CSE211

Computer Organization and Design

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

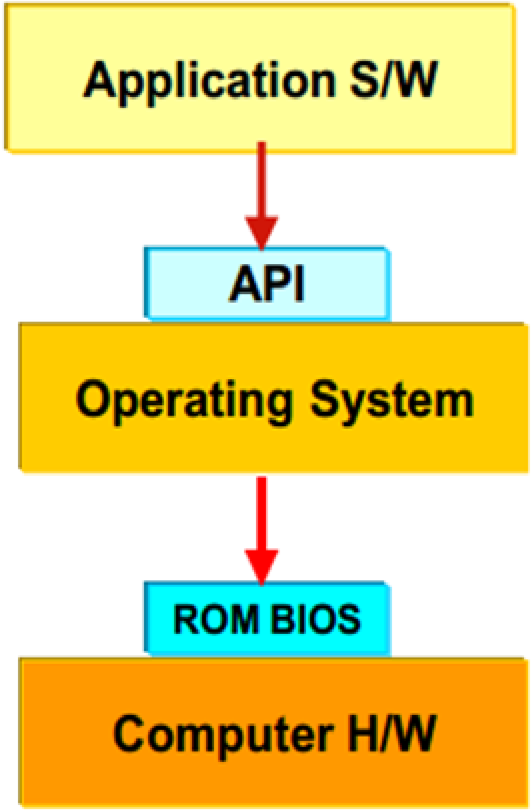
Unit 1 : Basics of Digital Electronics

- 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

- Computer Hardware
  - CPU
  - Memory
    - Program Memory(ROM)
    - Data Memory(RAM)
  - I/O Device
    - Interface
    - Input Device: Keyboard, Mouse, Scanner
    - Output Device: Printer, Plotter, Display
    - Storage Device(I/O): FDD, HDD, MOD

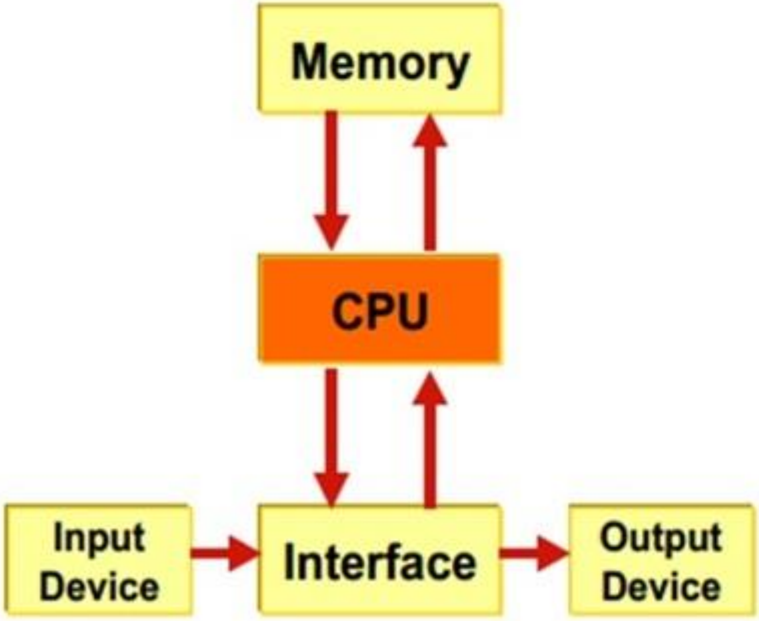


Figure *Block Diagram of a digital Computer*

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.

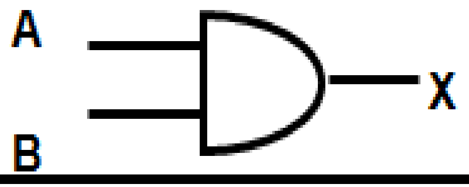
    - Amdahl, Blaaw, and Brooks(1964)

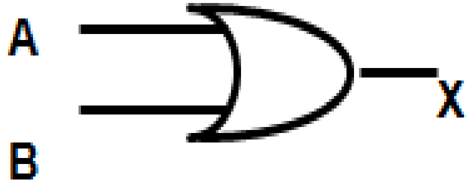
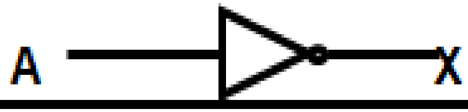
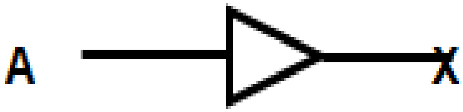
1-2 Logic Gates

- ADC(Analog to Digital Conversion)
  - Signal → Physical Quantity → Binary Information
    Discrete Value
    $\begin{cases} 0 : 0.5 \downarrow \\ 1 : 3 \uparrow \end{cases}$
- Gate
  - The manipulation of binary information is done by logic circuit called "gate".
- *Fig.* *Digital Logic Gates*
  - AND, OR, INVERTER, BUFFER, NAND, NOR, XOR, XNOR

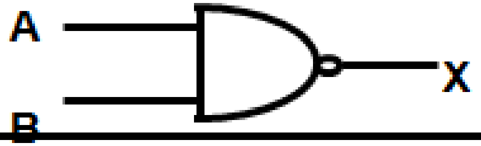
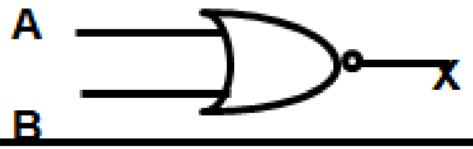
1-2 Logic Gates

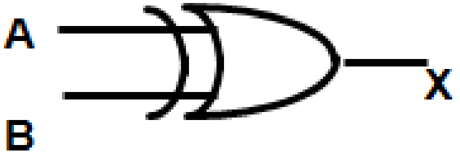
Name	Symbol	Function	Truth Table

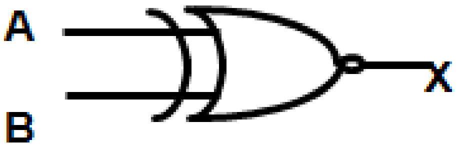
AND		X = A • B
		or
		X = AB
			A B | X
			0 0 | 0
			0 1 | 0
			1 0 | 0
			1 1 | 1

OR		X = A + B	A B | X 0 0 | 0 0 1 | 1 1 0 | 1 1 1 | 1
I		X = A’	A | X 0 | 1 1 | 0
Buffer		X = A	A | X 0 | 0 1 | 1

1-2 Logic Gates

Name	Symbol	Function	Truth Table
NAND		X = (AB)'	A B | X 0 0 | 1 0 1 | 1 1 0 | 1 1 1 | 0
NOR		X = (A + B)'	A B | X 0 0 | 1 0 1 | 0 1 0 | 0 1 1 | 0

XOR
Exclusive OR
		X = A ⊕ B
		or
		X = A'B + AB'
			A B | X
			0 0 | 0
			0 1 | 1
			1 0 | 1
			1 1 | 0

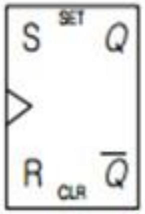
XNOR
Exclusive NOR
or Equivalence
		X = (A ⊕ B)'
		or
		X = A'B'+ AB
			A B | X
			0 0 | 1
			0 1 | 0
			1 0 | 0
			1 1 | 1

1-6 Flip-Flops

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

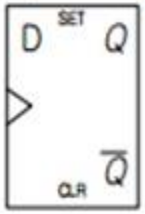
- Flip-Flop
  - The *storage elements* employed in clocked *sequential circuit*
  - A binary cell capable of storing one bit of information

- 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



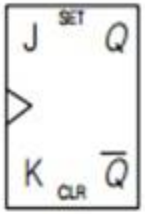
D	Q(t+1)	
0	0	clear to 0
1	1	set to 1

  - "no change" condition

    1) Disable Clock

    2) Feedback output into input

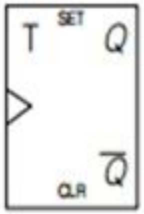
- JK*(Jack/King)* F/F



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

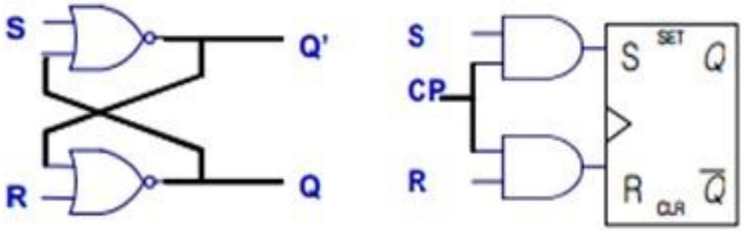


T	Q(t+1)	
0	Q(t)	no change
1	Q'(t)	Complement

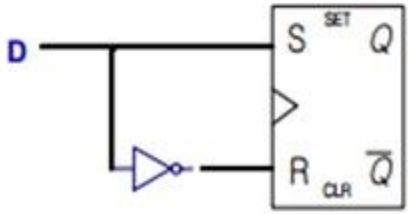
  - T=1(J=K=1), T=0(J=K=0)

1-6 Flip-Flops

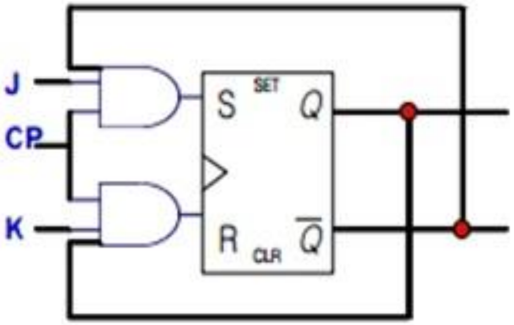
- SR*(Set/Reset)* F/F



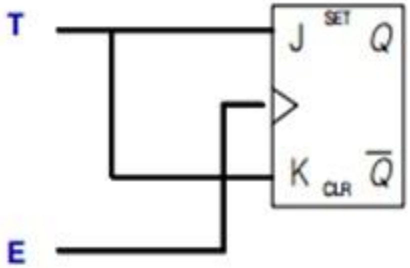
- D*(Data)* F/F



- JK*(Jack/King)* F/F

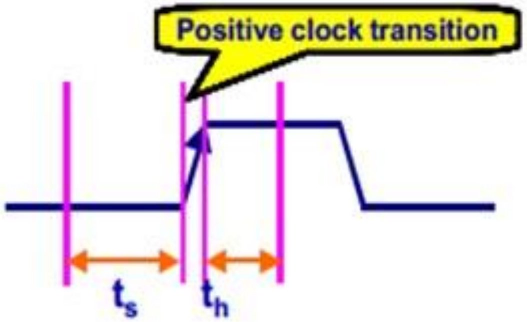


- T*(Toggle)* F/F



1-6 Flip-Flops

- Edge-Triggered F/F
  - State Change : *Clock Pulse*
    - Rising Edge(positive-edge transition) ↑
    - Falling Edge(negative-edge transition) ↓
  - Setup time(20ns)
    - minimum time that D input must remain at constant value before the transition.
  - Hold time(5ns)
    - minimum time that D input must not change after the positive transition.
  - Propagation delay(max 50ns)
    - time between the clock input and the response in Q



  - Master-Slave F/F

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)

CSE211

Computer Organization and Design

Lecture : 3 Tutorial: 2 Practical: 0 Credit: 4

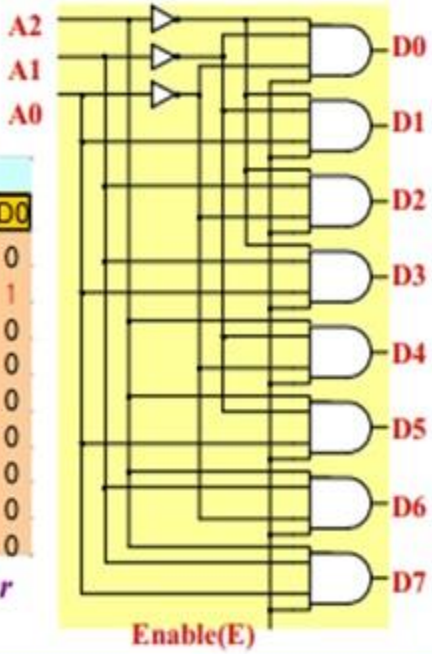
Unit 1 : Basics of Digital Electronics

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

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)

Fig. 2-1 3-to-8 Decoder



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

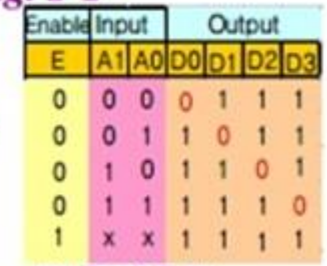
2-2 Decoder/Encoder

- NAND Gate Decoder

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

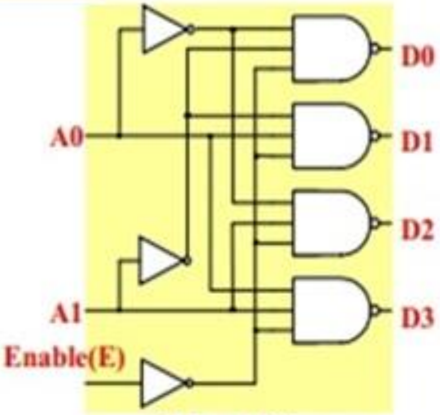
  - Constructed with NAND instead of AND gates
  - Logic Diagram/Truth Table : *Fig. 2-2*

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



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

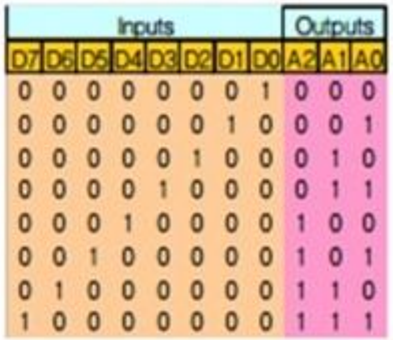
(a) Truth Table



(b) Logic Diagram

- 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

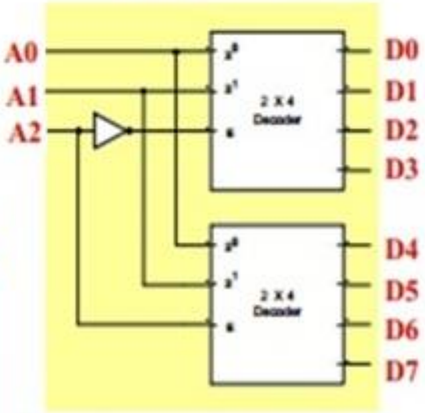


Fig. 2-3 A 3-to-8 Decoder constructed with two with 2-to-4 Decoder

2-2 Decoder/Encoder

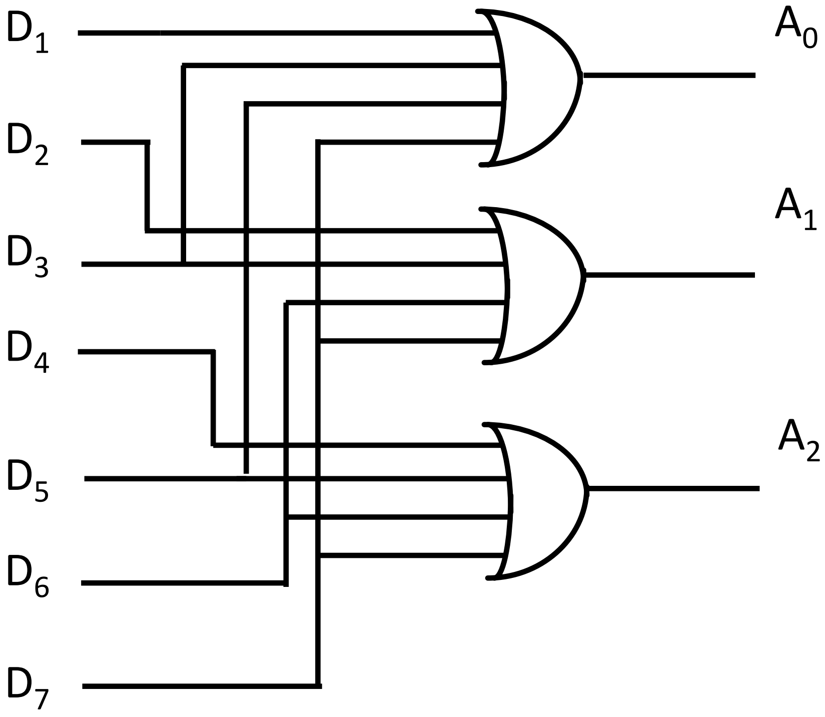
Octal to Binary Encoder

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

$$A_0 = D_1 + D_3 + D_5 + D_7$$

$$A_1 = D_2 + D_3 + D_6 + D_7$$

$$A_2 = D_4 + D_5 + D_6 + D_7$$



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

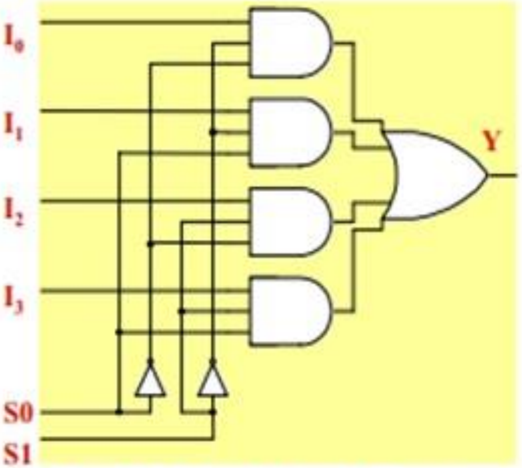


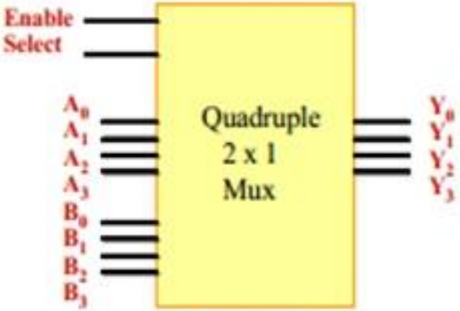
Fig. 2-4 4-to-1 Line Multiplexer

- 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

2-3 Multiplexers

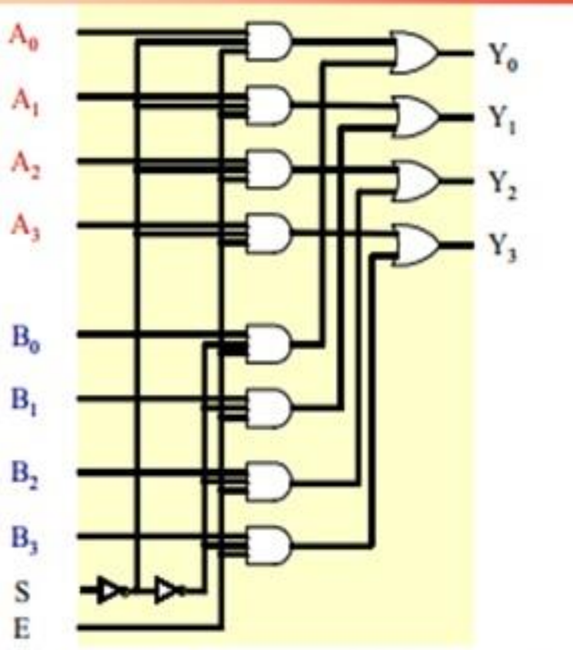
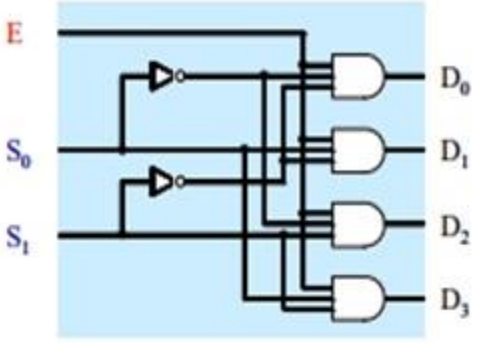


Fig A. Combinational logic diagram with four 2 × 1 multiplexer



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

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***)

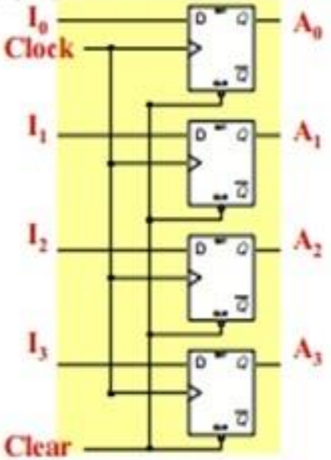


Fig. 2-6 4-bit register

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.

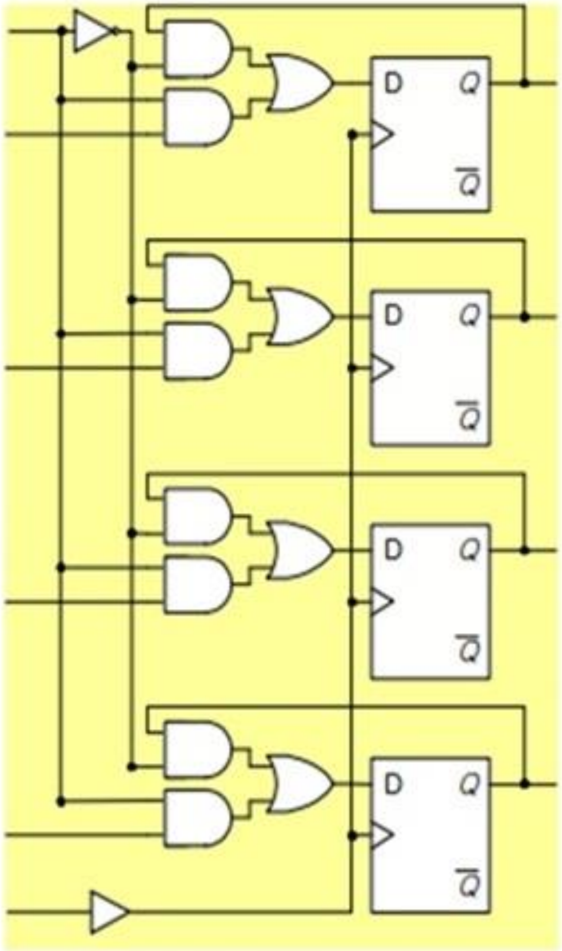


Fig. 2-7 4-bit register with parallel load

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

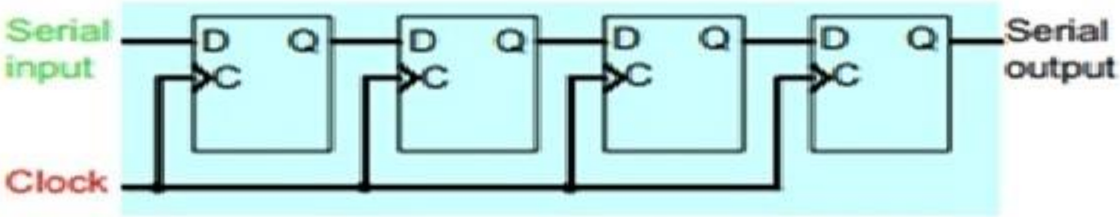
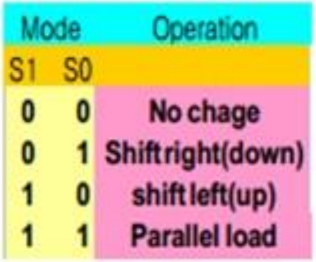


Fig. 2-8 4-bit shift register

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

2-5 Shift Registers

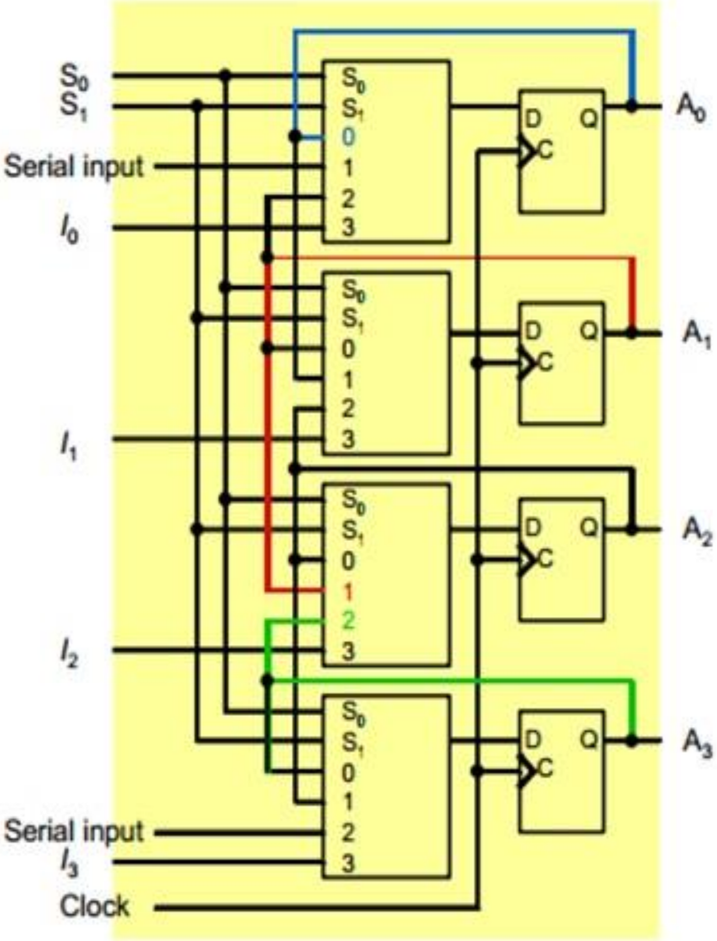


Fig. 2-9 Bidirectional shift register

$S_1S_0 = 00$: $A_i \rightarrow A_i$ (No change)

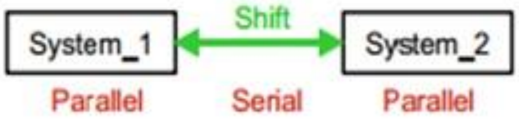
$S_1S_0 = 01$: $A_{i-1} \rightarrow A_i$ (Shift)

$S_1S_0 = 10$: $A_{i+1} \rightarrow A_i$ (Shift)

$S_1S_0 = 11$: Parallel load

- Shift Register

  Interface digital systems situated remotely from each other



CSE211

Computer Organization and Design

- *Register Transfer Language*
- *Register Transfer*

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 characterized 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

Internal hardware organization of a digital computer :

- **Set of registers and their functions**
- **Sequence of microoperations performed on binary information stored in registers**
- **Control signals that initiate the sequence of micro-operations (to perform the functions)**

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.
    - Can also be used to facilitate the design process of digital systems.

Register Transfer

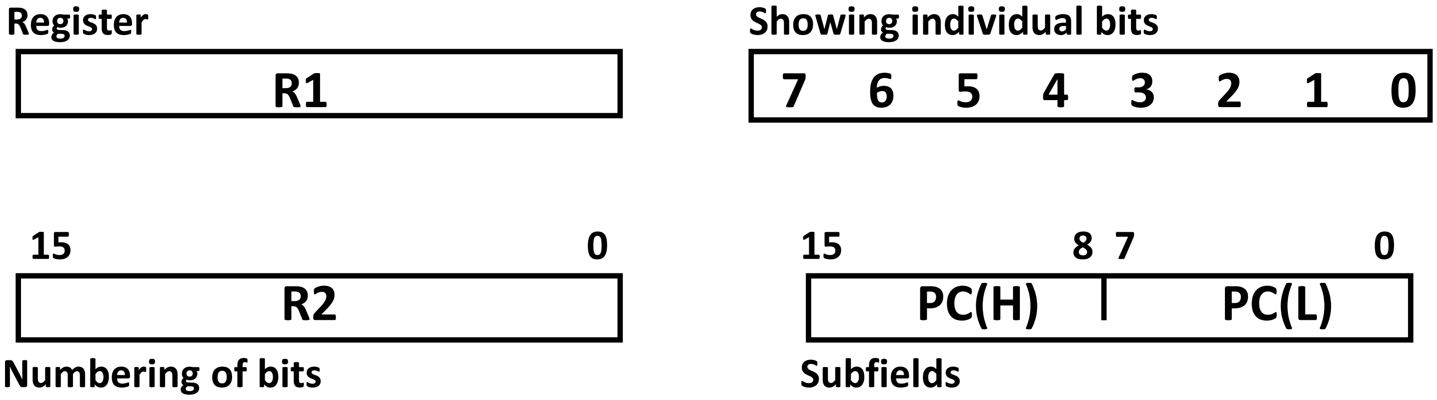
- **Registers are designated by capital letters, sometimes followed by numbers (e.g., A, R13, IR)**
- **Often the names indicate function:**
  - **MAR - memory address register**
  - **PC - program counter**
  - **IR - instruction register**
- **Registers and their contents can be viewed and represented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

Register Transfer

- **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

- **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**

Register Transfer

- **A register transfer such as**

  R3 ← R5

  Implies that the digital system has

  - **the data lines from the source register (R5) to the destination register (R3)**
  - **Parallel load in the destination register (R3)**
  - **Control lines to perform the action**

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

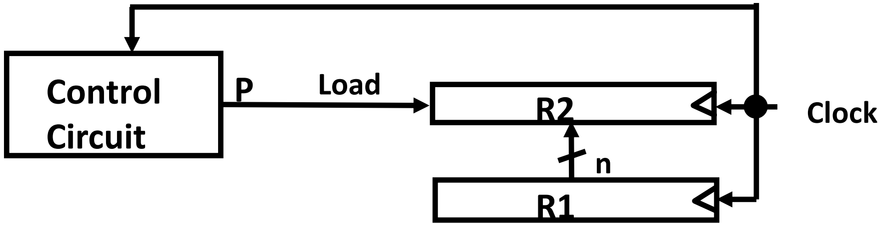
Which means "if P = 1, then load the contents of register R1 into register R2", i.e., if (P = 1) then (R2 ← R1)

Hardware Implementation of Controlled Transfers

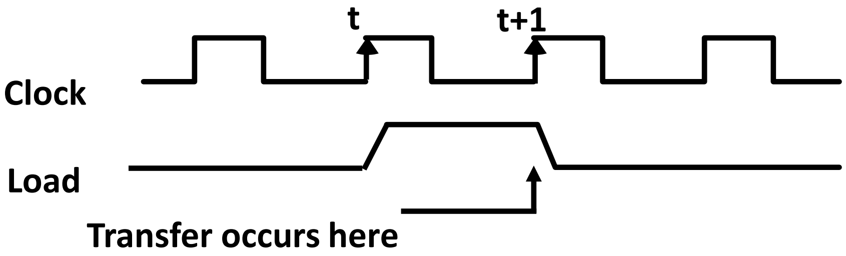
Implementation of controlled transfer

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

CSE211

Computer Organization and Design

- ***Bus and Memory Transfers***

Overview

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

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**

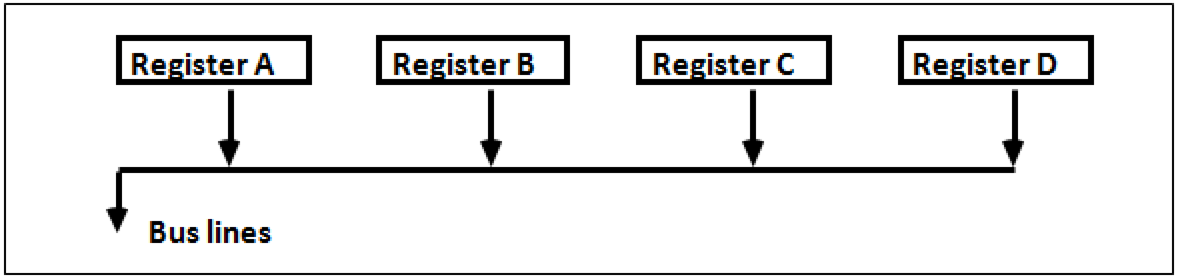
- **Instead, take a different approach**
- **Have one centralized set of circuits for data transfer – the bus**

- **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

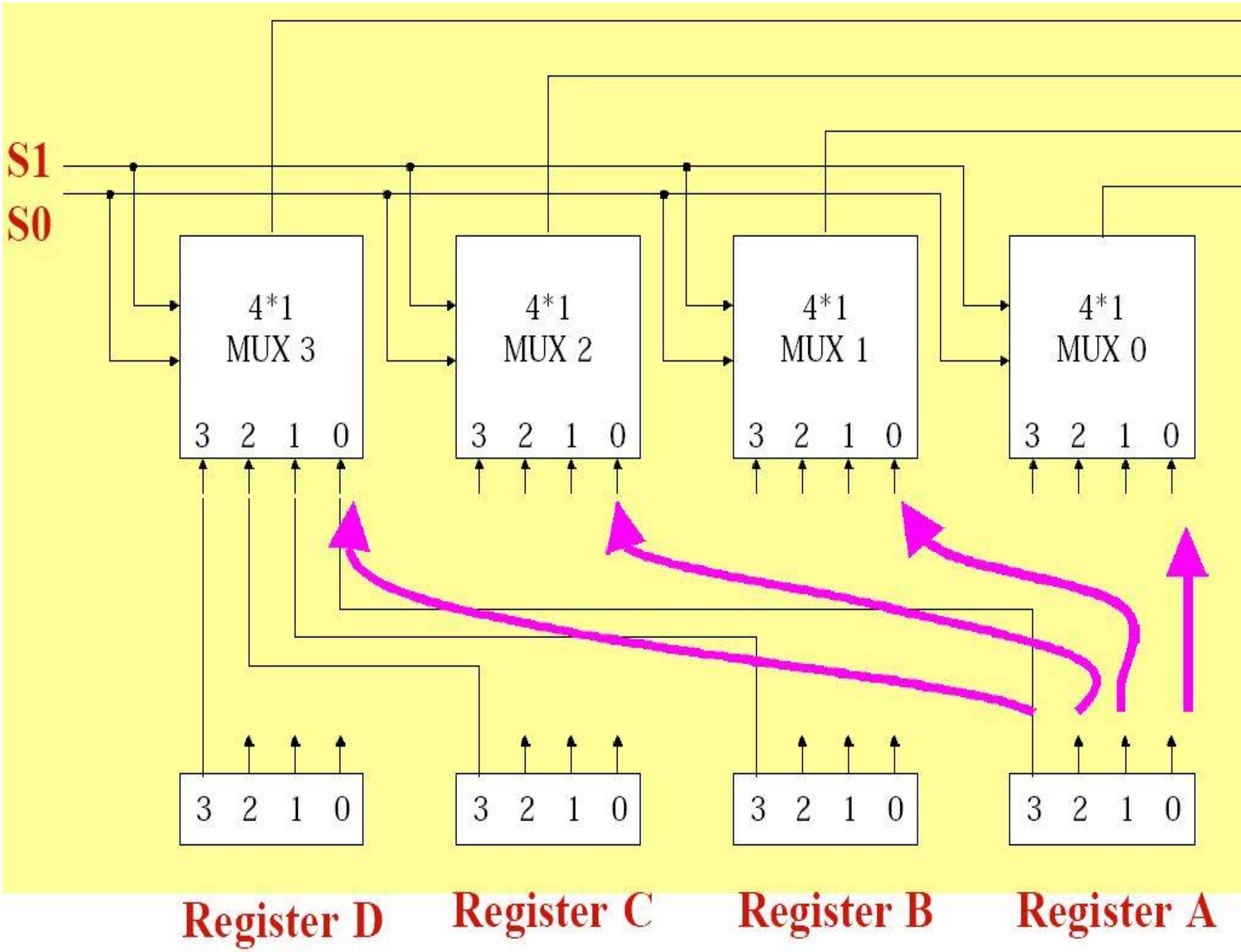
From a register to bus: BUS ← R



- **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

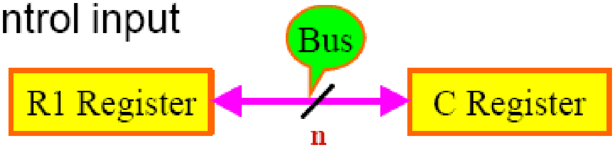
- For a bus system to multiplex k registers of n bits each
  - No. of multiplexer = n
  - Size of each multiplexer = k x 1

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

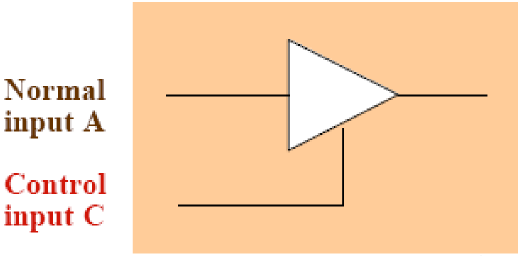
Connecting Registers - Bus Transfer

- 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)

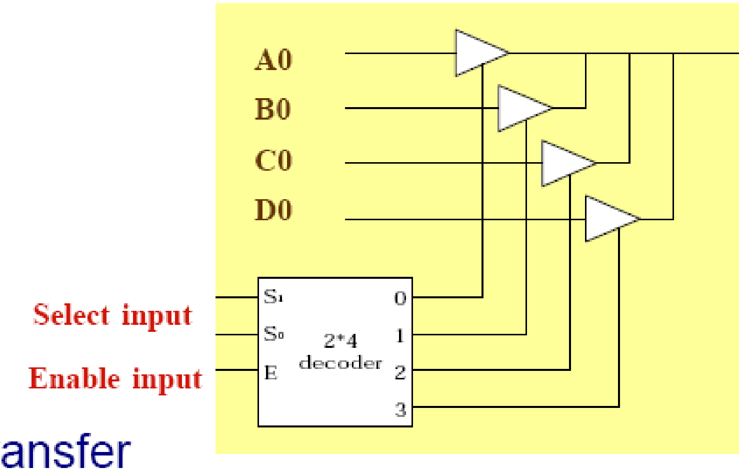


If C=1, Output Y = A

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)

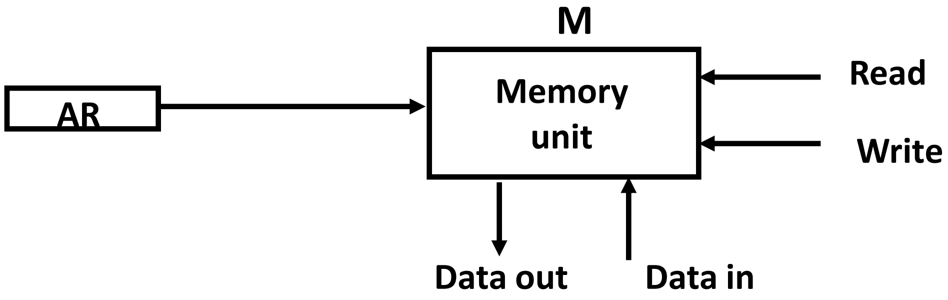
READ : $DR \leftarrow M[AR]$

WRITE : $M[AR] \leftarrow R1$

- 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

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 ← 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] ← 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**

SUMMARY OF R. TRANSFER MICROOPERATIONS

A ← B	**1.Transfer content of reg. B into reg. A**
AR ← DR(AD)	**2.Transfer content of AD portion of reg. DR into reg. AR**
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**

CSE211

Computer Organization and Design

- ***Arithmetic Microoperations***

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

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

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)

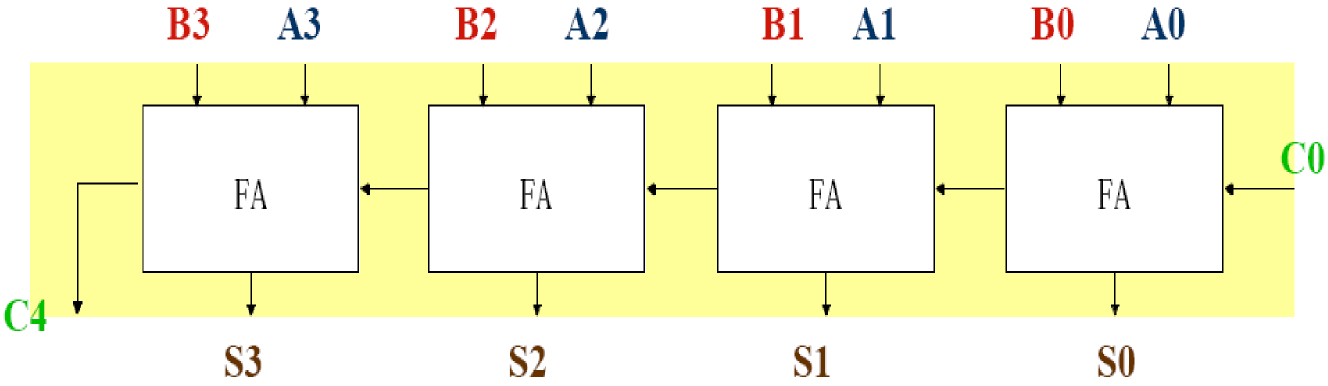
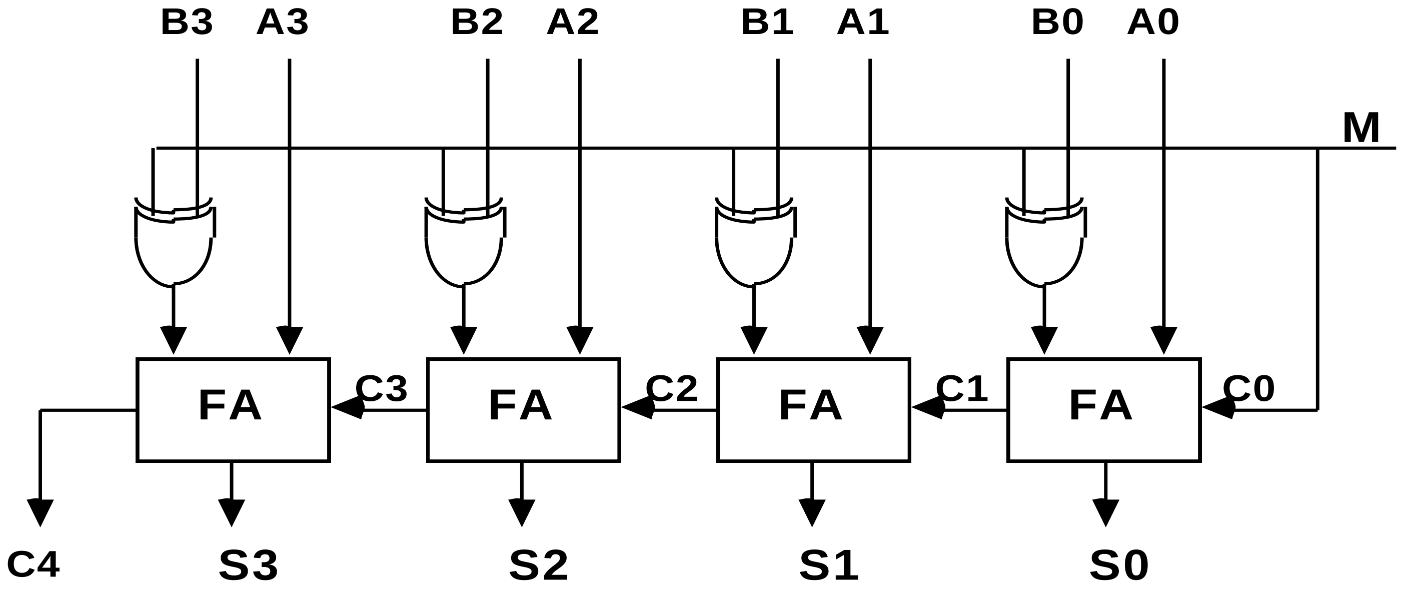


Figure 4-6. 4-bit binary adder

Binary Adder-Subtractor

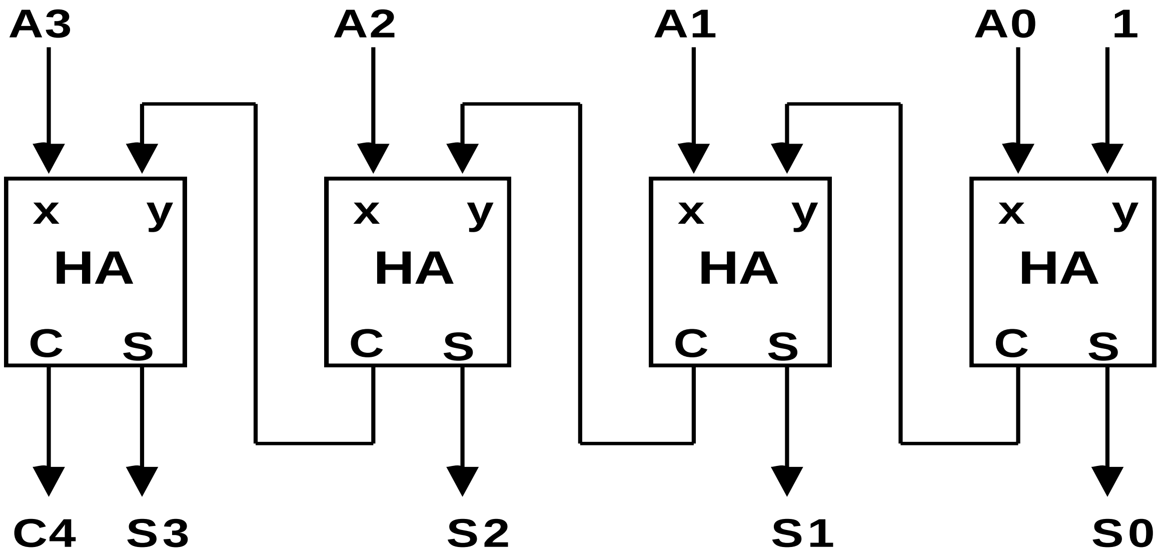
Binary Adder-Subtractor



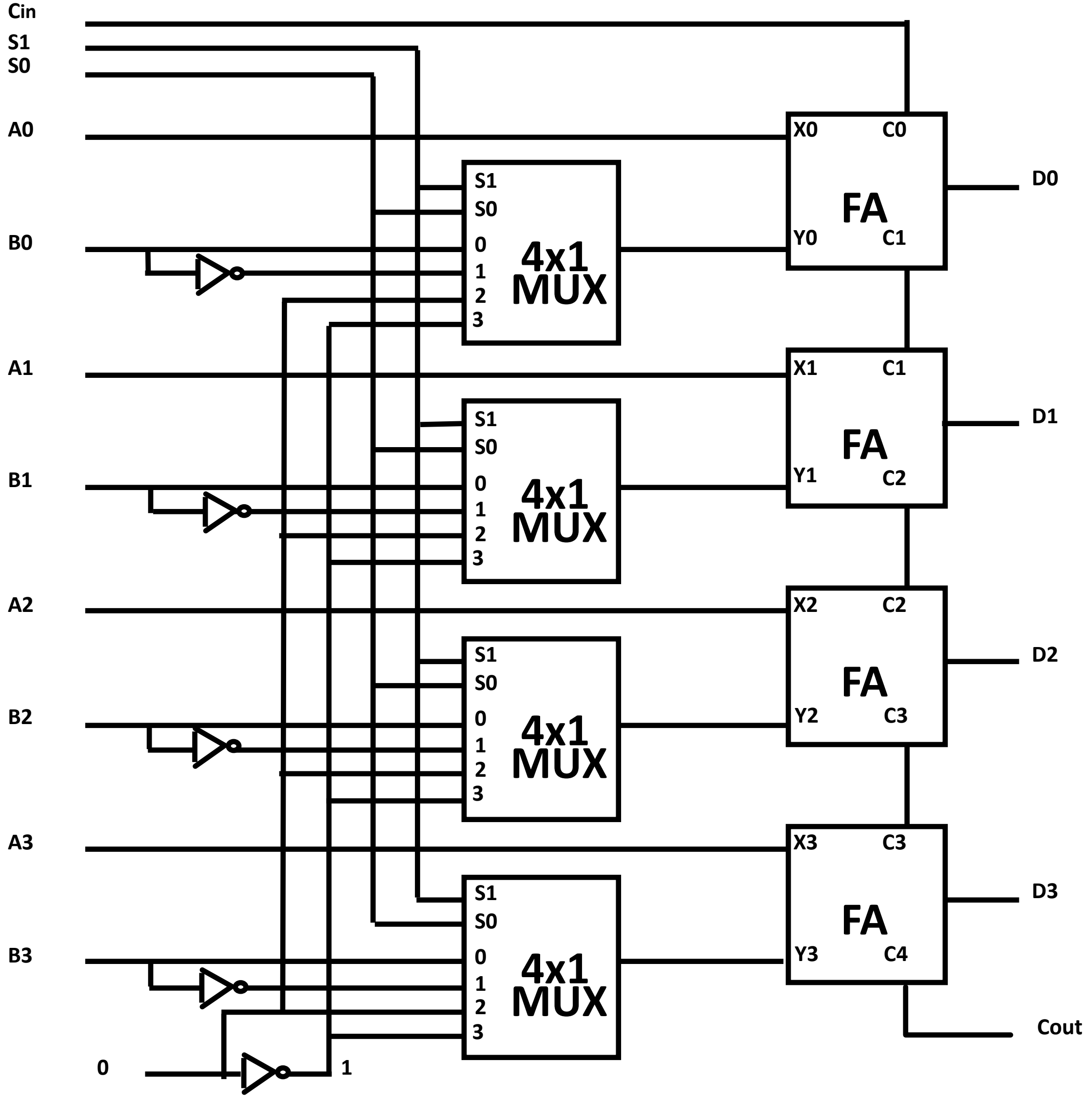
- ➢ Mode input M controls the operation
  - ➢ M=0 ---- adder
  - ➢ M=1 ---- subtractor

Binary Incrementer

Binary Incrementer



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

CSE211

Computer Organization and Design

- *Logic Microoperations*
- *Shift Microoperations*
- *Arithmetic Logic Shift Unit*

Overview

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

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

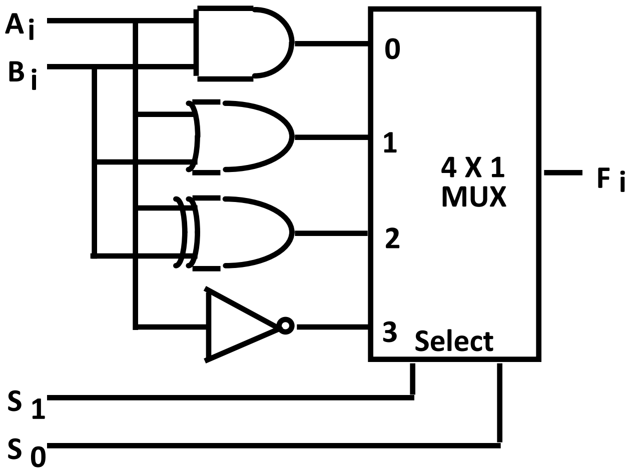
X	Y	F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}	F_{11}	F_{12}	F_{13}	F_{14}	F_{15}
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

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 ← 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 Microoperations

- 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 Microoperations

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} $(A \leftarrow A + B)$

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} $(A \leftarrow A \oplus B)$

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

Applications of Logic Microoperations

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

1 1 0 0 A_t

1 0 1 0 B

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

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 A_t

1 0 1 0 B

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

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 Microoperations

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 Microoperations

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:**
     -

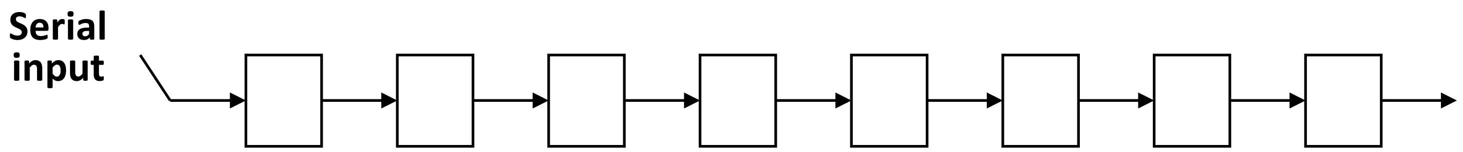
1101 1000 1011 0001	**A (Original)**
1101 1000 1011 1010	**A (Desired)**

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)**

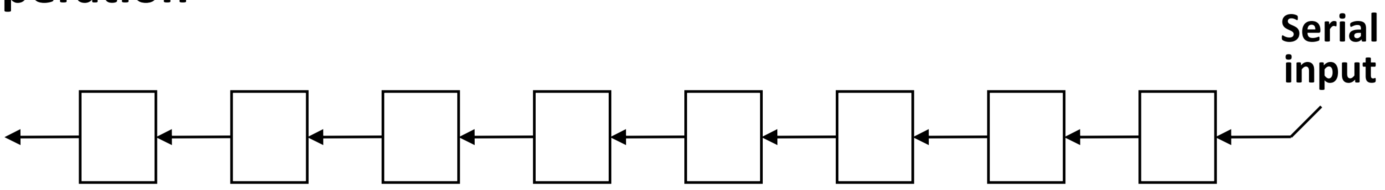
Shift Microoperations

- **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**



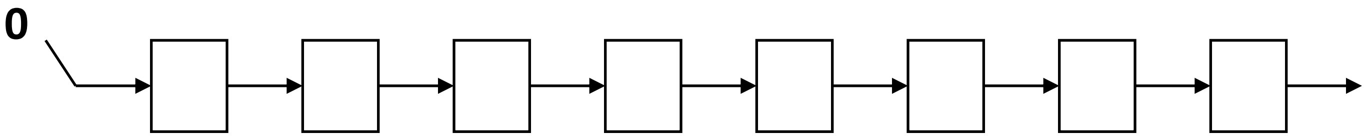
- **A left shift operation**



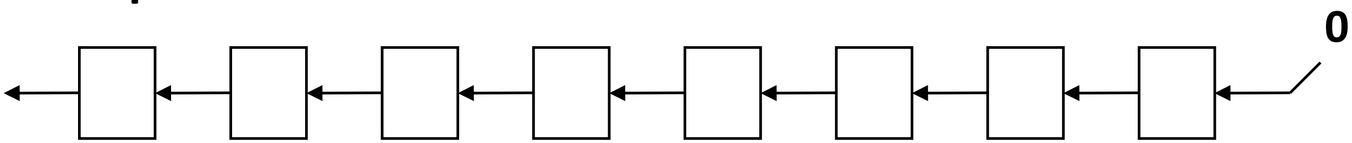
Logical Shift

- **In a logical shift the serial input to the shift is a 0.**

- **A right logical shift operation:**



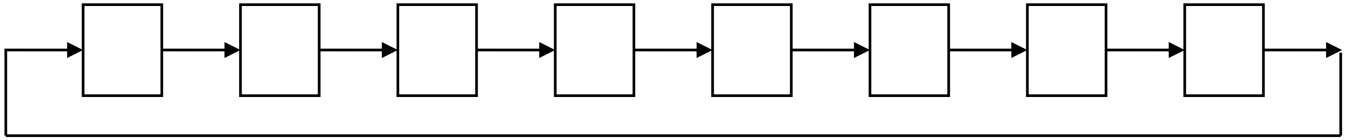
- **A left logical shift operation:**



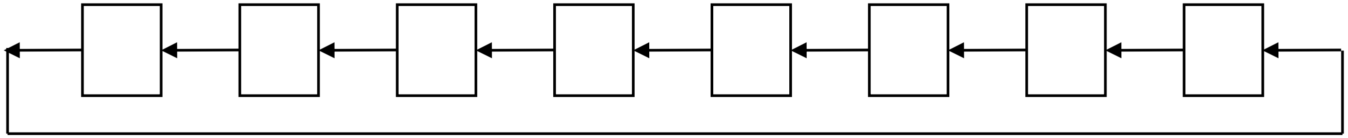
- **In a Register Transfer Language, the following notation is used**
  - ***shl*** **for a logical shift left**
  - ***shr*** **for a logical shift right**
  - **Examples:**
    - **R2 ← *shr* R2**
    - **R3 ← *shl* R3**

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:**

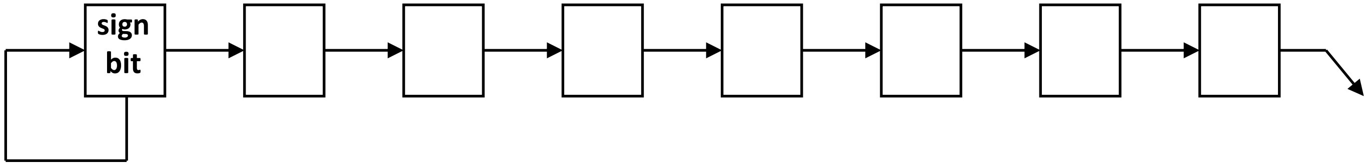


- **In a RTL, the following notation is used**
  - ***cil*** **for a circular shift left**
  - ***cir*** **for a circular shift right**
  - **Examples:**
    - **R2 ← *cir* R2**
    - **R3 ← *cil* R3**

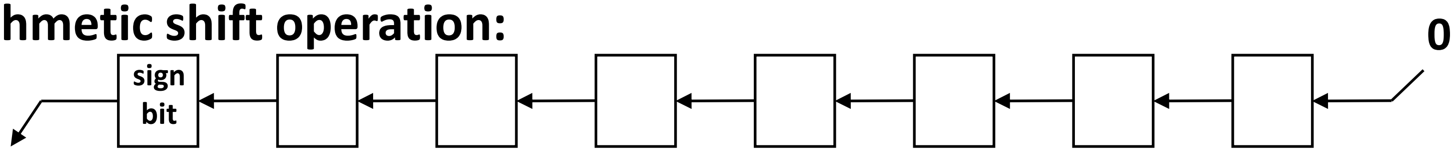
Arithmetic Shift

- **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:**

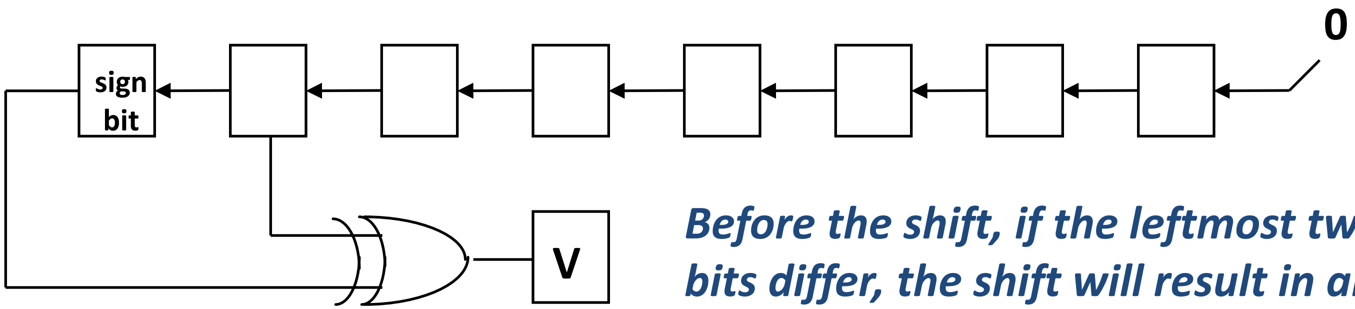


- **A left arithmetic shift operation:**



Arithmetic Shift

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

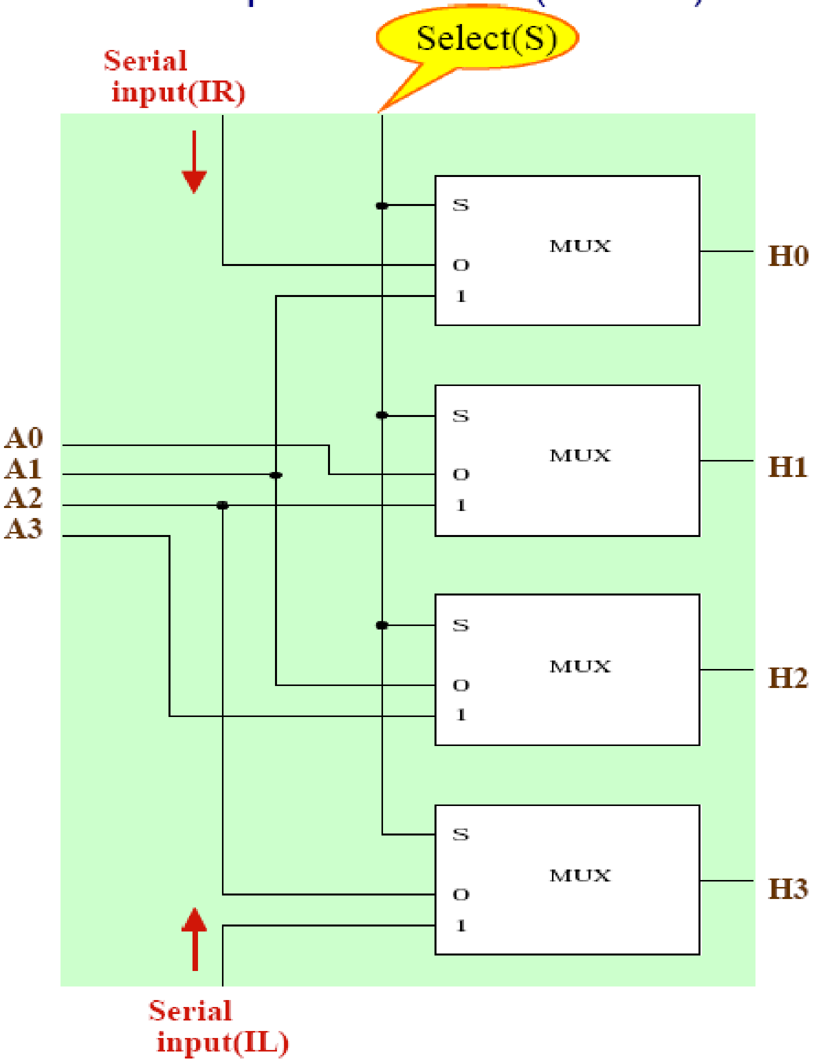


Before the shift, if the leftmost two bits differ, the shift will result in an overflow

- 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 Microoperation

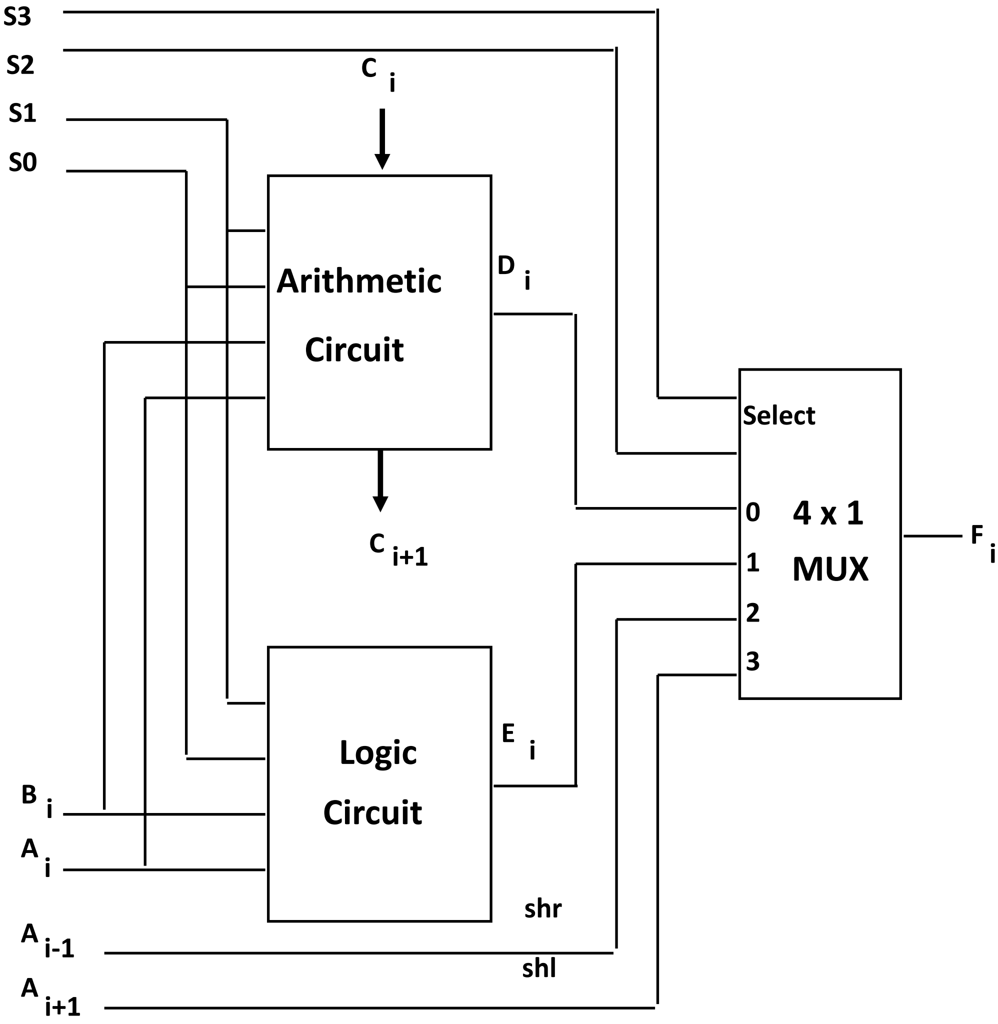
- Hardware Implementation(Shifter) :



Function Table

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

Arithmetic Logic and Shift Unit



S3	S2	S1	S0	Cin	Operation
0	0	0	0	0	F = A
0	0	0	0	1	F = A + 1
0	0	0	1	0	F = A + B
0	0	0	1	1	F = A + B + 1
0	0	1	0	0	F = A + B’
0	0	1	0	1	F = A + B’+ 1
0	0	1	1	0	F = A - 1
0	0	1	1	1	F = A
0	1	0	0	X	F = A ∧ B
0	1	0	1	X	F = A ∨ B
0	1	1	0	X	F = A ⊕ B
0	1	1	1	X	F = A’
1	0	X	X	X	F = shr A
1	1	X	X	X	F = shl A

CSE211

Computer Organization and Design

- ***Instruction Codes***
- ***Computer Registers***

Overview

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

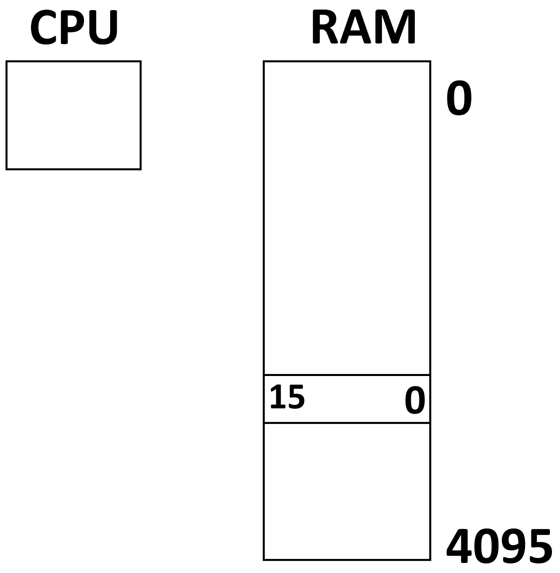
Introduction

- **Organization of computer is defined by its :**
  - **Internal Registers**
  - **Timing and Control Structure**
  - **Set of instructions that it uses**

- **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.**
- **However, to understand how processors work, we will start with a simplified processor model**

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 to select a word in memory**
- **Each word is 16 bits long**



Instruction

- **Program**
  - **A sequence of (machine) instructions**
- **Instruction**
  - **binary code that specifies a sequence of microoperations for a computer.**
- **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 microoperations necessary to implement it**

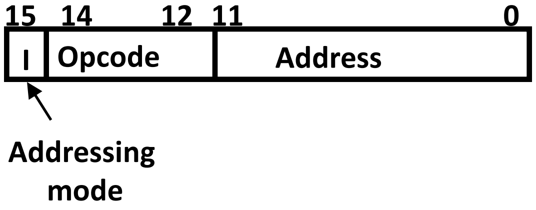
Instruction Format

- **Instruction Codes**
  - **A group of bits that tell the computer to *perform a specific operation* (a sequence of micro-operation)**
- **A computer instruction is often divided into two parts**
  - **An opcode (Operation Code) that specifies the operation for that instruction**
    - **Sometimes called as Macrooperation**
  - **An address that specifies the registers and/or locations in memory to use for that operation**
- **In the Basic Computer, 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 15 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

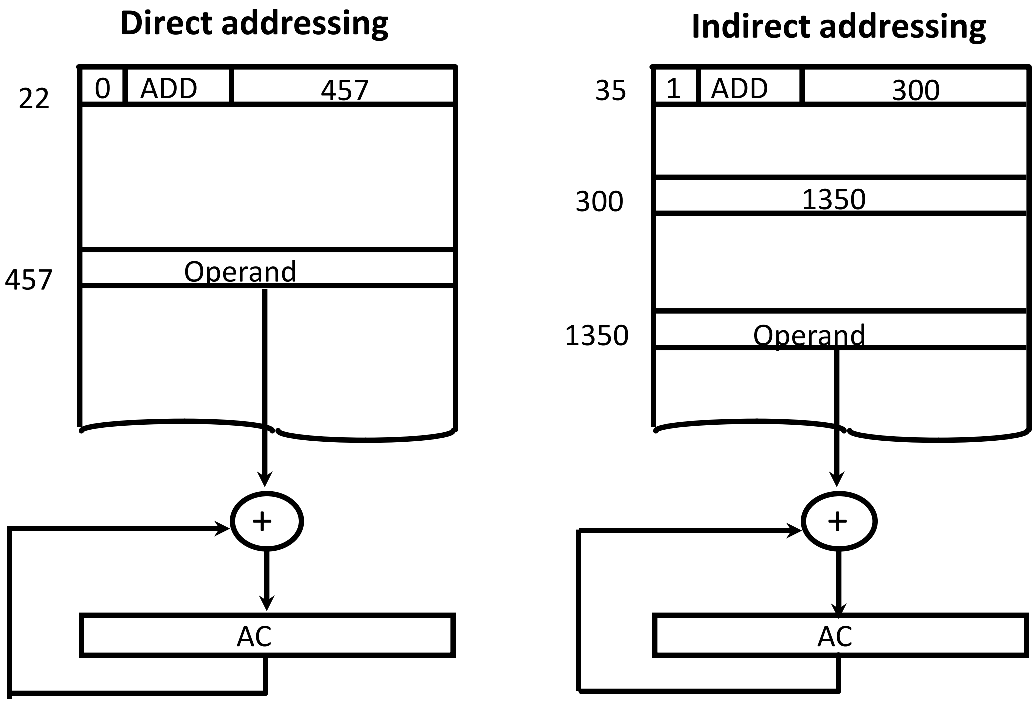
- **Sometimes the address bit of instruction code represent various different information, classified into different Instruction formats :**
  - **Immediate Instruction : when second part of instruction specifies operand**
  - **When second part of address specify address :**
    - **Direct Addressing : second part of instruction specifies address of an operand**
    - **Indirect Addressing : second part of instruction designates an address of a memory in which the address of the operand is found**

Instruction Format



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**

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 be executed

  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

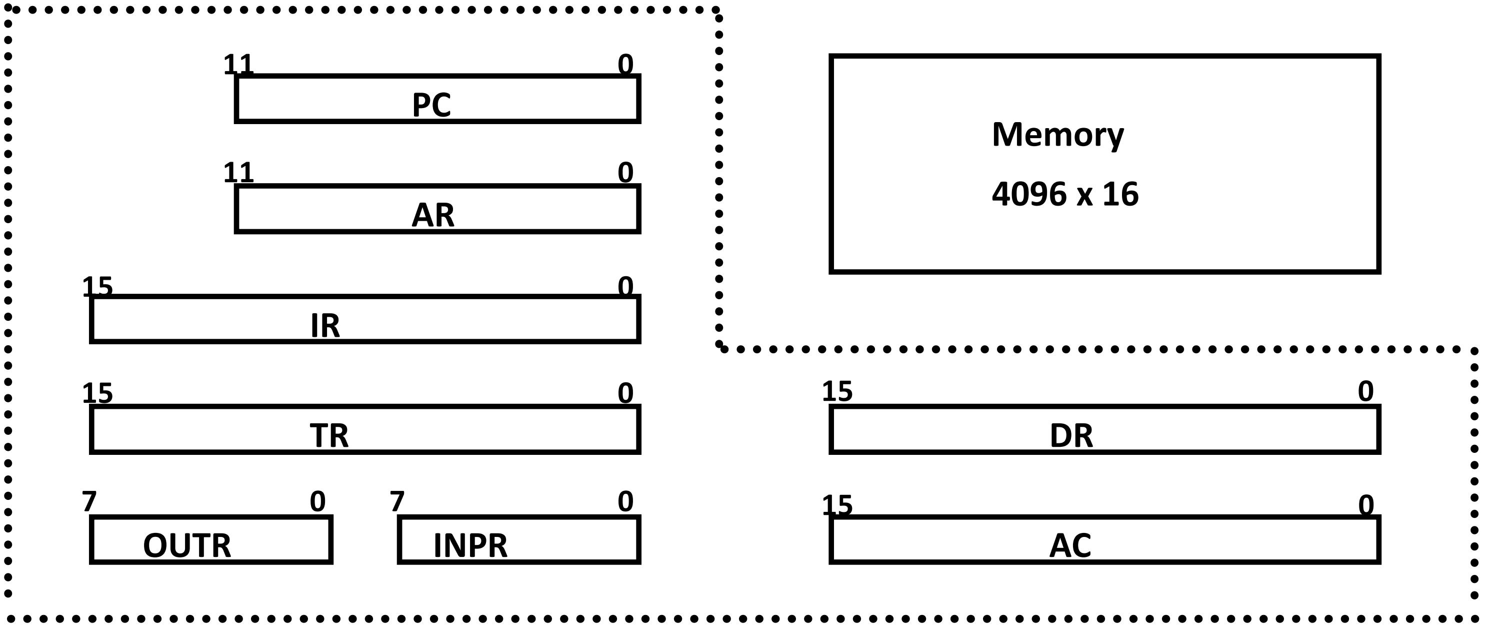
  The processor can send 8 bits of character data to output devices

- 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

Processor Register

Registers in the Basic Computer



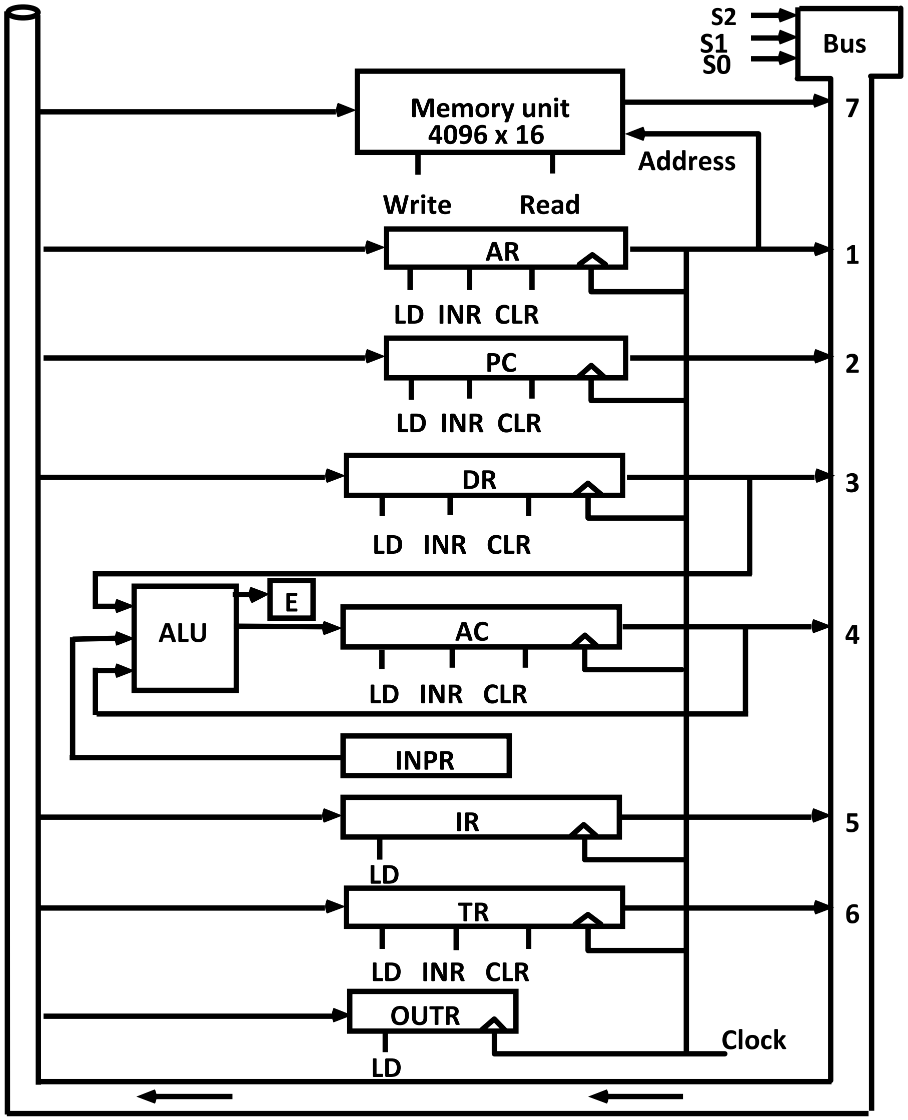
List of BC Registers

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

- **Basic computer : 8 register, a memory unit and a control unit**
- **The registers in the Basic Computer are connected using a bus**
- **This gives a savings in circuitry over complete connections between registers**
- **Output of 7 register and memory connected to input of bus**
- **Specific output that is selected for bus lines will be determined by selection variables S_2, S_1, S_0**

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 write signal activated**

  Will determine where the data from the bus gets loaded

- **Memory places its 16 bit output on bus when read input is activated and $S_2S_1S_0$=111**

Common Bus System

- **4 register DR, AC, IR, TR is 16 bit. 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**

- **INPR – connected to provide information to bus**

  - receives character from input device and transfer to AC

- **OUTR – can only receive information from bus**

  - receives a character from AC and delivers to Output device

- **Three types of input to AC :**
  - **from AC : complement AC, Shift AC**
  - **from DR : arithmetic and logic microoperation**
  - **from INPR**

Common Bus System

- Bus lines connected to inputs of 6 registers and memory

- Three types of input to AC :
  - from AC : complement AC, Shift AC
  - from DR : arithmetic and logic microoperation
  - from INPR

- Input/output data connected to common bus but memory address connected to AR

CSE211

Computer Organization and Design

- *Computer Instructions*
- *Timing and Control*
- *Instruction Cycles*
- *Memory Reference Instructions*
- *Input Output and Interrupts*
- *Complete Computer Description*

Overview

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

Basic Computer Instructions

Basic Computer Instruction Format

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

I=0 : Direct,
I=1 : Indirect

15	14 12	11 0
I	Opcode	Address

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

15 12	11 0
0 1 1 1	Register operation

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

15 12	11 0
1 1 1 1	I/O operation

Basic Computer Instructions

- **Only 3 bits are used for operation code**
- **It may seem computer is restricted to eight different operations**
- **however register reference and input output instructions use remaining 12 bit as part of operation code**
- **so total number of instruction can exceed 8**
- **Infact total no. of instructions chosen for basic computer is 25**

Basic Computer Instructions

Symbol	Hex Code I = 0	 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

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.

The set of instructions are said to be complete if computer includes a sufficient number of instruction in each of the following categories :

- **Functional Instructions**
  - **- Arithmetic, logic, and shift instructions**
  - **- ADD, CMA, INC, CIR, CIL, AND, CMA, 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**

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

  Advantage : optimized to provide fast mode of operations

  Disadvantage : requires changes in wiring if design has been modified

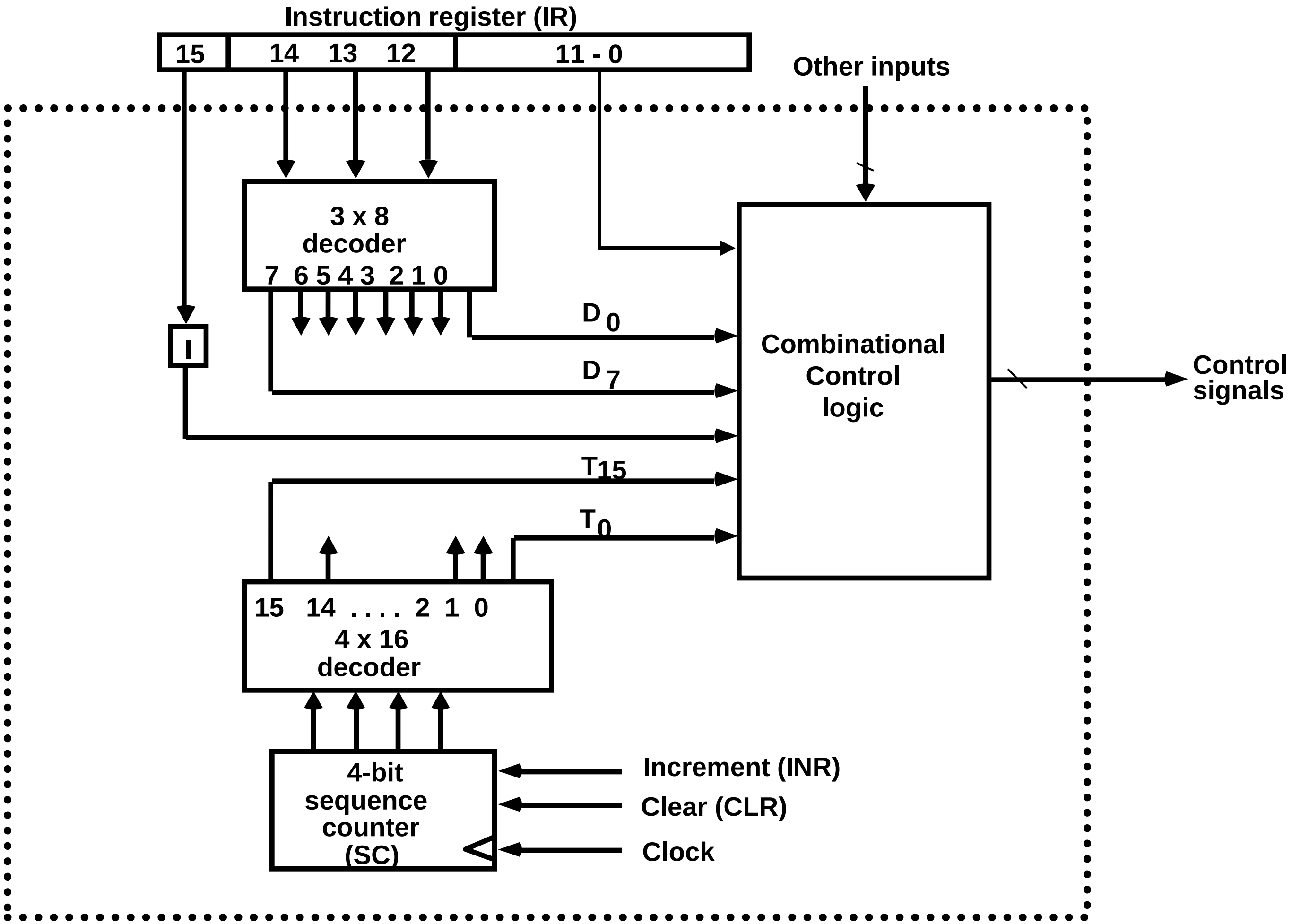
  Microprogrammed Control

  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

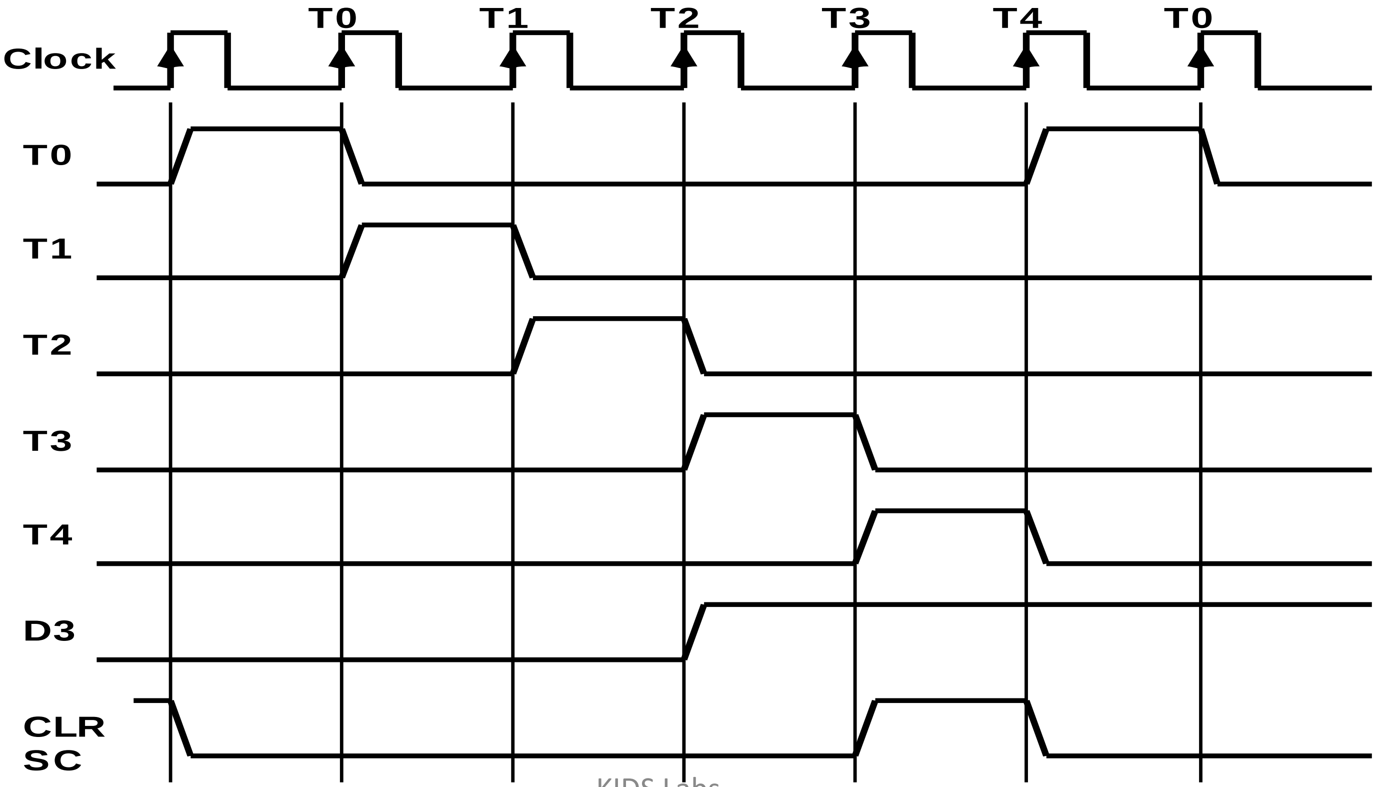


Timing Signals

- 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, . . .
  Assume: At time T_4, SC is cleared to 0 if decoder output D3 is active.

D_3T_4: SC ← 0



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

Initially PC loaded with address of first instruction and Sequence counter cleared to 0, giving timing signal T_0

T0: AR ←PC

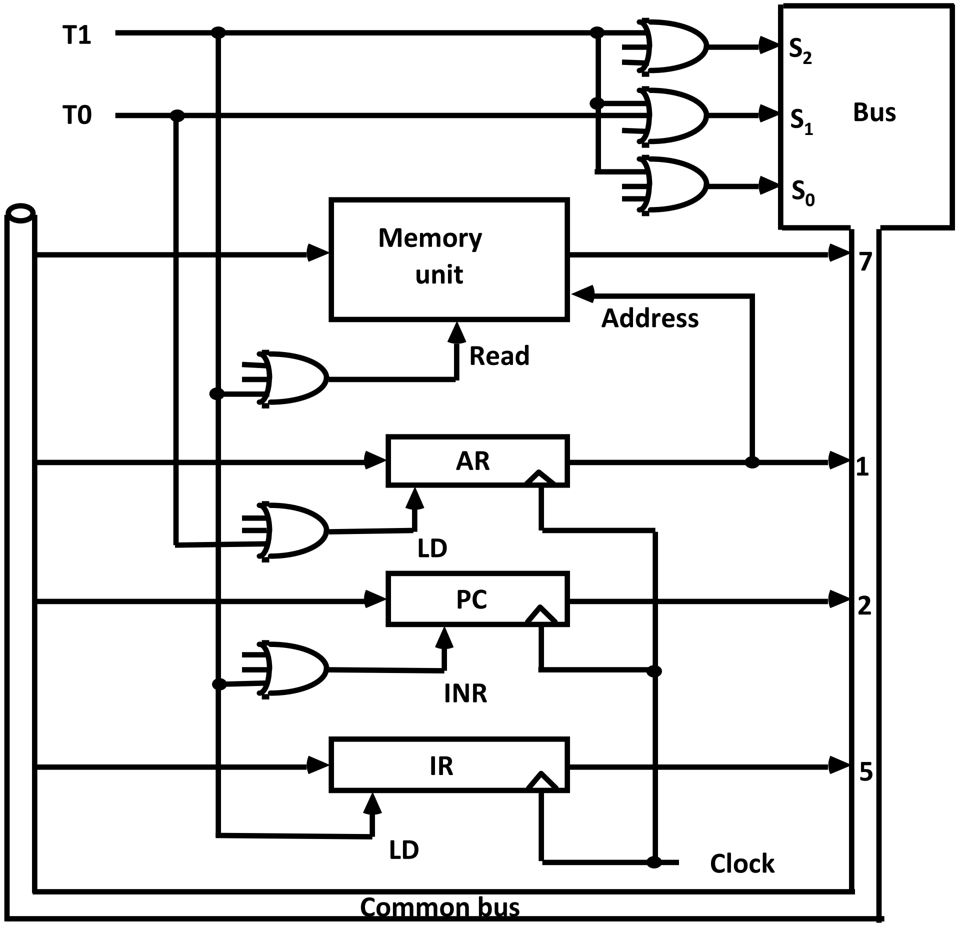
T1: IR ← M [AR], PC ← PC + 1

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

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)



Fetch and Decode

- Figure shows how first two statements are implemented in bus system

- At T_0 :
    - 1. Place the content of PC into bus by making $S_2S_1S_0$=010
    - Transfer the content of bus to AR by enabling the LD input of AR

- At T_1 :
    - 1. Enable read input of memory
    - 2. Place content of bus by making S2S1S0=111
    - 3. Transfer content of bus to IR by enabling the LD input of IR
    - 4. Increment PC by enabling the INR input of PC

Determine the Type of Instructions

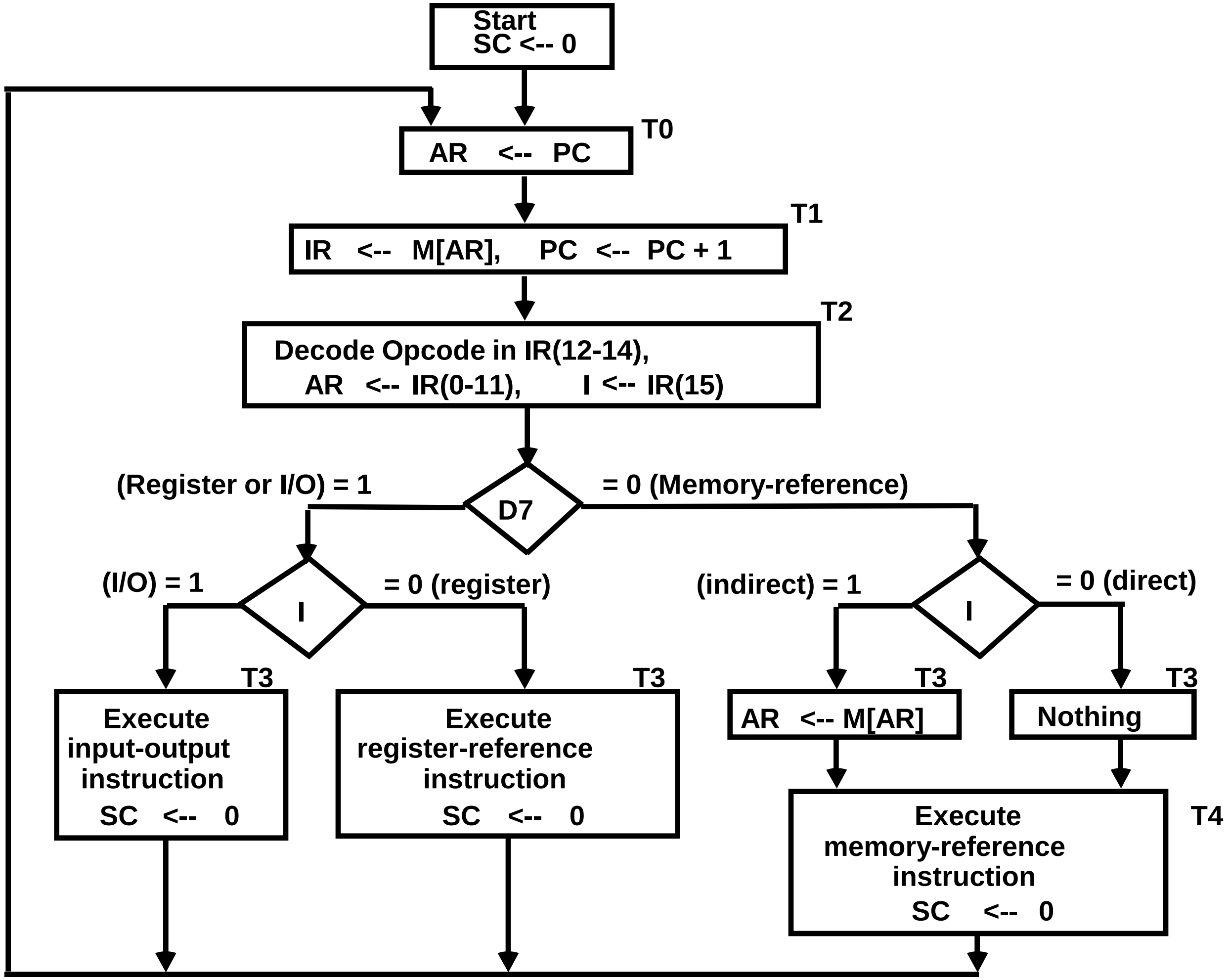

Start
SC <-- 0
T0
AR <-- PC
T1
IR <-- M[AR], PC <-- PC + 1
T2
Decode Opcode in IR(12-14),
AR <-- IR(0-11), I <-- IR(15)
(Register or I/O) = 1
D7
= 0 (Memory-reference)
(I/O) = 1
I
= 0 (register)
(indirect) = 1
I
= 0 (direct)
T3
Execute input-output instruction
SC <-- 0
T3
Execute register-reference instruction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

Fig : Flow chart for 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 ~ 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

e.g. rB_{11}=CLA

	r:	$SC \leftarrow 0$
CLA	rB_{11}:	$AC \leftarrow 0$
CLE	rB_{10}:	$E \leftarrow 0$
CMA	rB_9:	$AC \leftarrow AC'$
CME	rB_8:	$E \leftarrow E'$
CIR	rB_7:	$AC \leftarrow$ shr AC, $AC(15) \leftarrow E$, $E \leftarrow AC(0)$
CIL	rB_6:	$AC \leftarrow$ shl AC, $AC(0) \leftarrow E$, $E \leftarrow AC(15)$
INC	rB_5:	$AC \leftarrow AC + 1$
SPA	rB_4:	if $(AC(15) = 0)$ then $(PC \leftarrow PC+1)$
SNA	rB_3:	if $(AC(15) = 1)$ then $(PC \leftarrow PC+1)$
SZA	rB_2:	if $(AC = 0)$ then $(PC \leftarrow PC+1)$
SZE	rB_1:	if $(E = 0)$ then $(PC \leftarrow PC+1)$
HLT	rB_0:	$S \leftarrow 0$ (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 //performs AND logic with AC and memory word specified by EA

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

Memory Reference Instructions

ADD to AC // add content of memory word specified by EA to value of AC sum is transferred to AC and Carry to E (Extended Accumulator)

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

LDA: Load to AC // Transfers memory word specified by memory address to AC

D_2T_4: DR ← M[AR]

D_2T_5: AC ← DR, SC ← 0

STA: Store AC // Stores the content of AC into memory specified by EA

D_3T_4: M[AR] ← AC, SC ← 0

BUN: Branch Unconditionally // Transfer program to instruction specified by EA

D_4T_4: PC ← AR, SC ← 0

Memory Reference Instructions

BSA: Branch and Save Return Address **// 1. stores address of next instruction in sequence (PC) into address specified by EA 2. EA+1 transfer to PC serve as 1st inst. In subroutine**

$$M[AR] \leftarrow PC, PC \leftarrow AR + 1$$

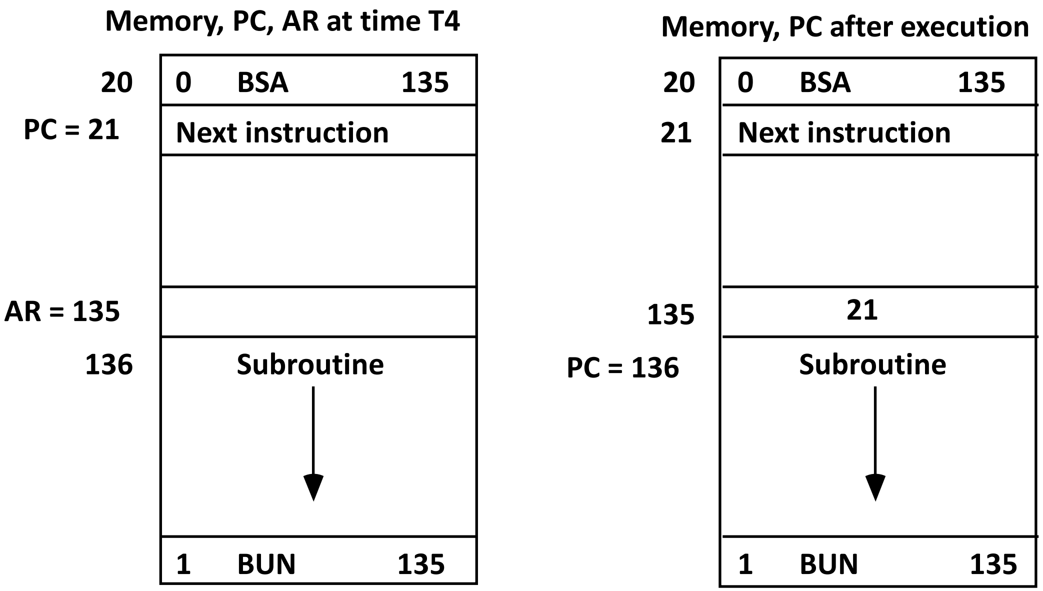
BSA:

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

D_5T_5: $PC \leftarrow AR, SC \leftarrow 0$

BSA: Example

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



Memory, PC, AR at time T4

20 | 0 BSA 135
PC = 21 | Next instruction
AR = 135 |
136 | Subroutine
| 1 BUN 135

Memory, PC after execution

20 | 0 BSA 135
21 | Next instruction
135 | 21
PC = 136 | Subroutine
| 1 BUN 135

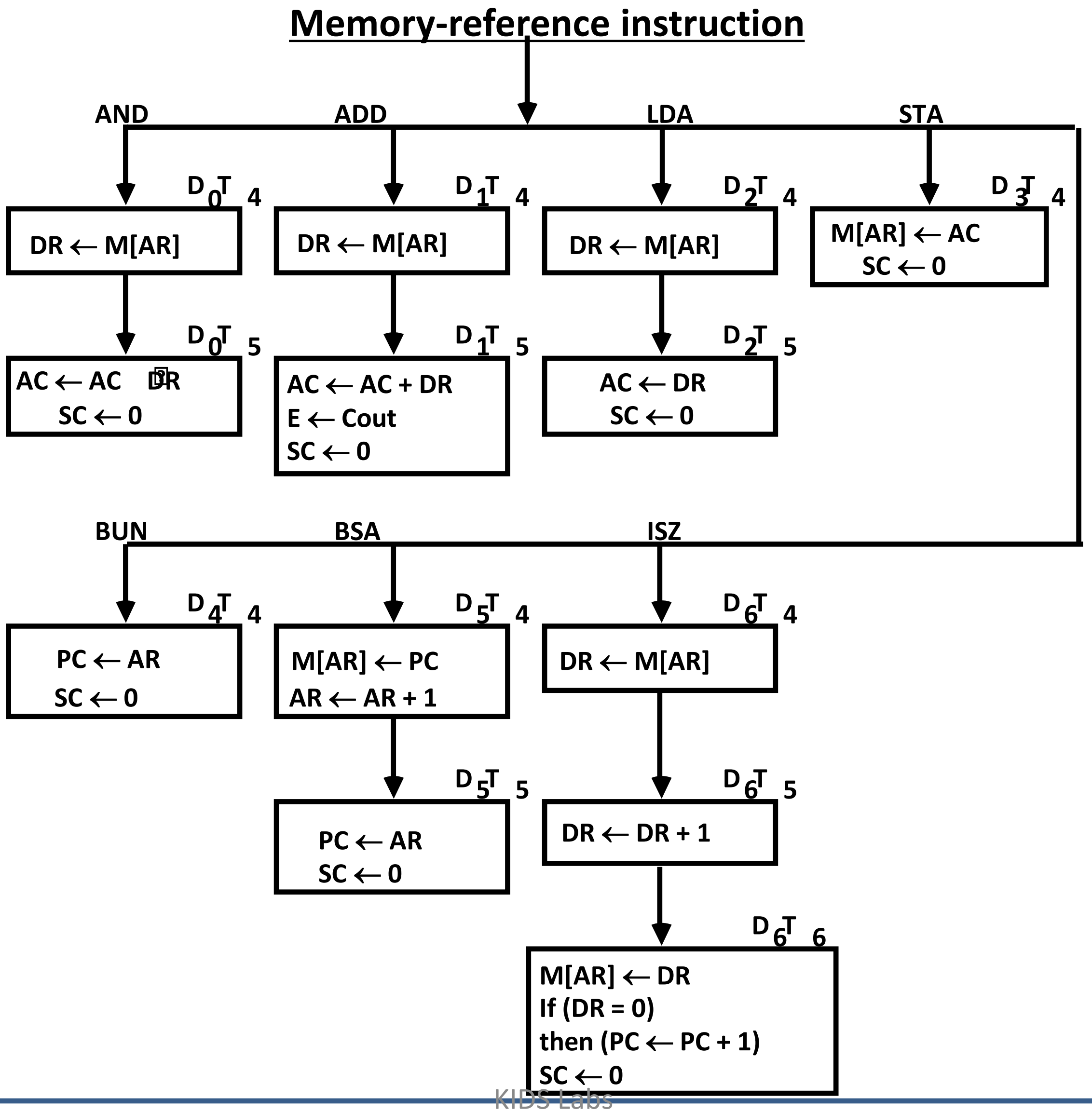
Memory Reference Instructions

ISZ: Increment and Skip-if-Zero

// increments the word specified by effective address, and if incremented value=0 , PC incremented by 1

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



Memory-reference instruction

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 ← Cout
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

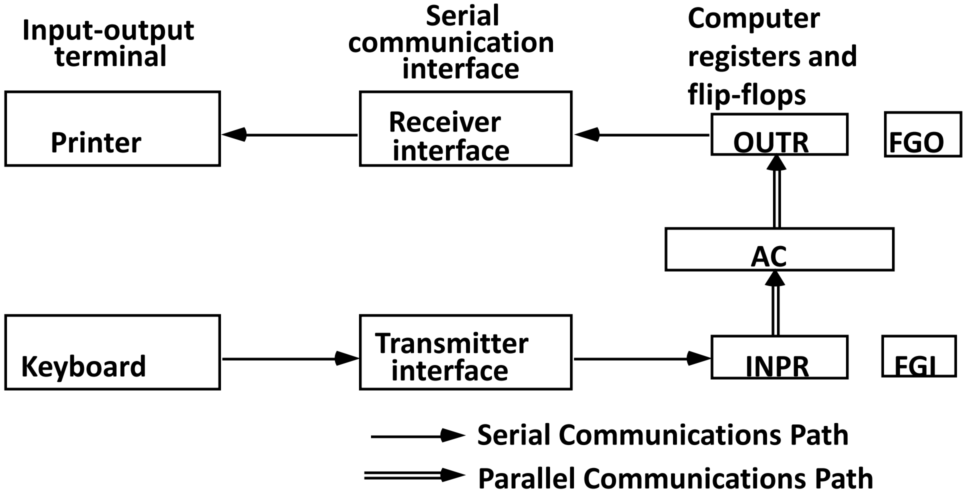
D_6T_6

M[AR] ← DR
If (DR = 0)
then (PC ← PC + 1)
SC ← 0

Input/Output and Interrupt

A Terminal with a keyboard and a Printer

Input-Output Configuration


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

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 communication interface serially and with the AC in parallel.
- The flags are needed to synchronize the timing difference between I/O device and the computer

Determining Type of Instruction

- **FGI =1 when new information available at input device, and cleared to 0 when information accepted by computer**
- **Initially FGI=0, new key pressed , 8 bit alphanumeric shifted to INPR and FGI=1, Computer checks flag if 1 then transfer content to AC and clear FGI to 0.**
- **Initially FGO=1,**
  - computer checks flag bit if 1, then OUTR ← AC and clears FGO=0
  - O/P device accepts information prints character and finally sets FGO=1.

Input/Output Instructions

I/O instructions are needed for transferring info to and from AC register, for checking the flag bits and for controlling interrupt facility

$D_7IT_3 = p$

$IR(i) = B_i$, $i = 6, \ldots, 11$

	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

Program controlled Input/Output

- **Program-controlled I/O**

  -**Continuous CPU involvement**

  CPU keeps checking flag bit. If 1 then initiates transfer

  I/O takes valuable CPU time

  -**Difference in information flow rate makes this type of transfer inefficient**

- **Alternative approach is to let external device inform the computer when it is ready for transfer, in meantime computer can be busy with other task**

- **Interrupt**

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

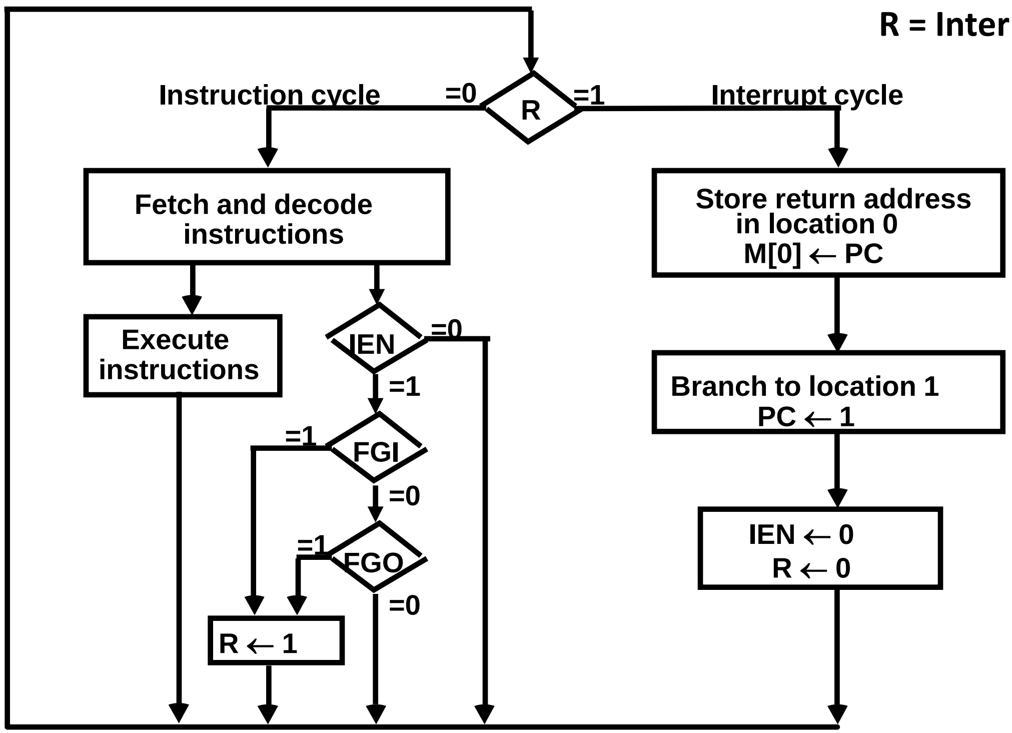
- Upon detecting an interrupt, the CPU stops momentarily the task it is doing, branches to the service routine to process the data transfer, and then returns to the task it was performing.

IEN (Interrupt-enable flip-flop)

- can be set and cleared by instructions

- **When cleared (IEN=0) the computer cannot be interrupted**
- **When set (IEN=1) the computer can be interrupted**

Flow Chart of Interrupt Cycle



R = Interrupt f/f

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

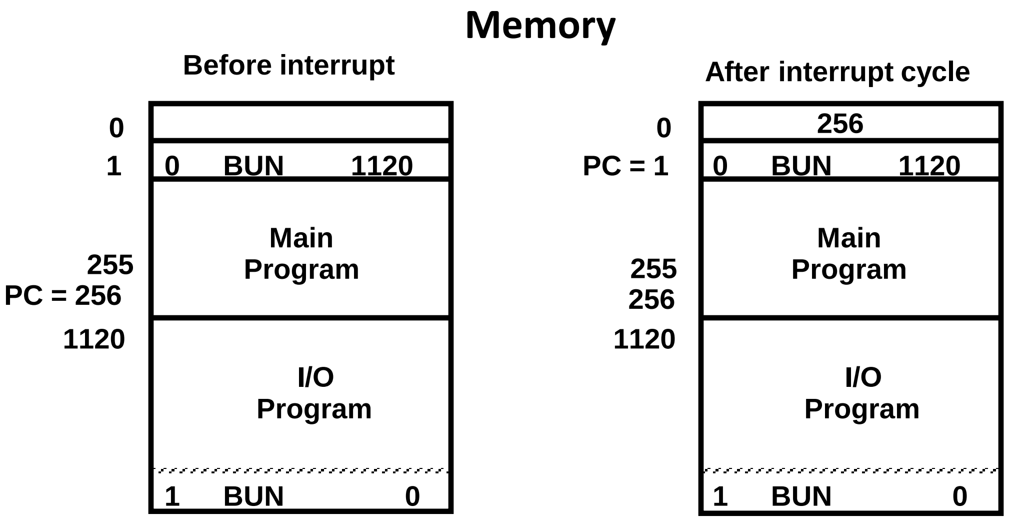
- The interrupt cycle is a HW implementation of a branch and save return address operation.

- 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 $R'T_0$, $R'T_1$, $R'T_2$

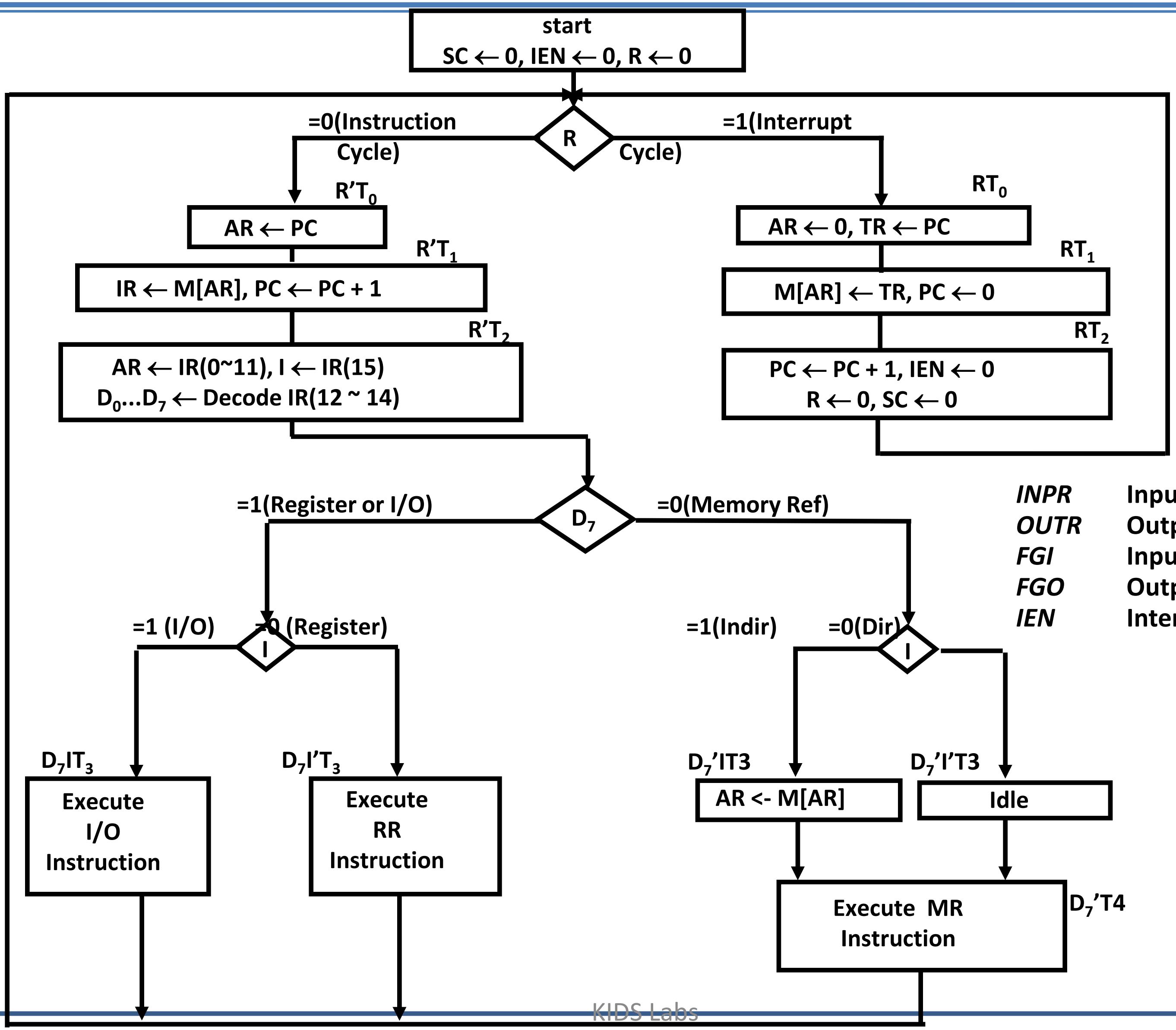
- The interrupt cycle :

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 Description



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

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	**D_7IT_3 = p**	**(Common to all input-output instructions)**
	IR(i) = B_i	**(i = 6,7,8,9,10,11)**
	p:	**SC ← 0**
INP	**pB_{11}:**	**AC(0-7) ← INPR, FGI ← 0**
OUT	**pB_{10}:**	**OUTR ← AC(0-7), FGO ← 0**
SKI	**pB_9:**	**If(FGI=1) then (PC ← PC + 1)**
SKO	**pB_8:**	**If(FGO=1) then (PC ← PC + 1)**
ION	**pB_7:**	**IEN ← 1**
IOF	**pB_6:**	**IEN ← 0**

Design of a Basic Computer(BC)

Hardware Components of BC

A memory unit: 4096 x 16.

Registers:

AR, PC, DR, AC, IR, TR, OUTR, INPR, and SC

Flip-Flops(Status):

I, E, R, IEN, FGI, and FGO

Decoders: a 3x8 Opcode decoder
a 4x16 timing decoder

Common bus: 16 bits

Control logic gates:

Adder and Logic circuit: Connected to AC

Control Logic Gates

- Input Controls of the nine registers

- Read and Write Controls of memory

- Set, Clear, or Complement Controls of the flip-flops

- S_2, S_1, S_0 Controls to select a register for the bus

- AC, and Adder and Logic circuit

Design of a Basic Computer(BC)

Register Control : AR

$AR \leftarrow ?$

- Control inputs of AR : ***LD, INR, CLR***
- ***Find all the statements that change the AR in Tab. 5-6***

$R'T_0 : AR \leftarrow PC$

$R'T_1 : AR \leftarrow IR(0-11)$

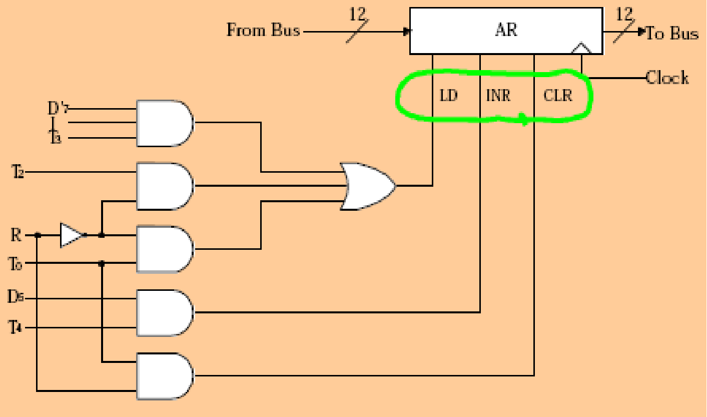
$D_7'IT_3 : AR \leftarrow M[AR]$

$RT_0 : AR \leftarrow 0$

$D_5T_4 : AR \leftarrow AR+1$

- Control functions

$$\begin{cases} LD(AR) = R'T_0 + R'T_1 + D_7'IT_3 \\ CLR(AR) = RT_0 \\ INR(AR) = D_5T_4 \end{cases}$$



Memory Control : READ

- Control inputs of Memory : ***READ, WRITE***

$M[AR] \leftarrow ?$

- Find all the statements that specify a ***read operation*** in Tab. 5-6

$? \leftarrow M[AR]$

- Control function

$$READ = R'T_1 + D_7'IT_3 + (D_0 + D_1 + D_2 + D_3)T_4$$

F/F Control : IEN

$IEN \leftarrow ?$

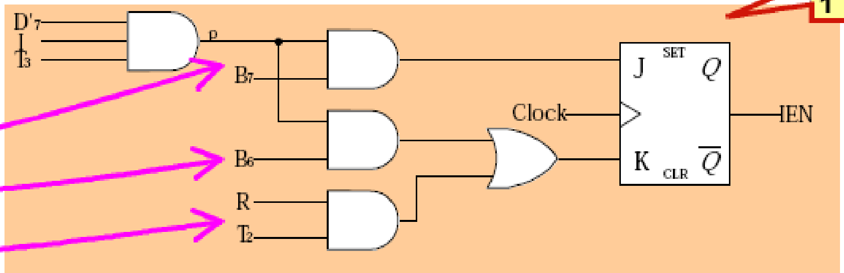
- Control functions

$pB_7 : IEN \leftarrow 1$

$pB_6 : IEN \leftarrow 0$

$RT_2 : IEN \leftarrow 0$

J	K	Q(t+1)
0	1	0
1	0	1



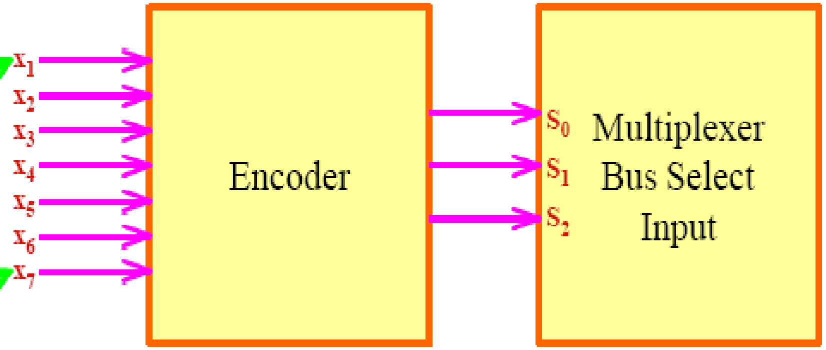
Design of a Basic Computer(BC)

Bus Control

- Encoder for Bus Selection :
  - » $S_0 = x_1 + x_3 + x_5 + x_7$
  - » $S_1 = x_2 + x_3 + x_6 + x_7$
  - » $S_0 = x_4 + x_5 + x_5 + x_7$
- $x_1 = 1$: $Bus \leftarrow AR = Find\ ? \leftarrow AR$
  - » $D_4T_4 : PC \leftarrow AR$
    $D_5T_5 : PC \leftarrow AR$
  - » Control Function : $x_1 = D_4T_4 + D_5T_5$
- $x_2 = 1$: $Bus \leftarrow PC = Find\ ? \leftarrow PC$

  “

  “
- $x_7 = 1$: $Bus \leftarrow Memory = Find\ ? \leftarrow M[AR]$
  - » Same as Memory Read
  - » Control Function : $x_7 = R'T_1 + D_7'IT_3 + (D_0 + D_1 + D_2 + D_3)T_4$


x_1
x_2
x_3
x_4
x_5
x_6
x_7
Encoder
S_0
S_1
S_2
Multiplexer
Bus Select
Input