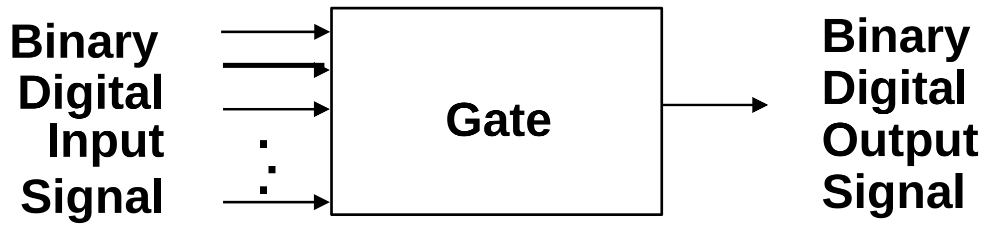
DIGITAL LOGIC CIRCUITS

BASIC LOGIC BLOCK - GATE -



Types of Basic Logic Blocks

- **Combinational Logic Block**
  Logic Blocks whose output logic value depends only on the input logic values

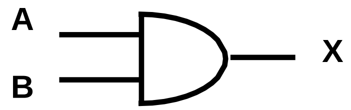
- **Sequential Logic Block**
  Logic Blocks whose output logic value depends on the input values and the state (stored information) of the blocks

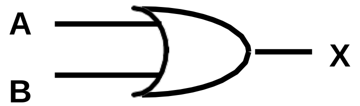
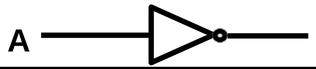
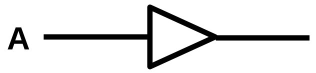
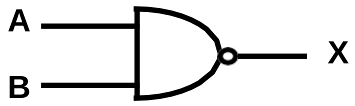
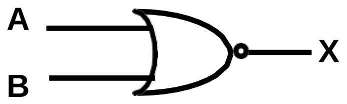
Functions of Gates can be described by

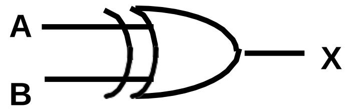
- **Truth Table**
- **Boolean Function**
- **Karnaugh Map**

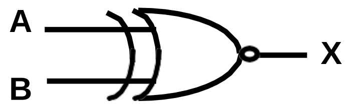
Logic Gates

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
Inverter		X = A'	A | X 0 | 1 1 | 0
Buffer		X = A	A | X 0 | 0 1 | 1
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

BOOLEAN ALGEBRA

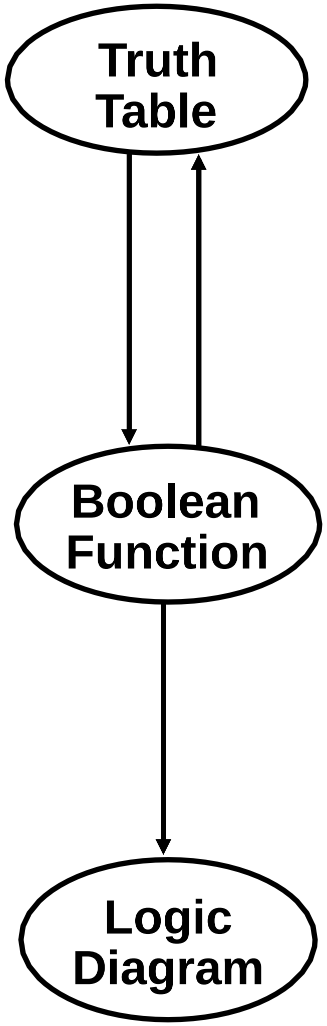
Boolean Algebra

1. **Algebra with Binary(Boolean) Variable and Logic Operations**
2. **Boolean Algebra is useful in Analysis and Synthesis of Digital Logic Circuits**
   1. **Input and Output signals can be represented by Boolean Variables, and**
   2. **Function of the Digital Logic Circuits can be represented by Logic Operations, i.e., Boolean Function(s)**
   3. **From a Boolean function, a logic diagram can be constructed using AND, OR, and INV (NOT)**

Truth Table

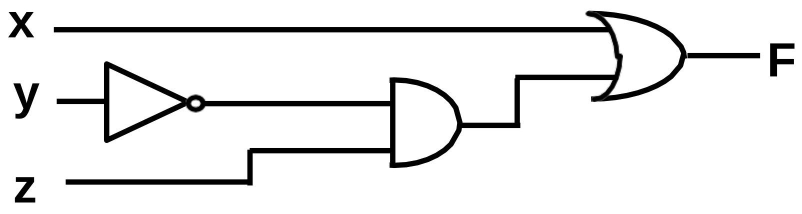
1. **The most elementary specification of the function of a Digital Logic Circuit is the Truth Table**
   1. **Table that describes the Output Values for all the combinations of the Input Values, called *MINTERMS***
   2. **n input variables --> 2^n minterms**

LOGIC CIRCUIT DESIGN



x	y	z	F
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

F = x + y'z



BASIC IDENTITIES OF BOOLEAN ALGEBRA

1. $x + 0 = x$
2. $x \bullet 0 = 0$
3. $x + 1 = 1$
4. $x \bullet 1 = x$
5. $x + x = x$
6. $x \bullet x = x$
7. $x + x' = 1$
8. $x \bullet x' = 0$
9. $x + y = y + x$
10. $xy = yx$
11. $x + (y + z) = (x + y) + z$
12. $x(yz) = (xy)z$
13. $x(y + z) = xy + xz$
14. $x + yz = (x + y)(x + z)$
15. $(x + y)' = x'y'$
16. $(xy)' = x' + y'$
17. $(x')' = x$
18. $x + xy = x$
19. $x + x'y = x + y$

Usefulness of this list

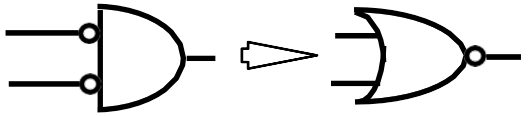
1. Simplification of the Boolean function
2. Derivation of equivalent Boolean functions to obtain logic diagrams utilizing different logic gates
    1. Ordinarily ANDs, ORs, and Inverters
    2. But a certain different form of Boolean function may be convenient to obtain circuits with NANDs or NORs

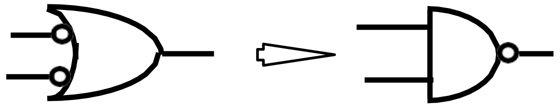
BASIC IDENTITIES OF BOOLEAN ALGEBRA

Applications of DeMorgans Theorem (15 and 16)

x'y' = (x + y)' **x'+ y'= (xy)'**

I, AND --> NOR **I, OR --> NAND**



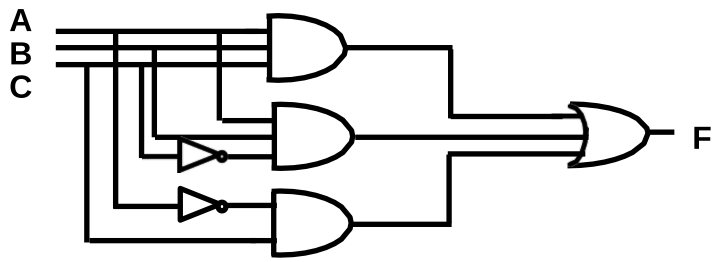


EQUIVALENT CIRCUITS

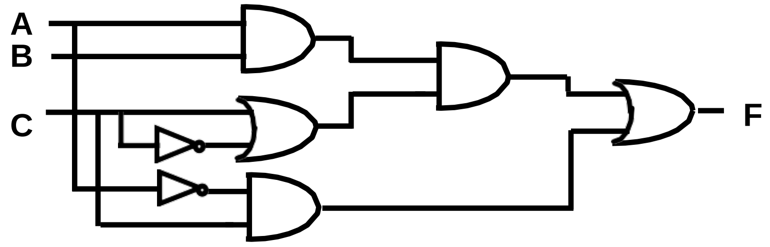
Many different logic diagrams are possible for a given Function

$$F = ABC + ABC' + A'C \qquad \ldots\ldots\ldots\ldots (1)$$
$$= AB(C + C') + A'C \qquad [13] \ldots\ldots (2)$$
$$= AB \cdot 1 + A'C \qquad [7]$$
$$= AB + A'C \qquad [4] \ldots\ldots (3)$$

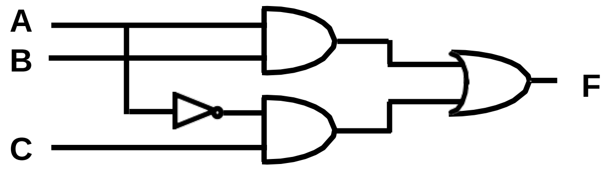
(1)



(2)



(3)



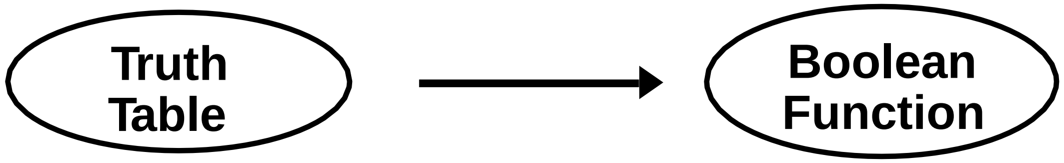
COMPLEMENT OF FUNCTIONS

A Boolean function of a digital logic circuit is represented by only using logical variables and AND, OR, and Invert operators.

--> Complement of a Boolean function

1. **First: Change all OR operations to AND operations and all AND operations to OR operations**
2. **Second: Complement each individual variable**
3. **Basically, extensive applications of the DeMorgan's theorem**
   $(x_1 + x_2 + ... + x_n)' \Rightarrow x_1'x_2'... x_n'$
   $(x_1x_2 ... x_n)' \Rightarrow x_1' + x_2' + ... + x_n'$
4. **Example:**
   $F = AB + C'D' + B'D$
   $F' = (A' + B')(C + D)(B + D')$

SIMPLIFICATION

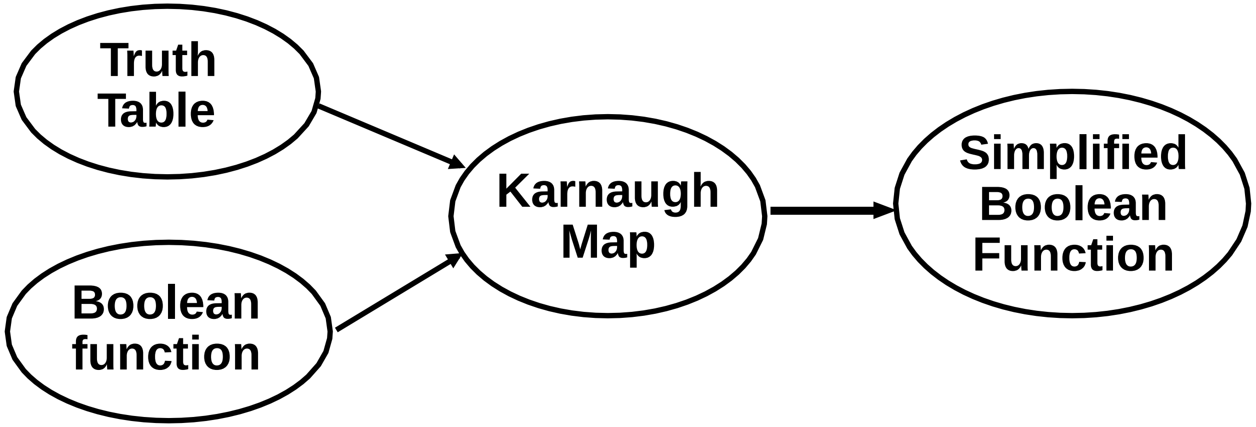


Unique **Many different expressions exist**

Simplification from Boolean function

- **Finding an equivalent expression that is least expensive to implement**
- **For a simple function, it is possible to obtain a simple expression for low cost implementation**
- **But, with complex functions, it is a very difficult task**

Karnaugh Map(K-map) is a simple procedure for simplifying Boolean expressions.



KARNAUGH MAP

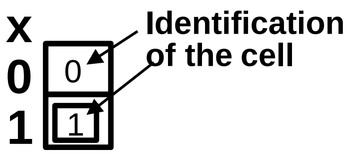
Karnaugh Map for an n-input digital logic circuit (n-variable sum-of-products form of Boolean Function, or Truth Table) is

- **Rectangle divided into 2^n cells**
- **Each cell is associated with a *Minterm***
- **An output(function) value for each input value associated with a minterm is written in the cell representing the minterm**

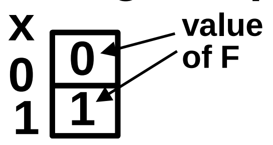
--> 1-cell, 0-cell

Each Minterm is identified by a decimal number whose binary representation is identical to the binary interpretation of the input values of the minterm.

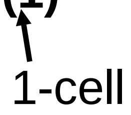
x	F
0	1
1	0



Karnaugh Map

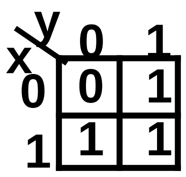


$F(x) = \sum (1)$

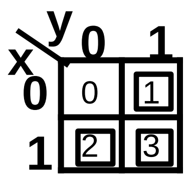


Σ - stands for the sum of the minterms. The minterms that produce 1 are listed. Minterms that produce 0 aren't listed

x	y	F
0	0	0
0	1	1
1	0	1
1	1	1



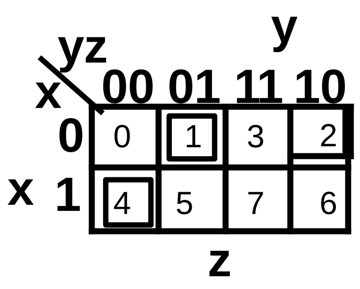
x	y	F
0	0	0
0	1	1
1	0	1
1	1	1



$F(x,y) = \sum (1,2,3)$

KARNAUGH MAP

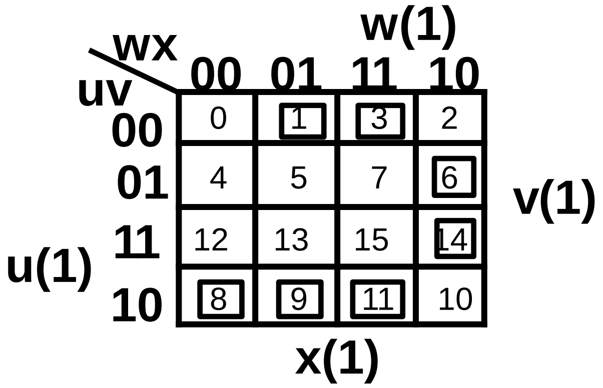
x	y	z	F
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0



x \ yz	00	01	11	10
0	0	1	0	1
1	1	0	0	0

$F(x,y,z) = \sum (1,2,4)$

u	v	w	x	F
0	0	0	0	0
0	0	0	1	1
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	1
0	1	1	1	0
1	0	0	0	1
1	0	0	1	1
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	0
1	1	1	0	1
1	1	1	1	0



uv \ wx	00	01	11	10
00	0	1	1	0
01	0	0	0	1
11	0	0	0	1
10	1	1	1	0

$F(u,v,w,x) = \sum (1,3,6,8,9,11,14)$

MAP SIMPLIFICATION - 2 ADJACENT CELLS -

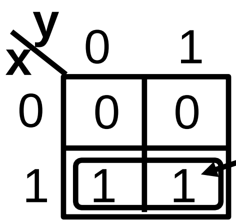
$\boxed{\text{Rule: } xy' + xy = x(y+y') = x}$

Adjacent cells

- **binary identifications are different in one bit**
  --> minterms associated with the adjacent cells have one variable complemented each other

Cells (1,0) and (1,1) are adjacent
Minterms for (1,0) and (1,1) are
x • y' --> x=1, y=0
x • y --> x=1, y=1

F = xy'+ xy can be reduced to F = x
From the map

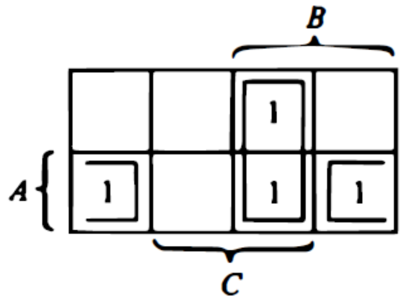


$$\begin{array}{c|cc} x \backslash y & 0 & 1 \\ \hline 0 & 0 & 0 \\ 1 & 1 & 1 \end{array}$$

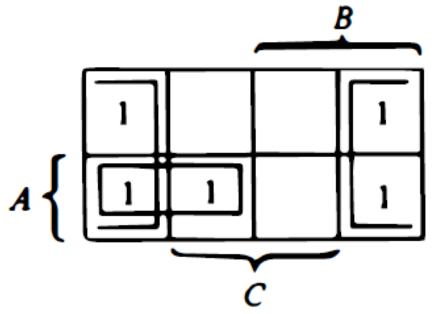
2 adjacent cells xy' and xy
--> merge them to a larger cell x

$$F(x,y) = \sum(2,3) = xy' + xy = x$$

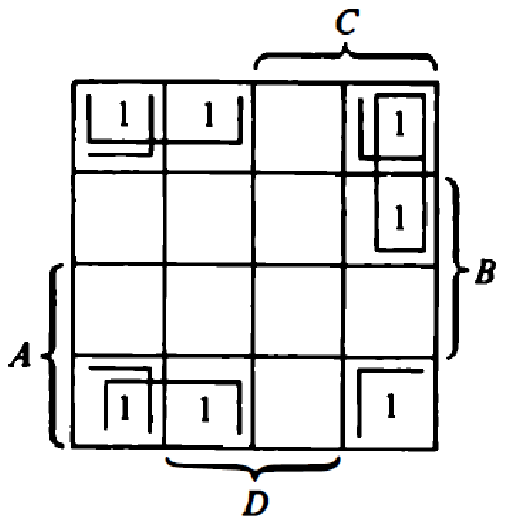
MAP SIMPLIFICATION - 2 ADJACENT CELLS -



F = BC + AC’



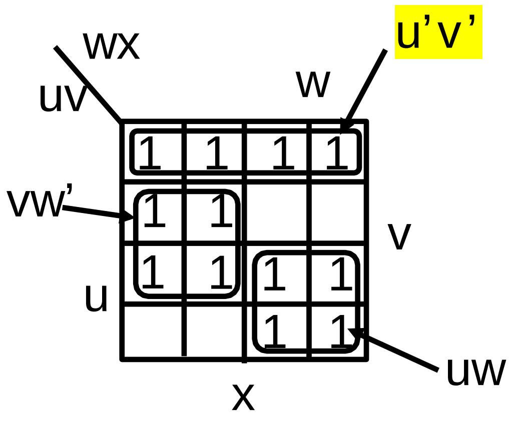
F = C’ + AB’

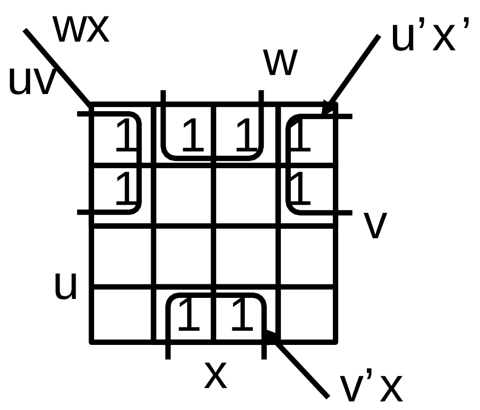


F = B’C’ + A’CD’ + B’D’

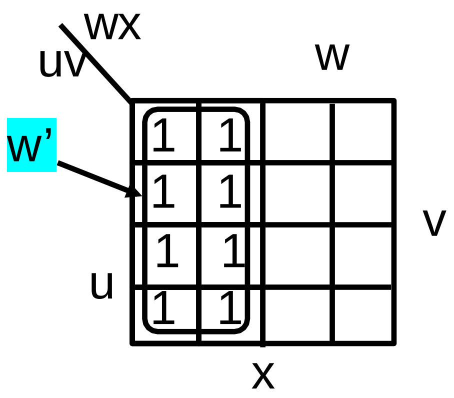
MAP SIMPLIFICATION - MORE THAN 2 CELLS -

Proof of u’v’:
u’v’w’x’ + u’v’w’x + u’v’wx + u’v’wx’
= u’v’w’(x’+x) + u’v’w(x+x’)
= u’v’w’ + u’v’w
= u’v’(w’+w)
= u’v’





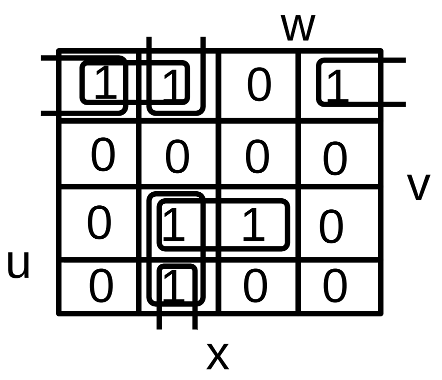
Proof of w’:
u’v’w’x’+u’v’w’x+u’vw’x’+u’vw’x+uvw’x’+uvw’x+uv’w’x’+uv’w’x
= u’v’w’(x’+x) + u’vw’(x’+x) + uvw’(x’+x) + uv’w’(x’+x)
= u’(v’+v)w’ + u(v’+v)w’
= (u’+u)w’ = w’



MAP SIMPLIFICATION

uv \ wx	00	01	11	10
00	1	1	0	1
01	0	0	0	0
11	0	1	1	0
10	0	1	0	0

$\longrightarrow$



	00	01	11	10
00	0	1	3	2
01	4	5	7	6
11	12	13	15	14
10	8	9	11	10

F(u,v,w,x) = ∑ (0,1,2,9,13,15)

(0,1), (0,2), (0,4), (0,8)

Adjacent Cells of 1

Adjacent Cells of 0

(1,0), (1,3), (1,5), (1,9)

...

...

Adjacent Cells of 15

(15,7), (15,11), (15,13), (15,14)

$\longrightarrow$

Merge (0,1) and (0,2)

--> u’v’w’ + u’v’x’

Merge (1,9)

--> v’w’x

Merge (9,13)

--> uw’x

Merge (13,15)

--> uvx

F = u’v’w’ + u’v’x’ + v’w’x + uw’x + uvx

But (9,13) is covered by (1,9) and (13,15)

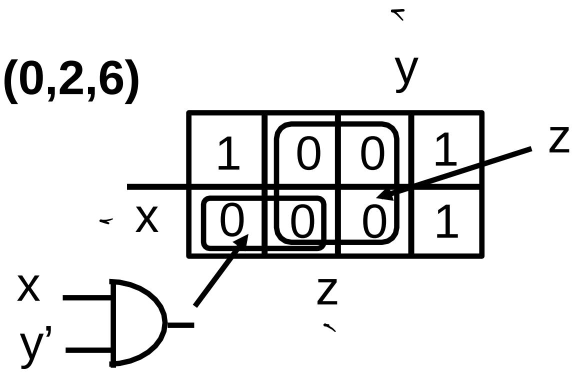
F = u’v’w’ + u’v’x’ + v’w’x + uvx

IMPLEMENTATION OF K-MAPS - Product-of-Sums Form -

Logic function represented by a Karnaugh map can be implemented in the form of I-OR-AND

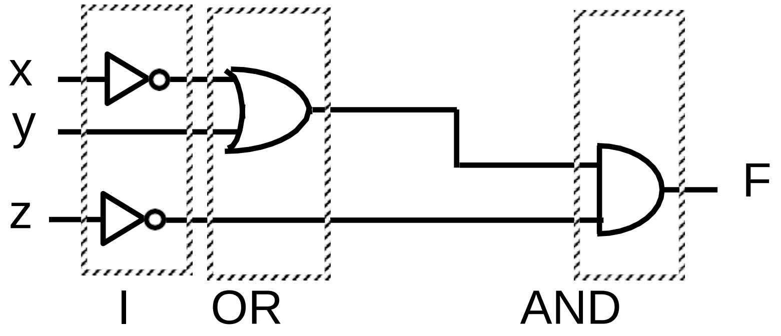
If we implement a Karnaugh map using 0-cells, the complement of F, i.e., F', can be obtained. Thus, by complementing F' using DeMorgan's theorem F can be obtained

F(x,y,z) = (0,2,6)



F' = xy' + z

F = (x' + y)z'

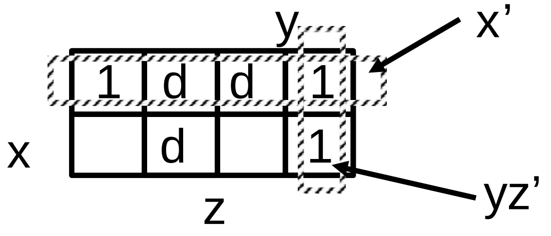


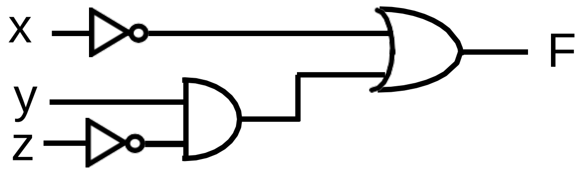
IMPLEMENTATION OF K-MAPS
- Don't-Care Conditions -

In some logic circuits, the output responses for some input conditions are don't care whether they are 1 or 0.

In K-maps, don't-care conditions are represented by d's in the corresponding cells.

Don't-care conditions are useful in minimizing the logic functions using K-map.
- **Can be considered either 1 or 0**
- **Thus increases the chances of merging cells into the larger cells**
  --> Reduce the number of variables in the product terms





COMBINATIONAL LOGIC CIRCUITS

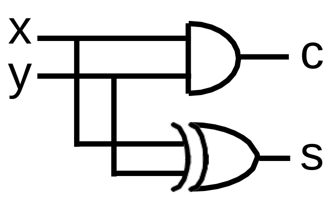
Half Adder

x	y	c	s
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$$\begin{array}{c|cc} & & y \\ \hline & 0 & 0 \\ x & 0 & 1 \end{array} \qquad \begin{array}{c|cc} & & y \\ \hline & 0 & 1 \\ x & 1 & 0 \end{array}$$

$c = xy$

$s = xy' + x'y$
$= x \oplus y$



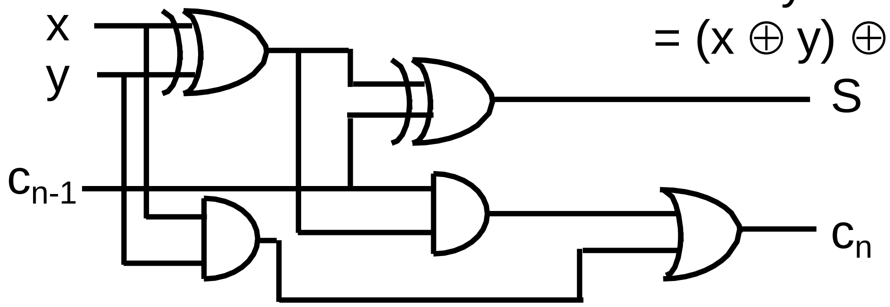
Full Adder

x	y	c_{n-1}	c_n	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

$$\begin{array}{c|cc|c} & & y & \\ \hline & 0 & 0 & \\ & 0 & 1 & c_{n-1} \\ x & 1 & 1 & \\ & 0 & 1 & \\ \hline & c_n & & \end{array} \qquad \begin{array}{c|cc|c} & & y & \\ \hline & 0 & \boxed{1} & \\ & \boxed{1} & 0 & c_{n-1} \\ x & 0 & \boxed{1} & \\ & \boxed{1} & 0 & \\ \hline & s & & \end{array}$$

$c_n = xy + xc_{n-1} + yc_{n-1}$
$= xy + (x \oplus y)c_{n-1}$

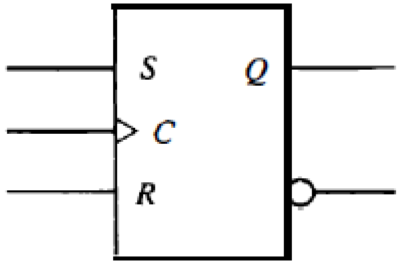
$s = x'y'c_{n-1} + x'yc'_{n-1} + xy'c'_{n-1} + xyc_{n-1}$
$= x \oplus y \oplus c_{n-1}$
$= (x \oplus y) \oplus c_{n-1}$



Sequential Storage Circuits - Flip-Flops

1. Flip flops will sample inputs and change the output only at certain instances in time.
2. Allows for storage of state information.
3. For SR flip-flop below, the output Q only changes when clock changes from 0 to 1

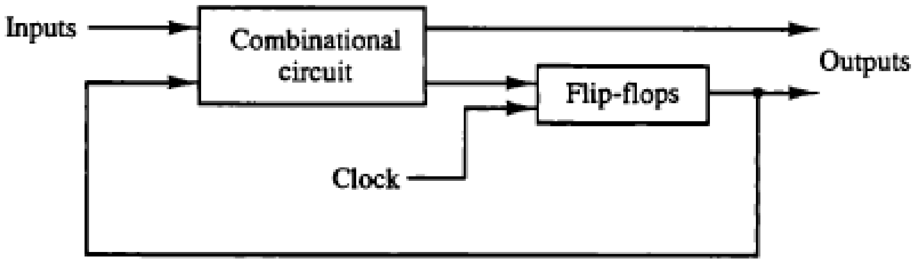
Note: C is for clock

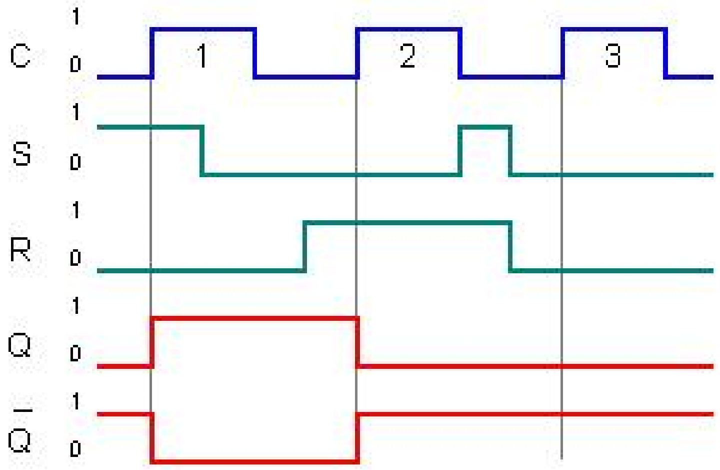


(a) Graphic symbol

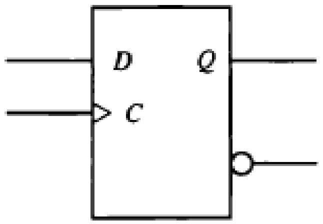
S	R	$Q(t+1)$	
0	0	$Q(t)$	No change
0	1	0	Clear to 0
1	0	1	Set to 1
1	1	?	Indeterminate

(b) Characteristic table





Some other Flip-Flops

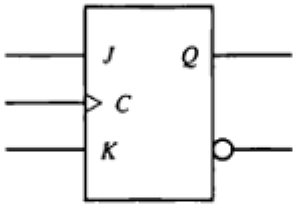


(a) Graphic symbol

D	Q (t + 1)	
0	0	Clear to 0
1	1	Set to 1

(b) Characteristic table

Figure 1-20 D flip-flop.

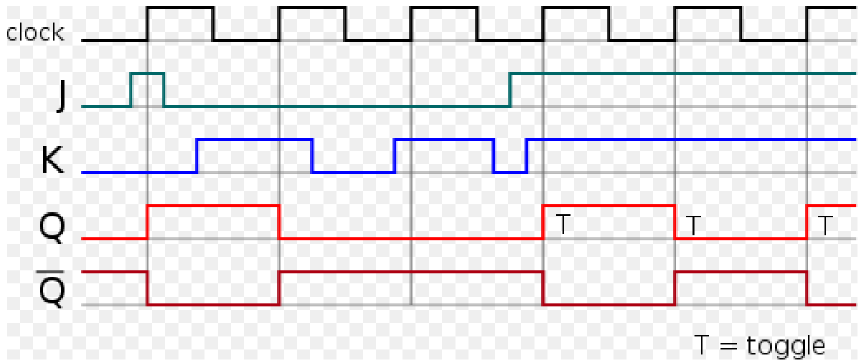


(a) Graphic symbol

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

(b) Characteristic table

Figure 1-21 JK flip-flop.



1. D Flip-Flop – D is sampled during the clock transition from 0 to 1 and passed to the output.
2. JK Flip-Flop – The indeterminate conditions of SR is defined to switch the outputs to their complement state.
   1. Toggle in the diagram is the same meaning as Complement in the table

Chapter 2 Digital Components

1. **Integrated Circuits**
2. **Decoder**
3. **Encoder**
4. **Multiplexer**
5. **Registers**
6. **Memory**

INTEGRATED CIRCUITS (IC)

Classification by the Circuit Density

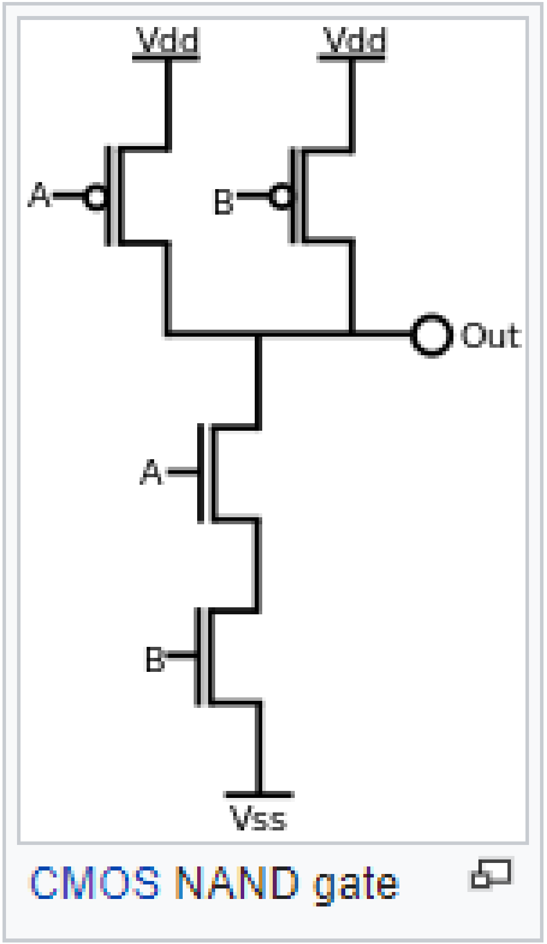
SSI -	**several (less than 10) independent gates**
MSI -	**10 to 200 gates; Perform elementary digital functions; Decoder, adder, register, parity checker, etc**
LSI -	**200 to few thousand gates; Digital subsystem Processor, memory, etc**
VLSI -	**Thousands of gates; Digital system Microprocessor, memory module**

Classification by Technology

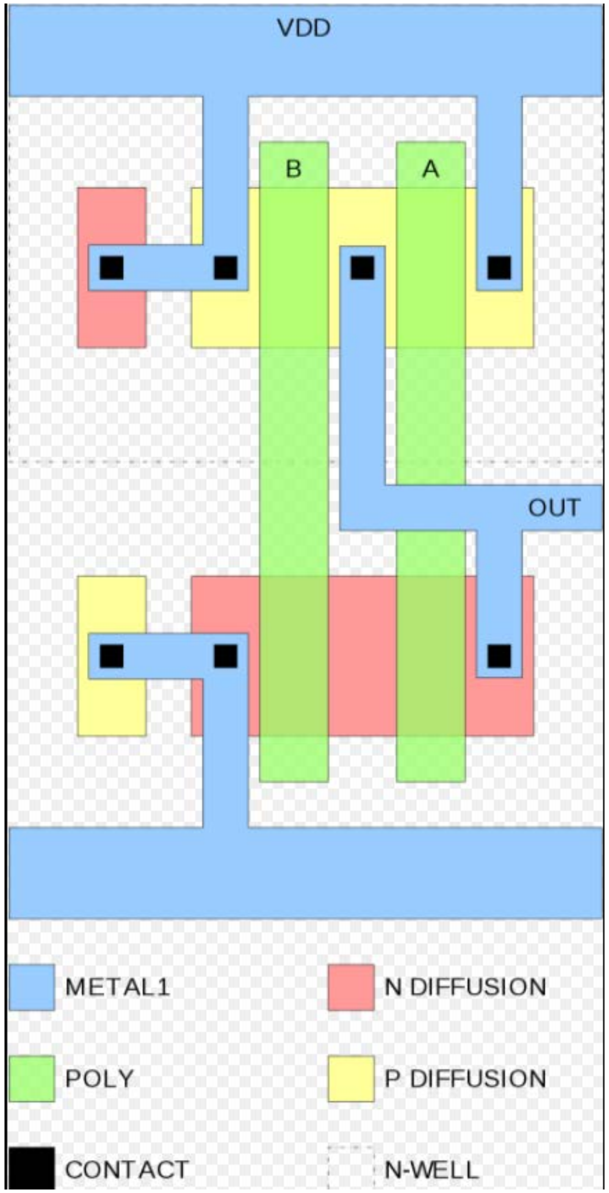
TTL -	**Transistor-Transistor Logic Bipolar transistors NAND**
ECL -	**Emitter-coupled Logic Bipolar transistor NOR**
MOS -	**Metal-Oxide Semiconductor Unipolar transistor High density**
CMOS -	**Complementary MOS Low power consumption. As of 2011, 99% of IC chips are fabricated using CMOS tech**

NAND Gate Example

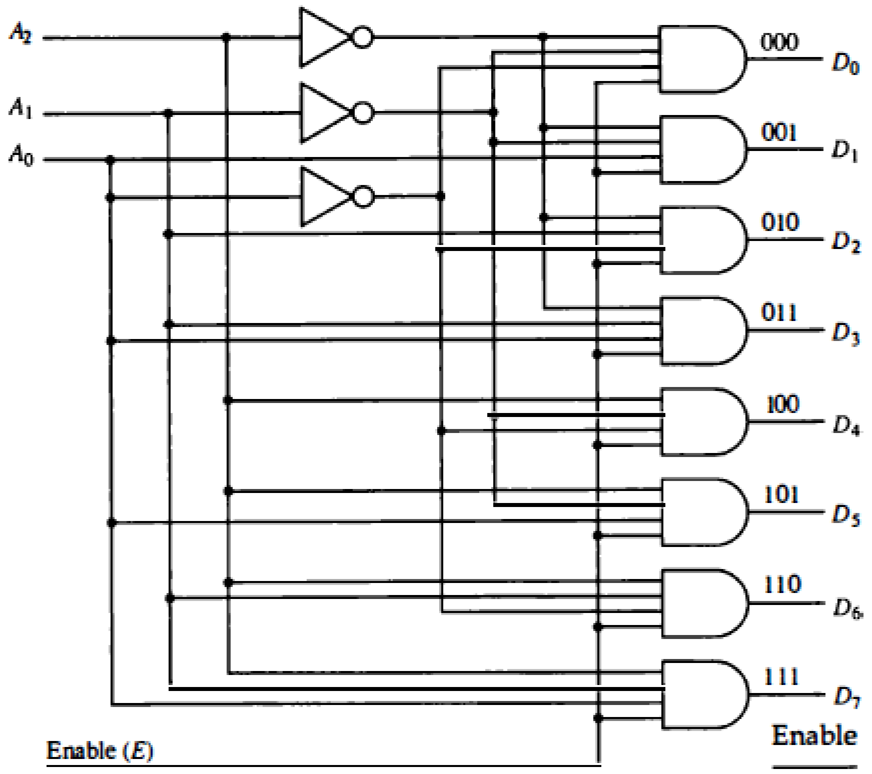
https://en.wikipedia.org/wiki/NAND_gate



CMOS NAND gate

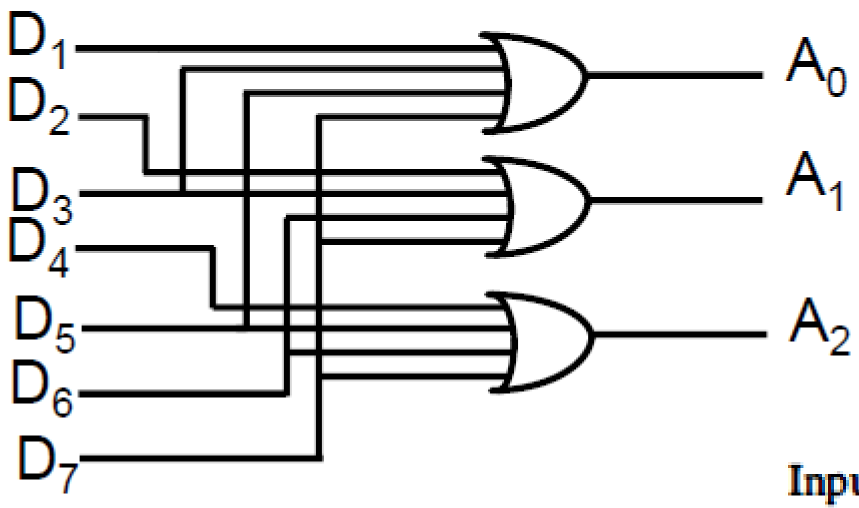


Decoder



Enable	Inputs			Outputs							
E	A_2	A_1	A_0	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
0	×	×	×	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

Encoder

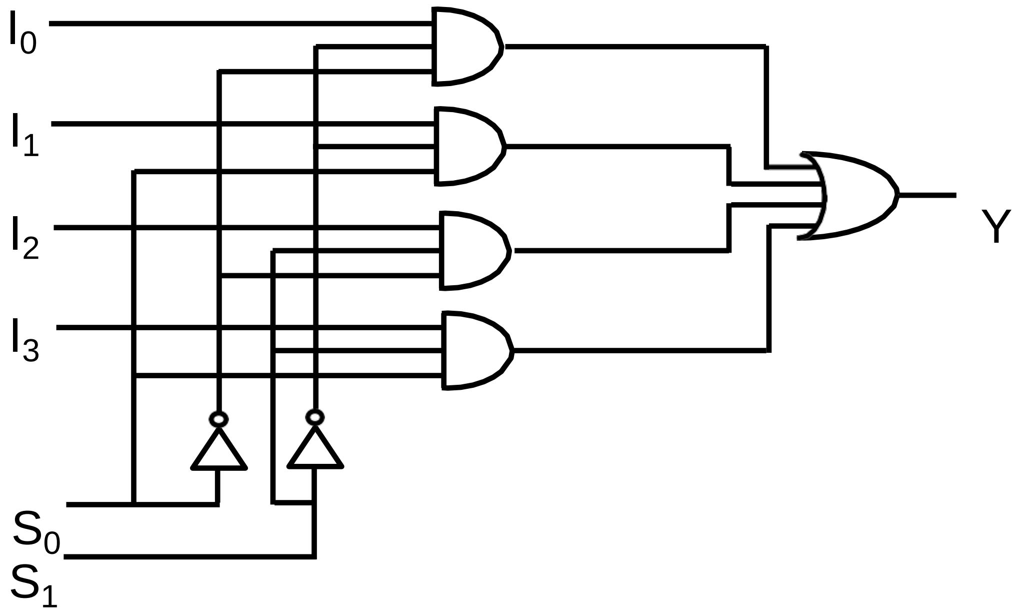


Inputs								Outputs		
D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	A_2	A_1	A_0
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	1	1	0
1	0	0	0	0	0	0	0	1	1	1

MULTIPLEXER (MUX)

4-to-1 Multiplexer

Select		Output
S_1	S_0	Y
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3



Registers

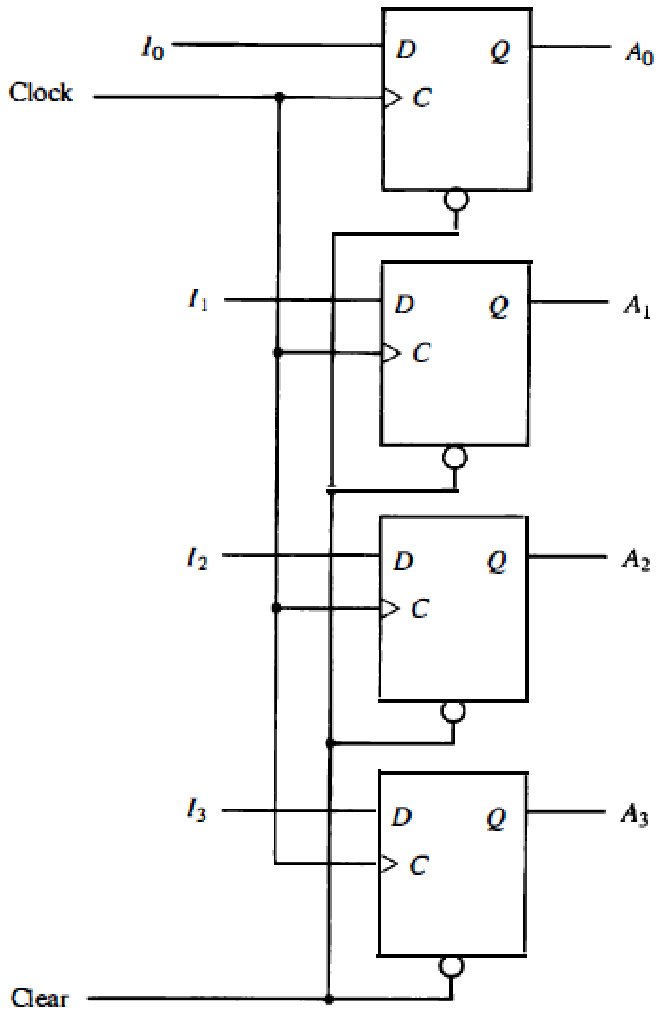


Figure 2-6 4-bit register.

Basic 4-bit Register example

1. 4-bit register requires 4 flip flops
2. The data needs to be clocked in
3. There is also a clear bit to return the registers to 0

Shift Registers- Simple Unidirectional

Figure 2-8 4-bit shift register.

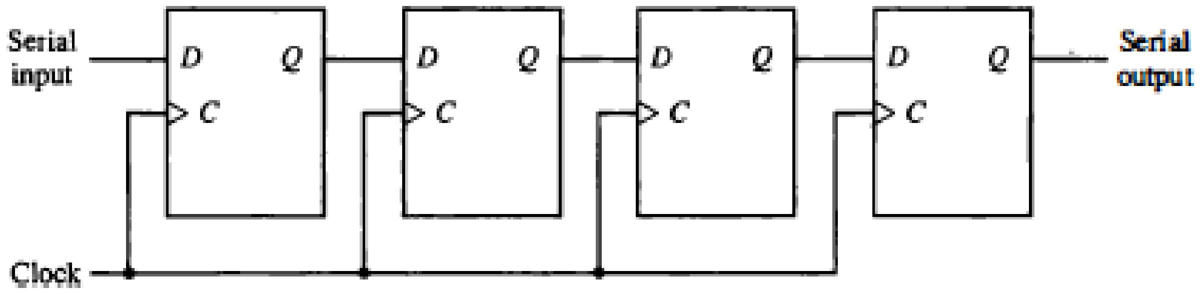


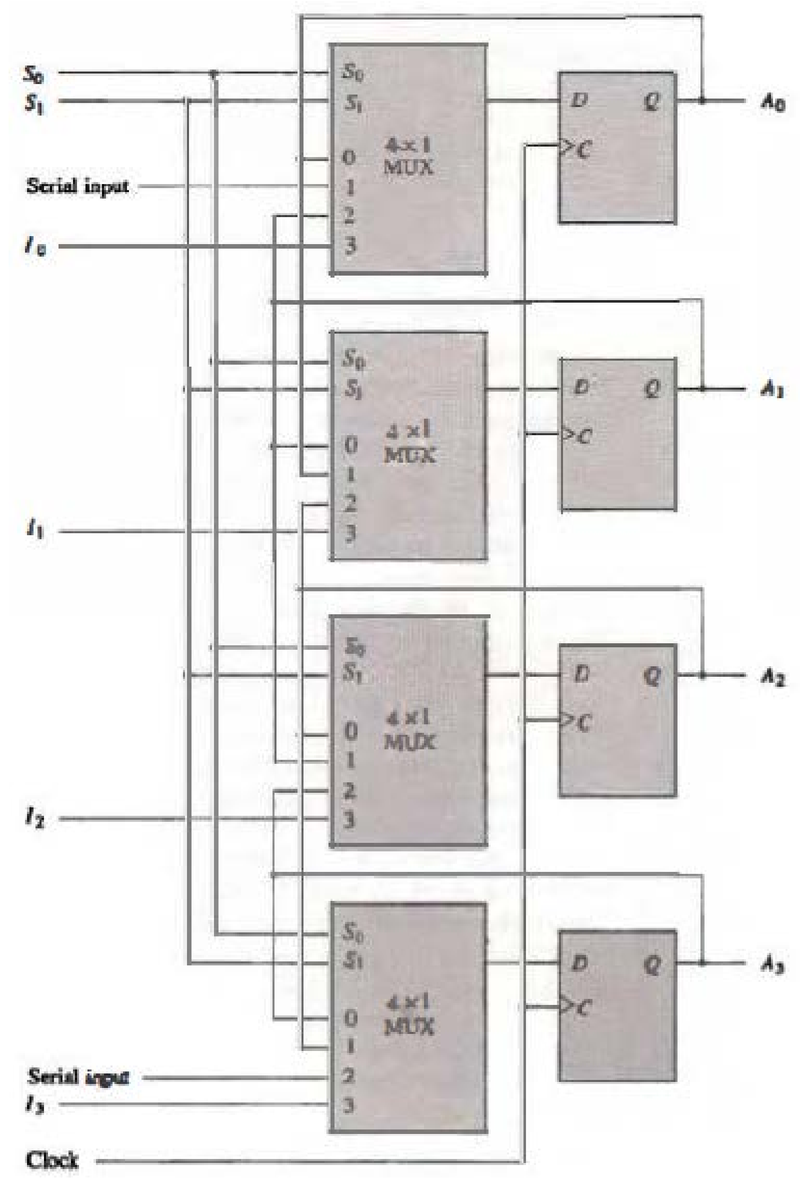
Figure 2-8 shows a simple unidirectional (only shifts one way) shift register.

What functionality is missing to make this more useful?

1. Shifting left/right
2. Parallel loading and reading
3. Control mechanism to leave values unchanged on clock

Next slide shows a more complete shift register

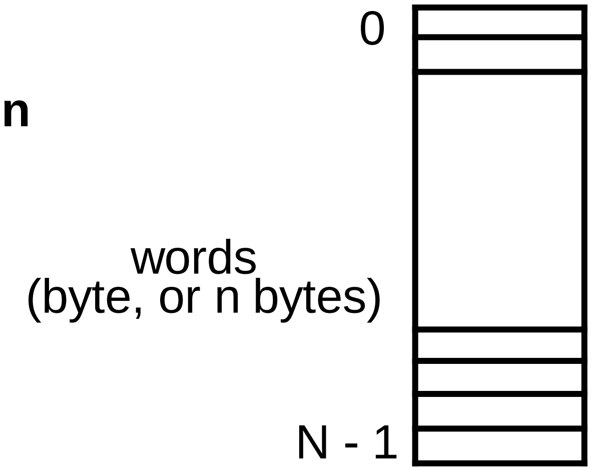
Shift Registers- Bidirectional



Mode control		
S_1	S_0	Register operation
0	0	No change
0	1	Shift right (down)
1	0	Shift left (up)
1	1	Parallel load

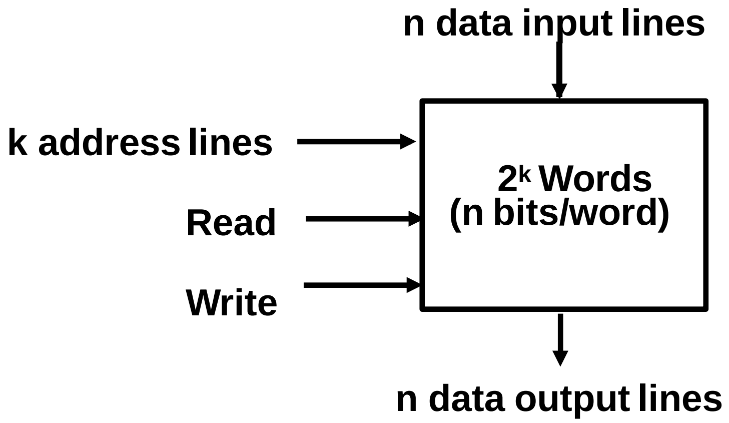
MEMORY COMPONENTS

Logical Organization

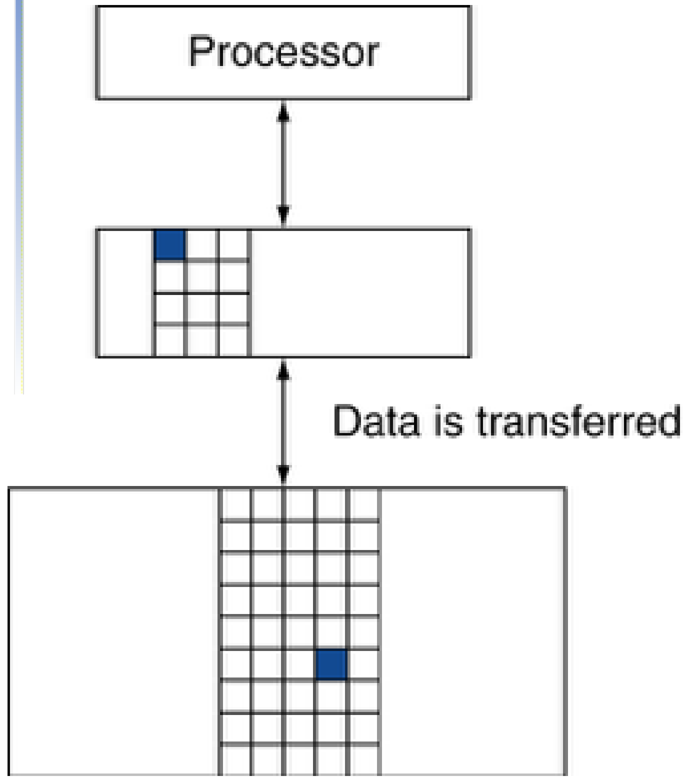


Random Access Memory

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



Memory Hierarchy Levels

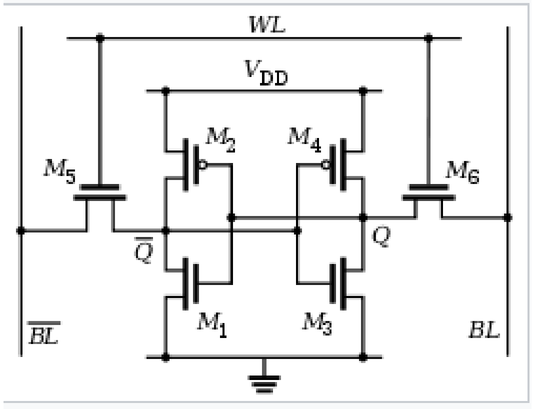


- Block (aka line): unit of copying
  - May be multiple words
- If accessed data is present in upper level
  - Hit: access satisfied by upper level
    - Hit ratio: hits/accesses
- If accessed data is absent
  - Miss: block copied from lower level
    - Time taken: miss penalty
    - Miss ratio: misses/accesses
      = 1 – hit ratio
  - Then accessed data supplied from upper level

SRAM Technology

An integrated circuit with 6-8 transistors per bit to prevent information from being disturbed while read.

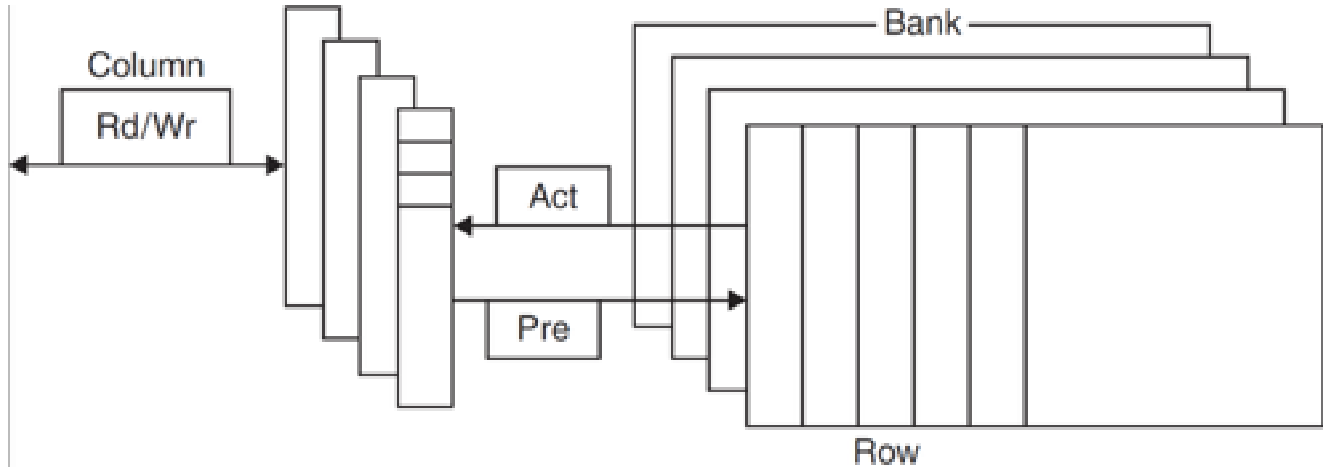
1. No refresh needed
2. Access time is close to the cycle time
3. Only need minimal power to retain value in standby mode



A six-transistor CMOS SRAM cell

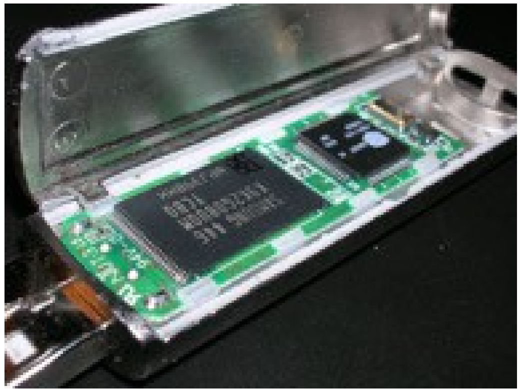
DRAM Technology

- Data stored as a charge in a capacitor
  - Single transistor used to access the charge
  - Must periodically be refreshed
    - Read contents and write back
    - Performed on a DRAM “row”



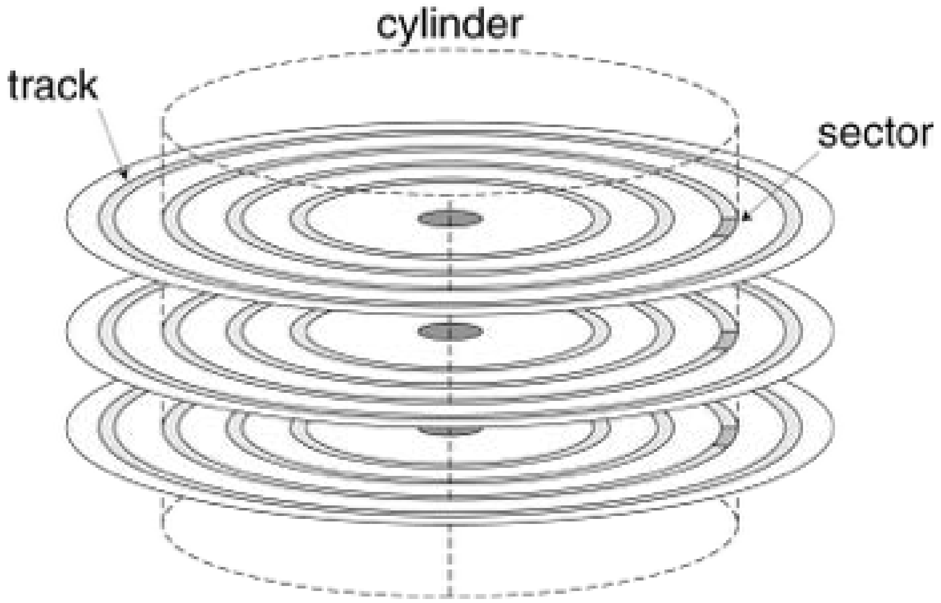
Flash Storage

- Nonvolatile semiconductor storage
  - 100× – 1000× faster than disk
  - Smaller, lower power, more robust
  - But more $/GB (between disk and DRAM)



Disk Storage

- Nonvolatile, rotating magnetic storage



Memory Technology

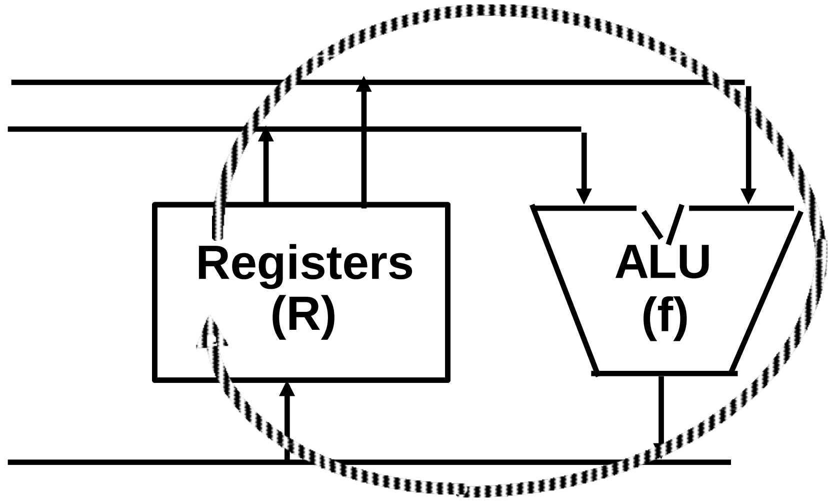
- Static RAM (SRAM)
  - 0.5ns – 2.5ns, $500 – $1000 per GB
- Dynamic RAM (DRAM)
  - 50ns – 70ns, $10 – $20 per GB
- Flash
  - 5,000-50,000ns - $0.75 - $1.00 per GB
- Magnetic disk
  - 5ms – 20ms, $0.05 – $0.10 per GB
- Ideal memory
  - Access time of SRAM
  - Capacity and cost/GB of disk

REGISTER TRANSFER AND MICROOPERATIONS

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

MICROOPERATION

An elementary operation performed during one clock pulse, on the information stored in one or more registers



1 clock cycle

$R \leftarrow f(R, R)$

f could be: shift, count, clear, load, add, etc…

REGISTER TRANSFER LANGUAGE

Definition of the (internal) organization of a computer

- **Set of registers and their functions**
- **Microoperations Set**

  Set of allowable microoperations provided by the organization of the computer
- **Control signals that initiate the sequence of microoperations**

For any function of the computer, a sequence of microoperations is used to describe it

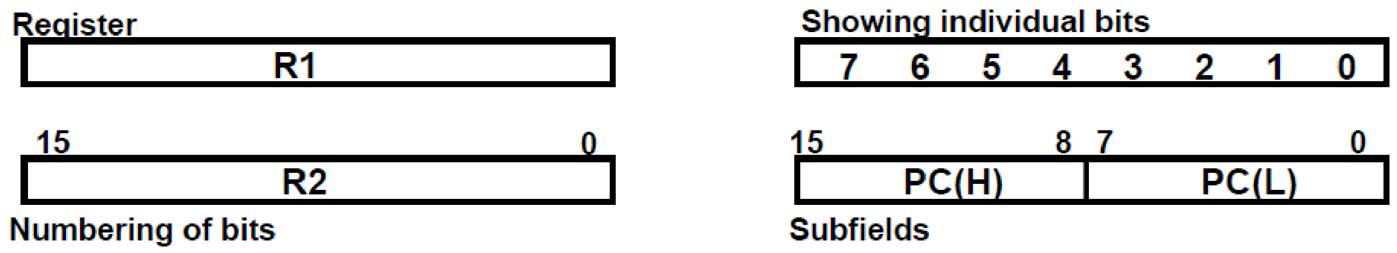
----> *Register transfer language*

- **A symbolic language**
- **A convenient tool for describing the internal organization of digital computers**
- **Can also be used to facilitate the design process of digital systems.**

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



Representation of a transfer(parallel)

$R2 \leftarrow R1$

A simultaneous transfer of all bits from the source to the destination register, during one clock pulse

Representation of a controlled(conditional) transfer

$P: \quad R2 \leftarrow R1$

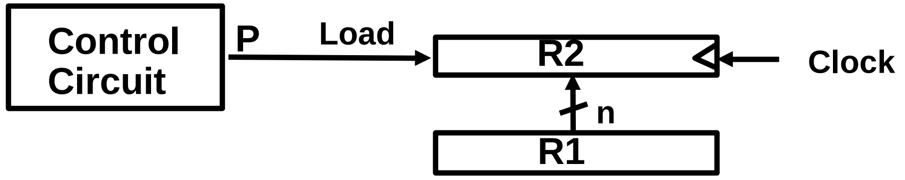
A binary condition(p=1) which determines when the transfer is to occur

If $(p=1)$ then $(R2 \leftarrow R1)$

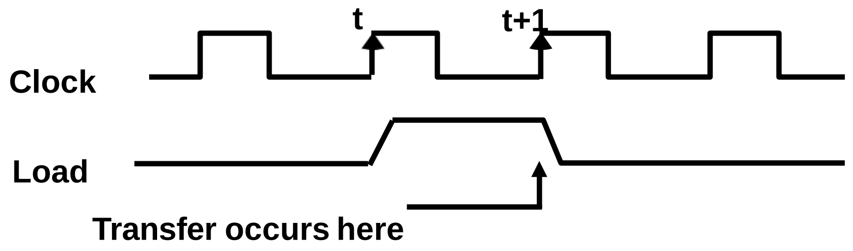
HARDWARE IMPLEMENTATION OF CONTROLLED TRANSFERS

Implementation of controlled transfer
P: R2 ← R1

Block diagram



Timing diagram



Basic Symbols for Register Transfers

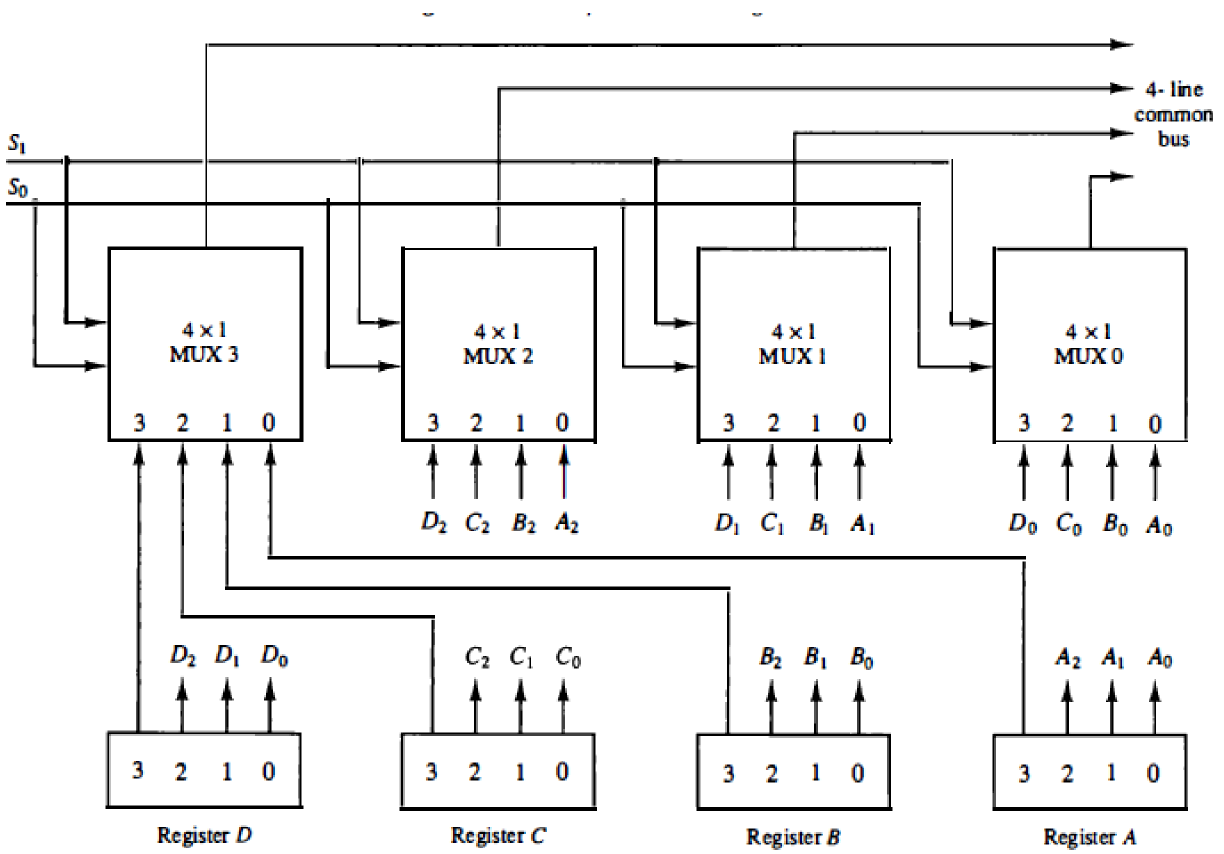
Symbols	Description	Meaning
Capital letters and numerals	Denotes a register	MAR, R2
Parentheses ( )	Denotes a part of a register	R2(0-7), R2(L)
Arrow ←	Denotes transfer of information	R2 ← R1
Colon :	Denotes termination of control function	P:
Comma ,	Separates two micro-operations	A ← B, B ← A

BUS AND MEMORY TRANSFER

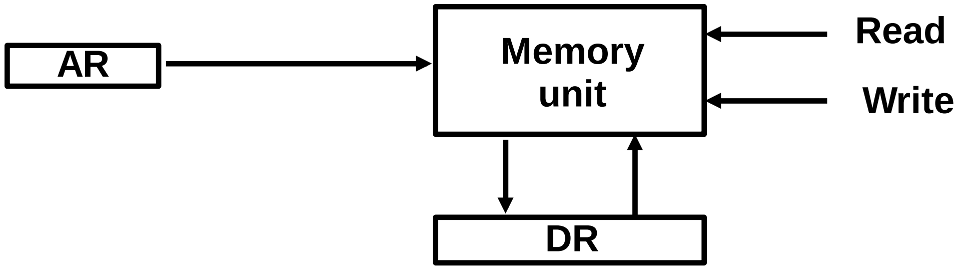
Bus is a path(of a group of wires) over which information is transferred, from any of several sources to any of several destinations.

From a register to bus: BUS <- A or A <- BUS

S_1	S_0	Register selected
0	0	A
0	1	B
1	0	C
1	1	D



MEMORY TRANSFERS



Memory *read* micro-op: DR ← M **(DR ← M[AR])**
Memory *write* micro-op: M ← DR **(M[AR] ← DR)**

Summary of Register Transfer Microoperations

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

ARITHMETIC MICROOPERATIONS

Four types of microoperations

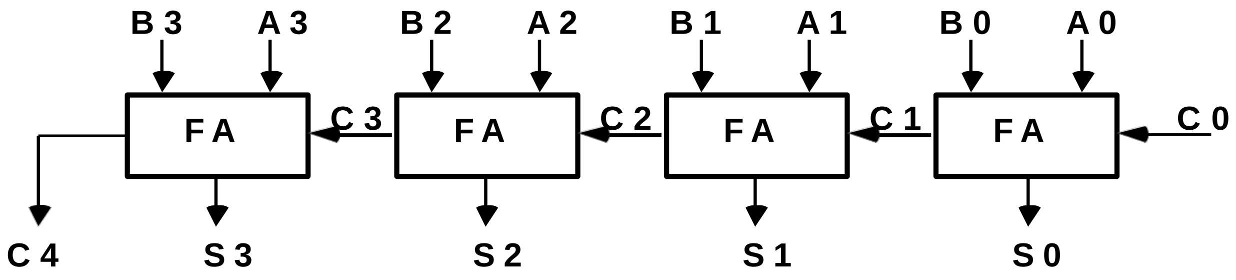
- **Register transfer microoperations**
- **Arithmetic microoperations**
- **Logic microoperations**
- **Shift microoperations**

*** Summary of Arithmetic Micro-Operations**

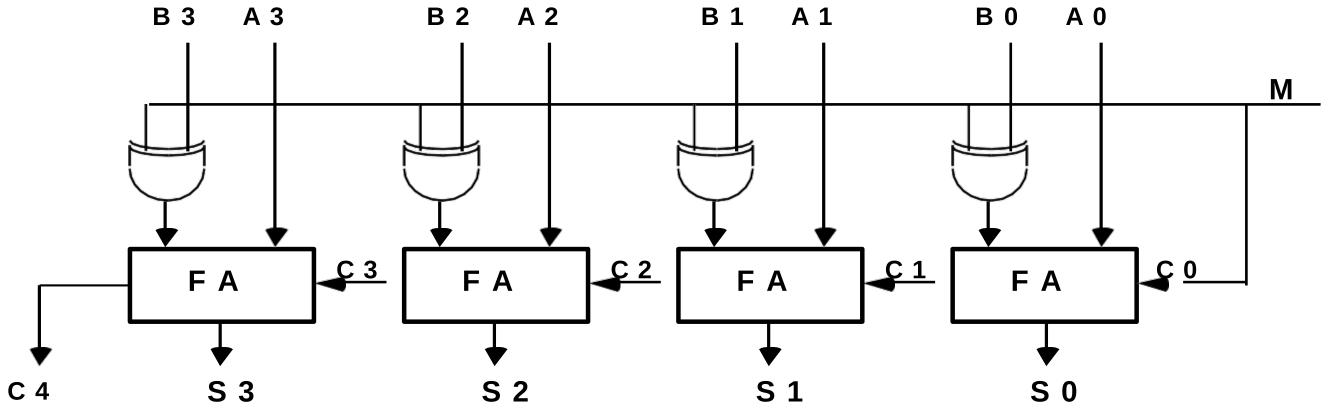
$R3 \leftarrow R1 + R2$	Contents of R1 plus R2 transferred to R3
$R3 \leftarrow R1 - R2$	Contents of R1 minus R2 transferred to R3
$R2 \leftarrow R2'$	Complement the contents of R2
$R2 \leftarrow R2' + 1$	2's complement the contents of R2 (negate)
$R3 \leftarrow R1 + R2' + 1$	subtraction
$R1 \leftarrow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

BINARY ADDER

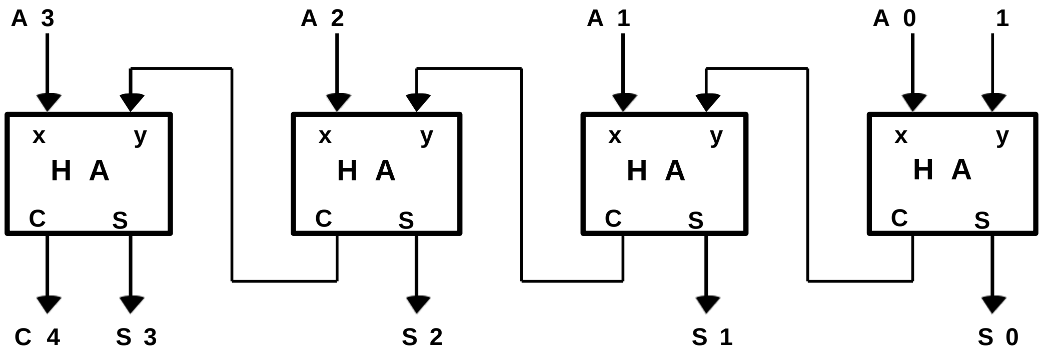
Binary Adder



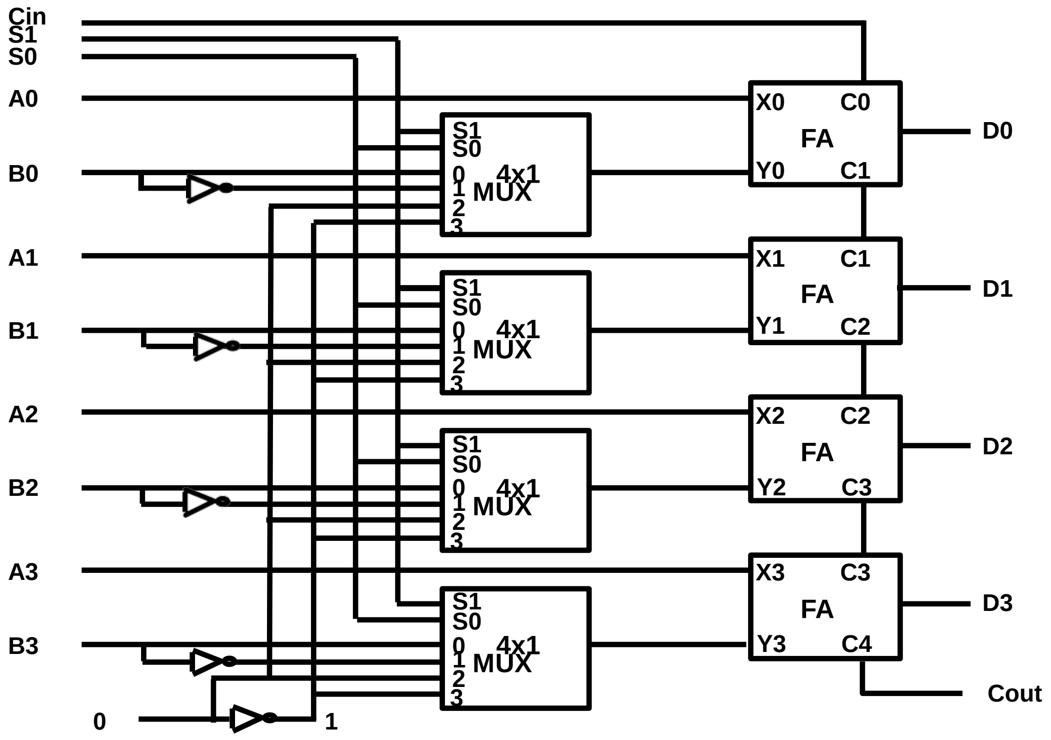
Binary Adder-Subtractor



Binary Incrementer



ARITHMETIC CIRCUIT



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

LOGIC MICROOPERATIONS

Specify binary operations on the strings of bits in registers.

- **useful for bit manipulations on binary data**

AND: Mask out certain group of bits
OR : Merge binary or character data

- **useful for making logical decisions based on the bit value**

Applications

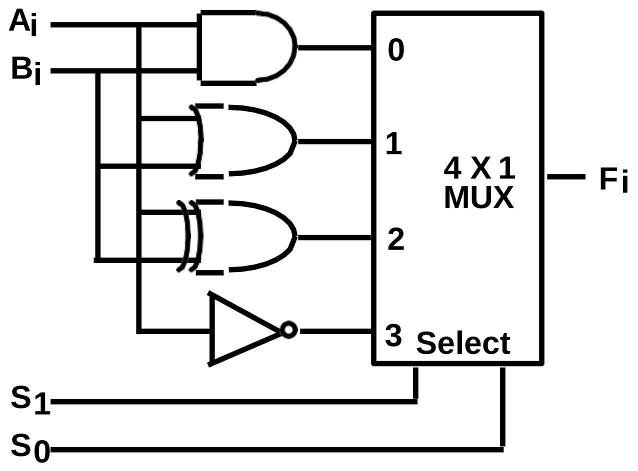
Manipulating individual bits or a field(portion) of a word in a register

LIST OF LOGIC MICROOPERATIONS

- **List of Logic Micro-Operations**
  - **16 different logic operations with 2 binary vars.**
  - **n binary vars -> 2^{2^n} functions**
- **Truth tables for 16 functions of 2 variables and the corresponding 16 logic micro-operations**

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

HARDWARE IMPLEMENTATION OF LOGIC MICROOPERATIONS



Function table

S1	S0	Output	μ-operation
0	0	$F = A \wedge B$	AND
0	1	$F = A \vee B$	OR
1	0	$F = A \oplus B$	XOR
1	1	F = A’	Complement

SHIFT MICROOPERATIONS

Shifts

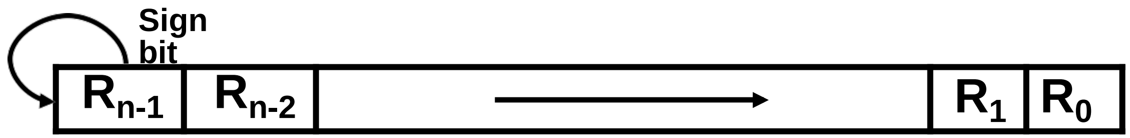
- *Logical shift* : shift in a 0 into the extreme flip-flop
- *Circular shift* : circulates the bits of the register around the two ends
- *Arithmetic shift* : shifts a signed number (shift with sign extension)

Left shift -> multiplied by 2

Right shift -> divided by 2

Arithmetic shifts for signed binary numbers

- Arithmetic shift-right

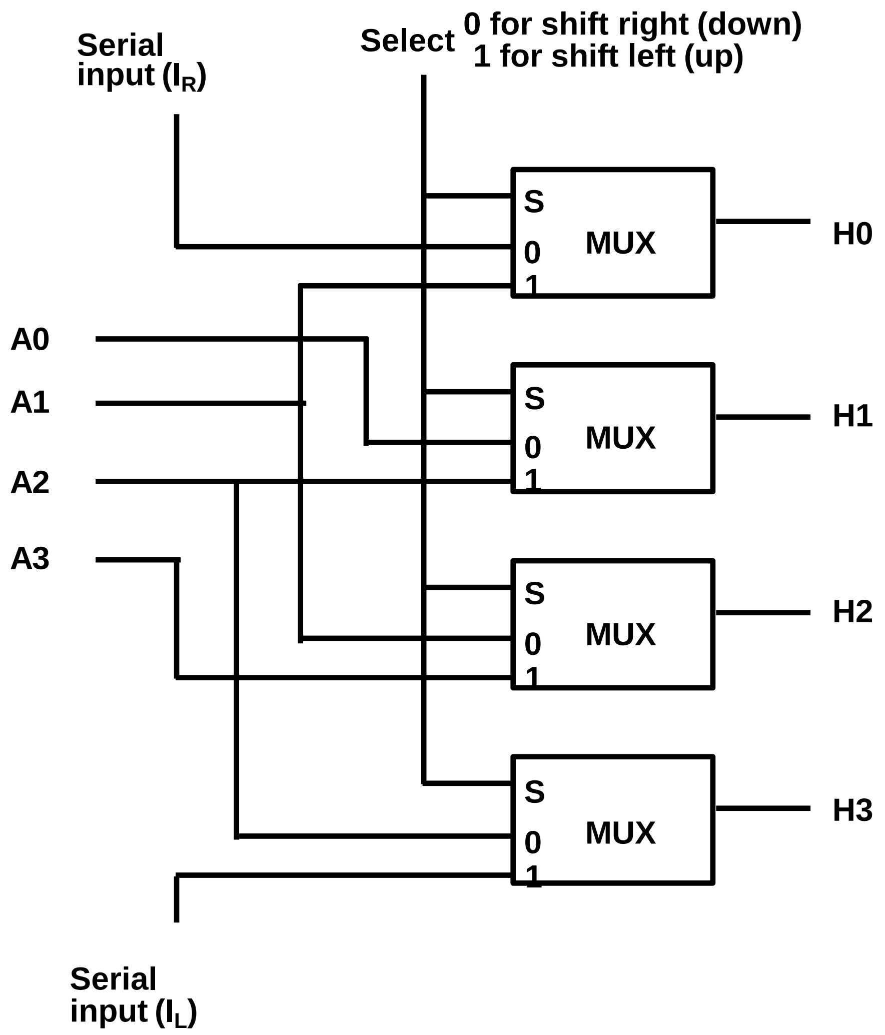


- Arithmetic shift-left Overflow $V = R_{n-1} \neq R_{n-2}$

Shift Micro-Operations

Symbol	*Description*
$R \leftarrow$ shl R	Shift-left register R
$R \leftarrow$ shr R	Shift-right register R
$R \leftarrow$ cil R	Circular shift-left register R
$R \leftarrow$ cir R	Circular right-shift register R
$R \leftarrow$ ashl R	Arithmetic shift-left register R
$R \leftarrow$ ashr R	Arithmetic shift-right register R

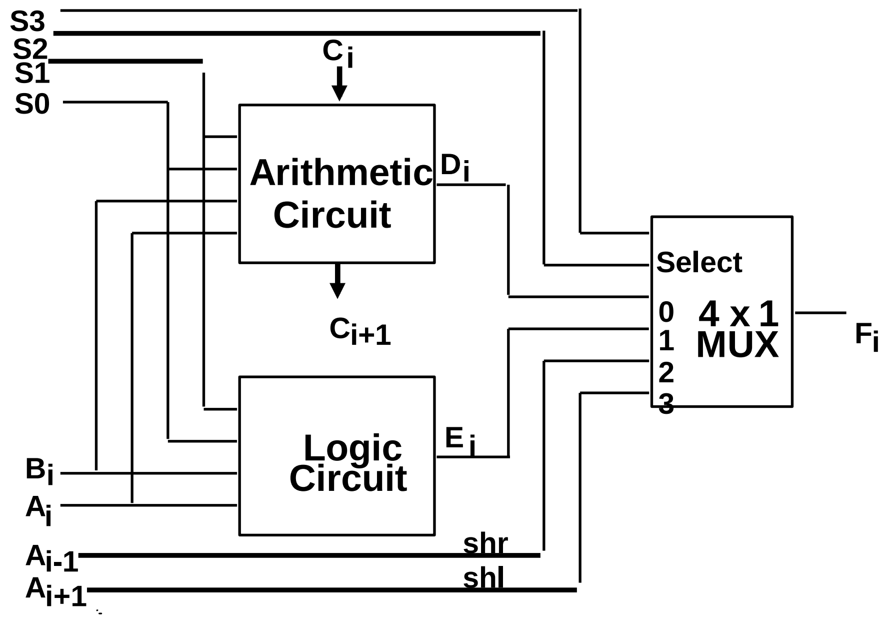
HARDWARE IMPLEMENTATION OF SHIFT MICROOPERATIONS



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

ARITHMETIC LOGIC SHIFT UNIT



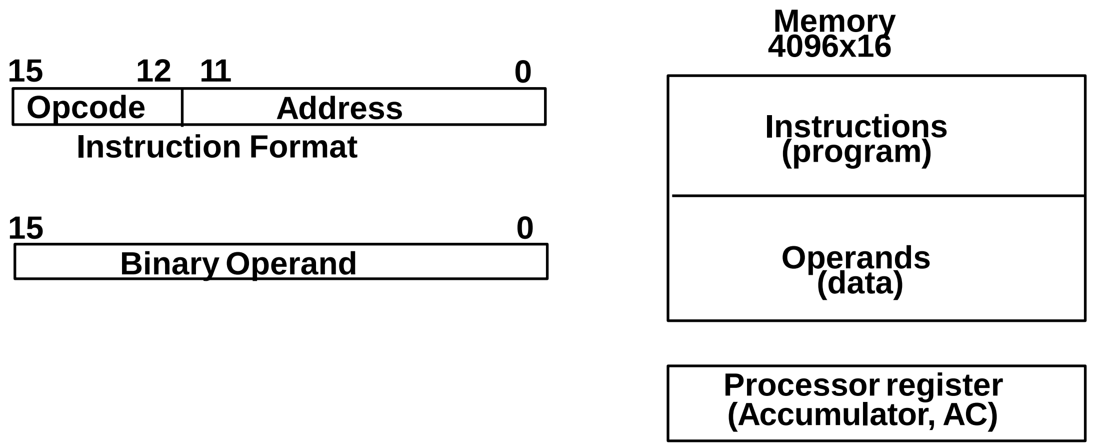
S3	S2	S1	S0	Cin	Operation	Function
0	0	0	0	0	F = A	Transfer A
0	0	0	0	1	F = A + 1	Increment A
0	0	0	1	0	F = A + B	Addition
0	0	0	1	1	F = A + B + 1	Add with carry
0	0	1	0	0	F = A + B’	Subtract with borrow
0	0	1	0	1	F = A + B’+ 1	Subtraction
0	0	1	1	0	F = A - 1	Decrement A
0	0	1	1	1	F = A	TransferA
0	1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

BASIC COMPUTER ORGANIZATION AND DESIGN

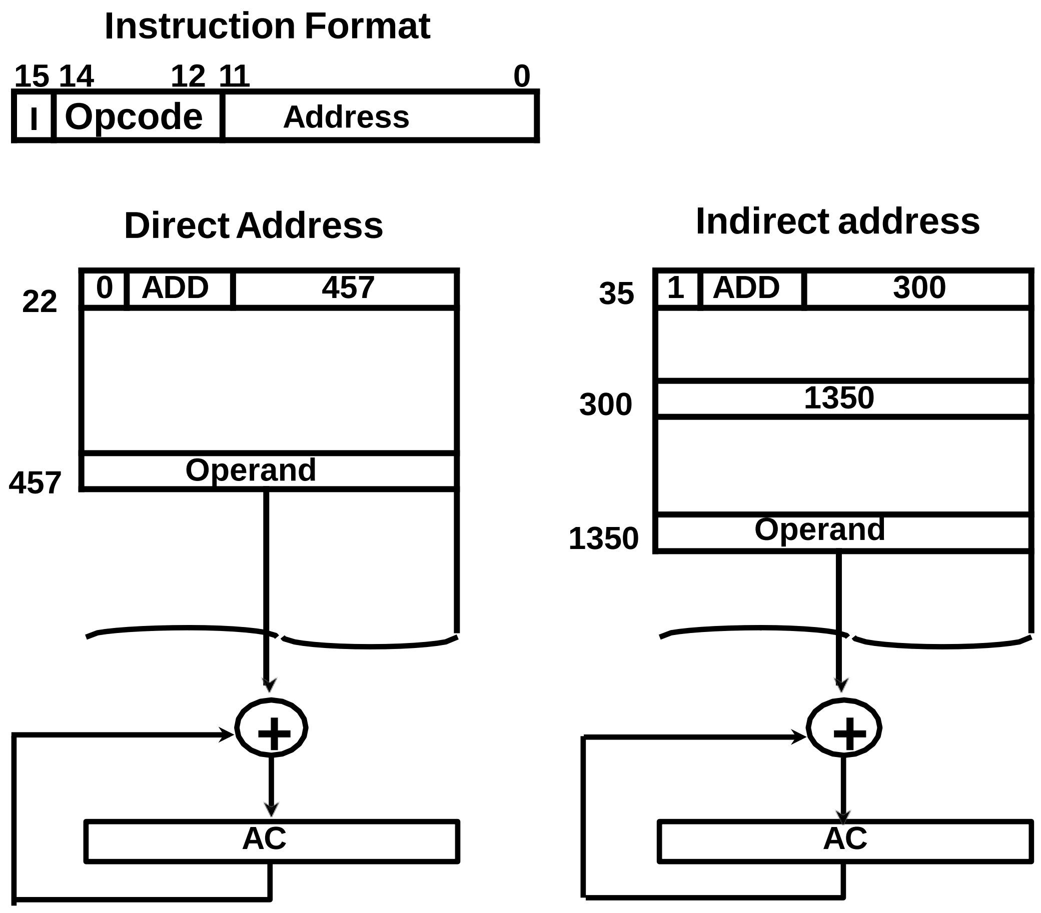
- **Instruction Codes**
- **Computer Registers**
- **Computer Instructions**
- **Timing and Control**
- **Instruction Cycle**
- **Memory Reference Instructions**
- **Input-Output and Interrupt**
- **Complete Computer Description**
- **Design of Basic Computer**
- **Design of Accumulator Logic**

INSTRUCTION CODES

- **Program:**
  A set of instructions that specify the *operations*, *operands*, and the *sequence* by which processing has to occur.
- **Instruction Code:**
  A group of bits that tell the computer to *perform a specific operation*



INDIRECT ADDRESS

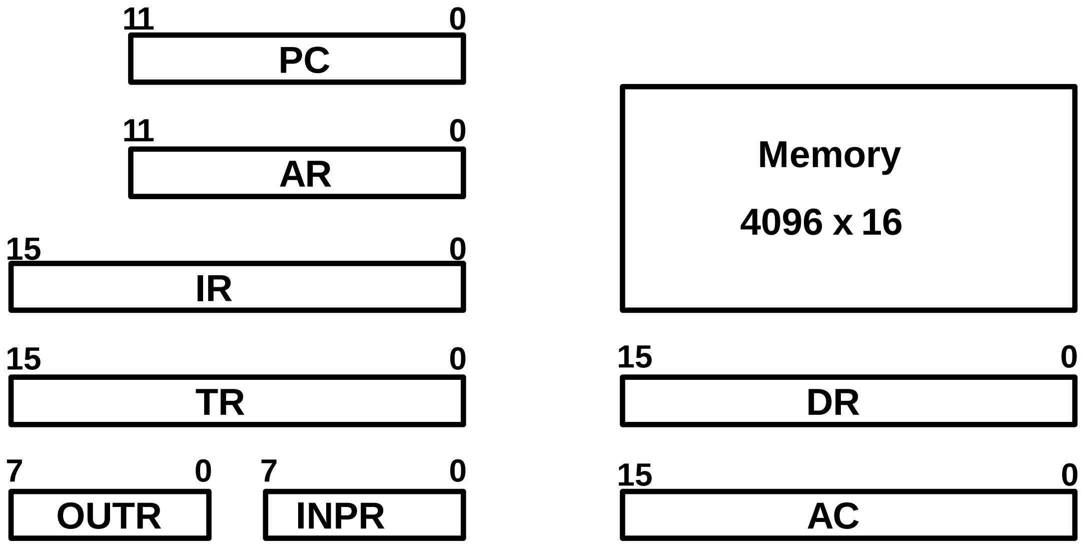


Effective Address(EFA, EA)

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

COMPUTER REGISTERS

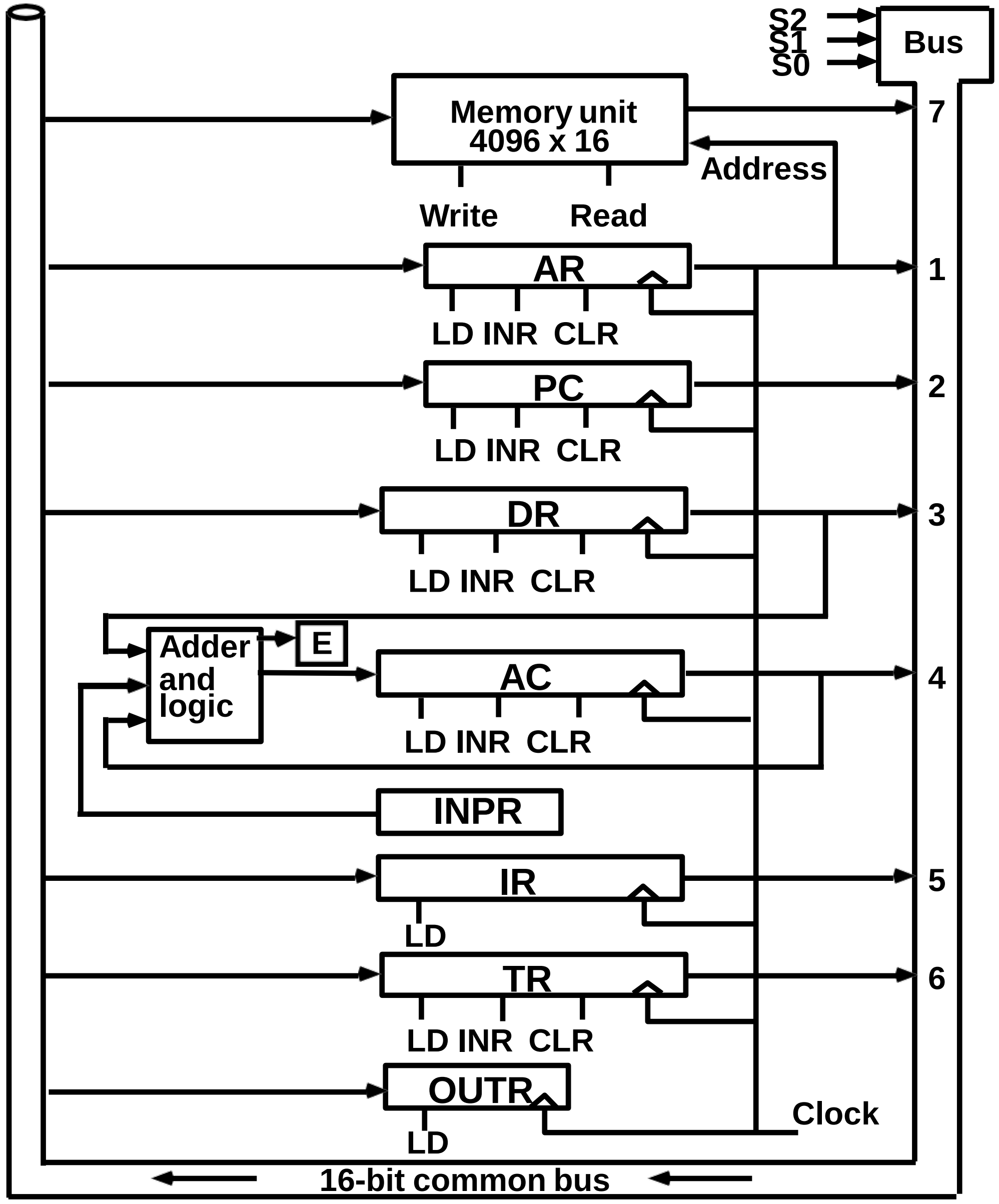
Registers in the Basic Computer (BC)



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

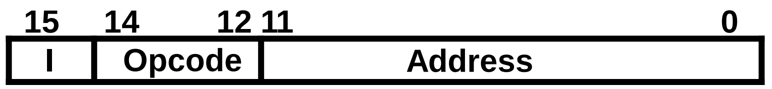
COMMON BUS SYSTEM



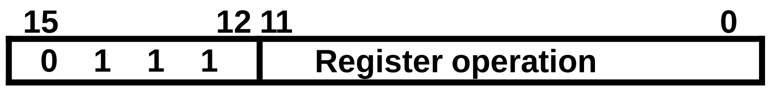
COMPUTER(BC) INSTRUCTIONS

Basic Computer Instruction code format

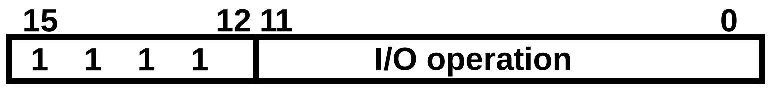
Memory-Reference Instructions (OP-code = 000 ~ 110)



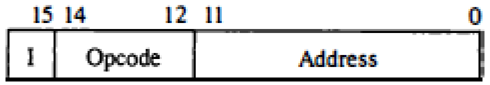
Register-Reference Instructions (OP-code = 111, I = 0)



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



BASIC COMPUTER INSTRUCTIONS

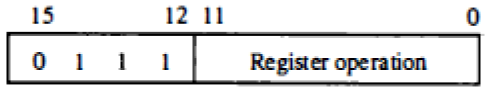


(a) Memory – reference instruction

(Opcode = 000 through 110)

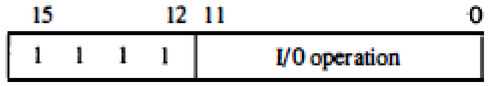
I = 0 for Direct

I = 1 for Indirect



(b) Register – reference instruction

(Opcode = 111, *I* = 0)



(c) Input – output instruction

(Opcode = 111, *I* = 1)

Symbol	***Hex Code***		***Description***
	I = 0	***I = 1***	
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

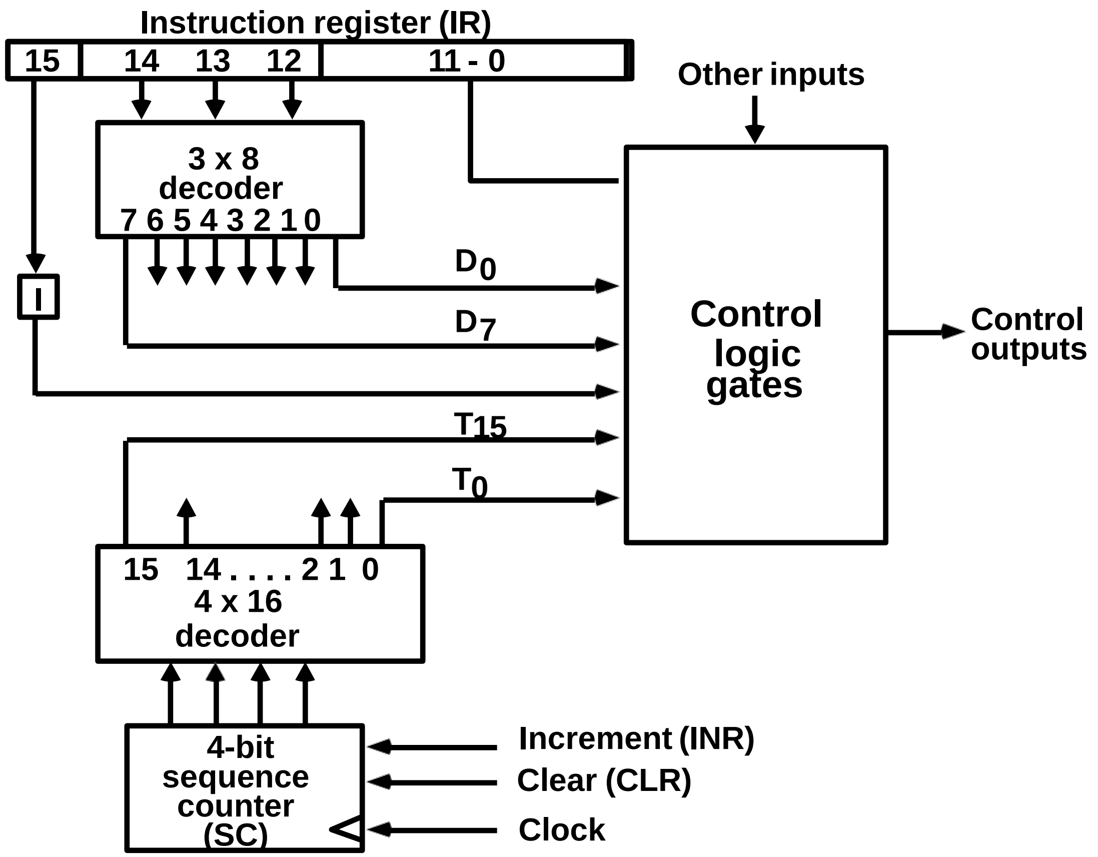
A computer should have a set of instructions so that the user can construct machine language programs to evaluate any function that is known to be computable.

Instruction Types

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

TIMING AND CONTROL

Control unit of basic computer



Control unit implementation

Hardwired Implementation

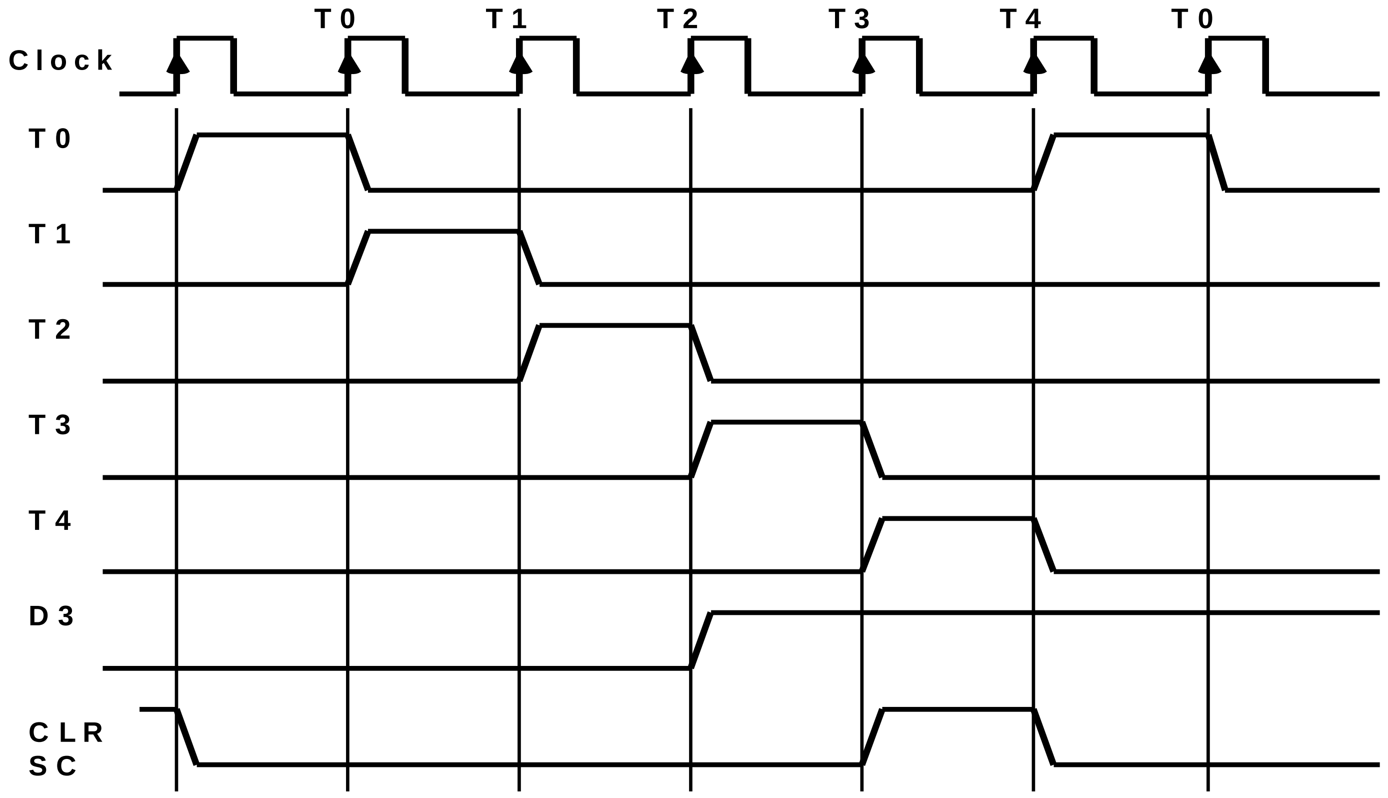
Microprogrammed Implementation

TIMING SIGNALS

- **Generated by 4-bit sequence counter and 4x16 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 $\leftarrow$ 0

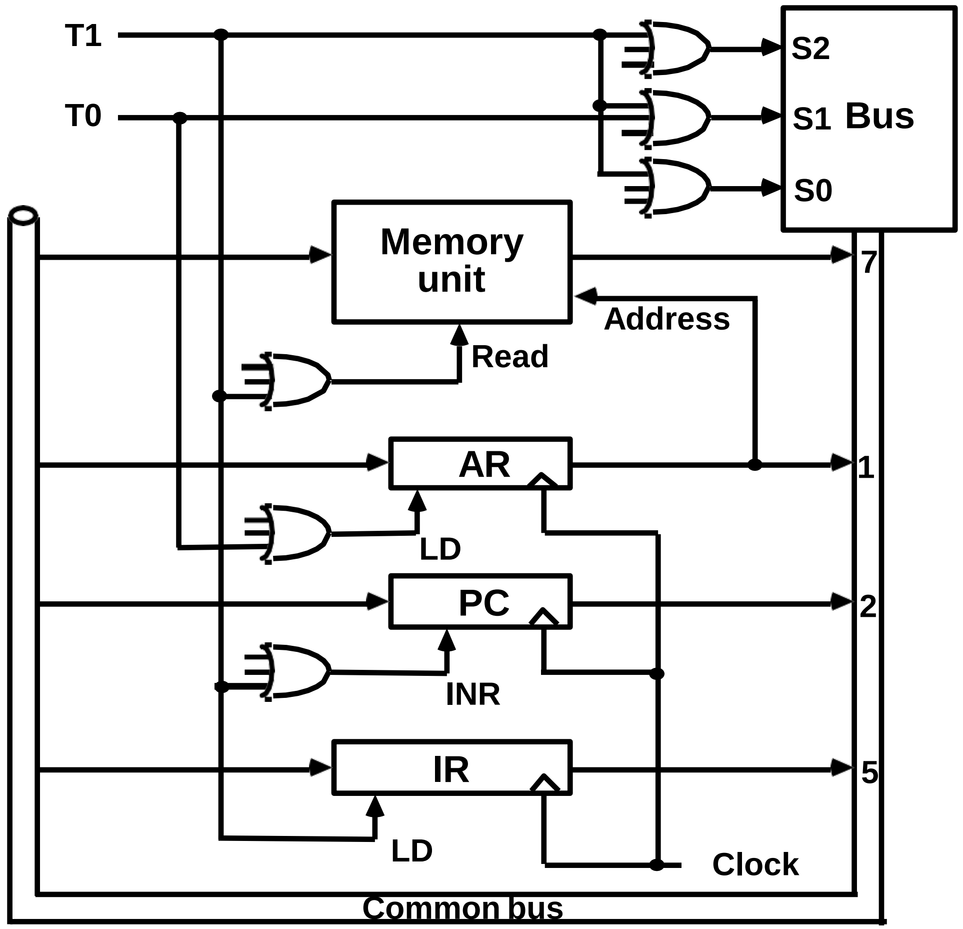


INSTRUCTION CYCLE

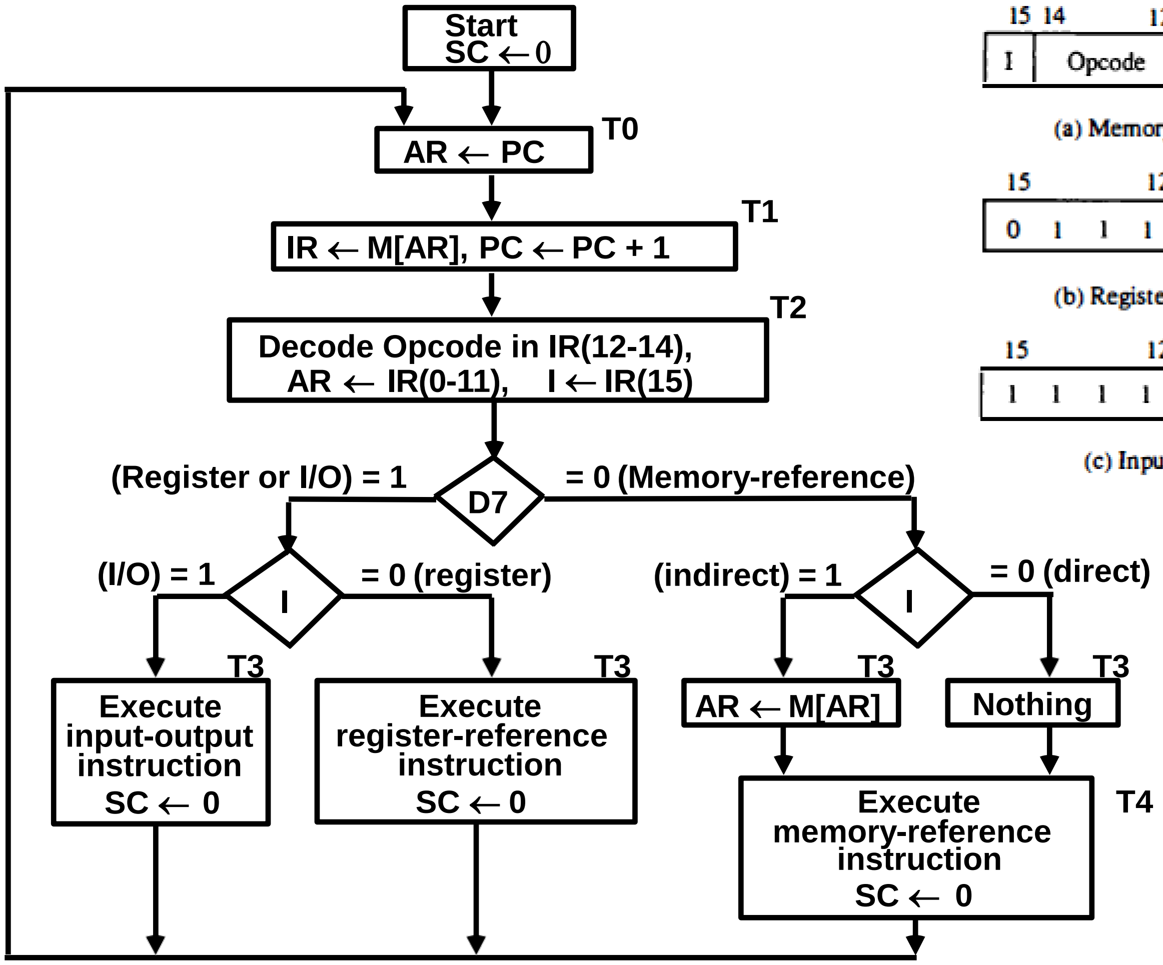
BC Instruction cycle: [Fetch Decode [Indirect] Execute]*

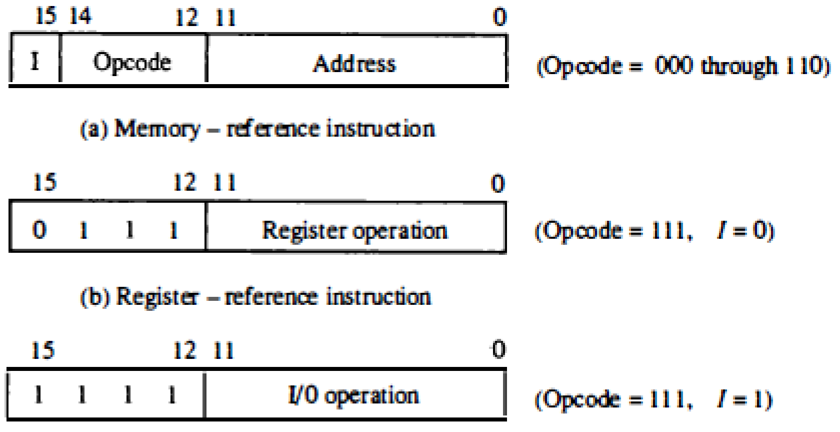
- **Fetch and Decode**

T0: AR ← PC (S0S1S2=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)



DETERMINE THE TYPE OF INSTRUCTION


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)
D7
(Register or I/O) = 1
= 0 (Memory-reference)
I
(I/O) = 1
= 0 (register)
I
(indirect) = 1
= 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



(a) Memory – reference instruction

(b) Register – reference instruction

(c) Input – output instruction

Symbol	Hexadecimal code $I = 0$	Hexadecimal code $I = 1$	Description
AND	0xxx	8xxx	AND memory word to *AC*
ADD	1xxx	9xxx	Add memory word to *AC*
LDA	2xxx	Axxx	Load memory word to *AC*
STA	3xxx	Bxxx	Store content of *AC* in 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 instruction if *AC* positive
SNA	7008		Skip next instruction if *AC* negative
SZA	7004		Skip next instruction if *AC* zero
SZE	7002		Skip next instruction if *E* is 0
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

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 INSTRUCTIONS

Register Reference Instructions are identified when

- **$D_7 = 1$, $I = 0$**
- **Register Ref. Instr. is specified in $b_0 \sim b_{11}$ of IR**
- **Execution starts with timing signal T_3**

$r = D_7 I' T_3$ => Register Reference Instruction
$B_i = IR(i)$, i=0,1,2,...,11

	r:	$SC \leftarrow 0$
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 T3 when I = 1
- Memory cycle is assumed to be short enough to complete in a CPU cycle
- The execution of MR Instruction starts with T_4 :

AND to AC

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

ADD to AC

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

MEMORY REFERENCE INSTRUCTIONS

LDA: Load to AC

D_2T_4: $DR \leftarrow M[AR]$

D_2T_5: $AC \leftarrow DR, SC \leftarrow 0$

STA: Store AC

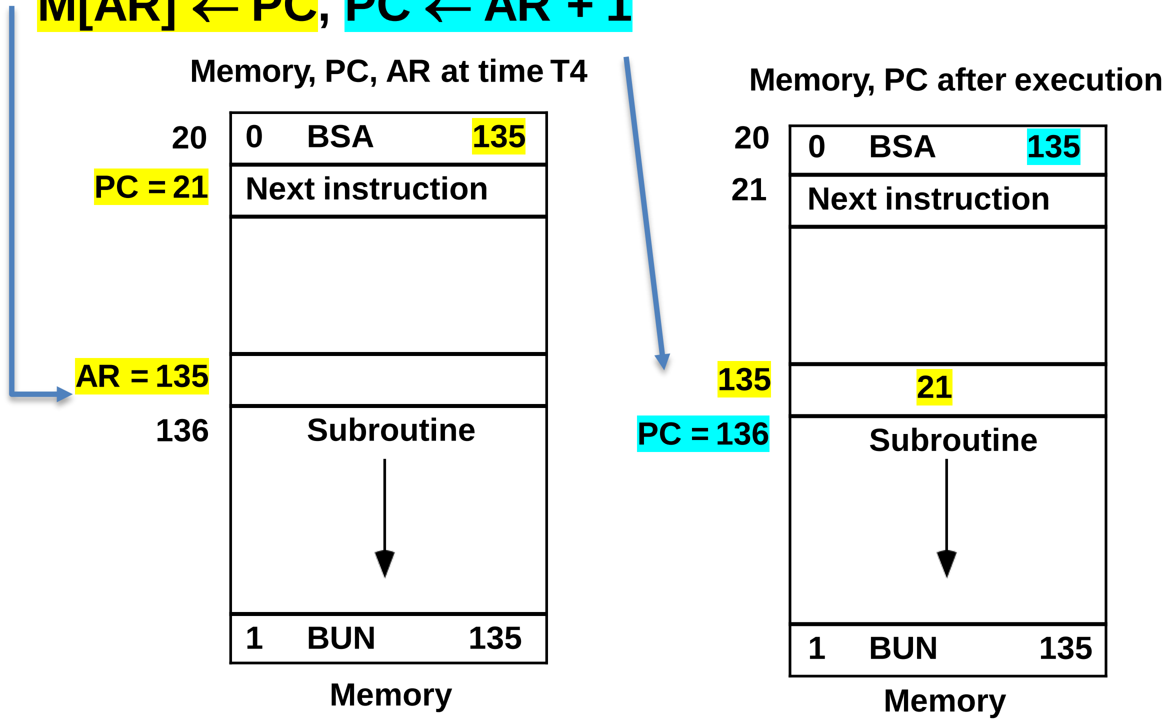
D_3T_4: $M[AR] \leftarrow AC, SC \leftarrow 0$

BUN: Branch Unconditionally

D_4T_4: $PC \leftarrow AR, SC \leftarrow 0$

BSA: Branch and Save Return Address (T4. T5 on next slide)

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



MEMORY REFERENCE INSTRUCTIONS

BSA:

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

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

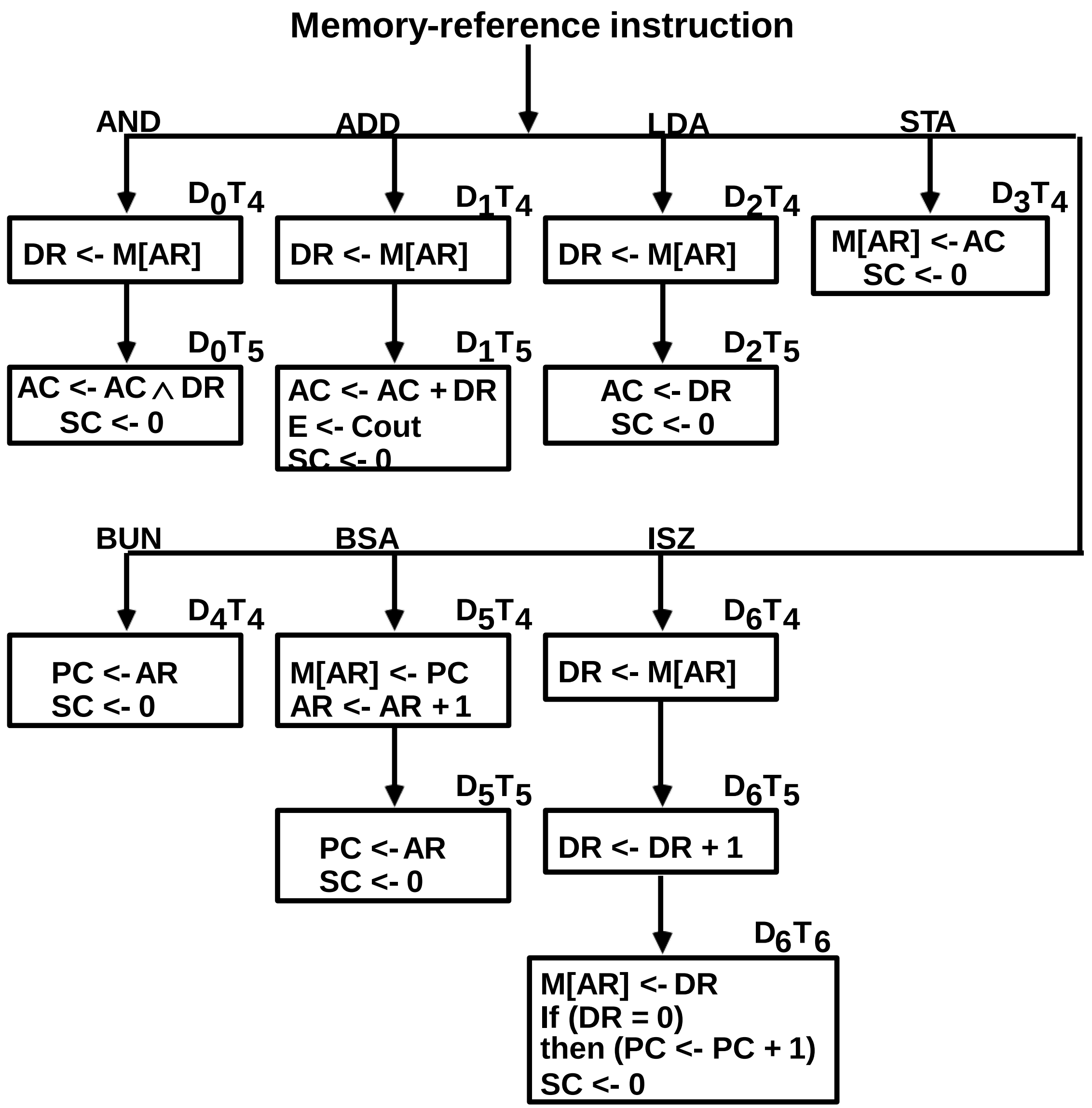
ISZ: Increment and Skip-if-Zero

D_6T_4: $DR \leftarrow M[AR]$

D_6T_5: $DR \leftarrow DR + 1$

D_6T_6: $M[AR] \leftarrow DR$, if $(DR = 0)$ then $(PC \leftarrow PC + 1)$, $SC \leftarrow 0$

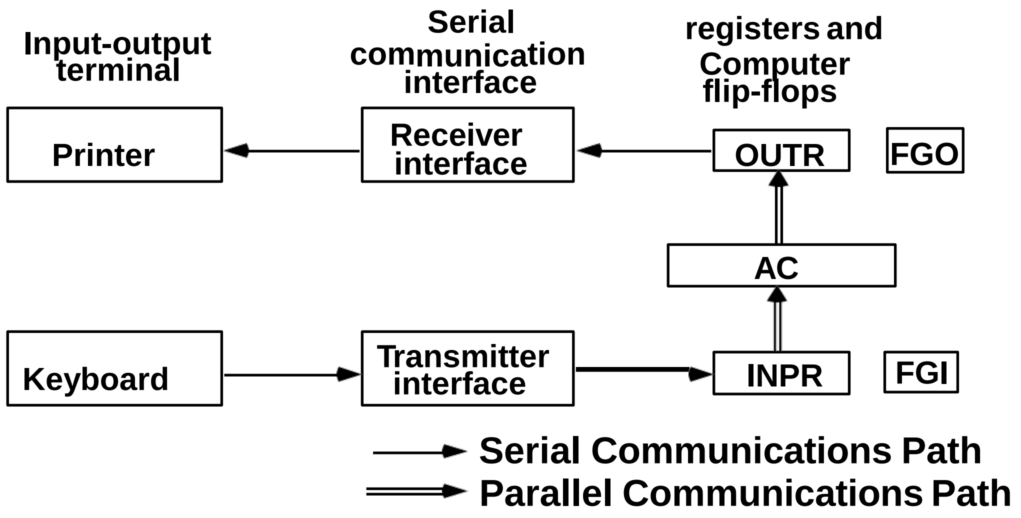
FLOWCHART FOR MEMORY REFERENCE INSTRUCTIONS



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

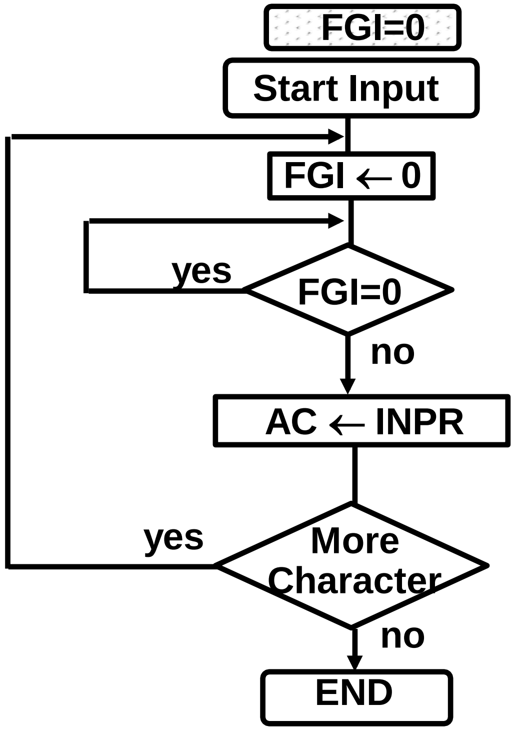
- The terminal sends and receives serial information
- The serial info. from the keyboard is shifted into INPR
- The serial info. for the printer is stored in the OUTR
- INPR and OUTR communicate with the terminal serially and with the AC in parallel.
- The flags are needed to synchronize the timing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

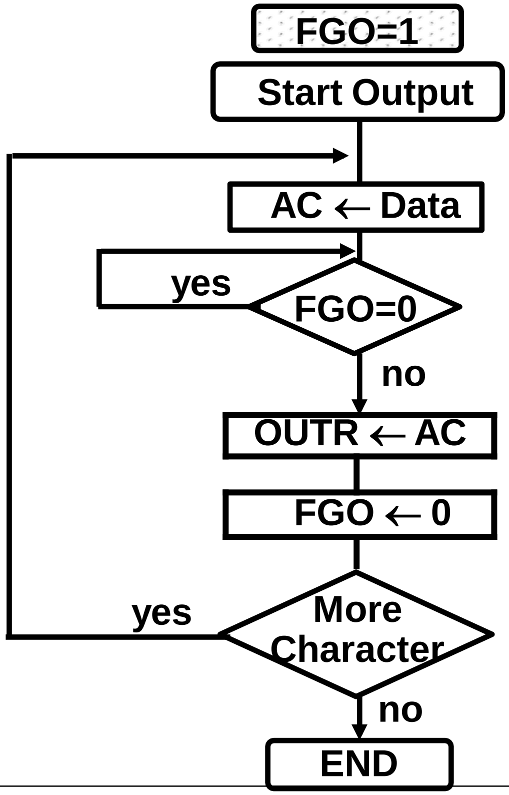
PROGRAM CONTROLLED DATA TRANSFER

-- CPU --

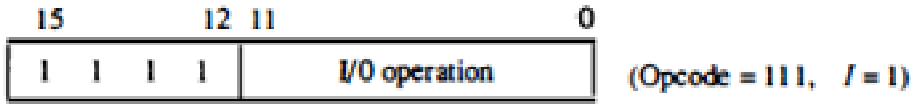
```
/* Input */        /* Initially FGI = 0 */
   loop: If FGI = 0 goto loop
         AC ← INPR, FGI ← 0

/* Output */       /* Initially FGO = 1 */
   loop: If FGO = 0 goto loop
         OUTR ← AC, FGO ← 0
```





INPUT-OUTPUT INSTRUCTIONS



(c) Input – output instruction

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

D_7IT_3 = p (common to all input-output instructions) = SC <- 0
IR(i) = B_i, B_0 to B_5 is always 0. B_6 to B_{11} specifies the I/O instruction.

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

INTERRUPT INITIATED INPUT/OUTPUT

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

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

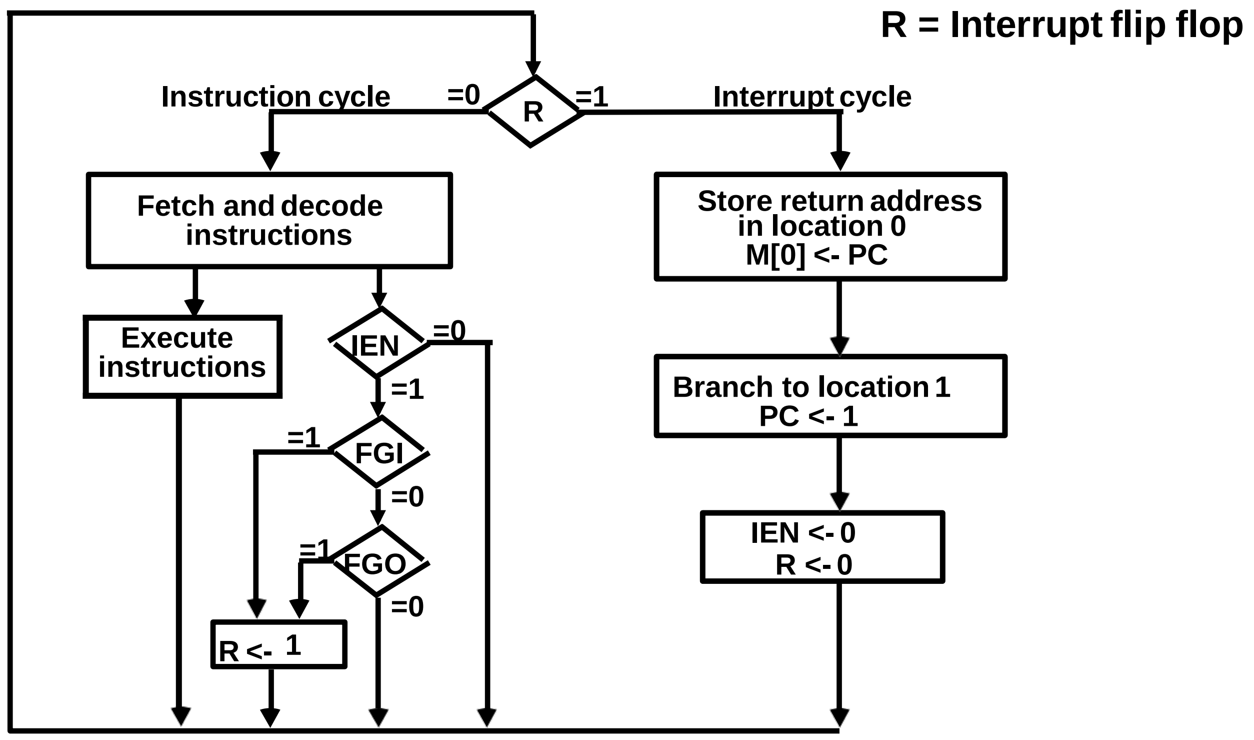
- **When the interface finds that the I/O device is ready for data transfer, it generates an interrupt request to the CPU**

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

* **IEN (Interrupt-enable flip-flop)**

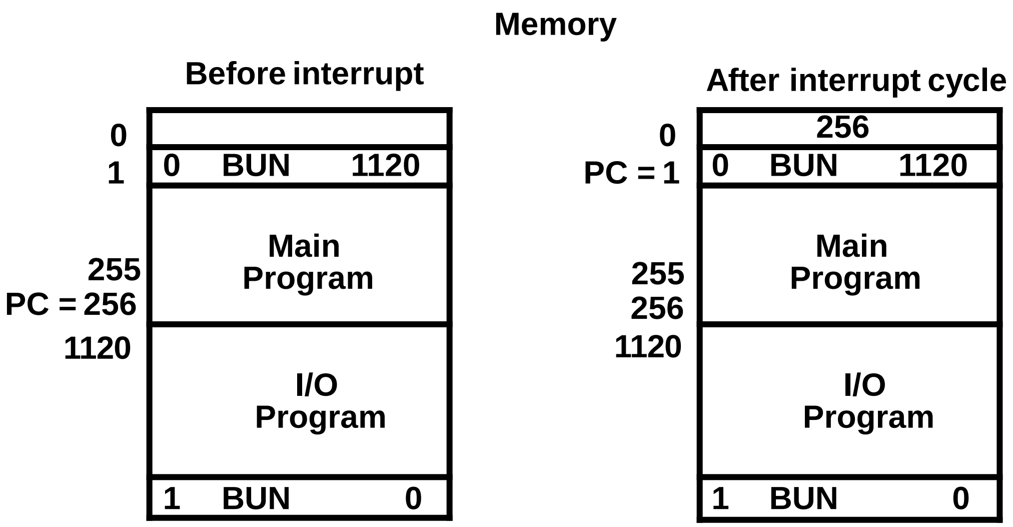
- **can be set and cleared by instructions**
- **when cleared, the computer cannot be interrupted**

FLOWCHART FOR INTERRUPT CYCLE



- 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 $\leftarrow$ 1 if IEN (FGI + FGO)$T_0'T_1'T_2'$

$\Leftrightarrow T_0'T_1'T_2'$ (IEN)(FGI + FGO): R $\leftarrow$ 1

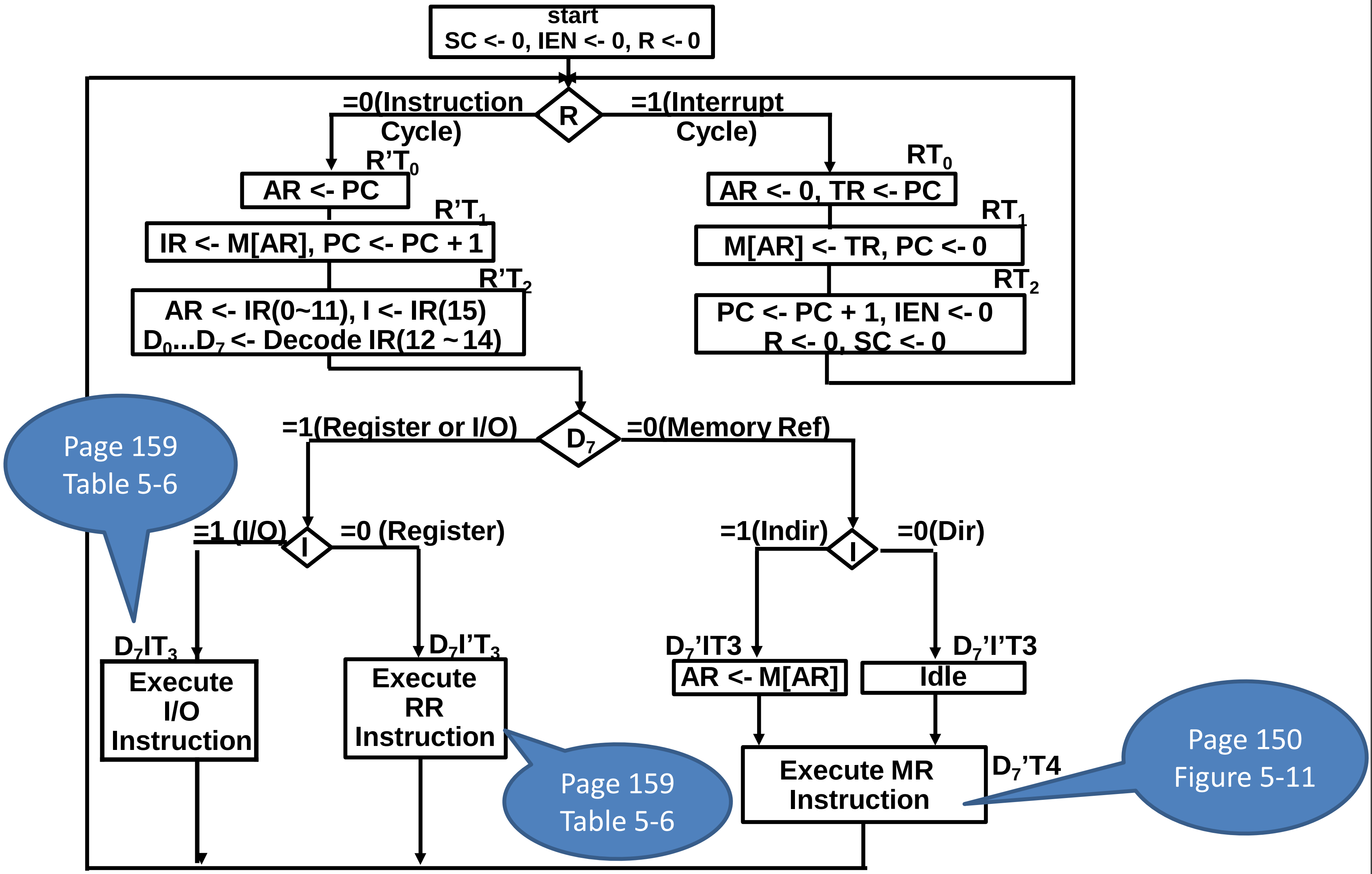
- **The fetch and decode phases of the instruction cycle must be modified:Replace T_0, T_1, T_2 with $R'T_0$, $R'T_1$, $R'T_2$**
- **The interrupt cycle :**

RT_0: AR $\leftarrow$ 0, TR $\leftarrow$ PC

RT_1: M[AR] $\leftarrow$ TR, PC $\leftarrow$ 0

RT_2: PC $\leftarrow$ PC + 1, IEN $\leftarrow$ 0, R $\leftarrow$ 0, SC $\leftarrow$ 0

COMPLETE COMPUTER DESCRIPTION
Flowchart of Operations



COMPLETE COMPUTER DESCRIPTION
Microoperations

Fetch	R'T0:	AR <- PC
	R'T1:	IR <- M[AR], PC <- PC + 1
Decode	R'T2:	D0, ..., D7 <- Decode IR(12 ~ 14), AR <- IR(0 ~ 11), I <- IR(15)
Indirect	D7'IT3:	AR <- M[AR]
Interrupt		
T0'T1'T2'(IEN)(FGI + FGO):		R <- 1
	RT0:	AR <- 0, TR <- PC
	RT1:	M[AR] <- TR, PC <- 0
	RT2:	PC <- PC + 1, IEN <- 0, R <- 0, SC <- 0
Memory-Reference		
AND	D0T4:	DR <- M[AR]
	D0T5:	AC <- AC & DR, SC <- 0
ADD	D1T4:	DR <- M[AR]
	D1T5:	AC <- AC + DR, E <- Cout, SC <- 0
LDA	D2T4:	DR <- M[AR]
	D2T5:	AC <- DR, SC <- 0
STA	D3T4:	M[AR] <- AC, SC <- 0
BUN	D4T4:	PC <- AR, SC <- 0
BSA	D5T4:	M[AR] <- PC, AR <- AR + 1
	D5T5:	PC <- AR, SC <- 0
ISZ	D6T4:	DR <- M[AR]
	D6T5:	DR <- DR + 1
	D6T6:	M[AR] <- DR, if(DR=0) then (PC <- PC + 1), SC <- 0

COMPLETE COMPUTER DESCRIPTION

Microoperations

Register-Reference		
	D7I'T3 = r	(Common to all register-reference instr)
	IR(i) = Bi	(i = 0,1,2, ..., 11)
	r:	SC <- 0
CLA	rB11:	AC <- 0
CLE	rB10:	E <- 0
CMA	rB9:	AC <- AC'
CME	rB8:	E <- E'
CIR	rB7:	AC <- shr AC, AC(15) <- E, E <- AC(0)
CIL	rB6:	AC <- shl AC, AC(0) <- E, E <- AC(15)
INC	rB5:	AC <- AC + 1
SPA	rB4:	If(AC(15) =0) then (PC <- PC + 1)
SNA	rB3:	If(AC(15) =1) then (PC <- PC + 1)
SZA	rB2:	If(AC = 0) then (PC <- PC + 1)
SZE	rB1:	If(E=0) then (PC <- PC + 1)
HLT	rB0:	S <- 0
Input-Output	D7IT3 = p	(Common to all input-output instructions)
	IR(i) = Bi	(i = 6,7,8,9,10,11)
	p:	SC <- 0
INP	pB11:	AC(0-7) <- INPR, FGI <- 0
OUT	pB10:	OUTR <- AC(0-7), FGO <- 0
SKI	pB9:	If(FGI=1) then (PC <- PC + 1)
SKO	pB8:	If(FGO=1) then (PC <- PC + 1)
ION	pB7:	IEN <- 1
IOF	pB6:	IEN <- 0

DESIGN OF 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, S, 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**
- **S2, S1, S0 Controls to select a register for the bus**
- **AC, and Adder and Logic circuit**

CONTROL OF REGISTERS AND MEMORY

Address Register; AR

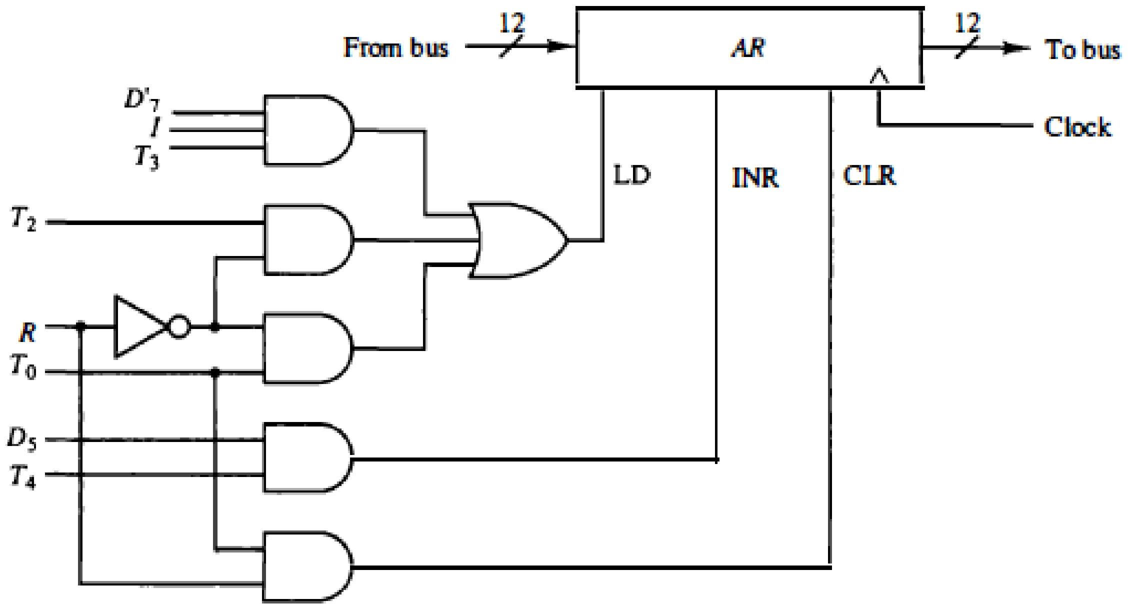
Scan all of the register transfer statements that change the content of AR:

$R'T_0$:	$AR \leftarrow PC$	LD(AR)
$R'T_2$:	$AR \leftarrow IR(0\text{-}11)$	LD(AR)
D'_7IT_3:	$AR \leftarrow M[AR]$	LD(AR)
RT_0:	$AR \leftarrow 0$	CLR(AR)
D_5T_4:	$AR \leftarrow AR + 1$	INR(AR)

⇩

$LD(AR) = R'T_0 + R'T_2 + D'_7IT_3$
$CLR(AR) = RT_0$
$INR(AR) = D_5T_4$

Figure 5-16 Control gates associated with AR.

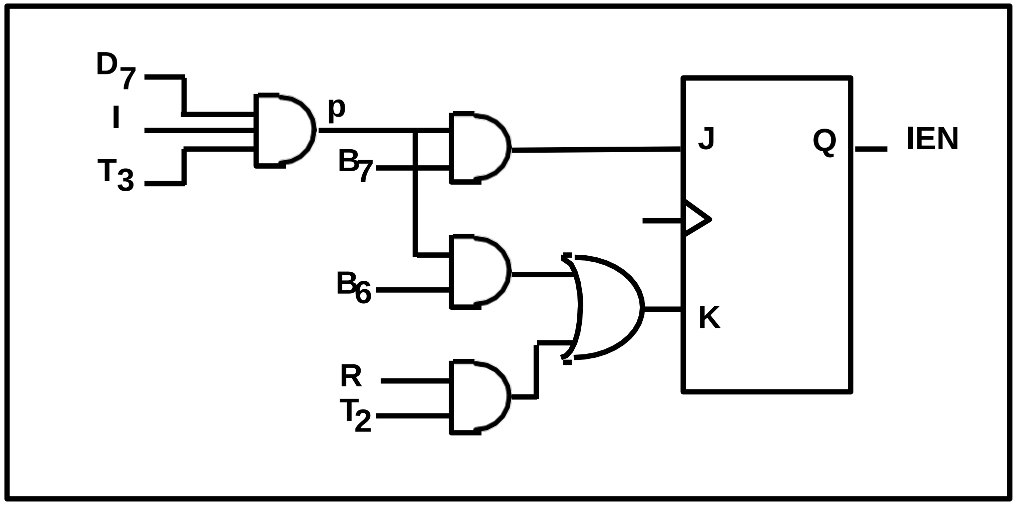


CONTROL OF FLAGS

IEN: Interrupt Enable Flag

$pB7$: IEN $\leftarrow$ 1 (I/O Instruction)
$pB6$: IEN $\leftarrow$ 0 (I/O Instruction)
RT_2: IEN $\leftarrow$ 0 (Interrupt)

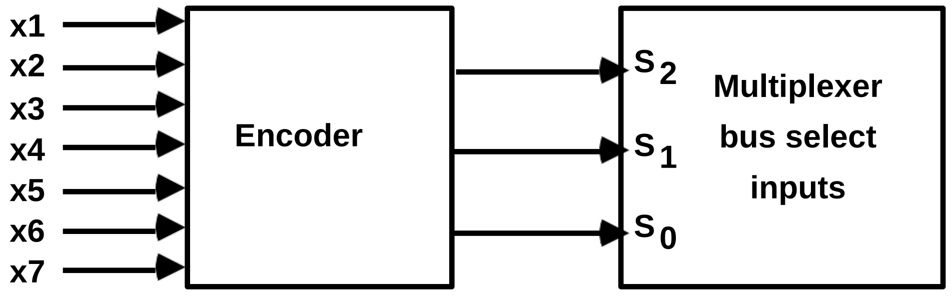
$p = D_7IT_3$ (Input/Output Instruction)



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

J – Sets
K – Clears
00 – Holds
11 – Compliments

CONTROL OF COMMON BUS



x1	x2	x3	x4	x5	x6	x7	S2	S1	S0	selected register
0	0	0	0	0	0	0	0	0	0	none
1	0	0	0	0	0	0	0	0	1	AR
0	1	0	0	0	0	0	0	1	0	PC
0	0	1	0	0	0	0	0	1	1	DR
0	0	0	1	0	0	0	1	0	0	AC
0	0	0	0	1	0	0	1	0	1	IR
0	0	0	0	0	1	0	1	1	0	TR
0	0	0	0	0	0	1	1	1	1	Memory

For AR

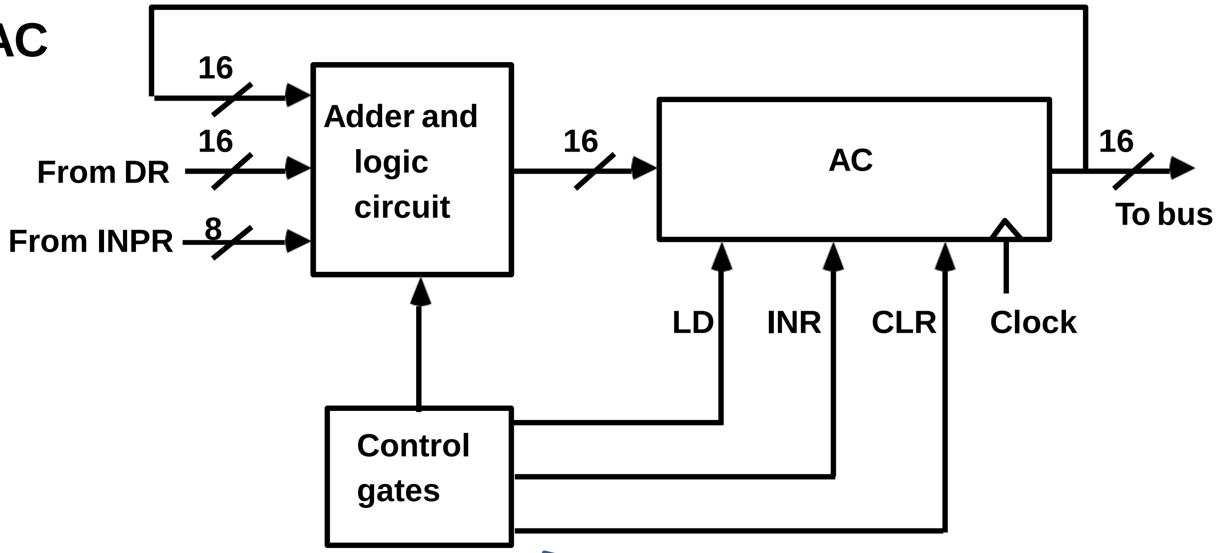
D_4T_4: PC $\leftarrow$ AR
D_5T_5: PC $\leftarrow$ AR

$\Downarrow$

$x1 = D_4T_4 + D_5T_5$

DESIGN OF ACCUMULATOR LOGIC

Circuits associated with AC



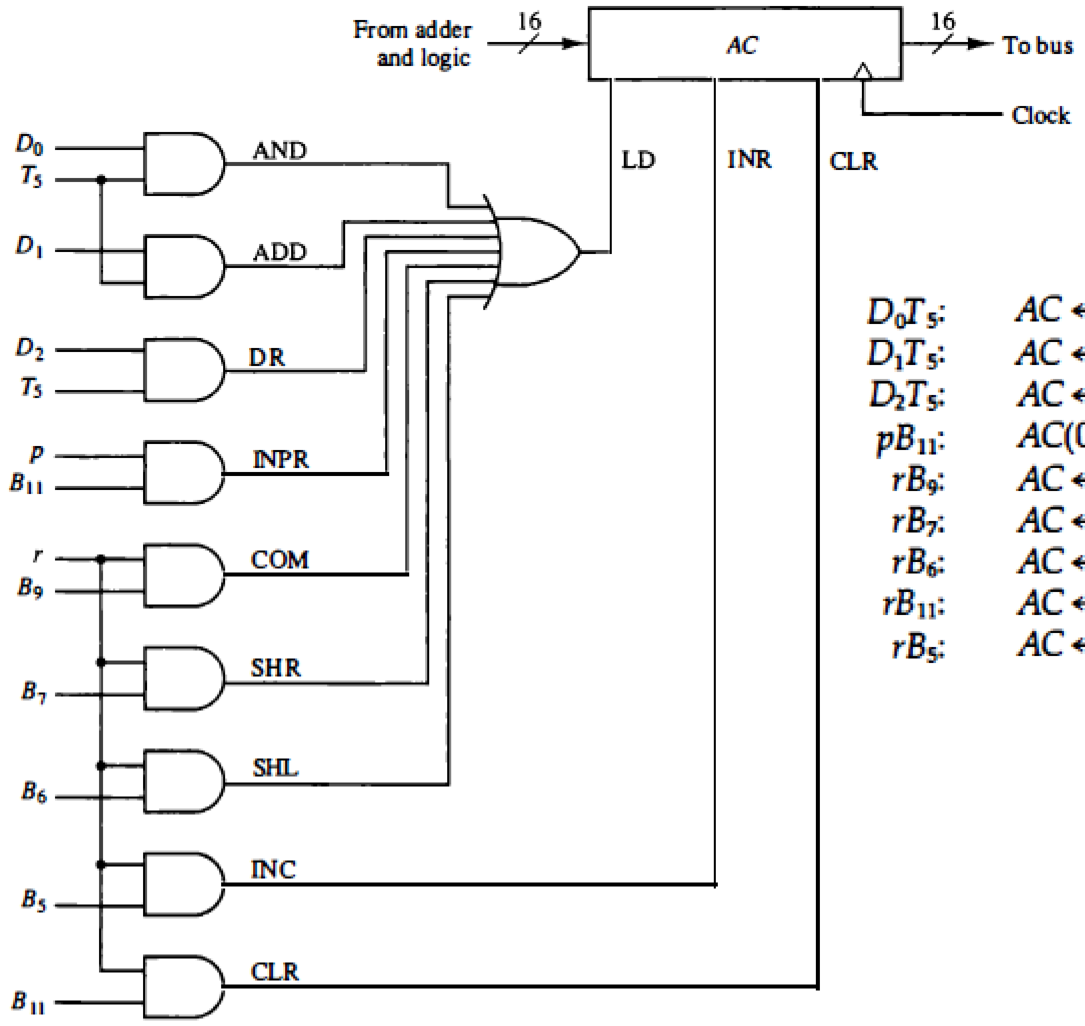
Control Gates on Next Slide

All the statements that change the content of AC

D_0T_5:	$AC \leftarrow AC \wedge DR$	AND with DR
D_1T_5:	$AC \leftarrow AC + DR$	Add with DR
D_2T_5:	$AC \leftarrow DR$	Transfer from DR
pB_{11}:	$AC(0\text{-}7) \leftarrow INPR$	Transfer from INPR
rB_9:	$AC \leftarrow AC'$	Complement
rB_7 :	$AC \leftarrow \text{shr } AC, AC(15) \leftarrow E$	Shift right
rB_6 :	$AC \leftarrow \text{shl } AC, AC(0) \leftarrow E$	Shift left
rB_{11} :	$AC \leftarrow 0$	Clear
rB_5 :	$AC \leftarrow AC + 1$	Increment

Control of AC Register

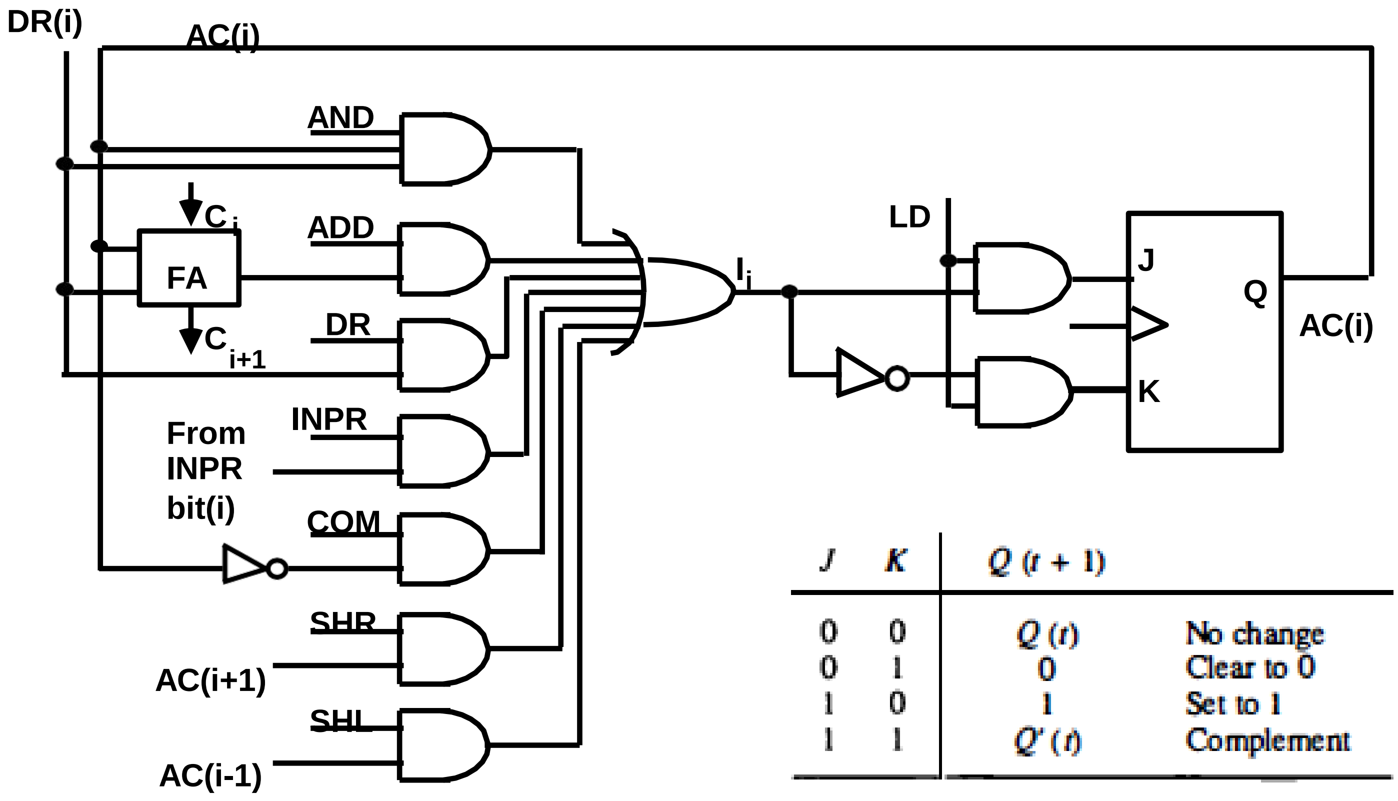
Gate structures for controlling the LD, INR, and CLR of AC

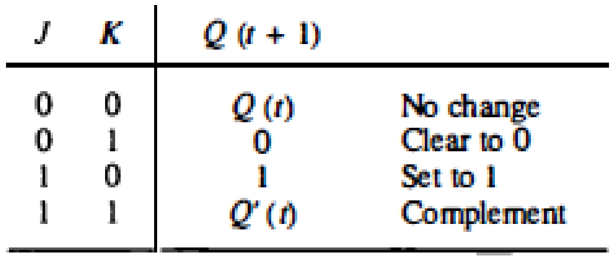


D_0T_5:	$AC \leftarrow AC \wedge DR$	AND with *DR*
D_1T_5:	$AC \leftarrow AC + DR$	Add with *DR*
D_2T_5:	$AC \leftarrow DR$	Transfer from *DR*
pB_{11}:	$AC(0\text{–}7) \leftarrow INPR$	Transfer from *INPR*
rB_9:	$AC \leftarrow \overline{AC}$	Complement
rB_7:	$AC \leftarrow \text{shr } AC, \quad AC(15) \leftarrow E$	Shift right
rB_6:	$AC \leftarrow \text{shl } AC, \quad AC(0) \leftarrow E$	Shift left
rB_{11}:	$AC \leftarrow 0$	Clear
rB_5:	$AC \leftarrow AC + 1$	Increment

ADDER AND LOGIC CIRCUIT

One stage of Adder and Logic circuit





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