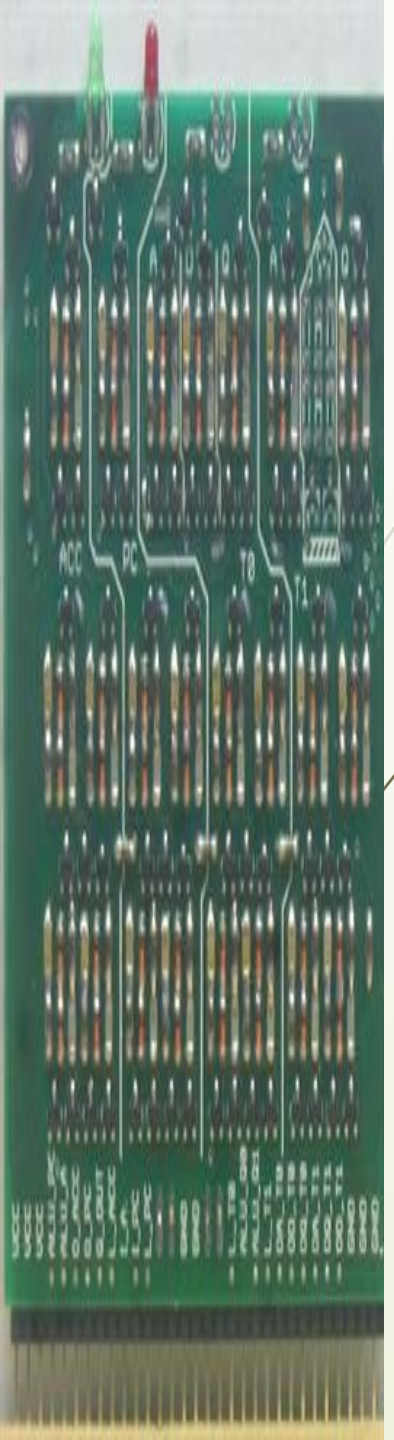
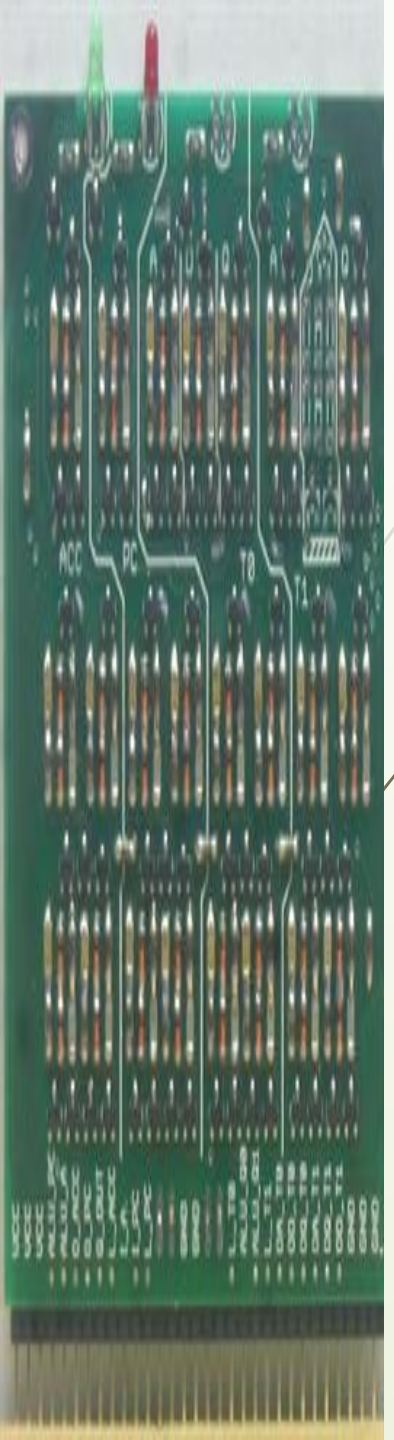
Computer Organization & Architecture UNIT 1:RTL

Register Transfer Language(RTL)



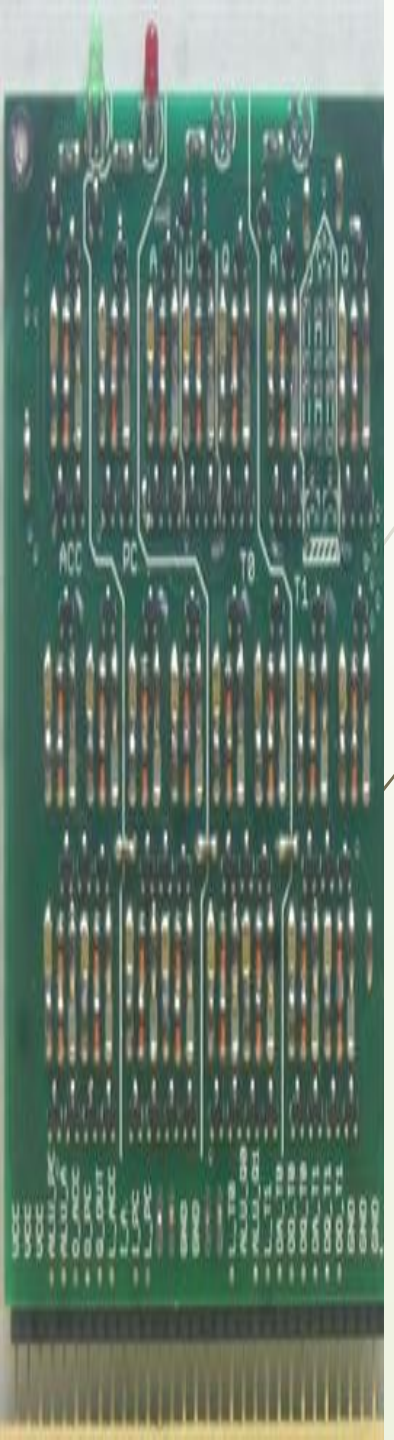
CONTENTS

- Register Transfer Language
- Register Transfer
- Bus and Memory Transfers
- Arithmetic Microoperations
- Logic Microoperations
- Shift Microoperations
- Arithmetic Logic Shift Unit



Registers

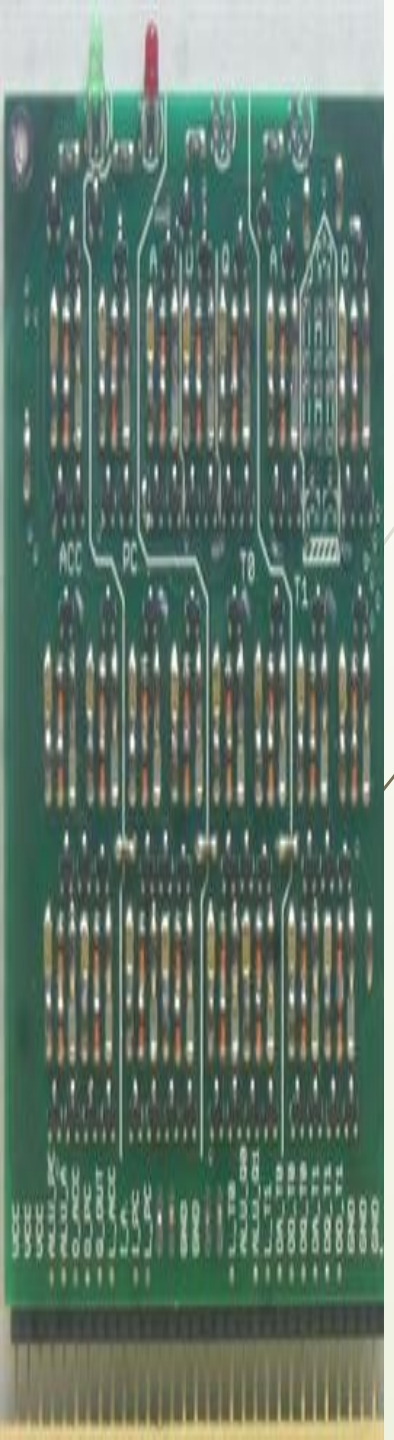
- Registers are a type of computer memory used to quickly accept, store, and transfer data and instructions that are being used immediately by the CPU. The registers used by the CPU are often termed as Processor registers.
- A processor register may hold an instruction, a storage address, or any data (such as bit sequence or individual characters).
- The computer needs processor registers for manipulating data and a register for holding a memory address. The register holding the memory location is used to calculate the address of the next instruction after the execution of the current instruction is completed.



Types of Registers

Following is the list of some of the most common registers used in a basic computer:

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



Descriptions of Registers

1. **Main Memory Unit (Registers) –Accumulator:** Stores the results of calculations made by ALU. This is the most common register, used to store data taken out from the memory.
2. **Special Purpose Registers**: Users do not access these registers. These registers are for Computer system such as MAR,MBR,PC, IR etc.

i) Program Counter (PC): Keeps track of the memory location of the next instructions to be dealt with. The PC then passes this next address to Memory Address Register (MAR).

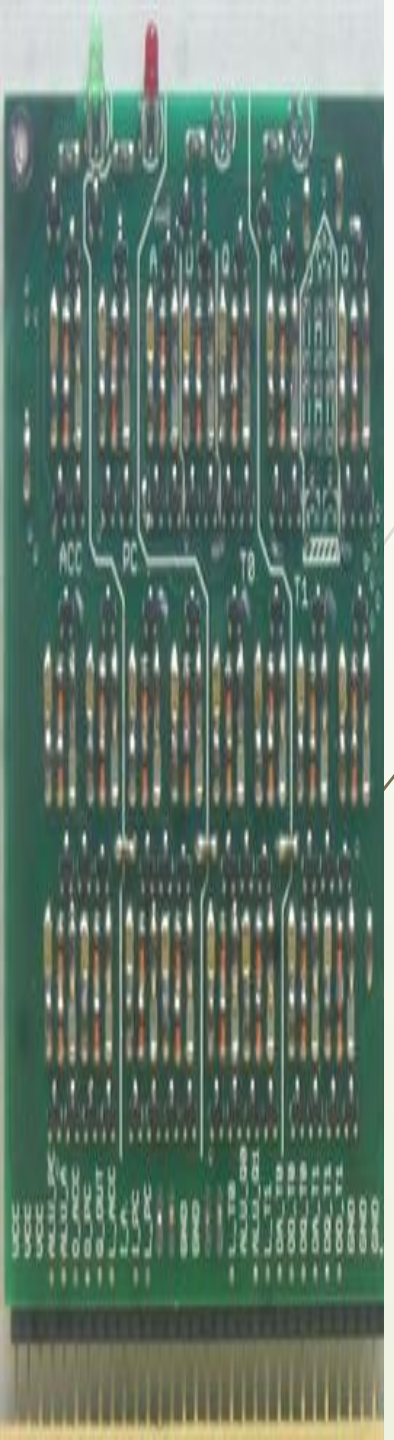
ii) Memory Address Register (MAR): It stores the memory locations of instructions that need to be fetched from memory or stored into memory.

iii) Memory Data Register (MDR): It stores instructions fetched from memory or any data that is to be transferred to, and stored in, memory.

iv) Current Instruction Register (CIR): It stores the most recently fetched instructions while it is waiting to be coded and executed.

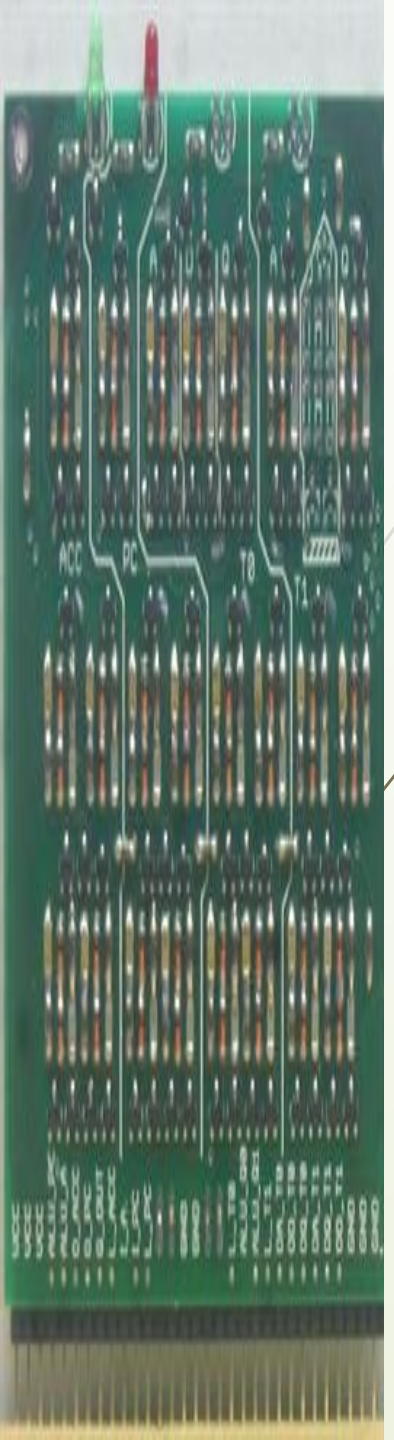
v) Instruction Buffer Register (IBR): The instruction that is not to be executed immediately is placed in the instruction buffer register IBR.

3. General Purpose Registers: This is used to store data intermediate results during program execution. It can be accessed via assembly programming.



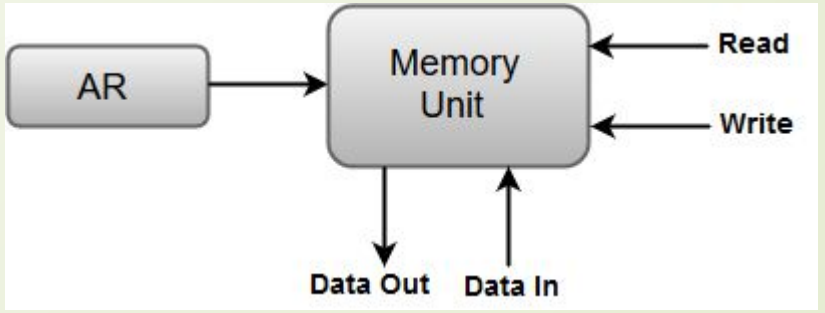
Memory Transfer

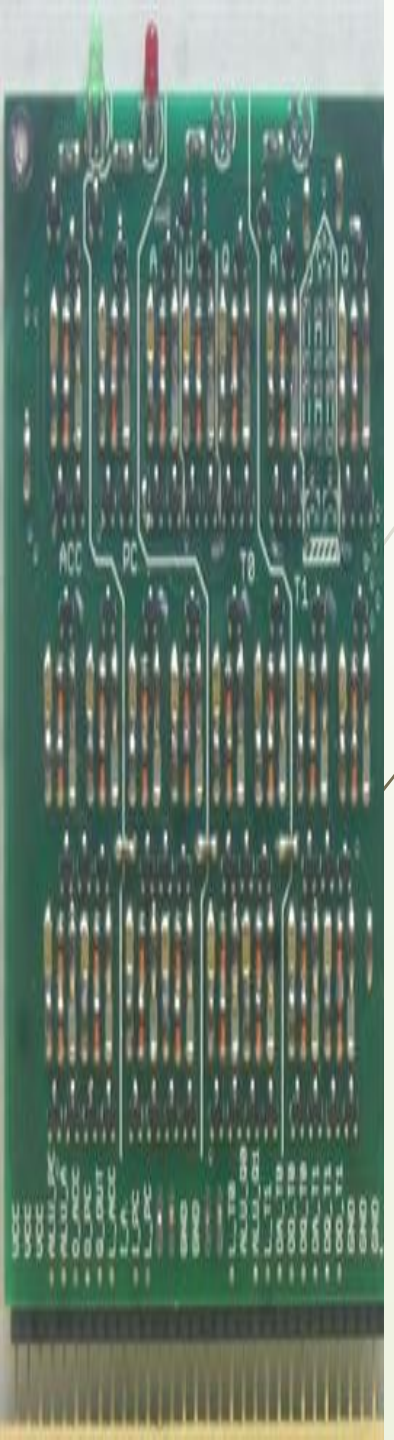
- Most of the standard notations used for specifying operations on memory transfer are stated below.
  - The transfer of information from a memory unit to the user end is called a **Read** operation.
  - The transfer of new information to be stored in the memory is called a **Write** operation.
  - A memory word is designated by the letter **M**.
  - We must specify the address of memory word while writing the memory transfer operations.
  - The **address register** is designated by **AR** and the **data register** by **DR**.
  - Thus, a read operation can be stated as:



Read/Write Operation

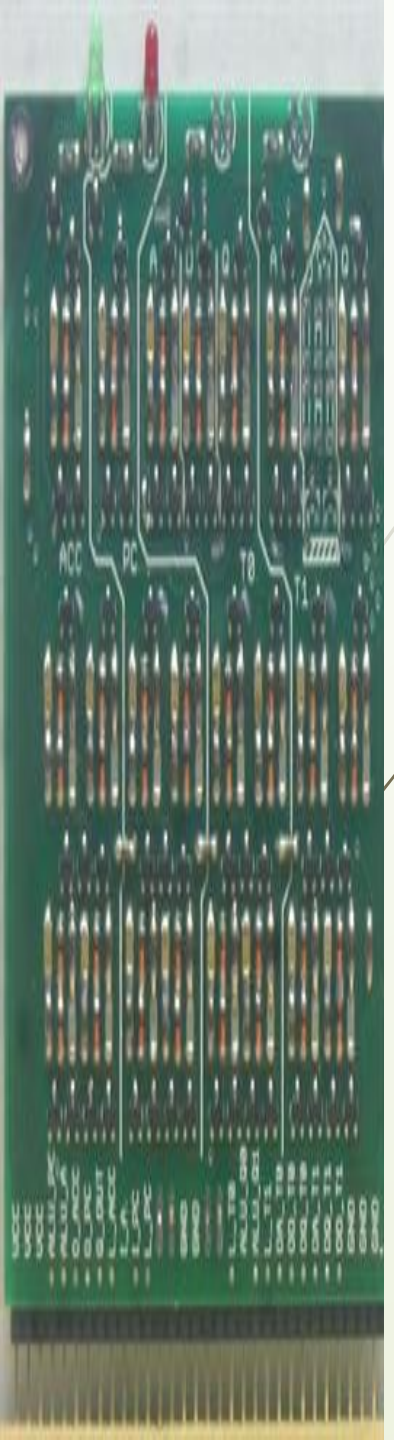
- Read: DR ← M [AR]
  - The **Read** statement causes a transfer of information into the data register (DR) from the memory word (M) selected by the address register (AR).
  - And the corresponding write operation can be stated as:
- Write: M [AR] ← R1
  - The Write statement causes a transfer of information from register R1 into the memory word (M) selected by address register (AR).





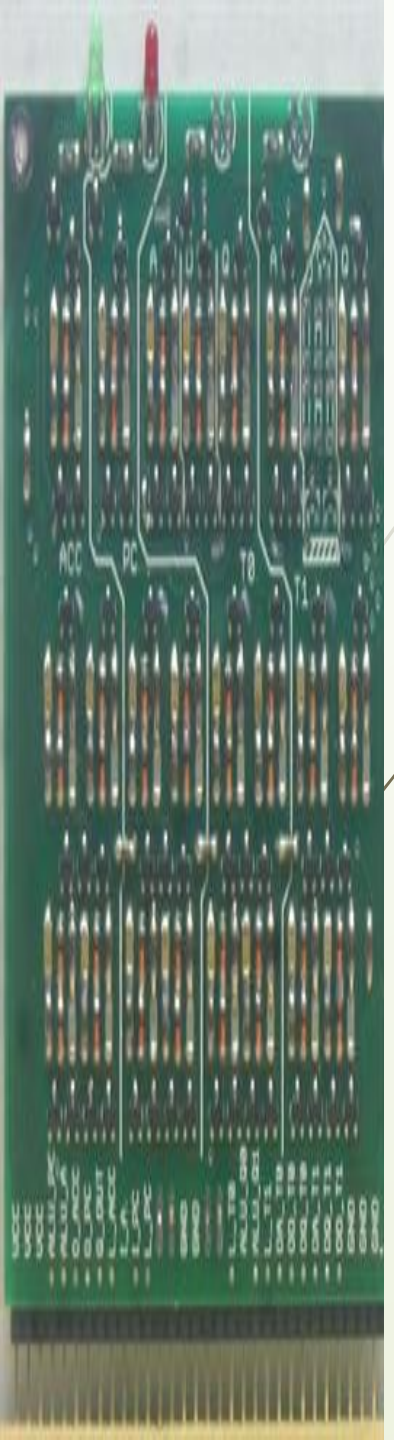
Register Transfer Language

- The symbolic notation used to describe the micro-operation transfers amongst registers is called **Register transfer language**.
- The term **register transfer** means the availability of **hardware logic circuits** that can perform a stated micro-operation and transfer the result of the operation to the same or another register.
- The word **language** is borrowed from programmers who apply this term to programming languages. This programming language is a procedure for writing symbols to specify a given computational process.
- The operations executed on data stored in registers are called microoperations.



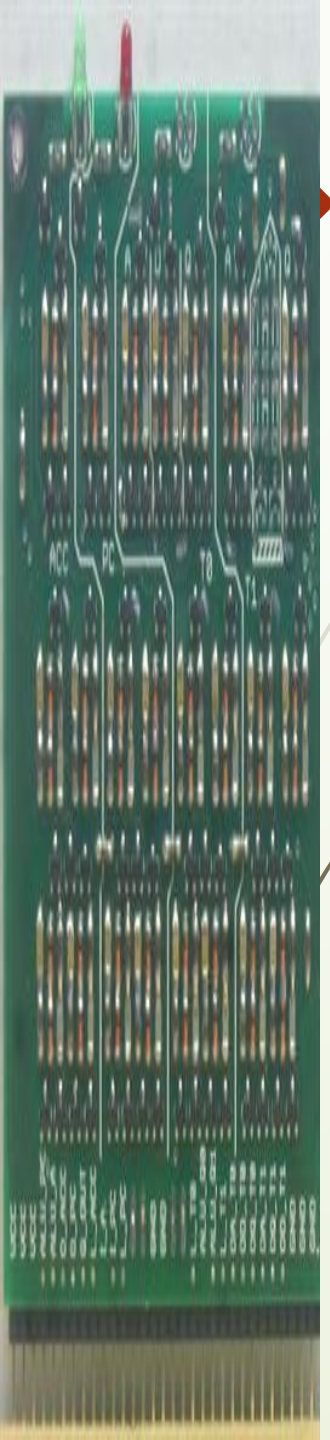
Register Transfer

- The term Register Transfer refers to the availability of hardware logic circuits that can perform a given micro-operation and transfer the result of the operation to the same or another register. Most of the standard notations used for specifying operations on various registers are stated below.
  - The memory address register is designated by **MAR**.
  - Program Counter **PC** holds the next instruction's address.
  - Instruction Register **IR** holds the instruction being executed.
  - **R1** (Processor Register).
  - We can also indicate individual bits by placing them in parenthesis. For instance, PC (8-15), R2 (5), etc.
  - Data Transfer from one register to another register is represented in symbolic form by means of replacement operator. For instance, the following statement denotes a transfer of the data of register R1 into register R2.



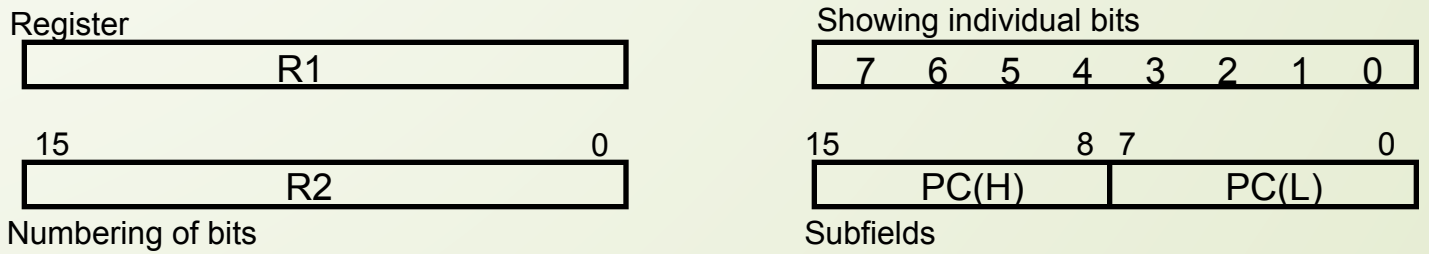
Contd..

- R2 ← R1
  - Typically, most of the users want the transfer to occur only in a predetermined control condition. This can be shown by following if-then statement: If (P=1) then (R2 ← R1); Here P is a control signal generated in the control section.
  - It is more convenient to specify a control function (P) by separating the control variables from the register transfer operation. For instance, the following statement defines the data transfer operation under a specific control function (P).

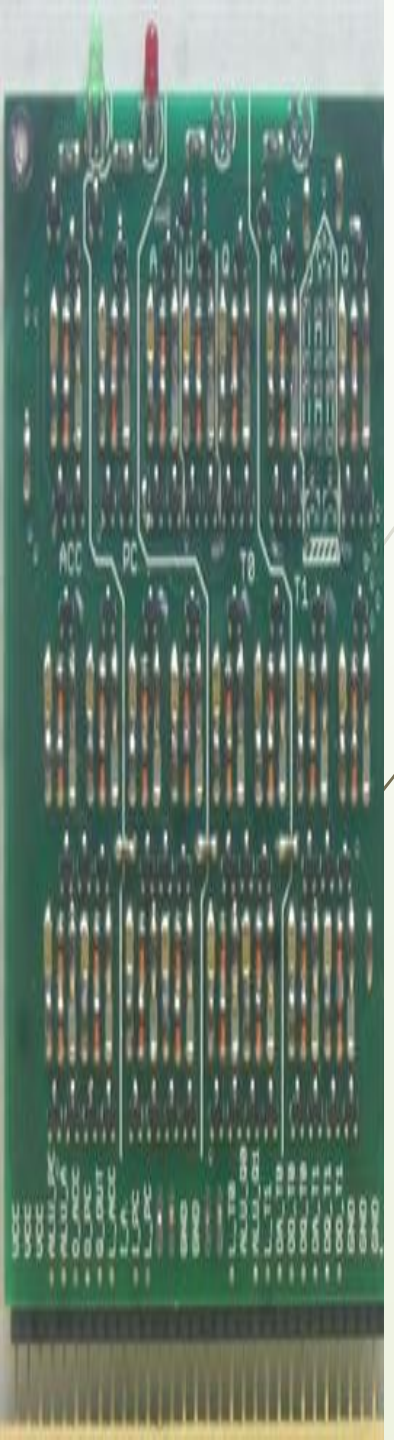


Notations of Registers

- Common ways of drawing the block diagram of a register



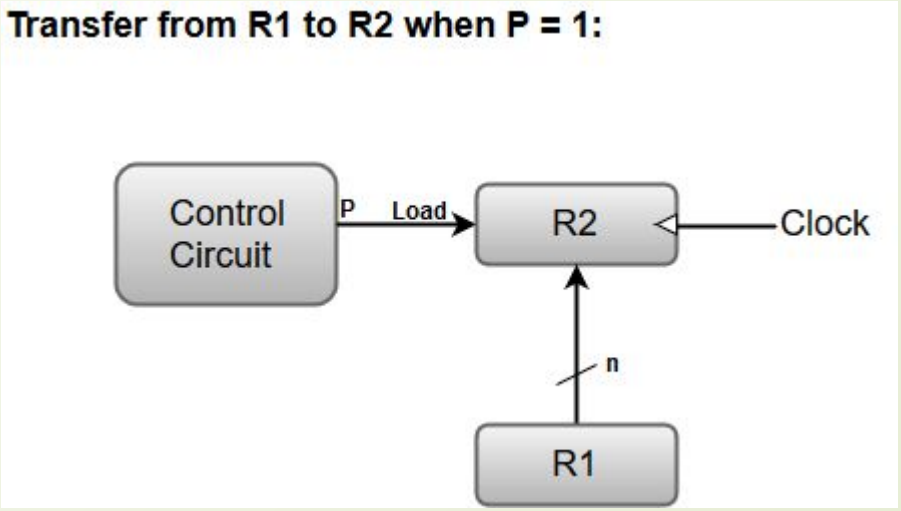
Divided into two parts

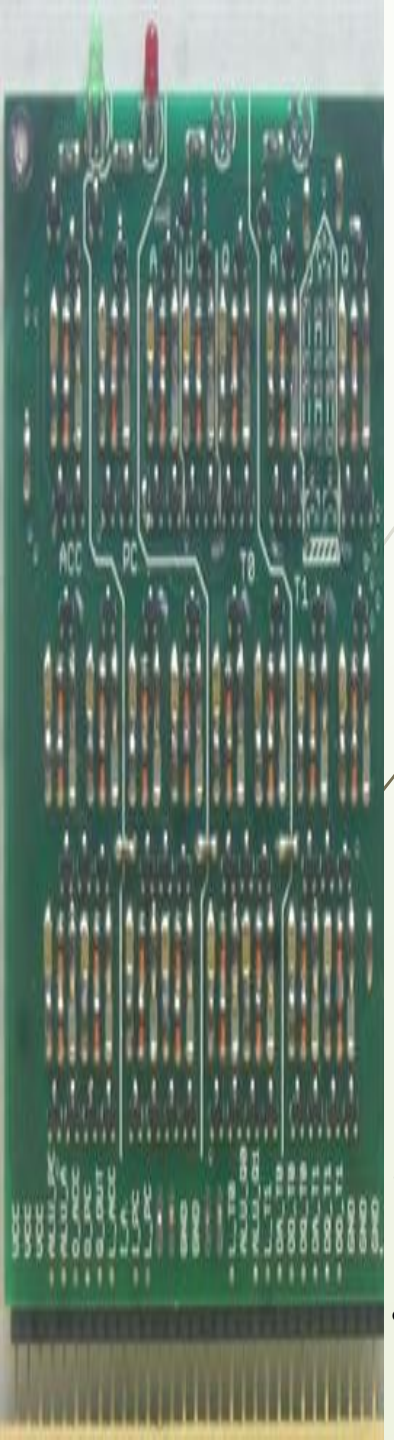


Ex.- Block Diagram for register Transfer

P: R2 ← R1

- The following image shows the block diagram that depicts the transfer of data from R1 to R2.



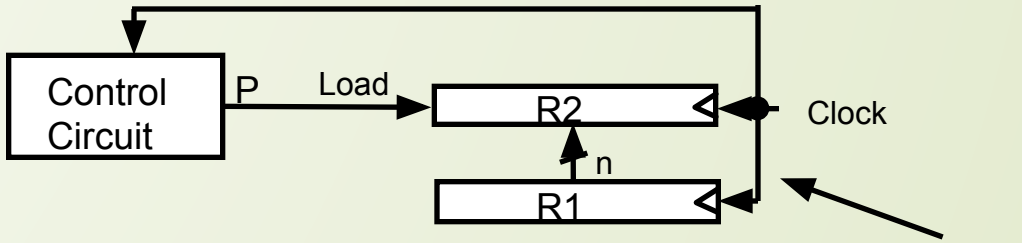


HARDWARE IMPLEMENTATION OF CONTROLLED TRANSFERS

Implementation of controlled transfer

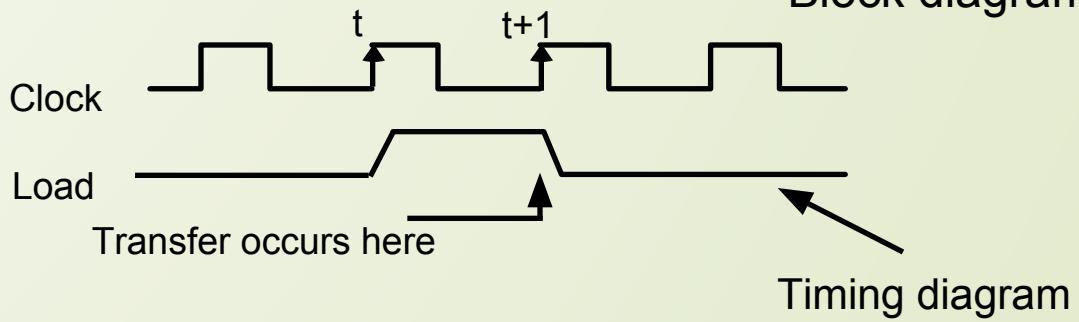
P: R2 ← R1

Block diagram



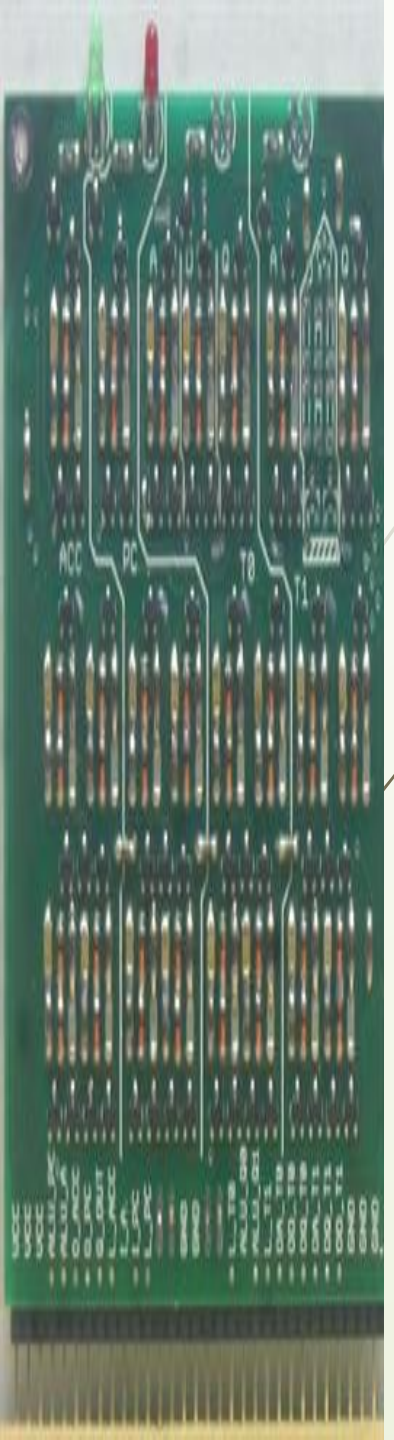
Block diagram

Timing diagram

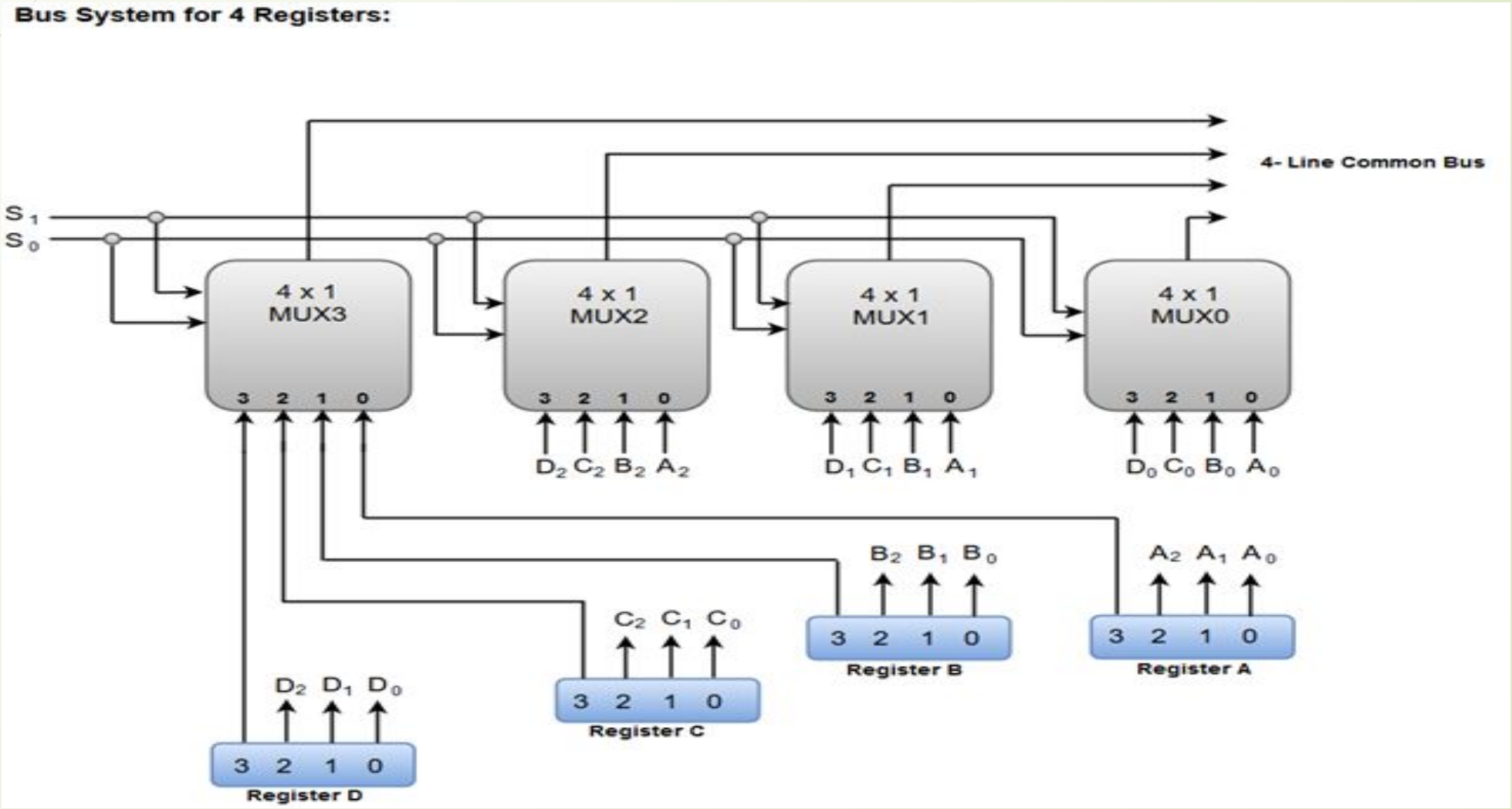


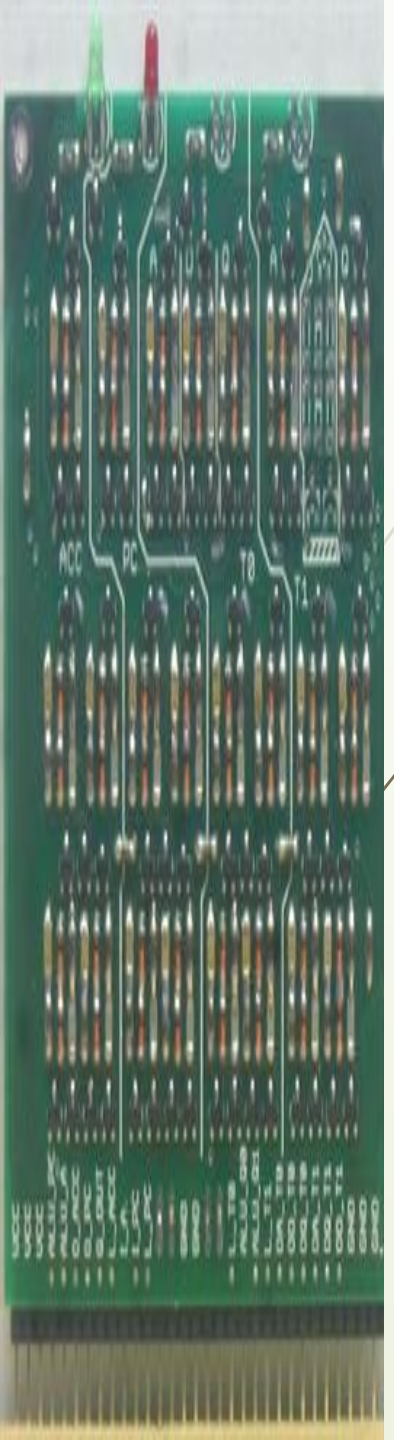
Timing diagram

- 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

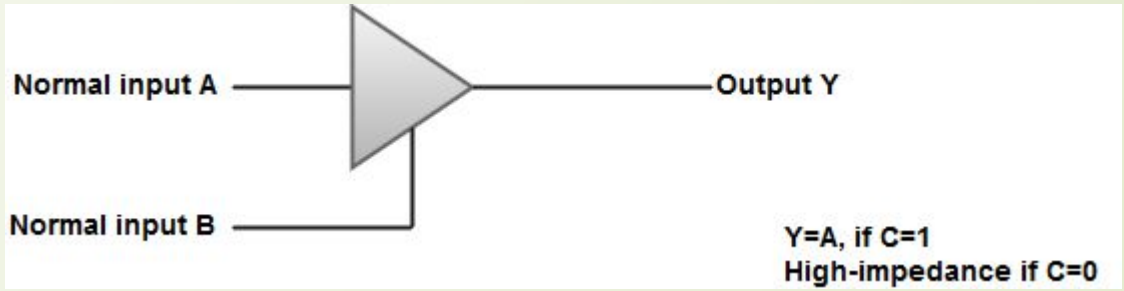


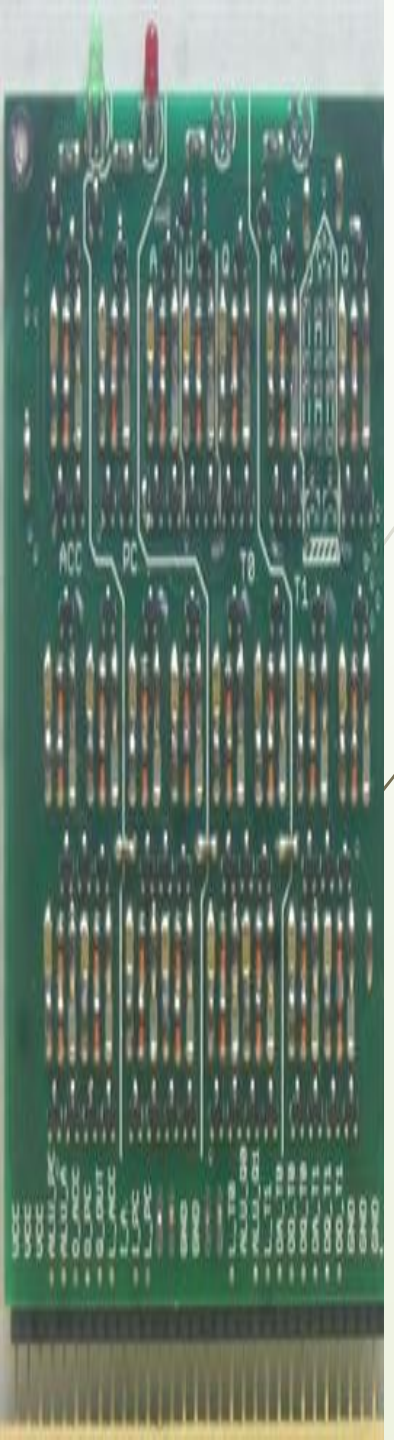
Bus Designing Using Multiplexer



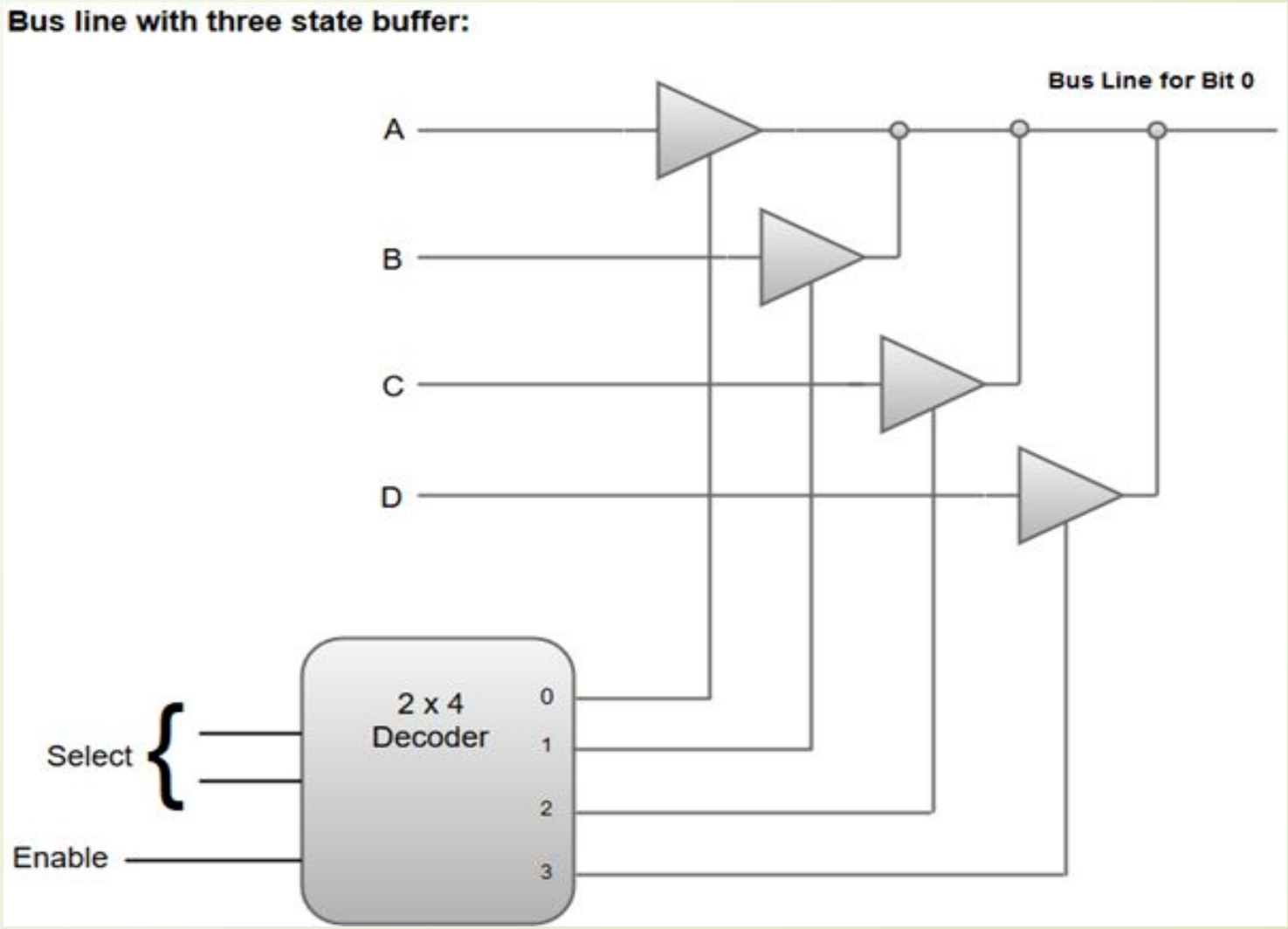


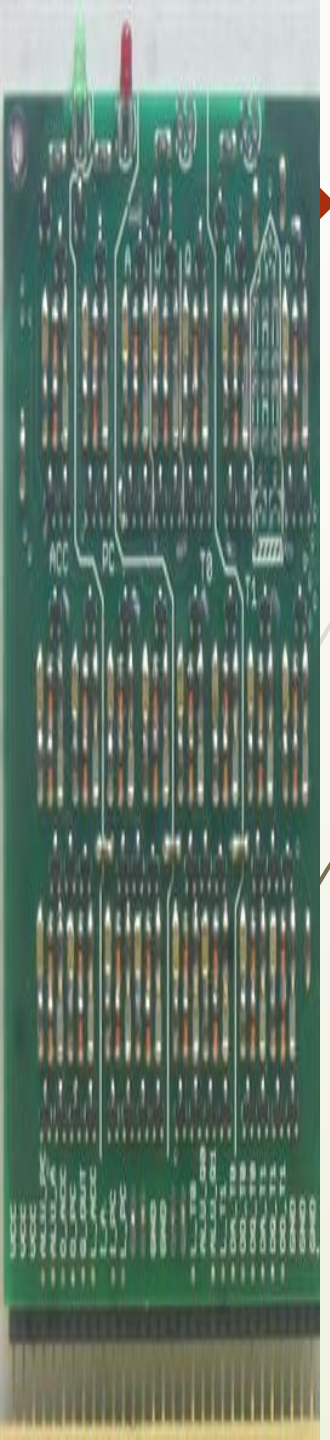
Three State Buffer





Bus Designing Using Three State Buffer

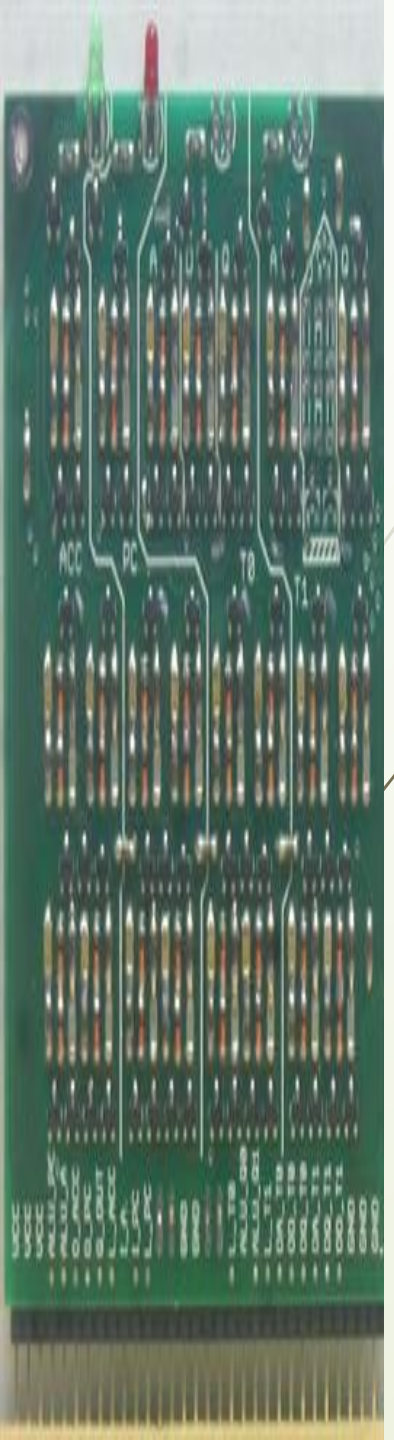




MICROOPERATIONS

- A microoperation is an elementary operation performed with data stored in registers.
- Computer system microoperations are usually of four types:

  - Register transfer microoperations
    - Transfer binary information from one register to another
  - Arithmetic microoperations
    - Performs arithmetic operations on numeric data stored in registers
  - Logic microoperations
    - Perform bit manipulation operations on nonnumeric data stored in registers
  - Shift microoperations
    - Performs shift operations on data stored in registers.



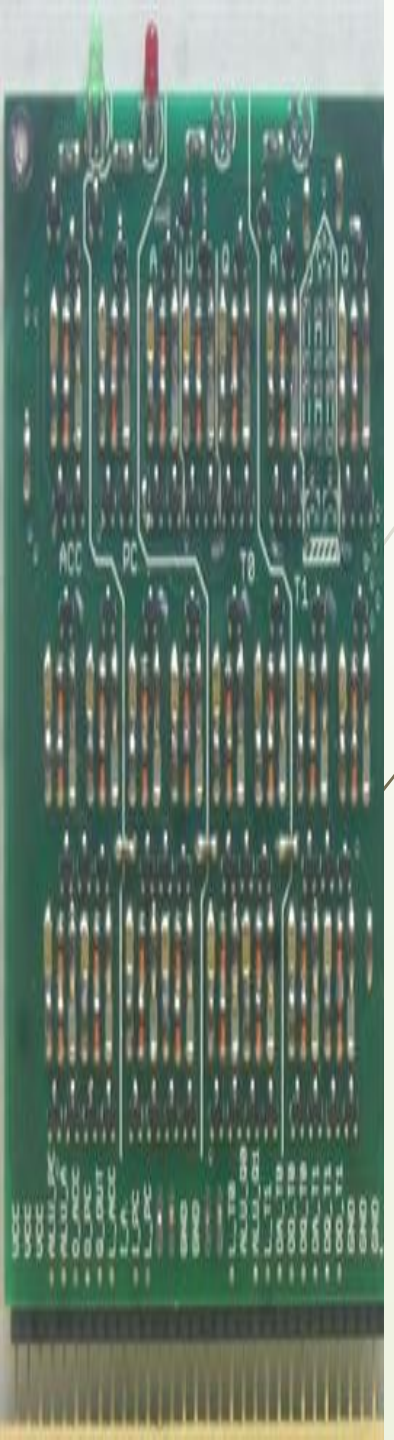
Arithmetic Micro operations

- In general, the Arithmetic Micro-operations deals with the operations performed on numeric data stored in the registers.
- The basic Arithmetic Micro-operations are classified in the following categories:

1. Addition
2. Subtraction
3. Increment
4. Decrement
5. Shift

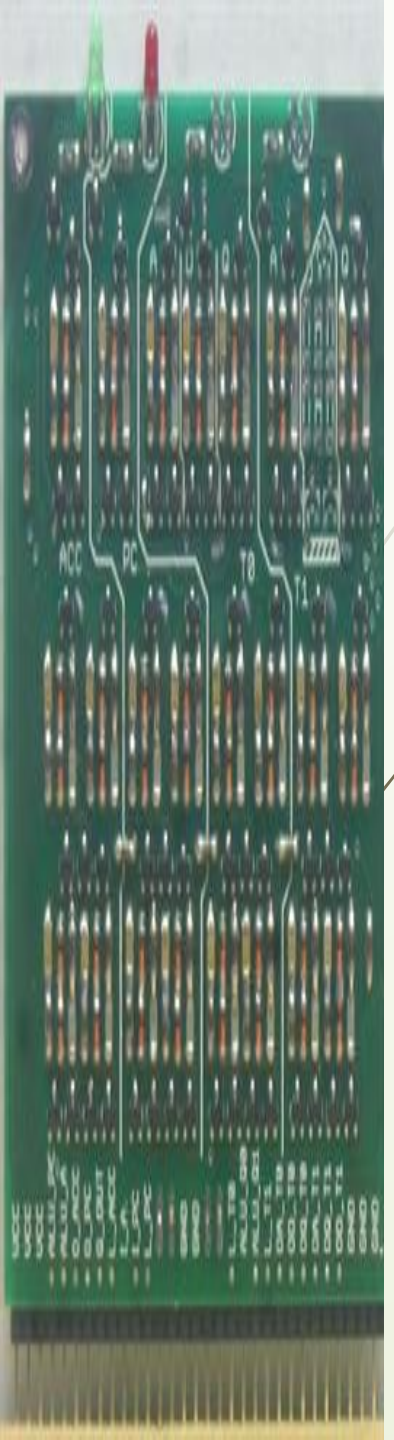
- Some additional Arithmetic Micro-operations are classified as:

1. Add with carry
2. Subtract with borrow
3. Transfer/Load, etc.

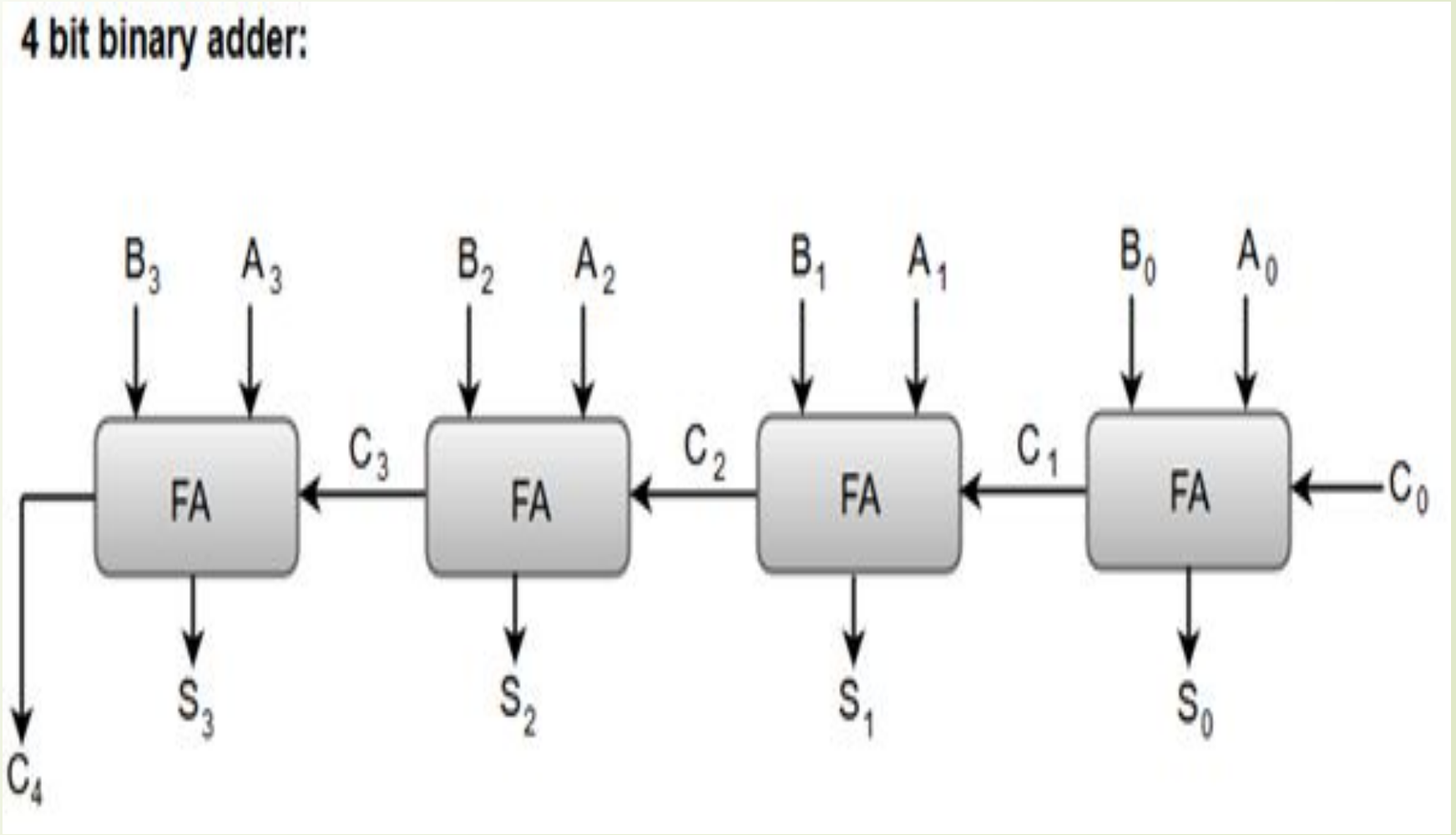


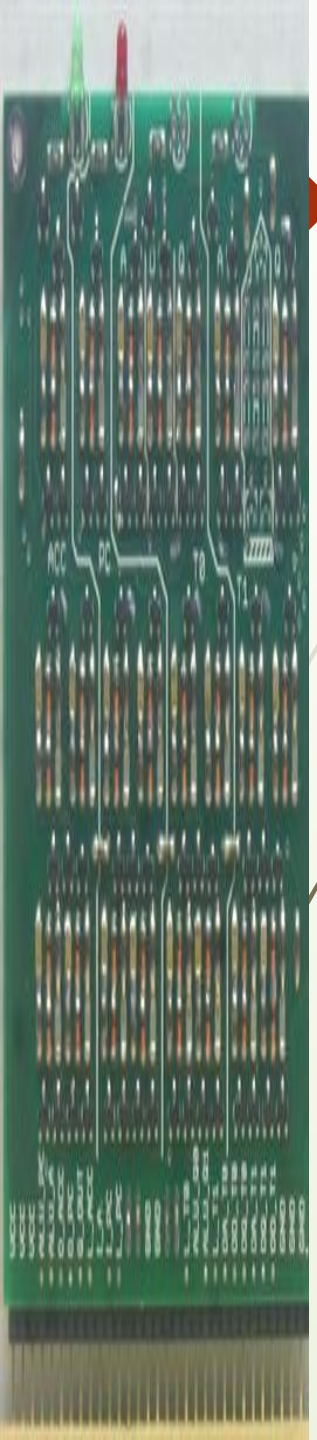
Arithmetic Micro-operations

Symbolic Representation	Description
R3 ← R1 + R2	The contents of R1 plus R2 are transferred to R3.
R3 ← R1 - R2	The contents of R1 minus R2 are transferred to R3.
R2 ← R2'	Complement the contents of R2 (1's complement)
R2 ← R2' + 1	2's complement the contents of R2 (negate)
R3 ← R1 + R2' + 1	R1 plus the 2's complement of R2 (subtraction)
R1 ← R1 + 1	Increment the contents of R1 by one
R1 ← R1 - 1	Decrement the contents of R1 by one



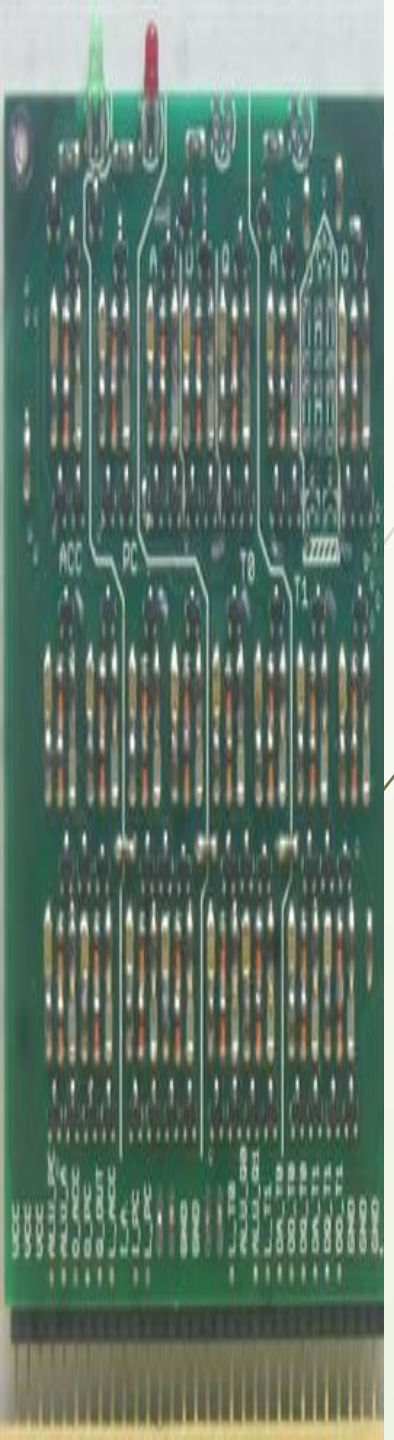
n Bit Binary Adder



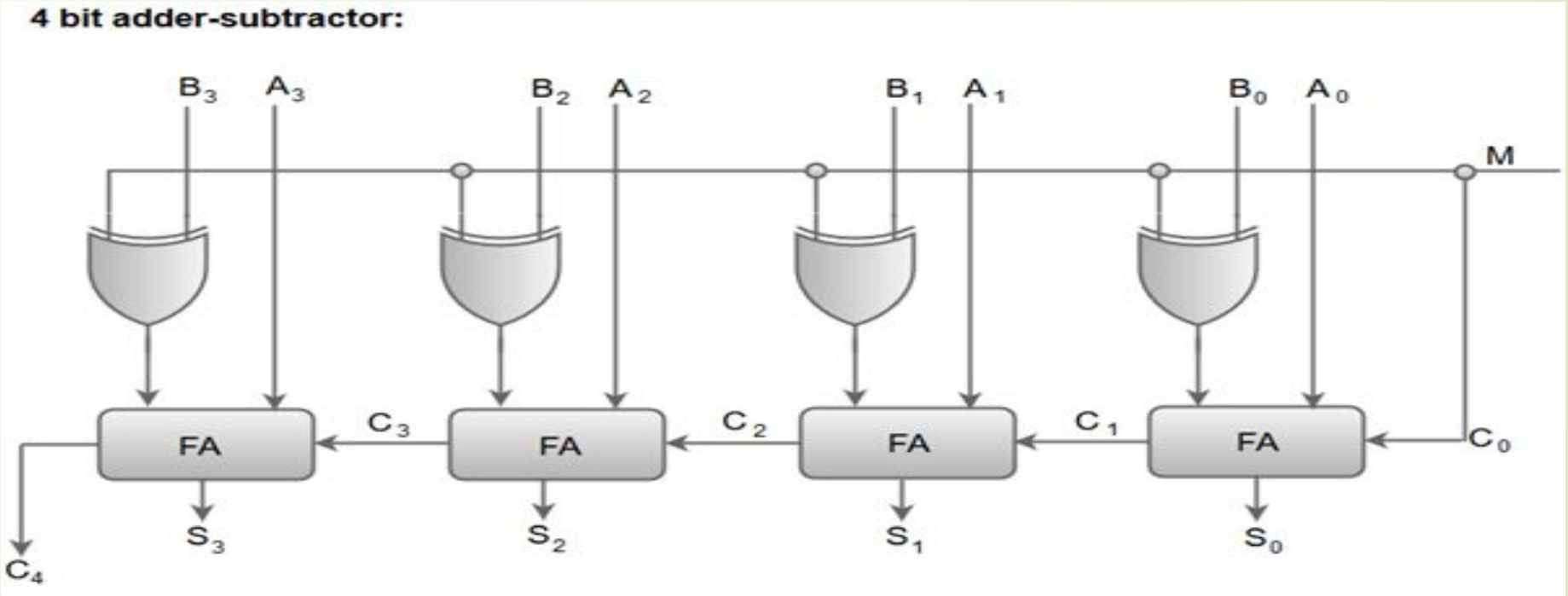


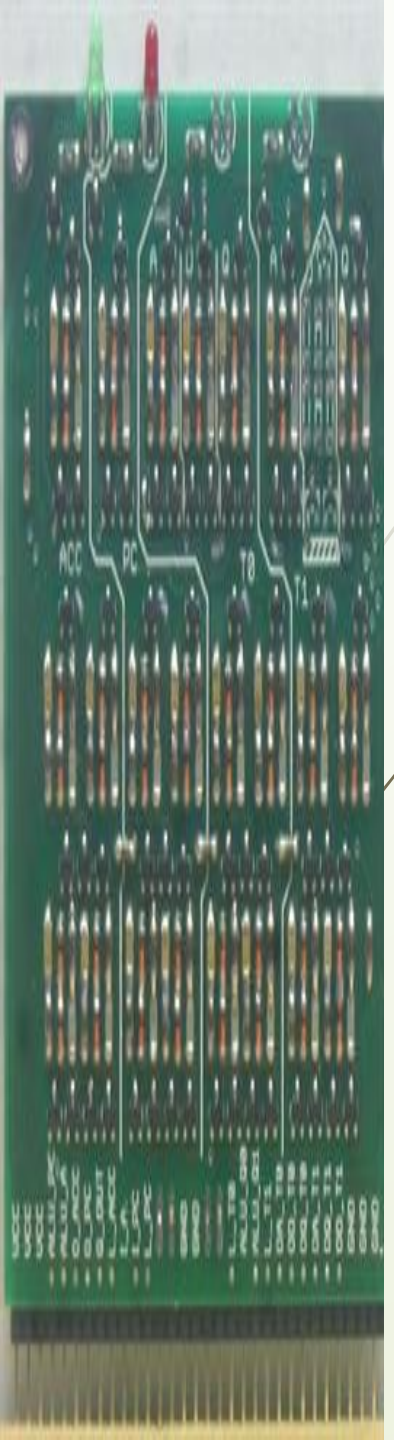
BINARY ADDER / SUBTRACTOR / INCREMENTER Cont. …

- Binary adder-Subtractor
  - If M = 0 , the circuit is adder and when M = 1, the circuit is a subtractor,
  - When M = 0, B = B ⊕ 0 = B, and the circuit performs A + B,
  - When M = 1, B = B ⊕ 1 = B′ and C_0 = 1, the circuit performs A plus 2's complement of B.
    - For unsigned number, this gives A – B if A ≥B or the 2's complement of (B – A) if A < B,
    - For signed numbers, the result is A – B provided there is no overflow.
- The increment microoperation adds one in a register and this can be implemented with a binary counter as sequential circuit.
  - Alternatively, this can be implemented as combination circuit through half adder
  - This approach can be extended to an n-bit binary incrementer by n-half adder

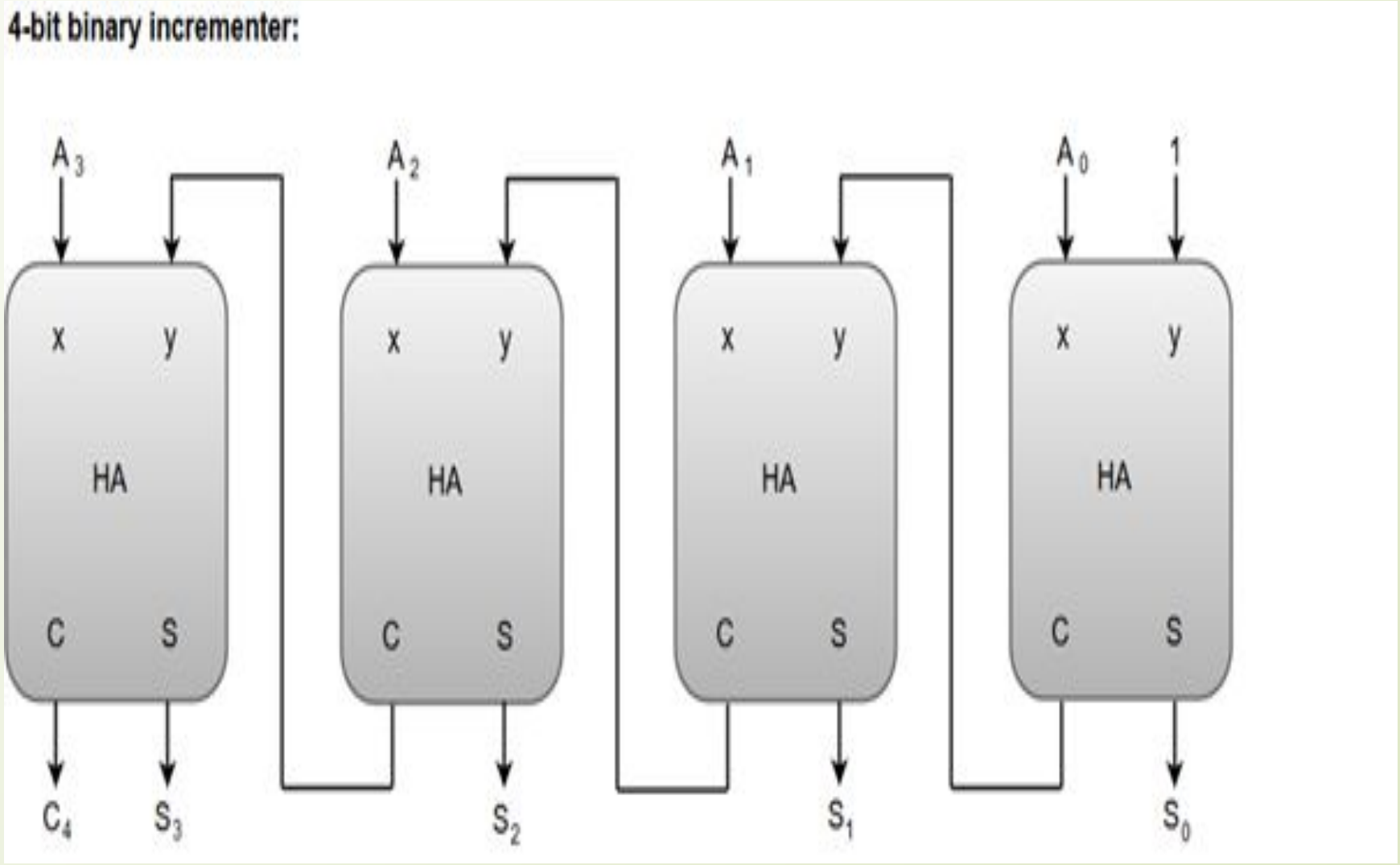


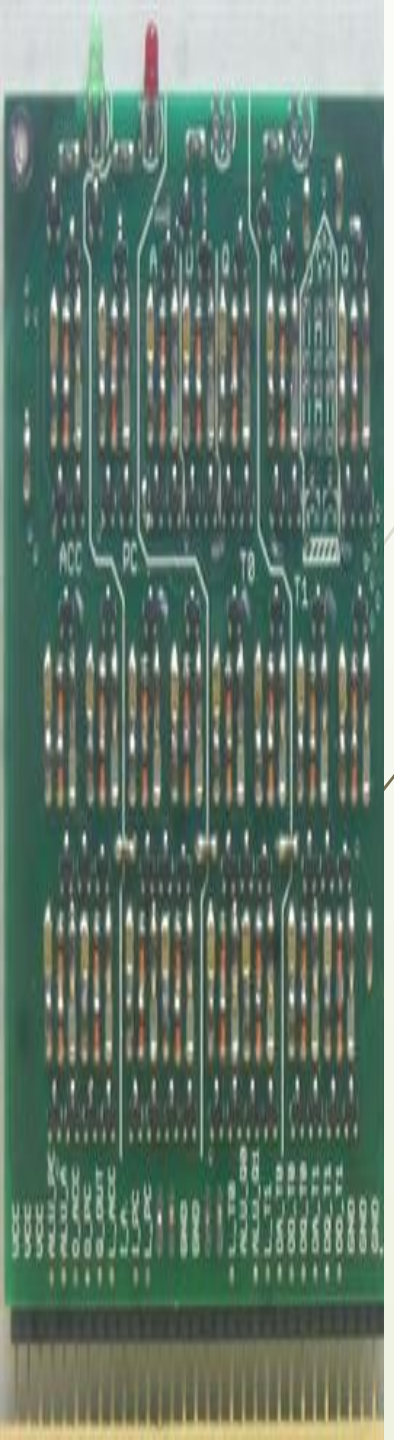
n Bit Adder Subtractor



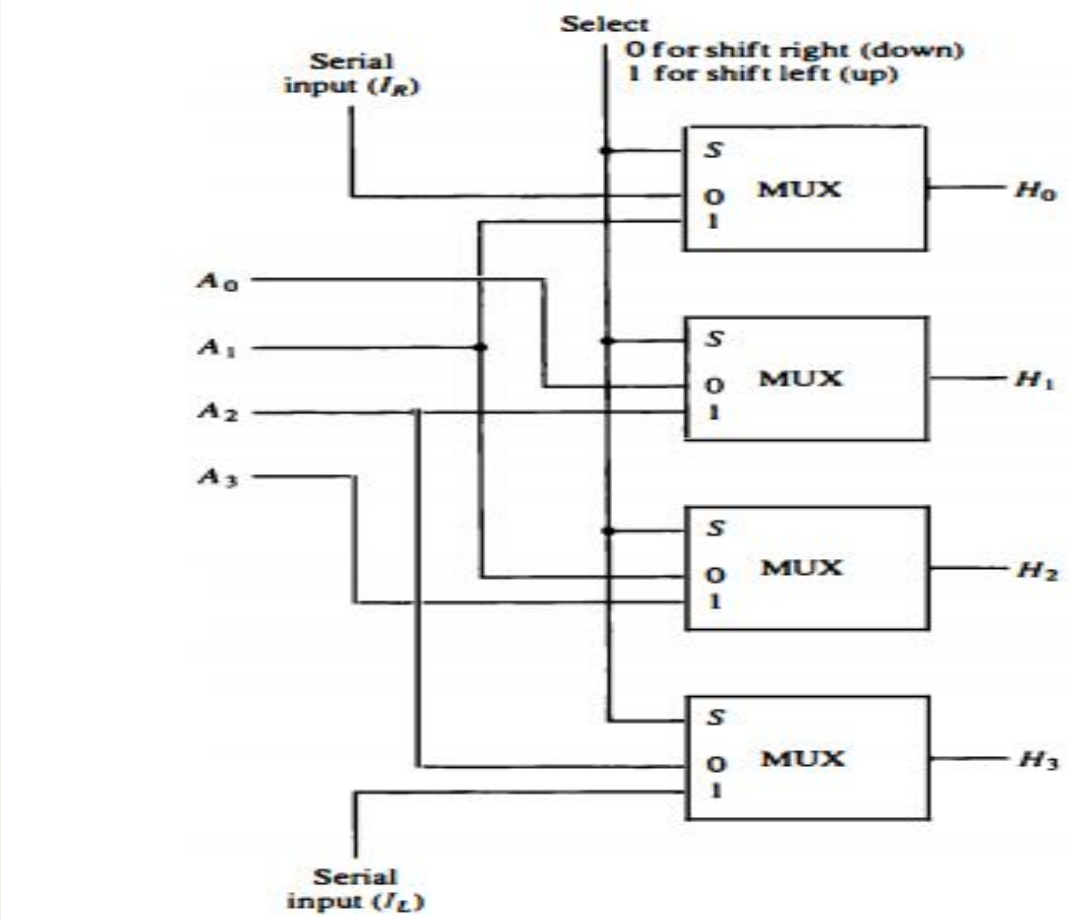


Binary Incrementer



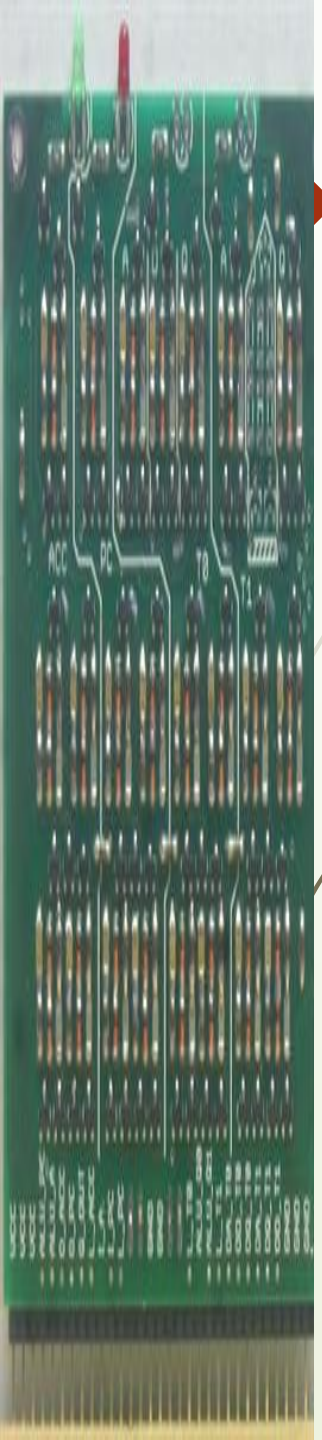


Hardware Implementation of Shift Micro-operations



Function table

Select	Output			
S	H_0	H_1	H_2	H_3
0	I_R	A_0	A_1	A_2
1	A_1	A_2	A_3	I_L

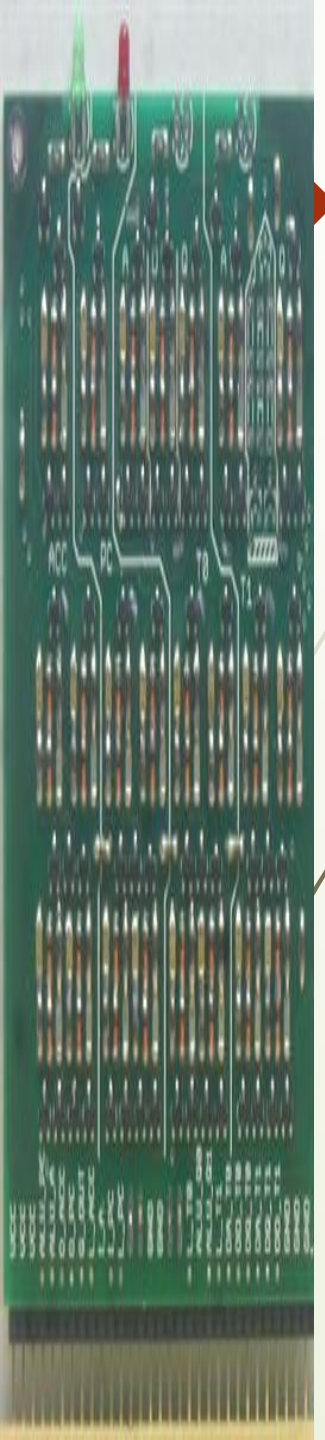


LOGIC MICROOPERATIONS

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

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

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



More Logic Microoperation

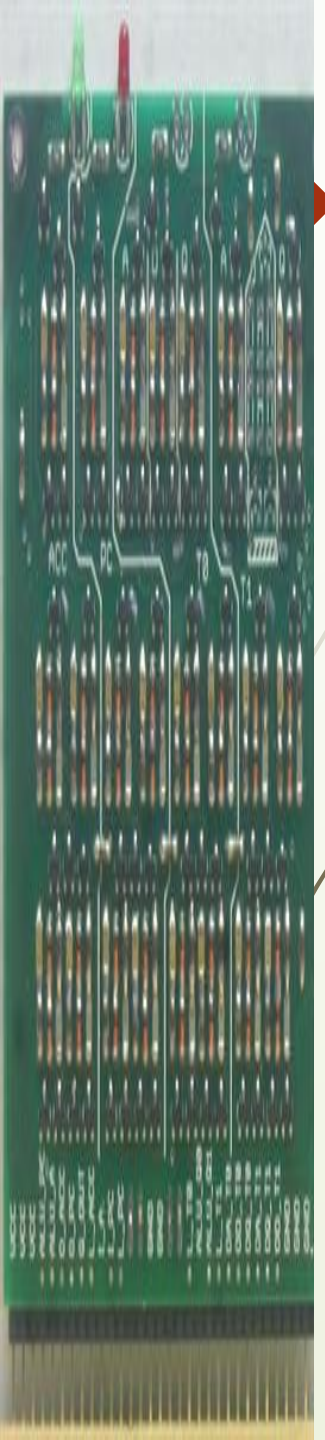
X	Y	F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}	F_{11}	F_{12}	F_{13}	F_{14}	F_{15}
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

TABLE 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

Boolean function	Microoperation	Name
$F_8 = (x+y)'$	$F \leftarrow \overline{A \vee B}$	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 Sixteen Logic Microoperations



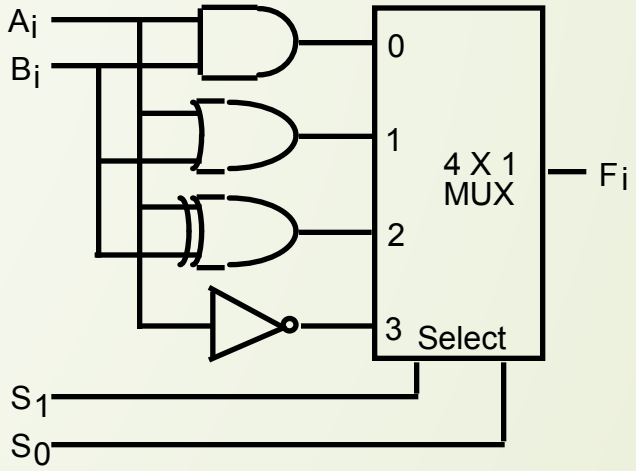
LIST OF LOGIC MICROOPERATIONS

- List of Logic Microoperations
  - 16 different logic operations with 2 binary variables.
  - n binary variables → 2^{2^n} functions

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

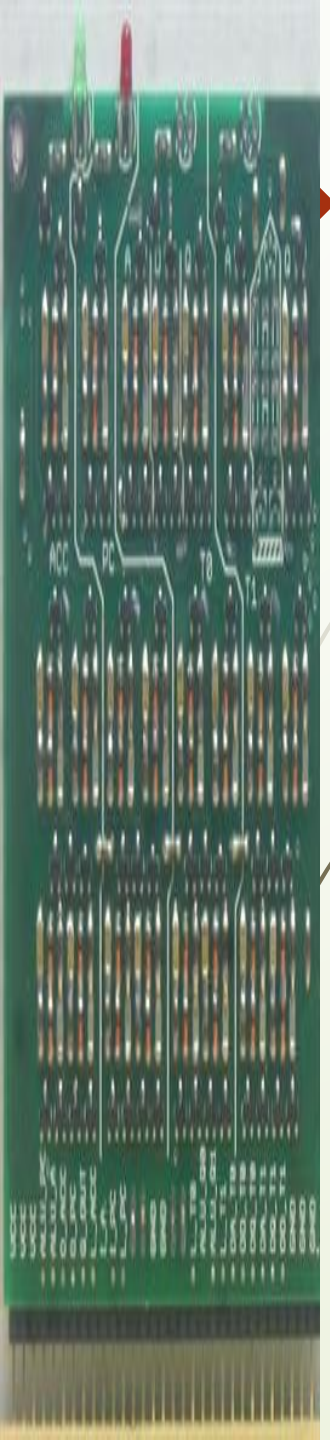
x	0 0 1 1	*Boolean*	*Micro-*	*Name*
y	0 1 0 1	*Function*	*Operations*	
	0 0 0 0	F0 = 0	F ← 0	Clear
	0 0 0 1	F1 = xy	F ← A ∧ B	AND
	0 0 1 0	F2 = xy'	F ← A ∧ B’	
	0 0 1 1	F3 = x	F ← A	Transfer A
	0 1 0 0	F4 = x'y	F ← A’∧ B	
	0 1 0 1	F5 = y	F ← B	Transfer B
	0 1 1 0	F6 = x ⊕ y	F ← A ⊕ B	Exclusive-OR
	0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

HARDWARE IMPLEMENTATION OF LOGIC MICROOPERATIONS



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

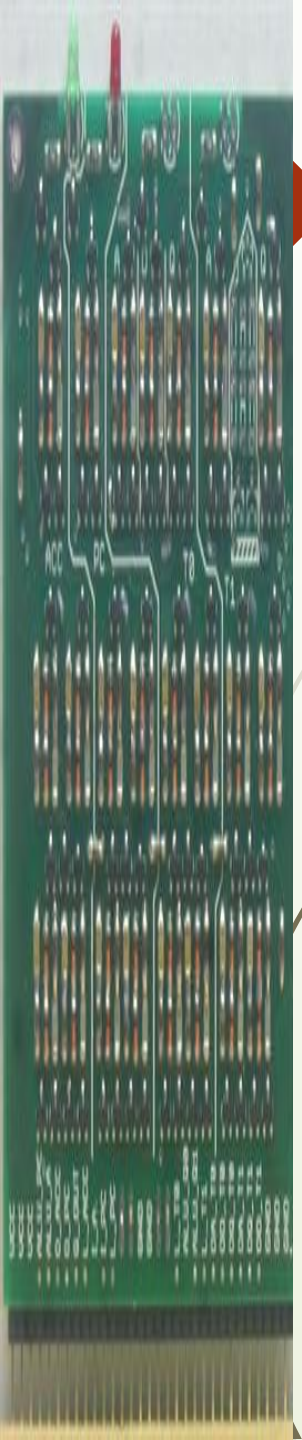


APPLICATIONS OF LOGIC MICROOPERATIONS

- 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 $A \leftarrow A \cdot B'$
  - Mask (Delete) $A \leftarrow A \cdot B$
  - Clear $A \leftarrow A \oplus B$
  - Insert $A \leftarrow (A \cdot B) + C$
  - Compare $A \leftarrow A \oplus B$
  - . . .

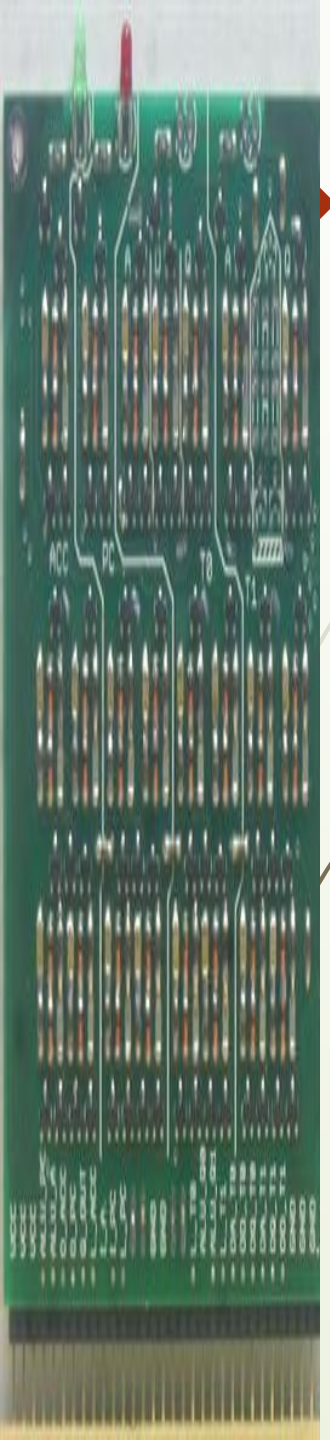


SELECTIVE-SET operation

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

1 1 0 0	A_t before
1 0 1 0	B (logic operand)
1 1 1 0	A_{t+1} (A ← A + B) after

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

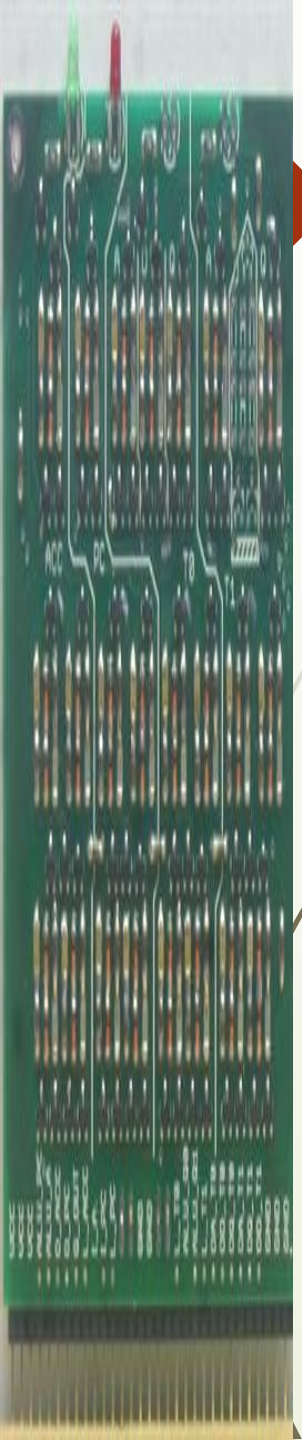


SELECTIVE-SET operation

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

1 1 0 0	A_t before
1 0 1 0	B (logic operand)
1 1 1 0	A_{t+1} ($A \leftarrow A + B$) after

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

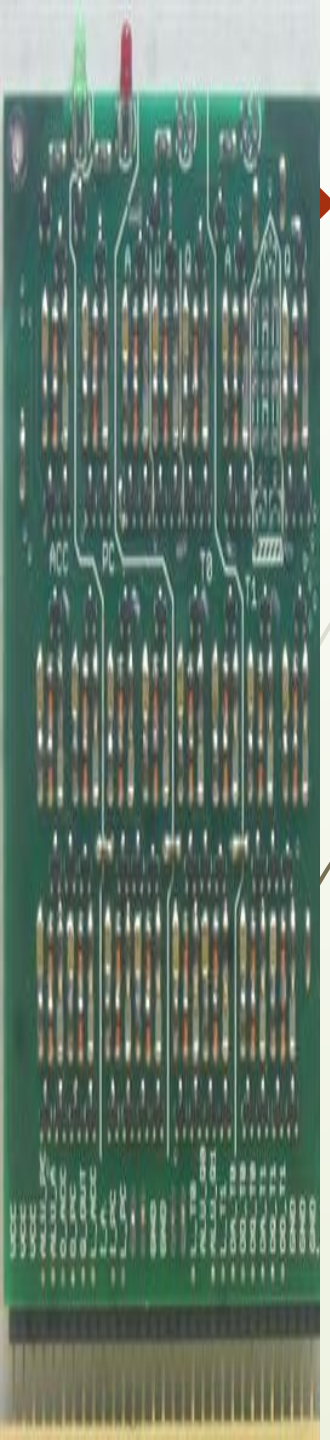


SELECTIVE-COMPLEMENT

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

1 1 0 0	A_t before
1 0 1 0	B (logic operand)
0 1 1 0	A_{t+1} $(A \leftarrow A \oplus B)$ after

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

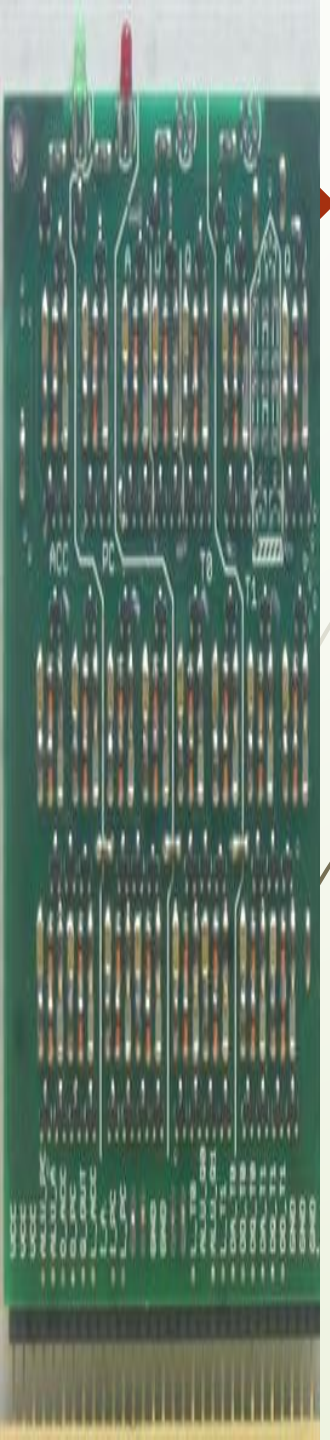


SELECTIVE-CLEAR

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

1 1 0 0	A_t before
1 0 1 0	B (logical operand)
0 1 0 0	A_{t+1} ($A \leftarrow A \cdot B'$) after

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



MASK OPERATION

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

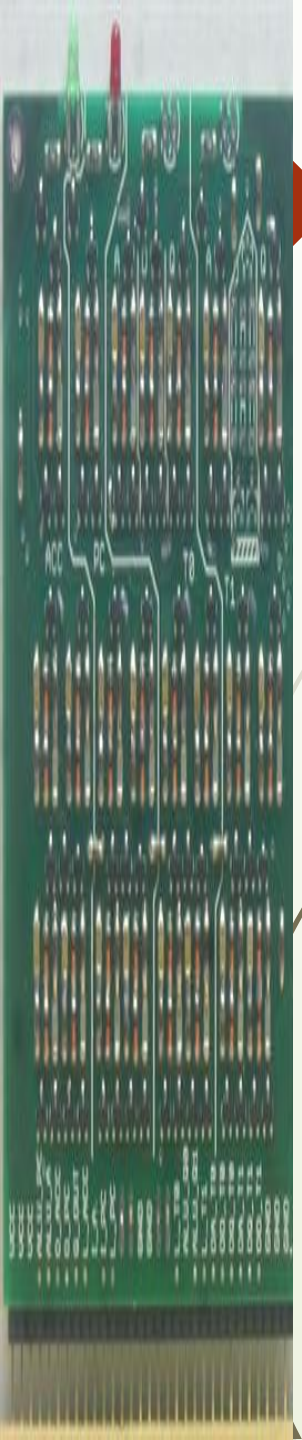
1 1 0 0	A_t before
1 0 1 0	B (logical operand)
1 0 0 0	A_{t+1} ($A \leftarrow A \cdot B$) after masking

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

CLEAR OPERATION

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

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

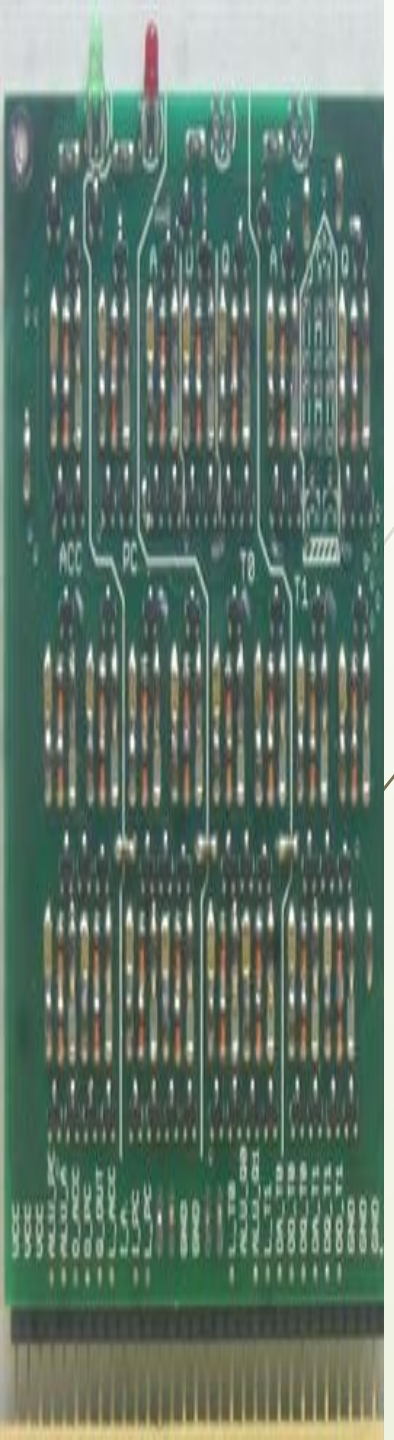


INSERT OPERATION

- An insert operation is used to introduce a specific bit pattern into A register, leaving the other bit positions unchanged
- This is done as
  - A mask operation to clear the desired bit positions, followed by
  - An OR operation to introduce the new bits into the desired positions
  - Example
    - Suppose you wanted to introduce 1010 into the low order four bits of A: 1101 1000 1011 0001
      A (Original) 1101 1000 1011 1010 A (Desired)

    - 1101 1000 1011 0001 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)

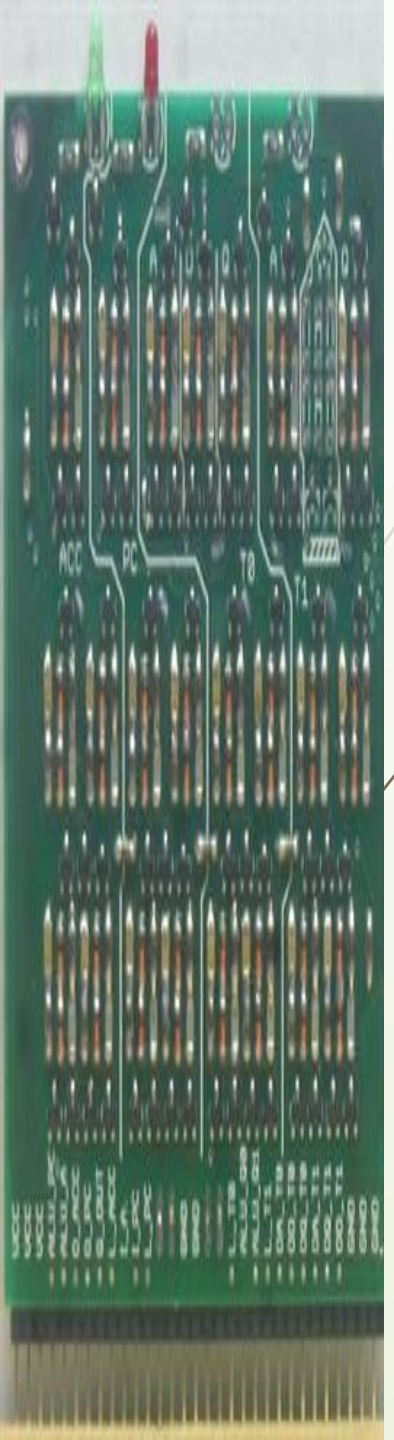


Shift Micro-operations

- ***Shift Micro-Operations***

- These are used for serial transfer of data. That means we can shift the contents of the register to the left or right. In the **shift left** operation the serial input transfers a bit to the right most position and in **shift right** operation the serial input transfers a bit to the left most position.

- There are three types of shifts as follows:



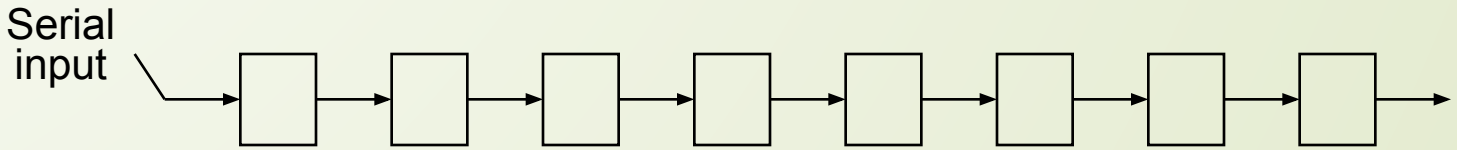
Types of Shift micro-operations

- **Circular Shift**
- This circulates or rotates the bits of register around the two ends without any loss of data or contents. In this, the serial output of the shift register is connected to its serial input. **"cil"** and **"cir"** is used for circular shift left and right respectively.
-
- **c) Arithmetic Shift**
- This shifts a signed binary number to left or right. An **arithmetic shift left** multiplies a signed binary number by 2 and **shift left** divides the number by 2. Arithmetic shift micro-operation leaves the sign bit unchanged because the signed number remains same when it is multiplied or divided by 2.

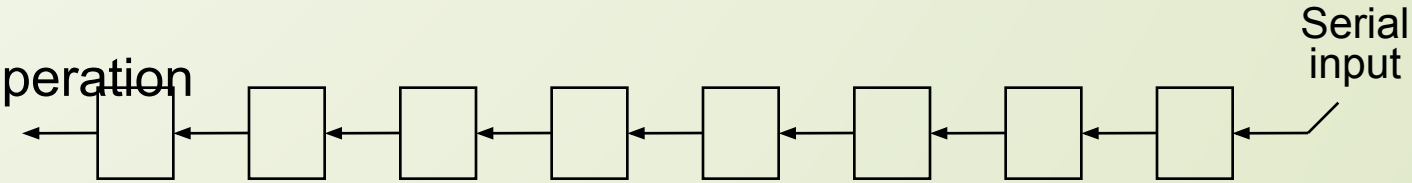
SHIFT MICROOPERATIONS

- Shift microoperations are used for serial transfer of data
- 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



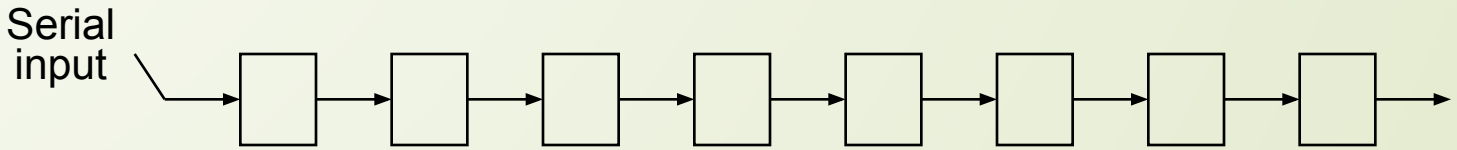
• A left shift operation



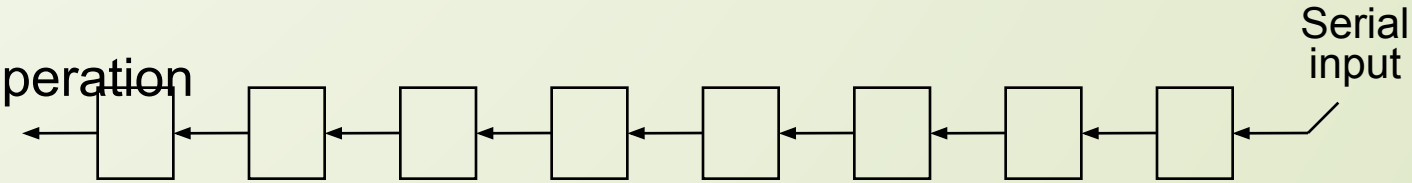
SHIFT MICROOPERATIONS

- Shift microoperations are used for serial transfer of data
- 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

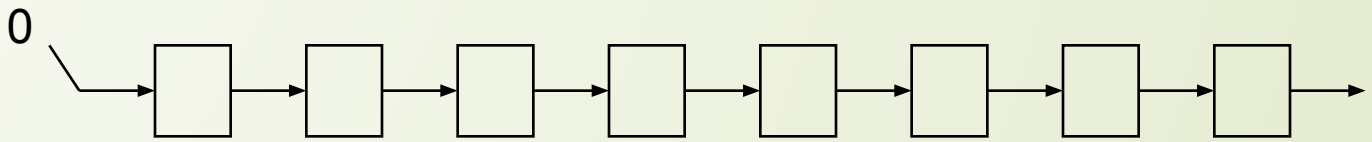


• A left shift operation

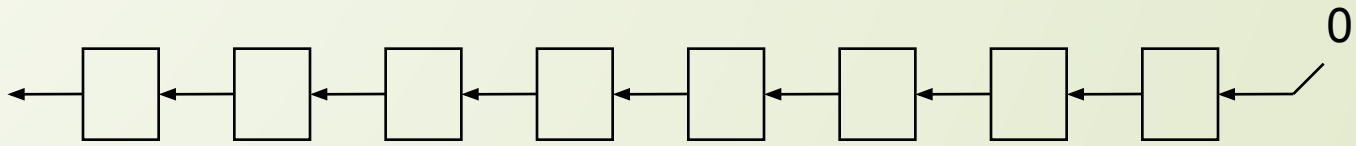


LOGICAL SHIFT

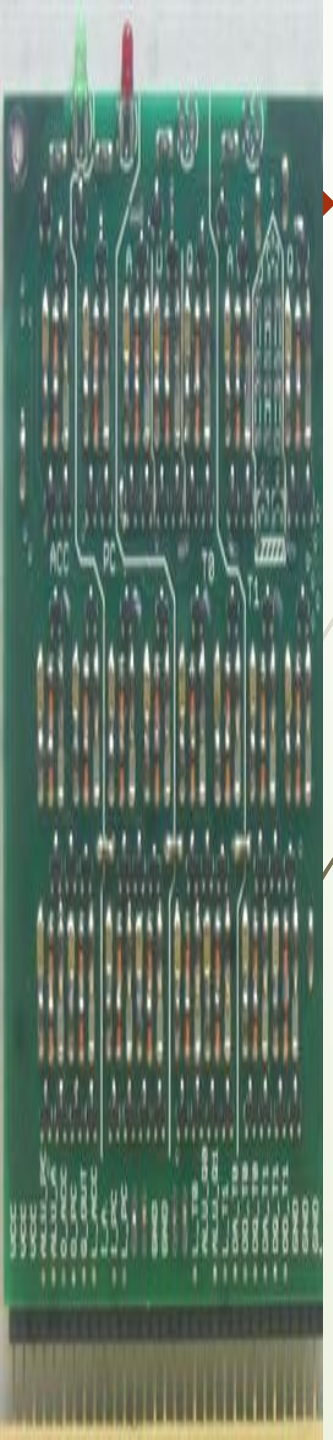
- In a logical shift the serial input to the shift is a 0.
- A right logical shift operation:



- A left logical shift operation:

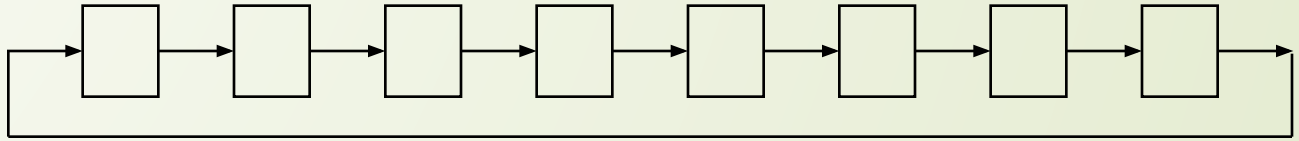


- In a Register Transfer Language, the following notation is used
  - *shl* for a logical shift left
  - *shr* for a logical shift right
  - Examples:
    - R2 ← shr R2
    - R3 ← shl R3

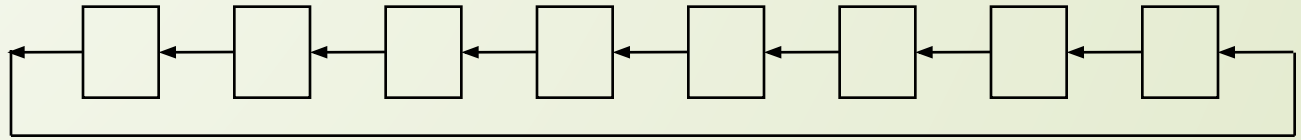


CIRCULAR SHIFT

- 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:

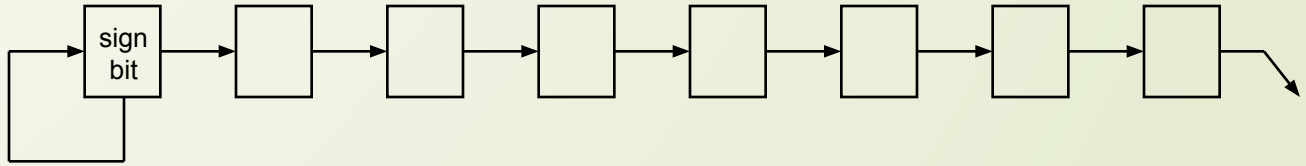


- In a RTL, the following notation is used
  - *cil* for a circular shift left
  - *cir* for a circular shift right
  - Examples:
    - R2 ← cir R2
    - R3 ← cil R3

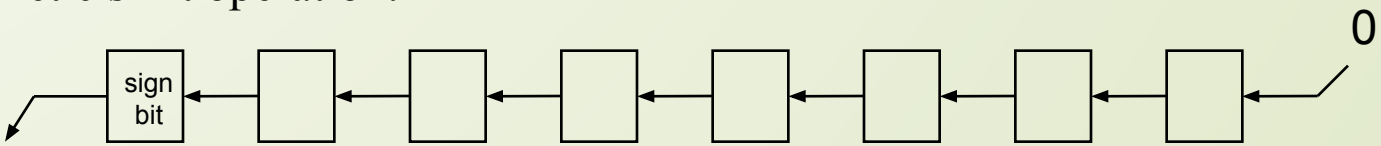
ARITHMETIC SHIFT

- 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
- 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:

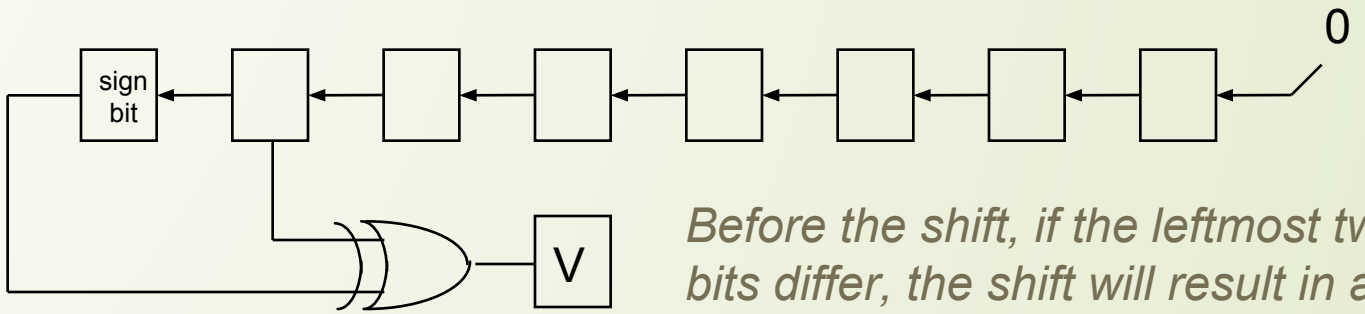


- A left arithmetic shift operation:



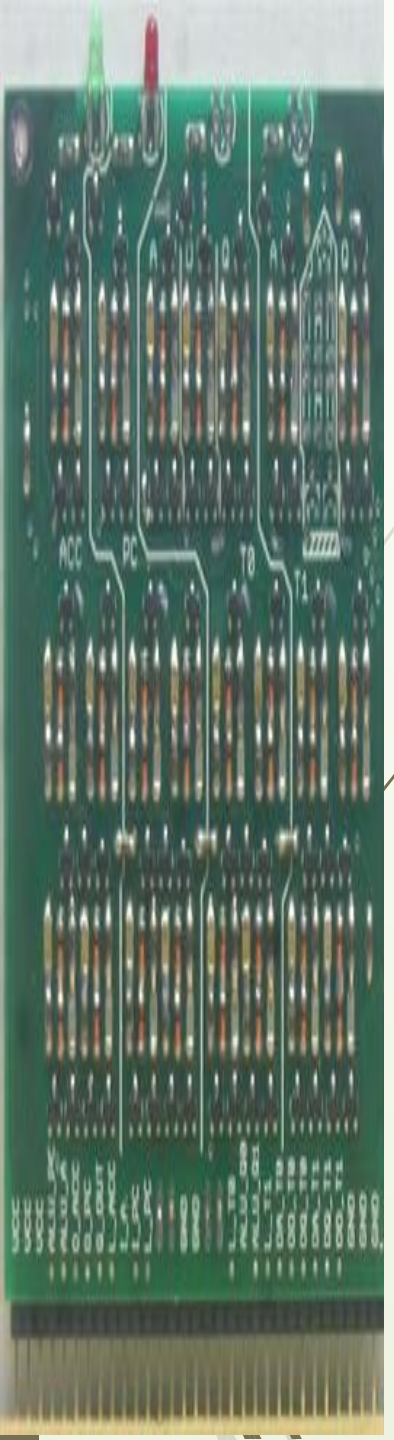
ARITHMETIC SHIFT

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

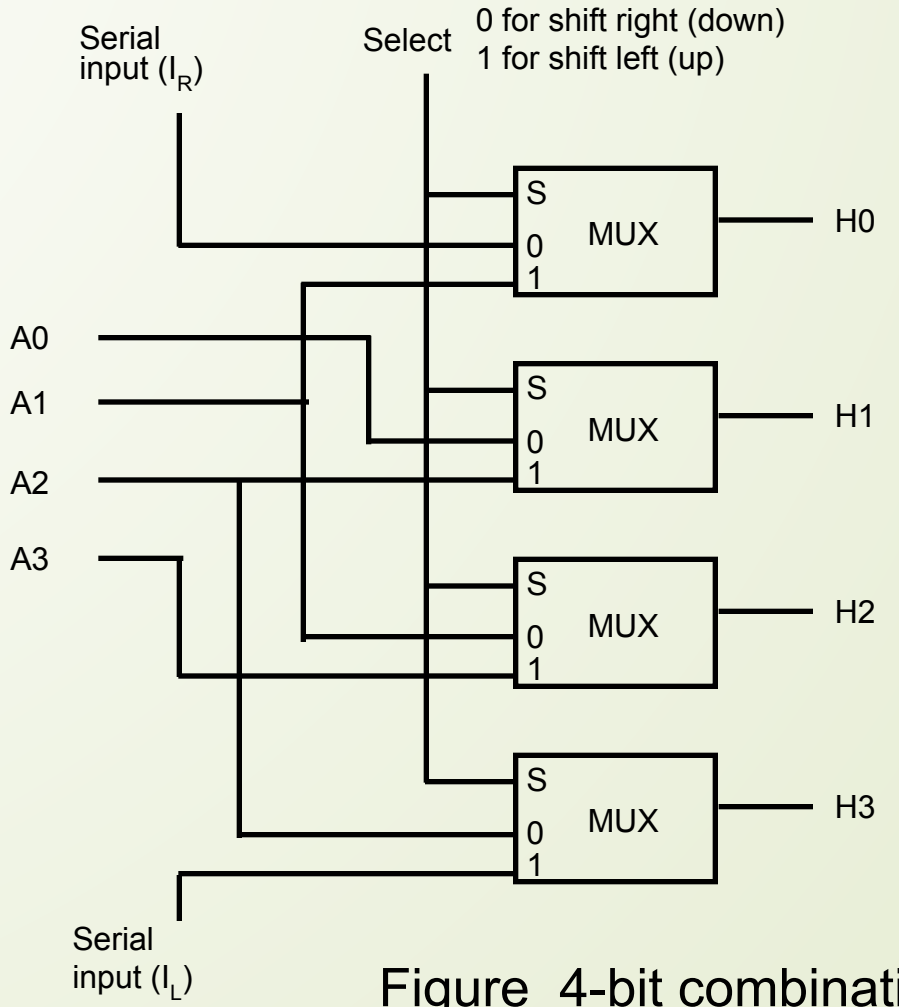


Before the shift, if the leftmost two bits differ, the shift will result in an overflow

- **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 MICROOPERATIONS

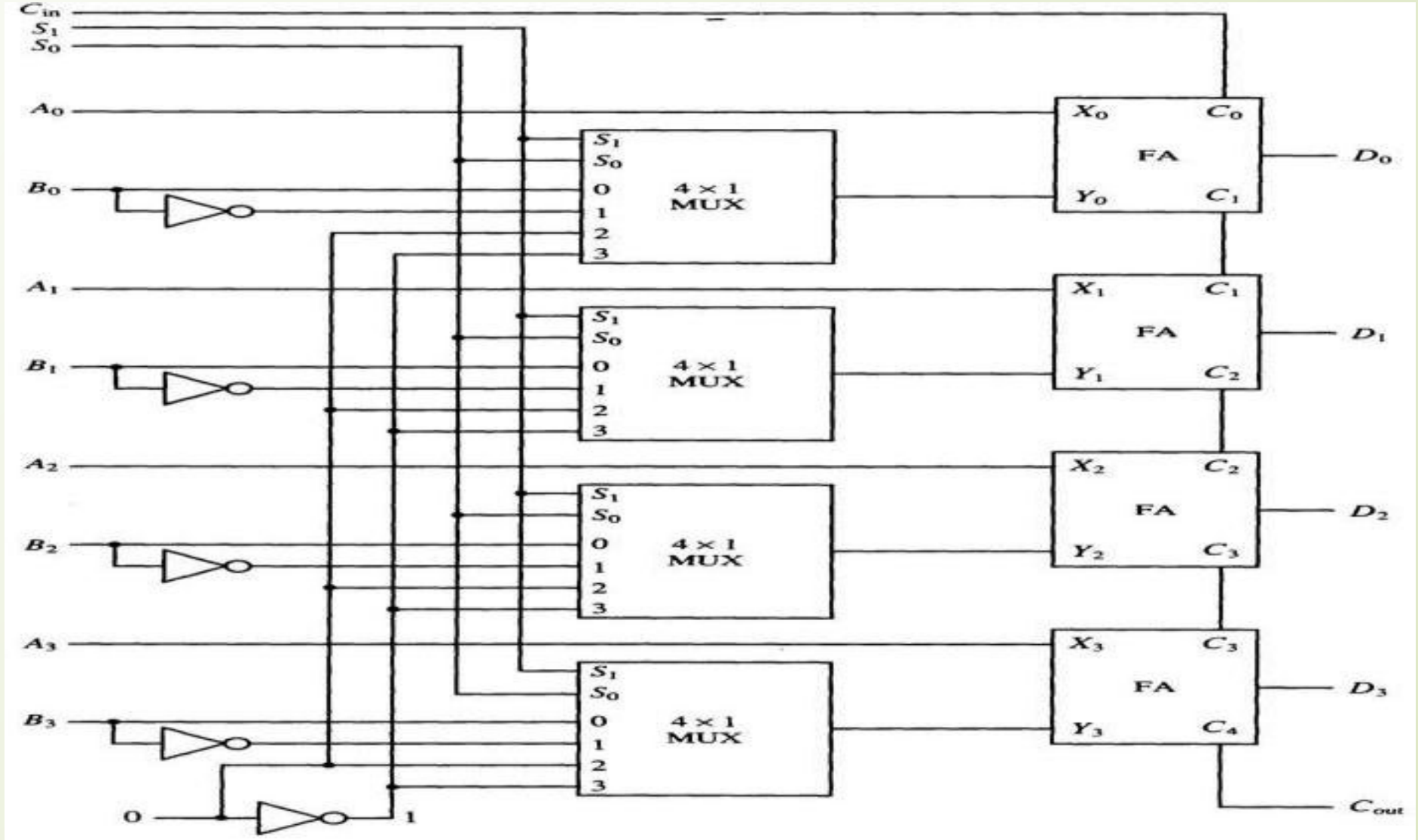


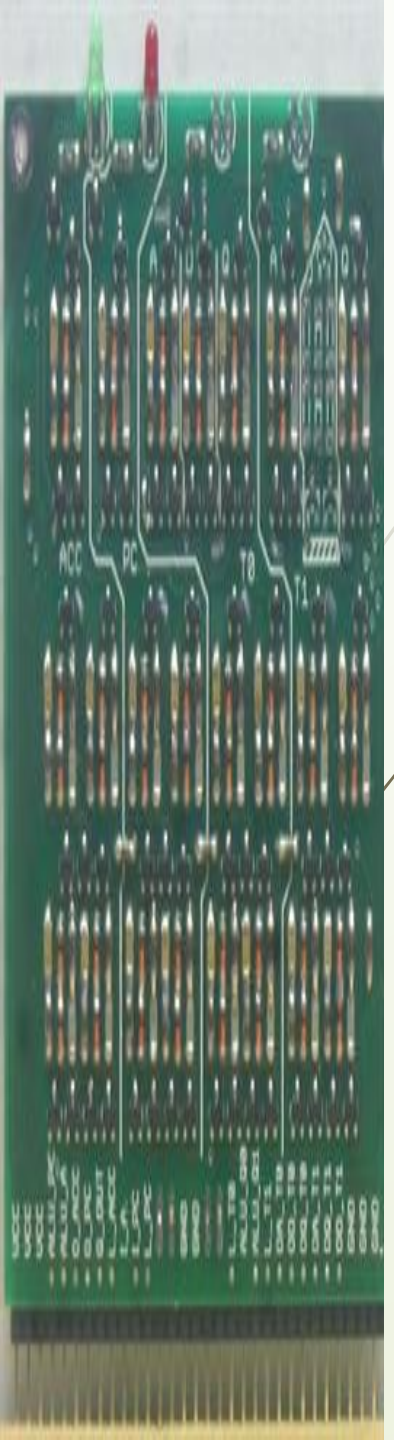
Function table

Select	output			
S	H0	H1	H2	H3
0	I_R	A0	A1	A2
1	A1	A2	A3	I_L

Figure 4-bit combinational circuit shifter

Arithmetic Circuit





Arithmetic Circuit Truth Table

Select			In	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 w/carry
0	1	0	$\overline{B}$	$D = A + \overline{B}$	Subtract w/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

A-B=A+2’S COMPLEMENT OF B

=B’+1

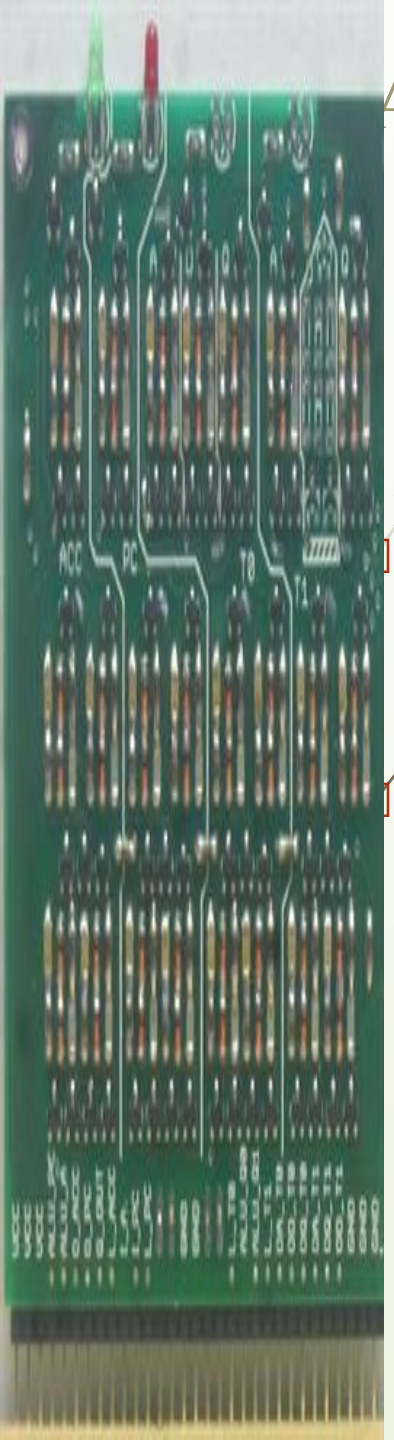
A-B=A+B’+1

b

c

s1	s0		cin	
0	0	B	0,1	
0	1	B’	0,1	
1	0	0		
1	1	1		

- 1=0001
- 1’= 1110
- 1’+1=1111=2’S COMPLEMENT OF 1
- A3A2A1A0+1111=A+2’S COMP OF 1=A-1
- A-1=



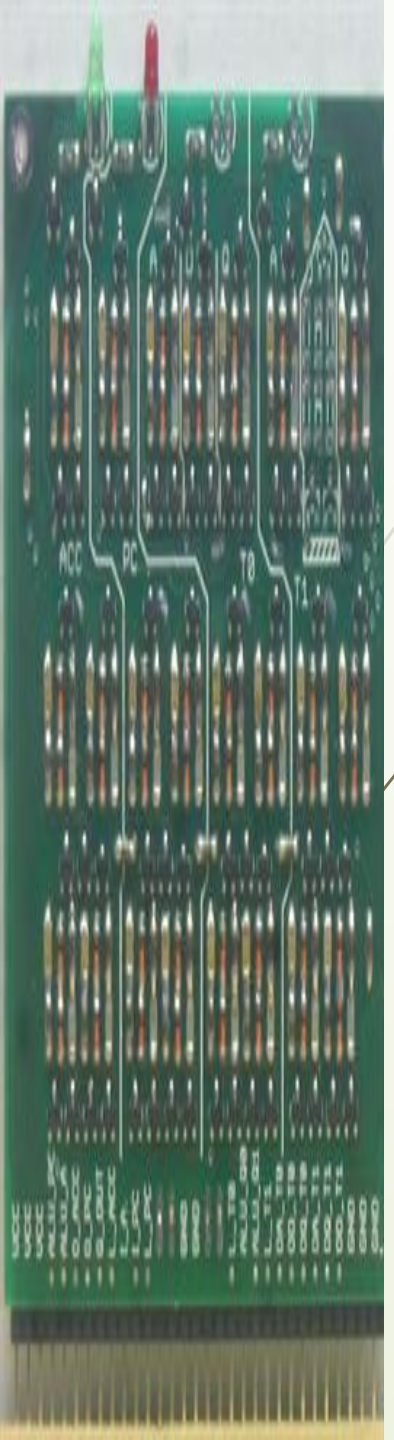
ARITHMETIC CIRCUIT Analysis

4-bit arithmetic circuit:

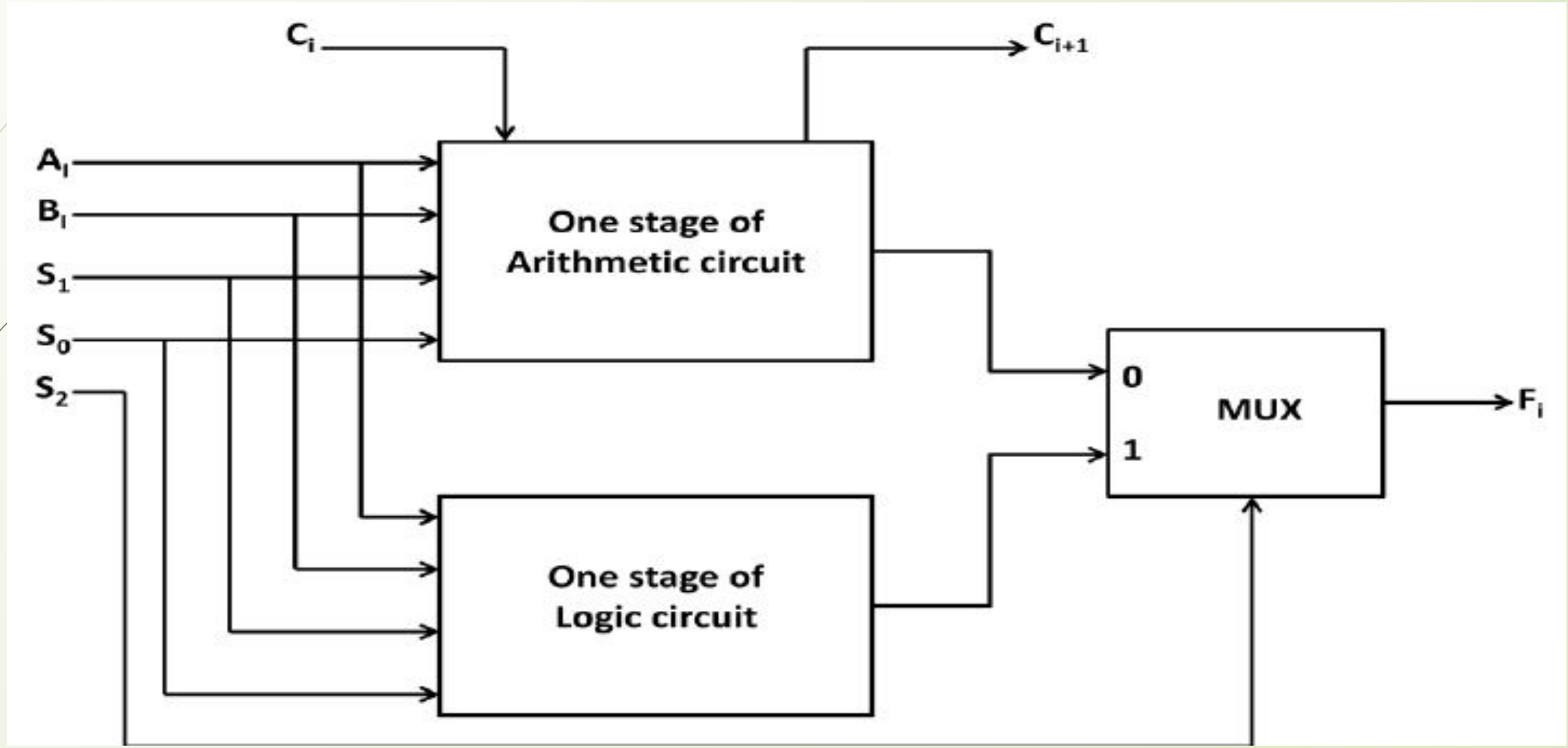
$$D = A + Y + C_{in}$$

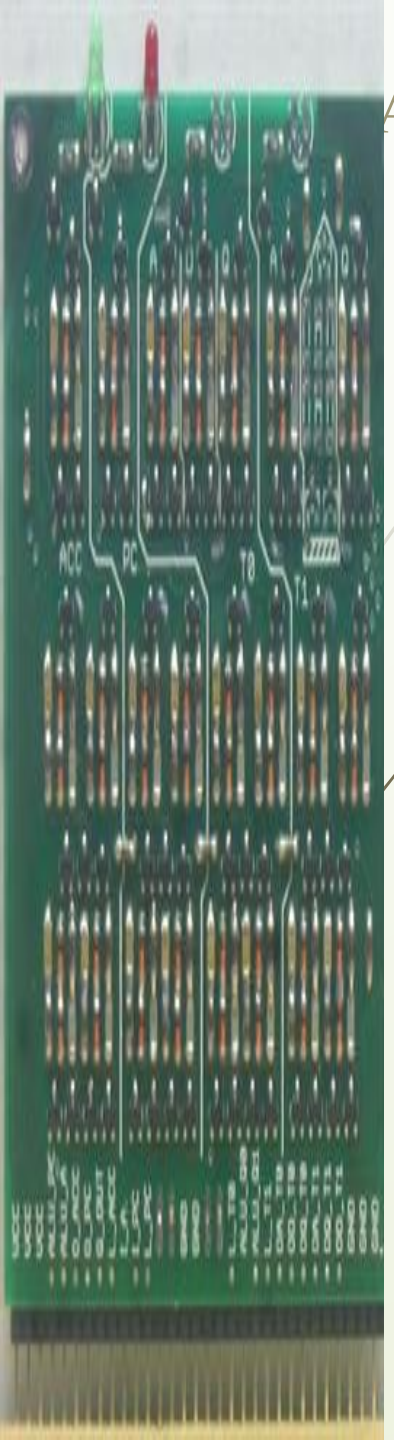
By controlling Y with the two selection inputs (S_1 and S_0) and C_{in} equals to 1 or 0, it is possible to generate eight microoperations.

- When $S_1S_0 = 00$, the value of B is applied to the Y inputs of the adder
  - If $C_{in} = 0$, the output $D = A + B$
  - If $C_{in} = 1$, the output $D = A + B + 1$
- …



Arithmetic logic Unit





ARITHMETIC LOGIC SHIFT UNIT

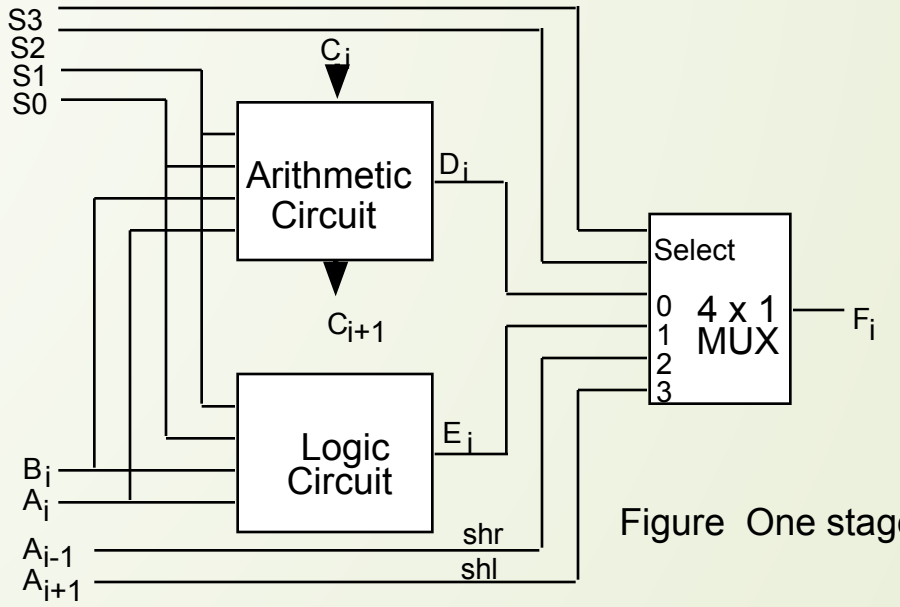
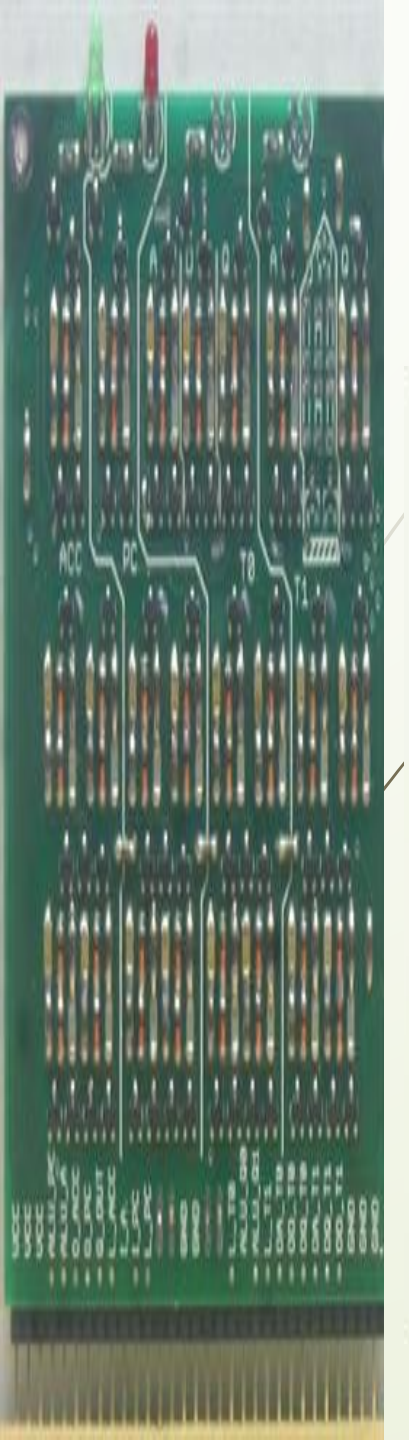


Figure One stage of arithmetic logic shift unit

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



Truth table for ALU

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