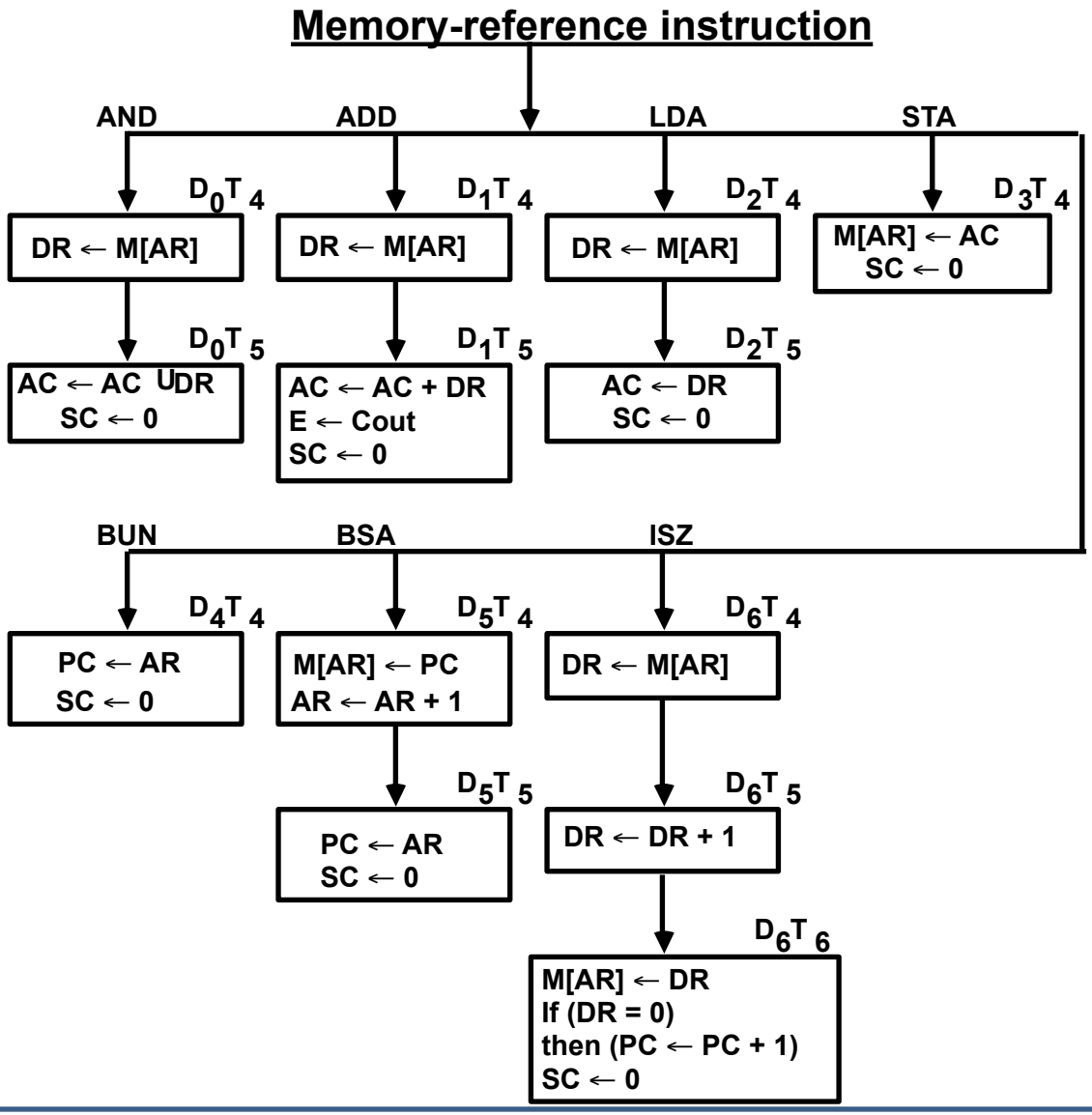
ISZ: Increment and Skip-if-Zero

D_6T_4: DR ← M[AR]

D_6T_5: DR ← DR + 1

D_6T_4: M[AR] ← DR, if (DR = 0) then (PC ← PC + 1), SC ← 0

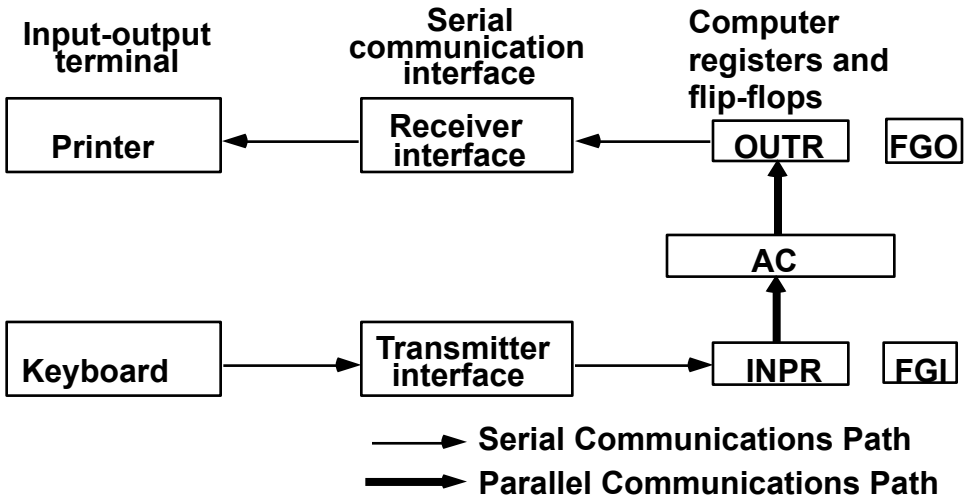
Flow Chart - Memory Reference Instructions



Input/Output and Interrupt

A Terminal with a keyboard and a Printer

Input-Output Configuration



INPR	Input register - 8 bits
OUTR	Output register - 8 bits
FGI	Input flag - 1 bit
FGO	Output flag - 1 bit
IEN	Interrupt enable - 1 bit

- The terminal sends and receives serial information
- The serial info. from the keyboard is shifted into INPR
- The serial info. for the printer is stored in the OUTR
- INPR and OUTR communicate with the terminal serially and with the AC in parallel.
- The flags are needed to synchronize the timing difference between I/O device and the computer

Input/Output Instructions

$D_7IT_3 = p$
$IR(i) = B_i$, i = 6, …, 11

	p:	SC ← 0	Clear SC
INP	pB_{11}:	AC(0-7) ← INPR, FGI ← 0	Input char. to AC
OUT	pB_{10}:	OUTR ← AC(0-7), FGO ← 0	Output char. from AC
SKI	pB_9:	if(FGI = 1) then (PC ← PC + 1)	Skip on input flag
SKO	pB_8:	if(FGO = 1) then (PC ← PC + 1)	Skip on output flag
ION	pB_7:	IEN ← 1	Interrupt enable on
IOF	pB_6:	IEN ← 0	Interrupt enable off

Interrupt Initiated Input/Output

- Open communication only when some data has to be passed --> *interrupt*.

- The I/O interface, instead of the CPU, monitors the I/O device.

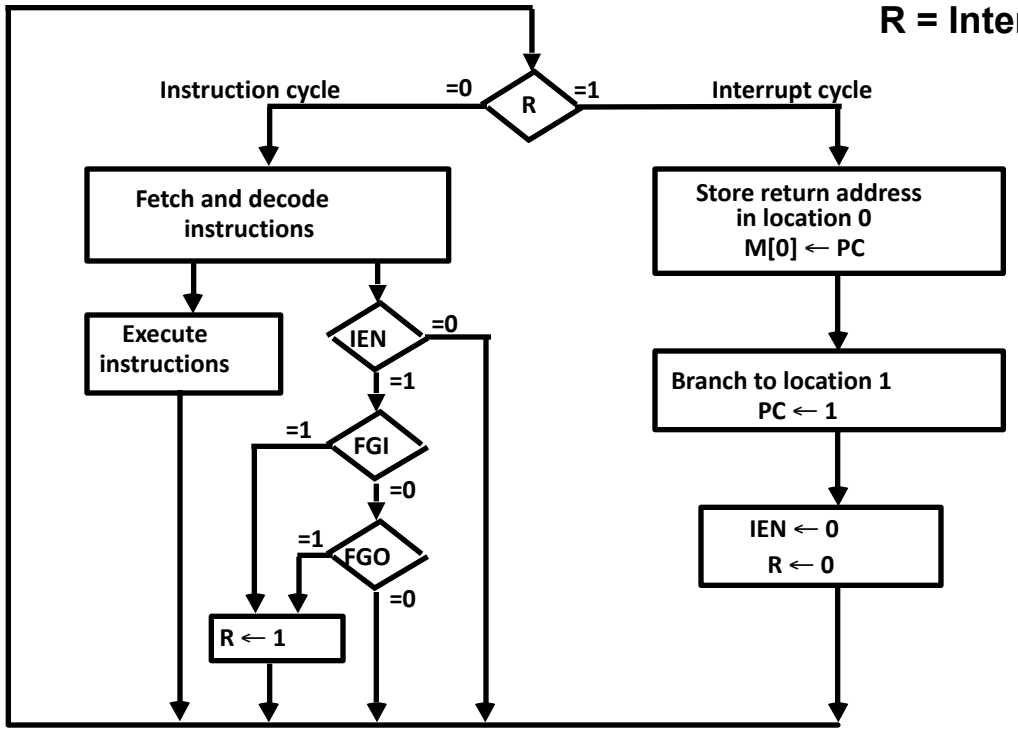
- When the interface founds that the I/O device is ready for data transfer, it generates an interrupt request to the CPU

- Upon detecting an interrupt, the CPU stops momentarily the task it is doing, branches to the service routine to process the data transfer, and then returns to the task it was performing.

IEN (Interrupt-enable flip-flop)

- can be set and cleared by instructions
- when cleared, the computer cannot be interrupted

Flow Chart of Interrupt Cycle



INPR	Input register - 8 bits
OUTR	Output register - 8 bits
FGI	Input flag - 1 bit
FGO	Output flag - 1 bit
IEN	Interrupt enable - 1 bit

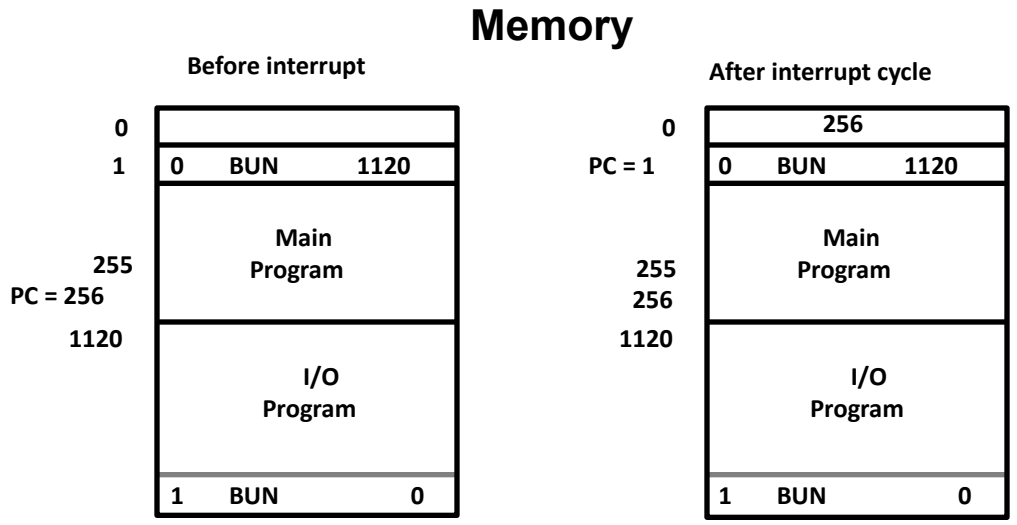
- The interrupt cycle is a HW implementation of a branch and save return address operation.

- At the beginning of the next instruction cycle, the instruction that is read from memory is in address 1.

- At memory address 1, the programmer must store a branch instruction that sends the control to an interrupt service routine

- The instruction that returns the control to the original program is "indirect BUN 0"

Register Transfer Operations in Interrupt Cycle



Register Transfer Statements for Interrupt Cycle

- R F/F ← 1 if IEN (FGI + FGO)$T_0'T_1'T_2'$

  ⇔ $T_0'T_1'T_2'$ (IEN)(FGI + FGO): R ← 1

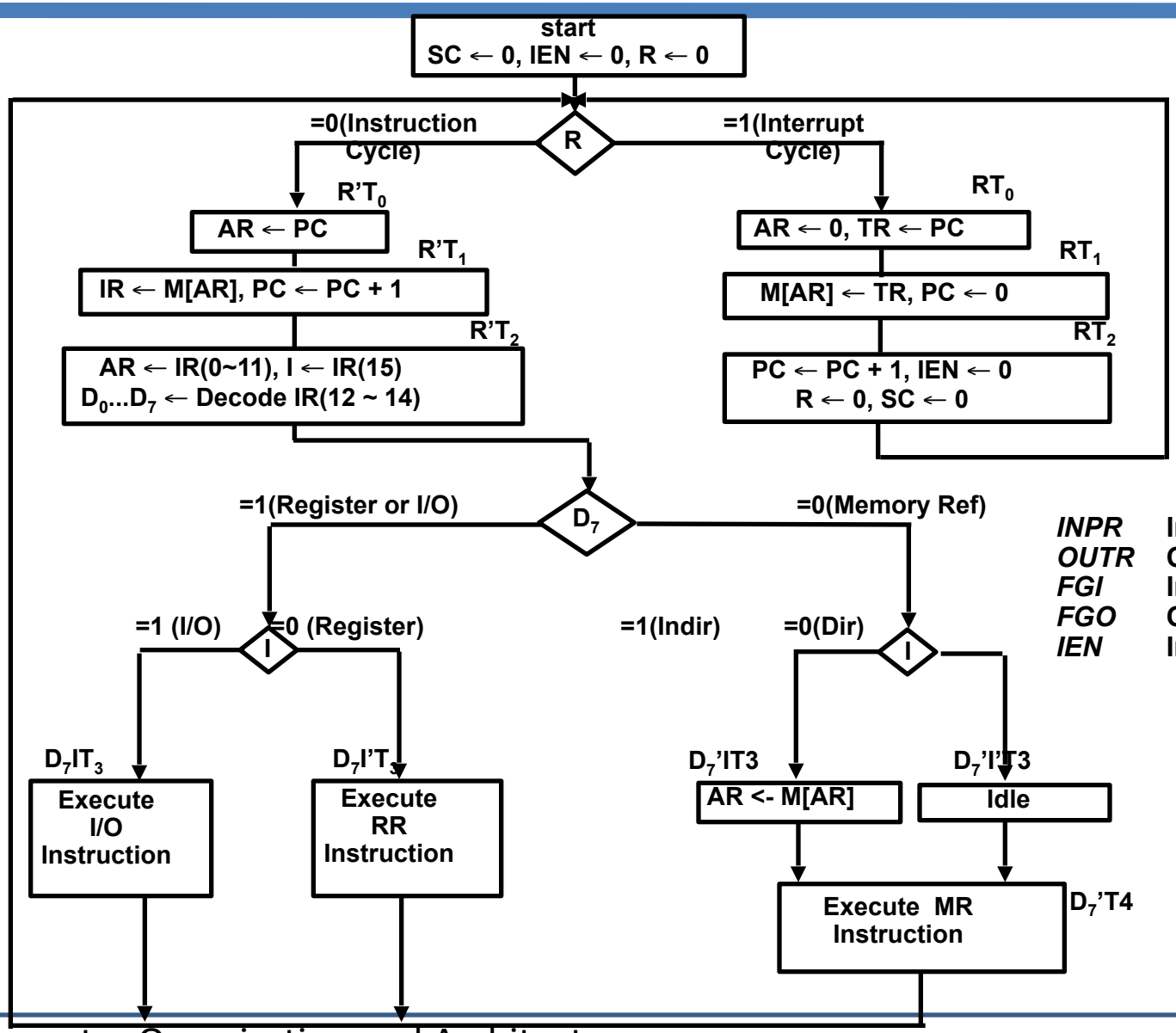
- The fetch and decode phases of the instruction cycle must be modified → Replace T_0, T_1, T_2 with $R'T_0$, $R'T_1$, $R'T_2$
- The interrupt cycle :

  RT_0: AR ← 0, TR ← PC

  RT_1: M[AR] ← TR, PC ← 0

  RT_2: PC ← PC + 1, IEN ← 0, R ← 0, SC ← 0

Complete Computer Design



Complete Computer Design

Fetch	$R'T_0$:	AR ← PC
	$R'T_1$:	IR ← M[AR], PC ← PC + 1
Decode	$R'T_2$:	D0, ..., D7 ← Decode IR(12 ~ 14), AR ← IR(0 ~ 11), I ← IR(15)
Indirect	$D_7'IT_3$:	AR ← M[AR]
Interrupt		R ← 1
	RT_0:	AR ← 0, TR ← PC
	RT_1:	M[AR] ← TR, PC ← 0
	RT_2:	PC ← PC + 1, IEN ← 0, R ← 0, SC ← 0
Memory-Reference		
AND	D_0T_4:	DR ← M[AR]
	D_0T_5:	AC ← AC ∧ DR, SC ← 0
ADD	D_1T_4:	DR ← M[AR]
	D_1T_5:	AC ← AC + DR, E ← C_{out}, SC ← 0
LDA	D_2T_4:	DR ← M[AR]
	D_2T_5:	AC ← DR, SC ← 0
STA	D_3T_4:	M[AR] ← AC, SC ← 0
BUN	D_4T_4:	PC ← AR, SC ← 0
BSA	D_5T_4:	M[AR] ← PC, AR ← AR + 1
	D_5T_5:	PC ← AR, SC ← 0
ISZ	D_6T_4:	DR ← M[AR]
	D_6T_5:	DR ← DR + 1
	D_6T_6:	M[AR] ← DR, if(DR=0) then (PC ← PC + 1), SC ← 0

Complete Computer Design

Register-Reference		
	$D_7I'T_3 = r$	(Common to all register-reference instr)
	$IR(i) = B_i$	(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
Input-Output		
		(Common to all input-output instructions)
	$D_7IT_3 = p$	(i = 6,7,8,9,10,11)
INP	$IR(i) = B_i$	SC ← 0
OUT	p:	AC(0-7) ← INPR, FGI ← 0
SKI	pB_{11}:	OUTR ← AC(0-7), FGO ← 0
SKO	pB_{10}:	If(FGI=1) then (PC ← PC + 1)
ION	pB_9:	If(FGO=1) then (PC ← PC + 1)
IOF	pB_8:	IEN ← 1
	pB_7:	IEN ← 0
	pB_6:	

COMPUTER ORGANIZATION

- The organization of a computer is defined by its internal registers, the timing and control structure, and the set of instructions that it uses.
- The internal organization of a digital system is defined by the sequence of micro-operations it performs on data stored in its registers.
- The general purpose digital computer is capable of executing various micro-operations and it can be instructed as to what specific instructions it must perform.

- The user of a computer can control the process by means of a program.
- A program is a set of instructions that specify the operations, operands, and the sequence by which processing has to occur.
- An instruction is a binary code that specifies a sequence of micro-operations.
- Instructions and data are stored in memory.
- The ability of store and execute instructions, the stored program concept (von Neumann architecture), is the most important property of a general-purpose computer.

- An instruction code is a group of bits that instruct the computer to perform a specific operation (set of micro-operations).
- Operation code is a part of instruction code; a group of bits that define such operations as add, subtract, multiply, shift, and complement.
- The operation code must consist of at least n bits for a given 2^n (or less) distinct operations.
- Control unit receives the instruction from memory and interprets the operation code bits. It the issues a sequence of control signals to initiate MOs in internal registers.

- For every operation code, the control issues a sequence of MOs.
- An operation code is called macro-operation because it specifies a set of micro-operations.
- An instruction code specify also the registers or the memory words for operands and results
  - Memory words can be specified by their address
  - Registers can be specified by a binary code of k bits specifying one of 2^k possible registers.
- Each computer (CPU) has its own instruction code format.

- A simple computer organization
  - One register
  - An instruction code format with two parts
    - Operation code
    - An address: tells the control where to find an operand from memory
- Fig. 5-1. Control reads 16-bit instruction from program memory. It uses the 12-bit address part of instruction to read 16-bit operand from data memory. It then executes the operation specified by the operation code.
- If an operation does not need an operand from memory, the address bits can be used for other purposes, e.g. for specifying other operations or an operand.

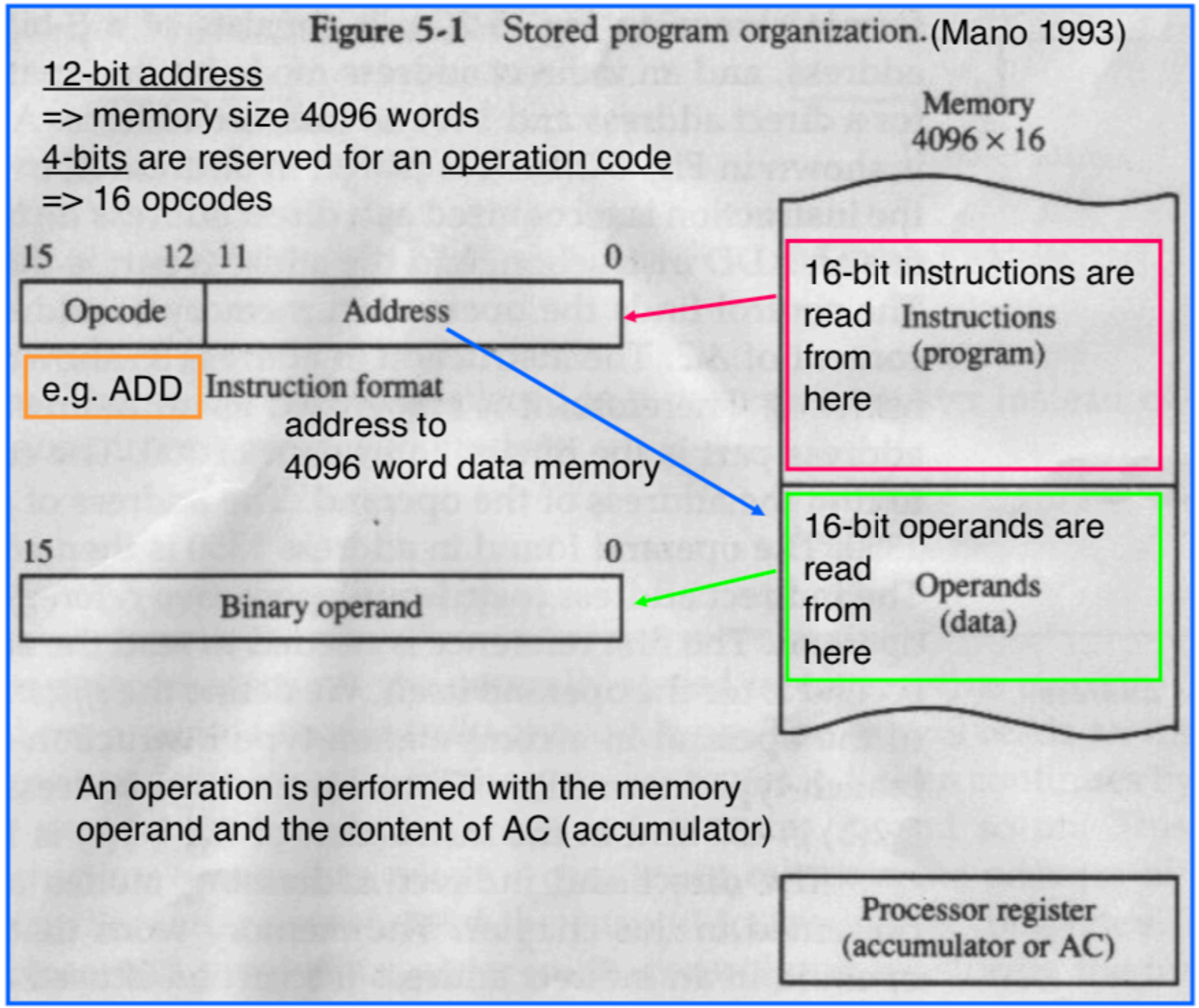
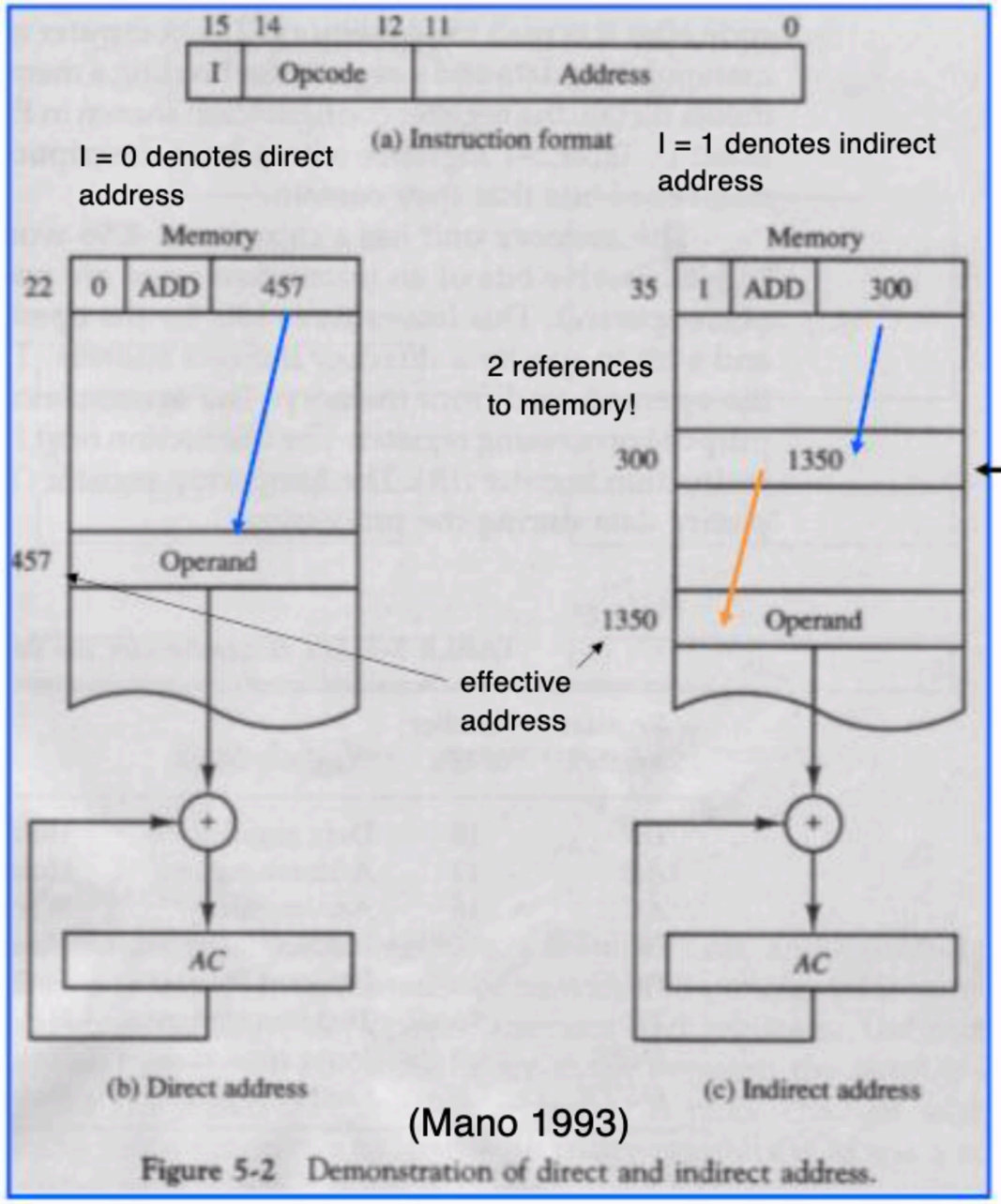


Figure 5-1 Stored program organization. (Mano 1993)
12-bit address
=> memory size 4096 words
4-bits are reserved for an operation code
=> 16 opcodes
Memory
4096 × 16
15
12
11
0
Opcode
Address
e.g. ADD
Instruction format
address to
4096 word data memory
16-bit instructions are read from here
Instructions (program)
15
0
Binary operand
16-bit operands are read from here
Operands (data)
An operation is performed with the memory operand and the content of AC (accumulator)
Processor register (accumulator or AC)

- When the second part of an instruction code specifies and operand, the instruction is said to have an immediate operand.
- When the second part specifies an address of an operand, the instruction is said to have a direct address.
- Indirect address: the second part specifies a memory location where the address of the operand is found.
- Indirect address increases addressable memory size => more bits for specifying addresses of operands.



(Mano 1993)

Figure 5-2 Demonstration of direct and indirect address.

Computer Registers

- Instructions are stored in consecutive memory locations and are executed sequentially one at a time.
- The control read an instruction from a specific address in memory and executes it: after that next instruction is read an executed, and so on.
- Register are needed for storing fetched instructions, and counters for computing the address of the next instruction.

- Computer needs processor registers for data manipulation and holding addresses (see. Fig. 5-3 and Table 5-1).
- Program counter (PC) goes through a counting sequence and causes the computer to read sequential instructions from memory.
- Instructions are read and executes in sequence unless a branch instruction is encountered
  - Calls for a transfer to a nonconsecutive instruction in the program
  - The address part of a branch instruction becomes the address of the next instruction in PC
  - Next instruction is read from the location indicated by PC

TABLE 5-1 List of Registers for the Basic Computer (Mano 1993)

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 (from an input device)
OUTR	8	Output register	Holds output character (for an output device)

(Mano 1993)

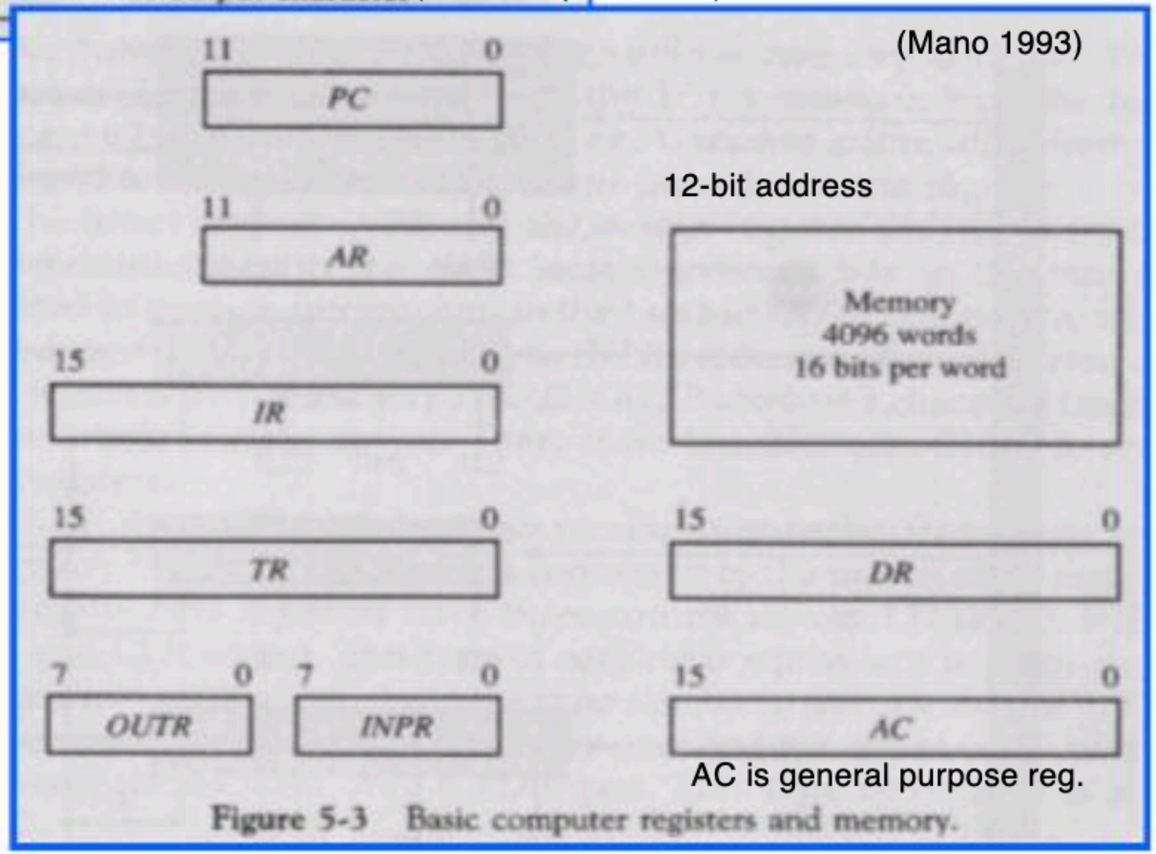


Figure 5-3 Basic computer registers and memory.

- The basic computer (introduced in this course) has (see Fig. 5-3):
  - 8 registers
  - 1 memory unit
  - 1 control unit
  - Common bus
- The outputs of 7 registers and memory are connected to the common bus.
- Connections to bus lines are specified by selection lines S0, S1, and S3.
- A register load during the next clock pulse transition is selected with a LD (load) input.
- Memory write/read is enabled with write/read signals.

(Mano 1993)

16-bit: DR, AC, IR, TR

12-bit: AR, PC

LD = load
INR = increment
CLR = clear

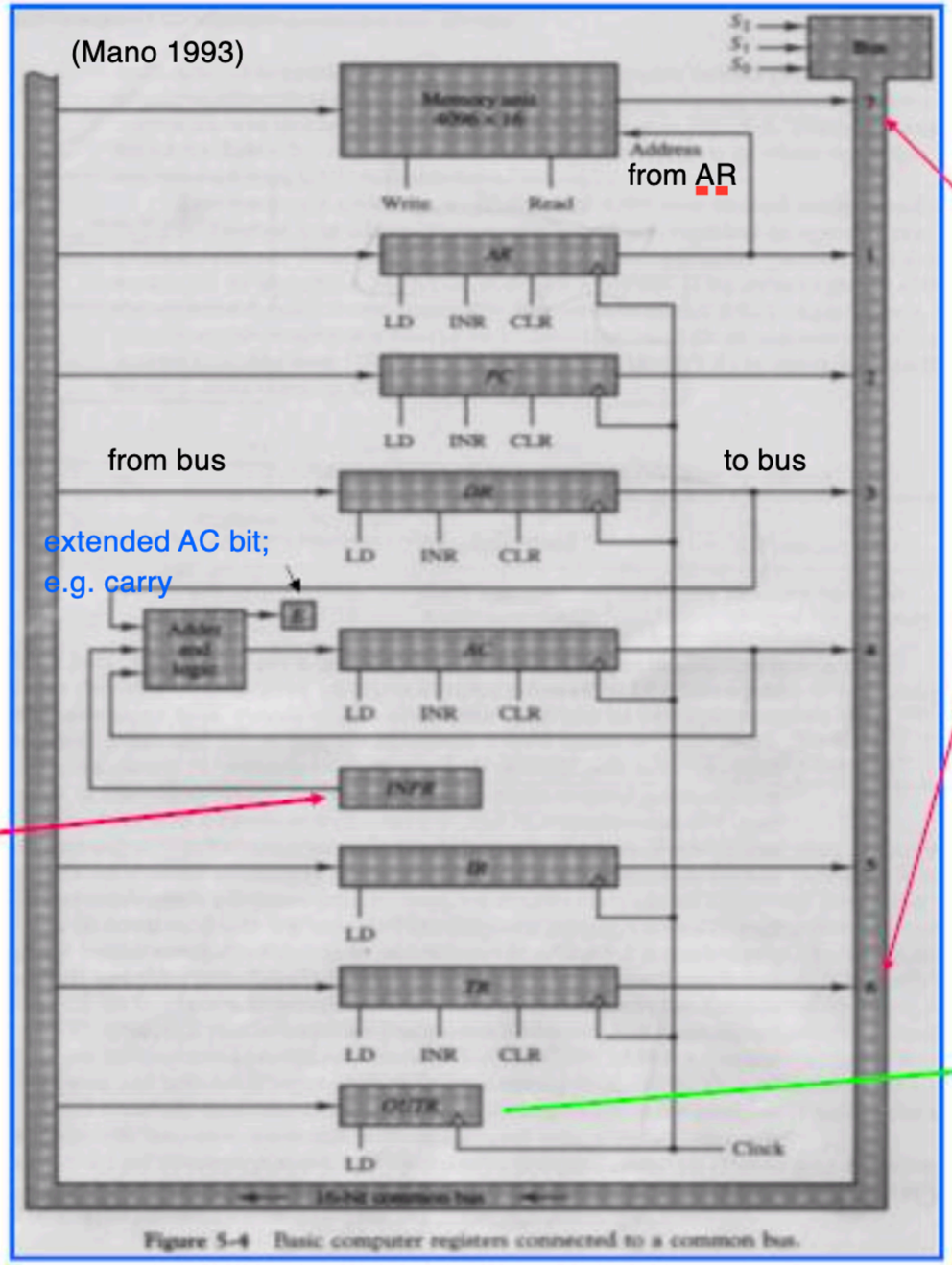


Figure 5-4 Basic computer registers connected to a common bus.

Address	Unit
111	MEM
001	AR
010	PC
011	DR
100	AC
101	IR
110	TR

- INPR receives a character from an input device.
- OUTR receives a character from AC and delivers it to an output device.
- Bus receives data from 6 registers and the memory unit.
- 5 registers have three control lines: LD (load), INR (increment), and CLR (clear): equivalent to a binary counter with parallel load and synchronous clear. 2 registers have only a LD input.
- AR is used to specify memory address: no need for an address bus.
- 16 inputs to AC come from an adder and logic circuit with three sets of inputs: AC output, DR, INPR.

- 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 register and the output of the adder and logic circuit into AC, e.g.:

$$DR \leftarrow AC \text{ and } AC \leftarrow DR$$

AC to the bus (S2S1S0 = 100), enabling the LD of DR, transferring DR into AC (through adder and logic unit), and enabling LD 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.

LOVELY PROFESSIONAL UNIVERSITY

CSE211

Computer Organization and Design

Overview

- **Introduction**
- **General Register Organization**
- Stack Organization
- Instruction Formats
- Addressing Modes
- Data Transfer and Manipulation
- Program Control and Program Interrupt
- Reduced Instruction Set Computer
- Complex Instruction Set Computer

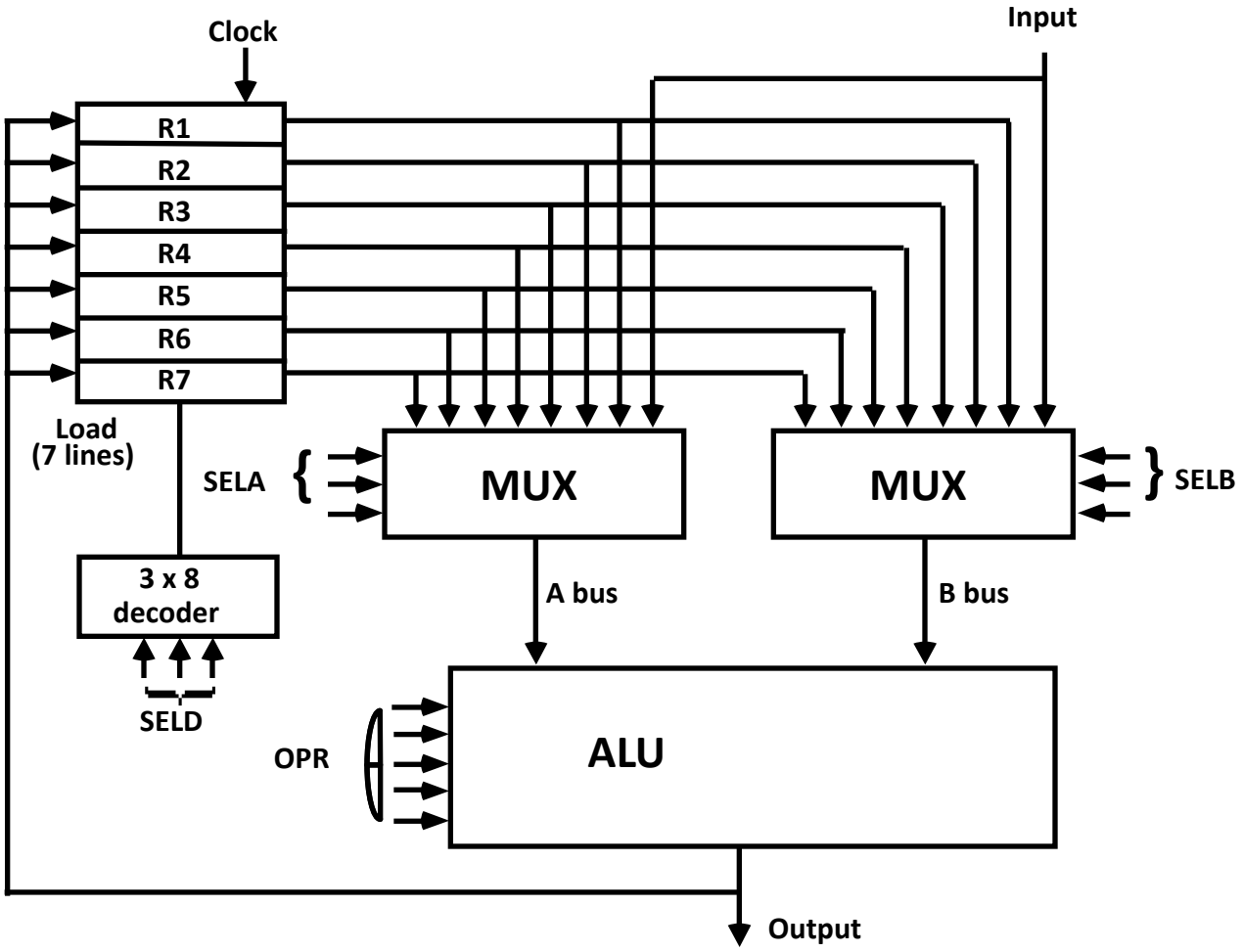
Major Components of CPU

- **Storage Components**
  - **Registers**
  - **Flags(Status registers)**
- **Execution (Processing) Components**
  - **Arithmetic Logic Unit(ALU)**
    - **Arithmetic calculations, Logical computations, Shifts/Rotates**
- **Transfer Components**
  - **Bus**
- **Control Components**
  - **Control Unit**

Processor Organization

- **In general, most processors are organized in one of 3 ways**
  - **Single register (Accumulator) organization**
    - **Basic Computer is a good example**
    - **Accumulator is the only general purpose register**
  - **General register organization**
    - **Used by most modern computer processors**
    - **Any of the registers can be used as the source or destination for computer operations**
  - **Stack organization**
    - **All operations are done using the hardware stack**
    - **For example, an OR instruction will pop the two top elements from the stack, do a logical OR on them, and push the result on the stack**

General Register Organization



$$R1 \leftarrow R2 - R3$$

Field:	SELA	SELB	SELD	OPR
Symbol:	R2	R3	R1	SUB
Control word:	010	011	001	00101

TABLE 8-3 Examples of Microoperations for the CPU

	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
$R4 \leftarrow$ sh1 $R4$	R4	—	R4	SHLA	100 000 100 11000
$R5 \leftarrow 0$	R5	R5	R5	XOR	101 101 101 01100

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

Q- Determine the micro operation that will be executed in the processor of general register organisations when 00101001100101. 14 bits control word is applied .

A-R3<—R1+R2
B-R3<—R1-R2
C-R4<—R1+R2
D-R1<—R1+R2
E-None of these

Stack Organization

LOVELY PROFESSIONAL UNIVERSITY

Stack

- Very useful feature for nested subroutines, nested interrupt services
- Also efficient for arithmetic expression evaluation
- Storage which can be accessed in LIFO
- Pointer: SP
- Only PUSH and POP operations are applicable

Stack Organization

- Register Stack Organization
- Memory Stack Organization

Register Stack Organization

- The computers which use Stack-based CPU Organization are based on a data structure called **stack**.
- The stack is a list of data words.
- It uses **Last In First Out (LIFO)** access method which is the most popular access method in most of the CPU.
- A register is used to store the address of the topmost element of the stack which is known as **Stack pointer (SP)**.
- In this organisation, ALU operations are performed on stack data.
- It means both the operands are always required on the stack. After manipulation, the result is placed in the stack.

In a 64-word stack, the stack pointer contains 6 bits because 2^6 =64. since SP has only six bits, it cannot exceed a number grater than 63(111111 in binary). When 63 is incremented by 1, the result is 0 since 111111 + 1 =1000000 in binary, but SP can accommodate only the six least significant bits. Similarly, when 000000 is decremented by 1, the result is 111111. The one bit register Full is set to 1 when the stack is full, and the one-bit register EMTY is set to 1 when the stack is empty of items. DR is the data register that holds the binary data to be written in to or read out of the stack.

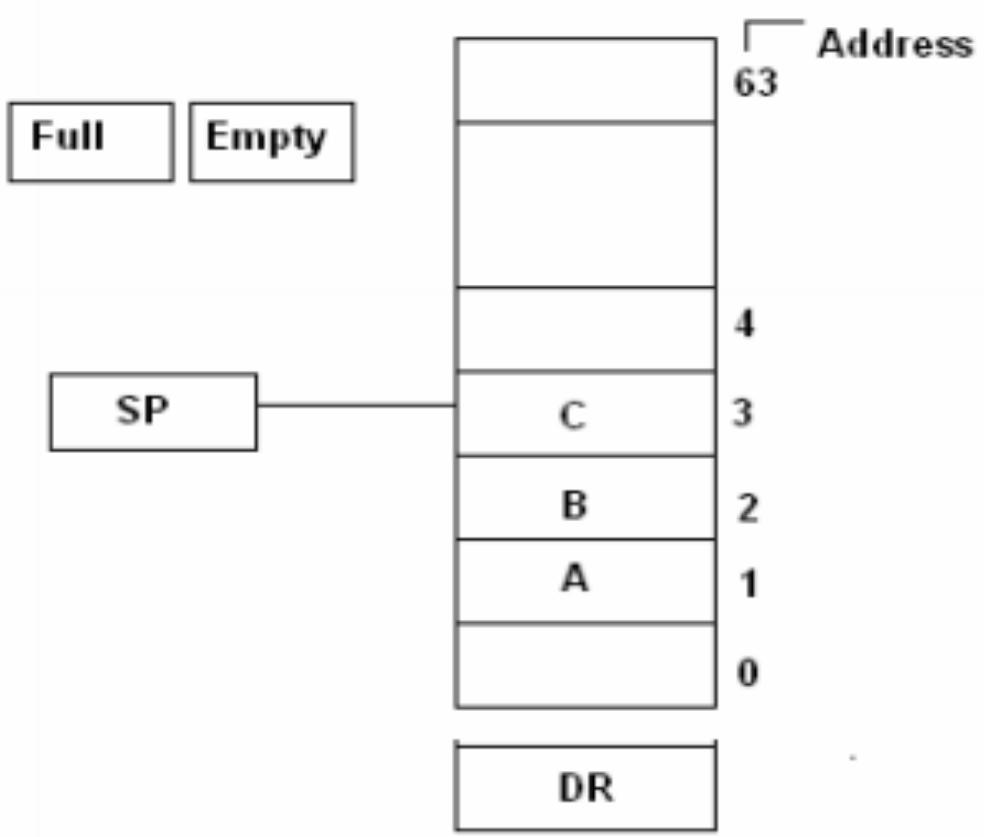
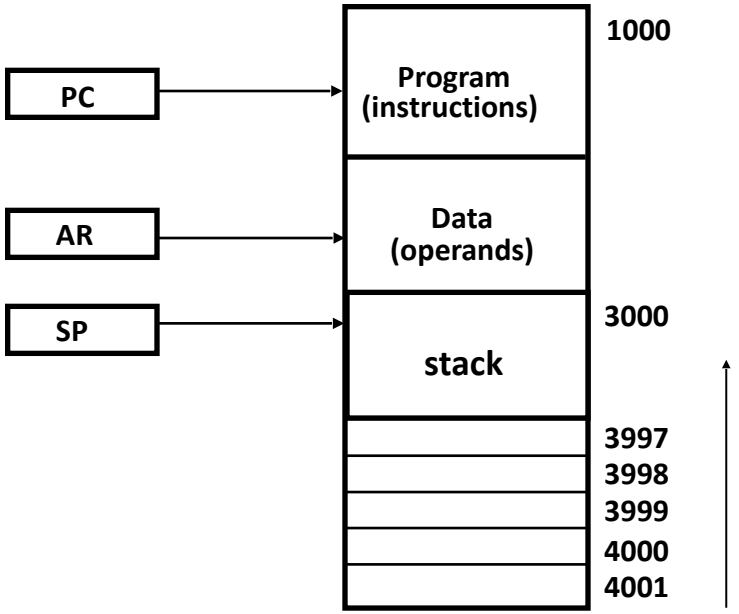


Figure :3 Block Diagram of a 64-word stack

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: $DR \leftarrow M[SP]$

  $SP \leftarrow SP + 1$

- Most computers do not provide hardware to check stack overflow (full stack) or underflow (empty stack) ? must be done in software

Instruction Format

- **Instruction Fields**

  OP-code field - specifies the operation to be performed

  Address field - designates memory address(es) or a processor register(s)

  Mode field - determines how the address field is to be interpreted (to get effective address or the operand)

- **The number of address fields in the instruction format depends on the internal organization of CPU**
- **The three most common CPU organizations:**

Single accumulator organization:

```
ADD     X                   /* AC ← AC + M[X]  */
```

General register organization:

```
ADD     R1, R2, R3    /* R1 ← R2 + R3  */
ADD     R1, R2              /* R1 ← R1 + R2  */
MOV     R1, R2              /* R1 ← R2  */
ADD     R1, X               /* R1 ← R1 + M[X]  */
```

Stack organization:

```
PUSH    X                   /* TOS ← M[X]  */
ADD
```

Three & Two Address Instruction

- **Three-Address Instructions**

Program to evaluate X = (A + B) * (C + D) :

```
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       */
```

- Results in short programs
- Instruction becomes long (many bits)

- **Two-Address Instructions**

Program to evaluate X = (A + B) * (C + D) :

```
MOV   R1, A       /* R1 ← M[A]       */
ADD   R1, B       /* R1 ← R1 + M[A]  */
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       */
```

-most common in commercial computer

One Address Instruction

- **One-Address Instructions**
  - **Use an implied AC register for all data manipulation**
  - **Program to evaluate X = (A + B) * (C + D) :**

```
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 Instruction

- **Zero-Address Instructions**

- Can be found in a stack-organized computer

- Program to evaluate X = (A + B) * (C + D) :

```
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             */
```

Reverse Polish Notation

- **Arithmetic Expressions: A + B**

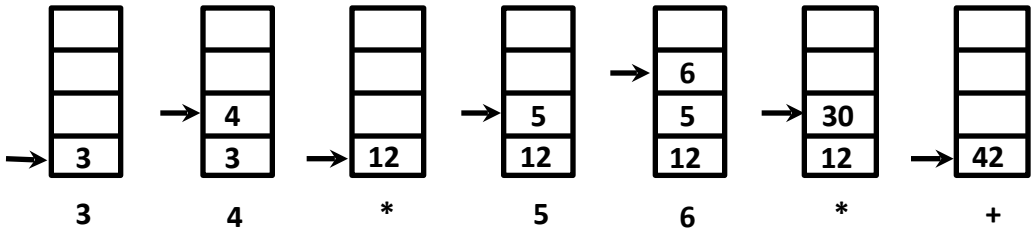
A + B	**Infix notation**
+ A B	**Prefix or Polish notation**
A B +	**Postfix or reverse Polish notation**

- The reverse Polish notation is suitable for stack manipulation

- **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 * +



Data Transfer and Manipulation

- Instruction set of different computers differ from each other mostly in way the operands are determined from the address and mode fields.

The basic set of operations available in a typical computer are :

- **Data Transfer Instructions**
- **Data Manipulation Instruction** :
  perform arithmetic, logic and shift operation
- **Program Control Instructions**
  decision making capabilities, change the path taken by the program when executed in computer.

Data Transfer Instructions

Move data from one place in computer to another without changing the data content

Most common transfer : processor reg -memory, processor reg -I/O, between processor register themselves

- **Typical Data Transfer Instructions**

Name	Mnemonic
Load	LD
Store	ST
Move	MOV
Exchange	XCH
Input	IN
Output	OUT
Push	PUSH
Pop	POP

Data Transfer Instructions

Some assembly language conventions modify the mnemonic symbol to differentiate between the different addressing modes

- **Data Transfer Instructions with Different Addressing Modes**

Mode	Assembly Convention	Register Transfer
Direct address	LD ADR	AC ← M[ADR]
Indirect address	LD @ADR	AC ← M[M[ADR]]
Relative address	LD $ADR	AC ← M[PC + ADR]
Immediate operand	LD #NBR	AC ← NBR
Index addressing	LD ADR(X)	AC ← M[ADR + XR]
Register	LD R1	AC ← R1
Register indirect	LD (R1)	AC ← M[R1]
Autoincrement	LD (R1)+	AC ← M[R1], R1 ← R1 + 1
Autodecrement	LD -(R1)	R1 ← R1 - 1, AC ← M[R1]

Data Maniplulation Instructions

These instruction performs operation on data and provide the computational capabilities for the computer

- **Three Basic Types:**
  - **Arithmetic instructions**
  - **Logical and bit manipulation instructions**
  - **Shift instructions**

Data Manipulation Instructions

Four basic arithmetic operations : + - * /

- **Arithmetic Instructions**

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

Data Manipulation Instructions

- Logical Instructions perform binary operations on string of bits stored in registers
- Useful for manipulating individual/ group of bits
- Consider each bit separately

- **Logical and Bit Manipulation Instructions**

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

AND ? Clear selected bits

OR ? Set selected bits

XOR ? Complement selected bits

Data Manipulation Instructions

- **Shift Instructions**

Shift Instructions

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

ROLC

Addressing Mode

Addressing Modes

- Specifies a rule for interpreting or modifying the address field of the instruction (before the operand is actually referenced)

- Variety of addressing modes

- to give programming flexibility to the user

- to use the bits in the address field of the instruction efficiently

1. Implied Mode

- **Address of the operands are specified implicitly in the definition of the instruction**
- **No need to specify address in the instruction**
- **EA = AC, or EA = Stack[SP]**
- **Examples from Basic Computer - CLA, CME, INP**

2. Immediate Mode

- **Instead of specifying the address of the operand, operand itself is specified**
- **No need to specify address in the instruction**
- **However, operand itself needs to be specified**
- **Sometimes, require more bits than the address**
- **Fast to acquire an operand**

Addressing Mode

3. Register Mode

- Address specified in the instruction is the register address
- Designated operand need to be in a register
- Shorter address than the memory address
- Saving address field in the instruction
- Faster to acquire an operand than the memory addressing
- EA = IR(R) (IR(R): Register field of IR)

4. Register Indirect Mode

- Instruction specifies a register which contains the memory address of the operand
- Saving instruction bits since register address is shorter than the memory address
- Slower to acquire an operand than both the register addressing or memory addressing
- EA = [IR(R)] ([x]: Content of x)

5. Autoincrement or Autodecrement Mode

- When the address in the register is used to access memory, the value in the register is incremented or decremented by 1 automatically

Addressing Mode

6. Direct Address Mode

- Instruction specifies the memory address which can be used directly to access the memory
- Faster than the other memory addressing modes
- Too many bits are needed to specify the address for a large physical memory space
- EA = IR(addr) (IR(addr): address field of IR)

7. Indirect Addressing Mode

- The address field of an instruction specifies the address of a memory location that contains the address of the operand
- When the abbreviated address is used large physical memory can be addressed with a relatively small number of bits
- Slow to acquire an operand because of an additional memory access
- EA = M[IR(address)]

Addressing Mode

8. Relative Addressing Modes

- The Address fields of an instruction specifies the part of the address (abbreviated address) which can be used along with a designated register to calculate the address of the operand

- Address field of the instruction is short

- Large physical memory can be accessed with a small number of address bits

- EA = f(IR(address), R), R is sometimes implied

-3 different Relative Addressing Modes depending on R;

PC Relative Addressing Mode (R = PC)

- EA = PC + IR(address)

Indexed Addressing Mode (R = IX, where IX: Index Register)

- EA = IX + IR(address)

Base Register Addressing Mode

(R = BAR, where BAR: Base Address Register)

- EA = BAR + IR(address)

Numerical Example

To show the differences between the various modes, we will show the effect of the addressing modes on the instruction defined in Fig. 8-7. The two-word instruction at address 200 and 201 is a "load to *AC*" instruction with an address field equal to 500. The first word of the instruction specifies the operation code and mode, and the second word specifies the address part. *PC* has the value 200 for fetching this instruction. The content of processor register *R*1 is 400, and the content of an index register *XR* is 100. *AC* receives the operand after the instruction is executed. The figure lists a few pertinent addresses and shows the memory content at each of these addresses.

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

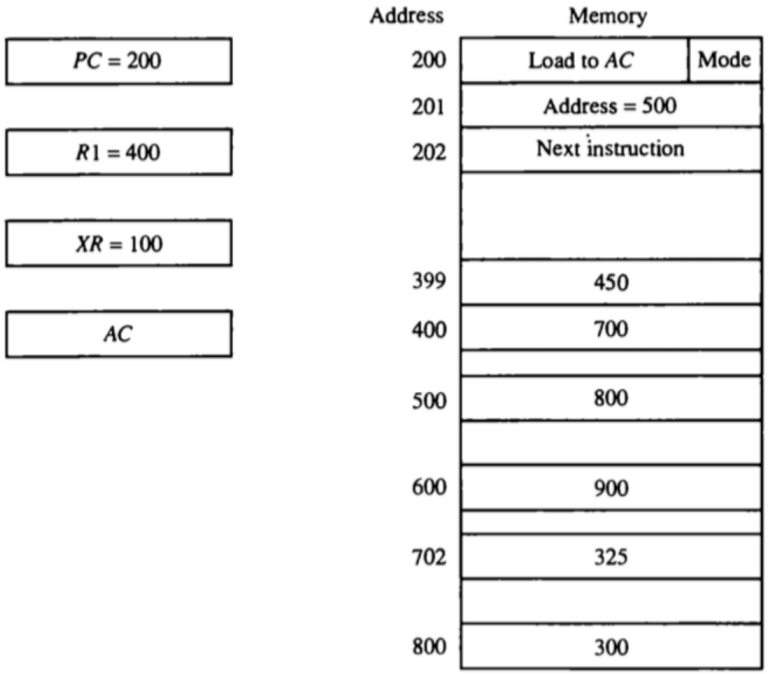
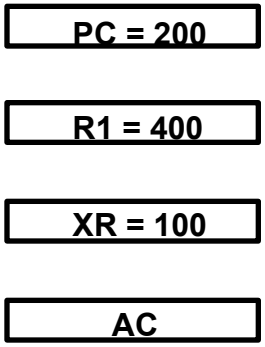
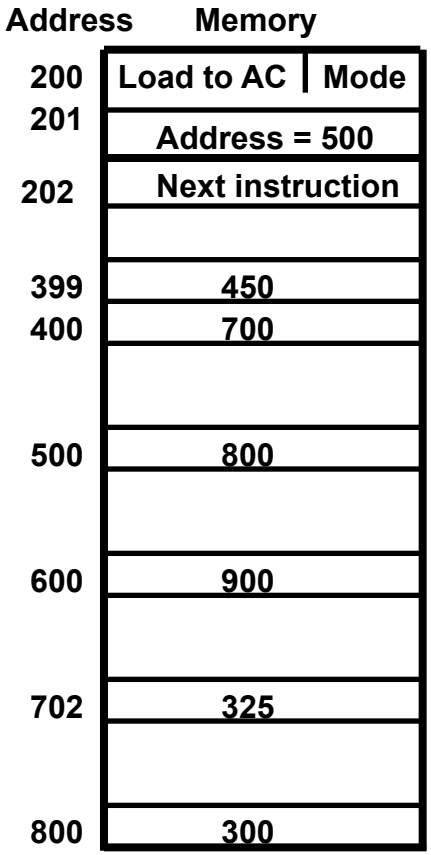


Figure 8-7 Numerical example for addressing modes.

Addressing Mode - Example





Addressing Mode	Effective Address		Content of AC
Direct address	500	/* AC ← (500) */	800
Immediate operand	-	/* AC ← 500 */	500
Indirect address	800	/* AC ← ((500)) */	300
Relative address	702	/* AC ← (PC+500) */	325
Indexed address	600	/* AC ← (RX+500) */	900
Register	-	/* AC ← R1 */	400
Register indirect	400	/* AC ← (R1) */	700
Autoincrement	400	/* AC ← (R1)+ */	700
Autodecrement	399	/* AC ← -(R) */	450

Program Interrupt

Types of Interrupts

External interrupts

External Interrupts initiated from the outside of CPU and Memory
- I/O Device → Data transfer request or Data transfer complete
- Timing Device → Timeout
- Power Failure
- Operator

Internal interrupts (traps)

Internal Interrupts are caused by the currently running program
- Register, Stack Overflow
- Divide by zero
- OP-code Violation
- Protection Violation

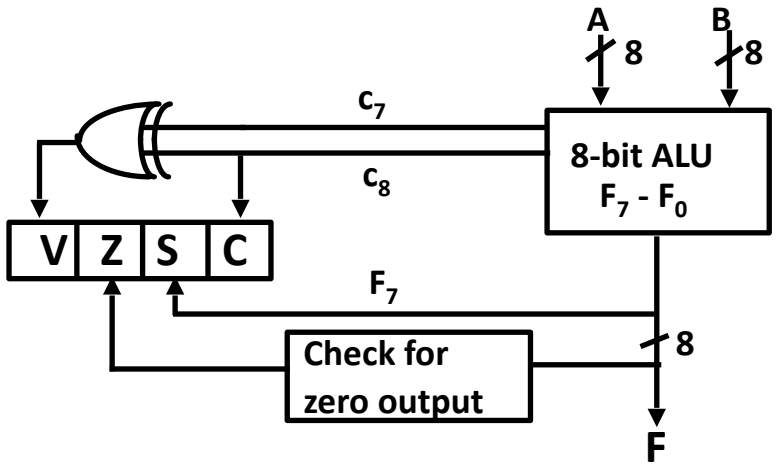
Software Interrupts

Both External and Internal Interrupts are initiated by the computer HW.
Software Interrupts are initiated by the executing an instruction.
- Supervisor Call **1. Switching from a user mode to the supervisor mode**
2. Allows to execute a certain class of operations which are not allowed in the user mode

Flag, Processor Status Word

- **In Basic Computer, the processor had several (status) flags – 1 bit value that indicated various information about the processor's state – E, FGI, FGO, I, IEN, R**
- **In some processors, flags like these are often combined into a register – the processor status register (PSR); sometimes called a processor status word (PSW)**
- **Common flags in PSW are**
  - **C (Carry): Set to 1 if the carry out of the ALU is 1**
  - **S (Sign): The MSB bit of the ALU's output**
  - **Z (Zero): Set to 1 if the ALU's output is all 0's**
  - **V (Overflow): Set to 1 if there is an overflow**

Status Flag Circuit



Status Bit Condition

Status Bit	Description	Set to 1	Clear to 0
C	Carry	If end carry C8 =1	If carry=0
S	Sign	Highest order bit F7=1	If F7=0
Z	Zero	O/P of ALU contains all 0	otherwise
V	Overflow	EX-OR of last two carries=1	otherwise

Mnemonic	Branch condition	Tested condition
BZ	Branch if zero	$Z = 1$
BNZ	Branch if not zero	$Z = 0$
BC	Branch if carry	$C = 1$
BNC	Branch if no carry	$C = 0$
BP	Branch if plus	$S = 0$
BM	Branch if minus	$S = 1$
BV	Branch if overflow	$V = 1$
BNV	Branch if no overflow	$V = 0$
	Unsigned compare conditions ($A - B$)	
BHI	Branch if higher	$A > B$
BHE	Branch if higher or equal	$A \geq B$
BLO	Branch if lower	$A < B$
BLOE	Branch if lower or equal	$A \leq B$
BE	Branch if equal	$A = B$
BNE	Branch if not equal	$A \neq B$
	Signed compare conditions ($A - B$)	
BGT	Branch if greater than	$A > B$
BGE	Branch if greater or equal	$A \geq B$
BLT	Branch if less than	$A < B$
BLE	Branch if less or equal	$A \leq B$
BE	Branch if equal	$A = B$
BNE	Branch if not equal	$A \neq B$

Q:—Consider the two 8 bit numbers in PSW A= 01000001 B= 10000100

A-Give the decimal equivalent of each number assuming that

(i)Unsigned

(ii)Signed

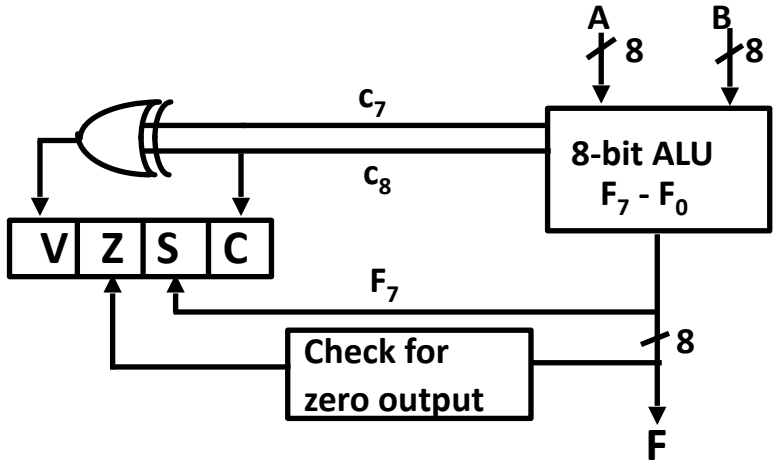
B-Add the two binary numbers and interpret the sum assuming that

(i)Unsigned

(ii)Signed

C- Determine the values of C,Z,S,V status bits after the addition.

D- Write all Branch instructions possible



Consider the two 8 bit numbers A= 01000001 B= 10000100

A-Give the decimal equivalent of each number assuming that

(i)Unsigned (I)- 65+132= 197

(ii)Signed (II)— +65-124= (-59)

B-Add the two binary numbers and interpret the sum assuming that

(i)Unsigned (I)— UNSIGNED = 197

(II)— SIGNED = -59

(ii)Signed

C- Determine the values of C,Z,S,V status bits after the addition. C= 0. Z=0. S=1. V=0

D- Write all Branch instructions possible

D— BL , BLE , BNE

Reduced Set Instruction Set Architecture (RISC) –
The main idea behind is to make hardware simpler by using an instruction set composed of a few basic steps for loading, evaluating and storing operations just like a load command will load data, store command will store the data.

Complex Instruction Set Architecture (CISC) –
The main idea is that a single instruction will do all loading, evaluating and storing operations just like a multiplication command will do stuff like loading data, evaluating and storing it, hence it's complex.

Both approaches try to increase the CPU performance

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

RISC – Reduced Instruction Set Computers

- **RISC processors often feature:**
  - **Few instructions**
  - **Few addressing modes**
  - **Only load and store instructions access memory**
  - **All other operations are done using on-processor registers**
  - **Fixed length instructions**
  - **Single cycle execution of instructions**
  - **The control unit is hardwired, not microprogrammed**

Complex Instruction Set Computers

- **Another characteristic of CISC computers is that they have instructions that act directly on memory addresses**
  - **For example,**

    ADD L1, L2, L3

    that takes the contents of *M*[*L1*] adds it to the contents of *M*[*L2*] and stores the result in location *M*[*L3*]
- **An instruction like this takes three memory access cycles to execute**
- **That makes for a potentially very long instruction execution cycle**
- **The problems with CISC computers are**
  - **The complexity of the design may slow down the processor,**
  - **The complexity of the design may result in costly errors in the processor design and implementation,**
  - **Many of the instructions and addressing modes are used rarely, if ever**

Characteristic of CISC –

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

RISC	CISC
Focus on software	Focus on hardware
Uses only Hardwired control unit	Uses both hardwired and micro programmed control unit
Transistors are used for more registers	Transistors are used for storing complex Instructions
Fixed sized instructions	Variable sized instructions
Can perform only Register to Register Arthmetic operations	Can perform REG to REG or REG to MEM or MEM to MEM
Requires more number of registers	Requires less number of registers
Code size is large	Code size is small
A instruction execute in single clock cycle	Instruction take more than one clock cycle
A instruction fit in one word	Instruction are larger than size of one word

Difference Between CISC and RISC

Architectural Characterstics	Complex Instruction Set Computer(CISC)	Reduced Instruction Set Computer(RISC)
Instruction size and format	Large set of instructions with variable formats (16-64 bits per instruction).	Small set of instructions with fixed format (32 bit).
Data transfer	Memory to memory.	Register to register.
CPU control	Most micro coded using control memory (ROM) but modern CISC use hardwired control.	Mostly hardwired without control memory.
Instruction type	Not register based instructions.	Register based instructions.
Memory access	More memory access.	Less memory access.
Clocks	Includes multi-clocks.	Includes single clock.
Instruction nature	Instructions are complex.	Instructions are reduced and simple.