

SWITCHING THEORY AND LOGIC DESIGN

Prepared by:
V.SESHAGIRI RAO, Professor
S.RAMBABU(Asst.Prof)
LINJU T T(Asst.Prof)

Unit 1

NUMBER SYSTEMS

Any number in one base system can be converted into another base system

Types

- 1) decimal to any base
- 2) Any base to decimal
- 3) Any base to Any base

Number Systems

Decimal number: $123.45 = 1 \cdot 10^2 + 2 \cdot 10^1 + 3 \cdot 10^0 + 4 \cdot 10^{-1} + 5 \cdot 10^{-2}$ •

Base b number: $N = a_{q-1}b^{q-1} + \dots + a_0b^0 + \dots + a_{-p}b^{-p}$
 $b > 1, \quad 0 \leq a_i \leq b-1$

Integer part: $a_{q-1}a_{q-2} \dots a_0$

Fractional part: $a_{-1}a_{-2} \dots a_{-p}$ •

Most significant digit: $a_{q-1} \dots$

Least significant digit: a_{-p}

Binary number ($b=2$): $1101.01 = 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 + 0 \cdot 2^{-1} + 1 \cdot 2^{-2}$

Representing number N in base b : $(N)_b$ • • • • •

Complement of digit a : $a' = (b-1)-a$

Decimal system: 9's complement of 3 = $9-3 = 6$

Binary system: 1's complement of 1 = $1-1 = 0$

Representation of Integers

	<i>Base</i>				
	2	4	8	10	12
0000	0	0	0	0	0
0001	1	1	1	1	1
0010	2	2	2	2	2
0011	3	3	3	3	3
0100	10	4	4	4	4
0101	11	5	5	5	5
0110	12	6	6	6	6
0111	13	7	7	7	7
1000	20	10	8	8	8
1001	21	11	9	9	9
1010	22	12	10	α	
1011	23	13	11	β	
1100	30	14	12	10	
1101	31	15	13	11	
1110	32	16	14	12	
1111	33	17	15	13	

Base Conversions

Example: Base 8 to base 10

$$(432.2)_8 = 4 \cdot 8^2 + 3 \cdot 8^1 + 2 \cdot 8^0 + 2 \cdot 8^{-1} = (282.25)_{10}$$

Example: Base 2 to base 10

$$(1101.01)_2 = 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 + 0 \cdot 2^{-1} + 1 \cdot 2^{-2} = (13.25)_{10}$$

Base b_1 to b_2 , where $b_1 > b_2$:

$$(N)_{b_1} = a_{q-1}b_2^{q-1} + a_{q-2}b_2^{q-2} + \cdots + a_1b_2^1 + a_0b_2^0$$

$$\frac{(N)_{b_1}}{b_2} = \underbrace{a_{q-1}b_2^{q-2} + a_{q-2}b_2^{q-3} + \cdots + a_1}_{Q_0} + \frac{a_0}{b_2}$$

$$\left(\frac{Q_0}{b_2}\right)_{b_1} = \underbrace{a_{q-1}b_2^{q-3} + a_{q-2}b_2^{q-4} + \cdots}_{Q_1} + \frac{a_1}{b_2}$$

Conversion of Bases (Contd.)

Example: Convert $(548)_{10}$ to base 8

Q_i	r_i
68	$4 = a_0$
8	$4 = a_1$
1	$0 = a_2$
	$1 = a_3$

Thus, $(548)_{10} = (1044)_8$

Example: Convert $(345)_{10}$ to base 6

Q_i	r_i
57	$3 = a_0$
9	$3 = a_1$
1	$3 = a_2$
	$1 = a_3$

5 Thus, $(345)_{10} = (1333)_6$

Conversions of fractional numbers

Fractional number:

$$(N)_{b_1} = a_{-1}b_2^{-1} + a_{-2}b_2^{-2} + \cdots + a_{-p}b_2^{-p}$$

$$b_2 \cdot (N)_{b_1} = a_{-1} + a_{-2}b_2^{-1} + \cdots + a_{-p}b_2^{-p+1}$$

Example: Convert $(0.3125)_{10}$ to base 8

$$0.3125 \times 8 = 2.5000 \text{ hence } a_{-1} = 2$$

$$0.5000 \times 8 = 4.0000 \text{ hence } a_{-2} = 4$$

Thus, $(0.3125)_{10} = (0.24)_8$

Decimal to Binary

Example: Convert $(432.354)_{10}$ to binary

Q_i	r_i	
216	$0 = a_0$	$0.354 \times 2 = 0.708$ hence $a_{-1} = 0$
108	$0 = a_1$	$0.708 \times 2 = 1.416$ hence $a_{-2} = 1$
54	$0 = a_2$	$0.416 \times 2 = 0.832$ hence $a_{-3} = 0$
27	$0 = a_3$	$0.832 \times 2 = 1.664$ hence $a_{-4} = 1$
13	$1 = a_4$	$0.664 \times 2 = 1.328$ hence $a_{-5} = 1$
6	$1 = a_5$	$0.328 \times 2 = 0.656$ hence $a_{-6} = 0$
3	$0 = a_6$	$\cdot$ $a_{-7} = 1$
1	$1 = a_7$	etc.
	$1 = a_8$	

Thus, $(432.354)_{10} = (110110000.0101101...)_{2}$

Octal to Binary Conversion

Example: Convert $(123.4)_8$ to binary

$$(123.4)_8 = (001\ 010\ 011.100)_2$$

Example: Convert $(1010110.0101)_2$ to octal

$$(1010110.0101)_2 = (001\ 010\ 110.010\ 100)_2 = (126.24)_8$$

Complements

- Complements are used in digital computers to simplify the subtraction operation and for logical manipulation
- They are two types of complements
 - 1) Diminished radix complement
$$(r^n - 1) - N \quad \{r \text{ is the base of number system}\}$$
 - 2) Radix Complement
$$(r^n - 1) - N + 1$$

(r-1)'s complement

- If the base = 10
- The 9's complement of 546700 is
 $999999 - 546700 = 453299$.
- If the base = 2
- The 1's complement of 1011000 is
0100111.

r 's complement

- the 10's complement of 012398 is 987602
- the 1's complement of 1101100 is 0010100

Subtraction using complements

- Discard end carry for r's complement
Using 10's complement subtract $72532 - 3250$.

$$M = 72532$$

$$10\text{'s complement of } N = \underline{+ 96750}$$

$$\text{Sum} = 169282$$

Discard end carry for 10's complement

Answer =

69282

Subtraction using (r-1)'s complement

- $X - Y = 1010100 - 1000011$

$$X = 1010100$$

$$1's \text{ comp of } Y = + \underline{0111100}$$

$$\text{Sum} = 1\ 0010000$$

$$\text{Add End-around carry} = \underline{\hspace{2cm}} + 1$$

$$\textcircled{1} \ X - Y = 0010001$$

Binary Codes

<i>Decimal digit</i>	$w_4w_3w_2w_1$				$w_4w_3w_2w_1$				$w_4w_3w_2w_1$			
	8	4	2	1	2	4	2	1	6	4	2	-3
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	1	0	0	0	1	0	1	0	1
2	0	0	1	0	0	0	1	0	0	0	1	0
3	0	0	1	1	0	0	1	1	1	0	0	1
4	0	1	0	0	0	1	0	0	0	1	0	0
5	0	1	0	1	1	0	1	1	1	0	1	1
6	0	1	1	0	1	1	0	0	0	1	1	0
7	0	1	1	1	1	1	0	1	1	1	0	1
8	1	0	0	0	1	1	1	0	1	0	1	0
9	1	0	0	1	1	1	1	1	1	1	1	1



BCD



Self-complementing Codes

Self-complementing code: Code word of 9's complement of N obtained by interchanging 1's and 0's in the code word of N

Non Weighted Codes

<i>Decimal digit</i>	<i>Excess-3</i>				<i>Cyclic</i>			
0	0	0	1	1	0	0	0	0
1	0	1	0	0	0	0	0	1
2	0	1	0	1	0	0	1	1
3	0	1	1	0	0	0	1	0
4	0	1	1	1	0	1	1	0
5	1	0	0	0	1	1	1	0
6	1	0	0	1	1	0	1	0
7	1	0	1	0	1	0	0	0
8	1	0	1	1	1	1	0	0
9	1	1	0	0	0	1	0	0



**Add 3 to
BCD**



**Successive code words
differ in only one digit**

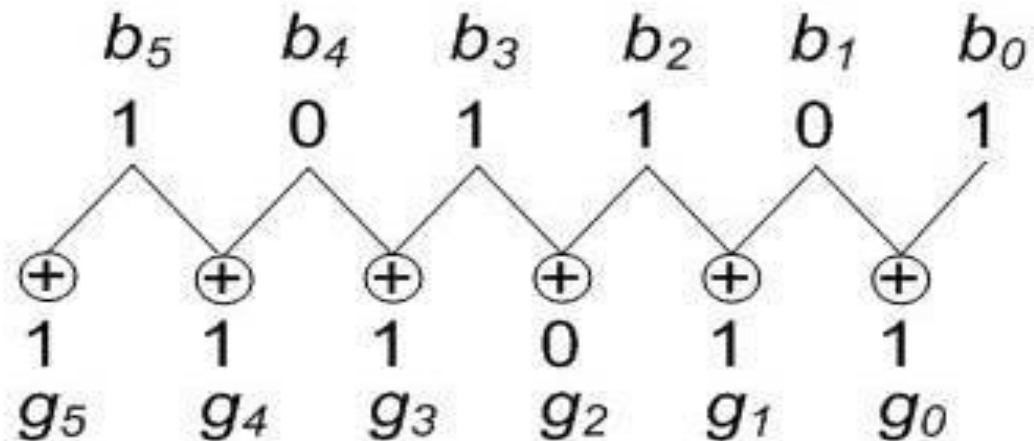
Gray Code(Unit distance Code)

<i>Decimal number</i>	<i>Gray</i>				<i>Binary</i>			
	g_3	g_2	g_1	g_0	b_3	b_2	b_1	b_0
0	0	0	0	0	0	0	0	0
1	0	0	0	1	0	0	0	1
2	0	0	1	1	0	0	1	0
3	0	0	1	0	0	0	1	1
4	0	1	1	0	0	1	0	0
5	0	1	1	1	0	1	0	1
6	0	1	0	1	0	1	1	0
7	0	1	0	0	0	1	1	1
8	1	1	0	0	1	0	0	0
9	1	1	0	1	1	0	0	1
10	1	1	1	1	1	0	1	0
11	1	1	1	0	1	0	1	1
12	1	0	1	0	1	1	0	0
13	1	0	1	1	1	1	0	1
14	1	0	0	1	1	1	1	0
15	1	0	0	0	1	1	1	1

Binary to Gray Conversion

Example:

Binary:



Gray:

Gray-to-binary:

- $b_i = g_i$ if no. of 1's preceding g_i is even
- $b_i = g_i'$ if no. of 1's preceding g_i is odd

Mirror Image Representation in Gray Code

00	0	00	0	000
01	0	01	0	001
11	0	11	0	011
10	0	10	0	010
	1	10	0	110
	1	11	0	111
	1	01	0	101
	1	00	0	100
			1	100
			1	101
			1	111
			1	110
			1	010
			1	011
			1	001
			1	000

Error Detection and Correction

- No communication channel or storage device is completely error-free
- As the number of bits per area or the transmission rate increases, more errors occur.
- Impossible to detect or correct 100% of the errors

Types of Error Detection

- 3 Types of Error Detection/Correction Methods
 - Cyclic Redundancy Check (CRC)
 - Hamming Codes
 - Reed-Solomon (RS)

$$\begin{array}{ccccc} 10011001011 & = & 1001100 & + & 1011 \\ \wedge & & \wedge & & \wedge \\ \text{Code word} & & \text{information} & & \text{error-checking bits/} \\ & & \text{bits} & & \text{parity bits/} \\ & & & & \text{syndrome/} \\ & & & & \text{redundant bits} \end{array}$$

Hamming Codes

- One of the most effective codes for error-recovery
- Used in situations where random errors are likely to occur
- Error detection and correction increases in proportion to the number of **parity bits (error-checking bits)** added to the end of the information bits

code word = information bits + parity bits

Hamming distance: the number of bit positions in which two code words differ.

```
10001001
10110001
  * * *
```

- **Minimum Hamming distance** or **D(min)** : determines its error detecting and correcting capability.
Hamming codes can always detect $D(\min) - 1$ errors, but can only correct half of those errors.

Hamming Codes

EX. Data

Parity

Code

Bits

Bit

Word

00

0

000

01

1

011

10

1

101

11

0

110

000* 100

001 101*

010 110*

011* 111

- Single parity bit can only detect error, not correct it
- Error-correcting codes require more than a single parity bit

EX. 0 0 0 0 0
 0 1 0 1 1
 1 0 1 1 0
 1 1 1 0 1

Minimum Hamming distance = 3

Can detect up to 2 errors and correct 1 error

Cyclic Redundancy Check

1. Let the information byte $F = 1001011$
2. The sender and receiver agree on an arbitrary binary pattern P . Let $P = 1011$.
3. Shift F to the left by 1 less than the number of bits in P . Now, $F = 1001011000$.
4. Let F be the dividend and P be the divisor. Perform "modulo 2 division".
5. After performing the division, we ignore the quotient. We got 100 for the remainder, which becomes the actual CRC checksum.
6. Add the remainder to F , giving the message M :

$$1001011 + 100 = 1001011100 = M$$

Calculating and Using CRCs

7. M is decoded and checked by the message receiver using the reverse process.

$$\begin{array}{r} \underline{\hspace{2cm}1010100} \\ 1011 \mid 1001011100 \\ \quad \underline{1011} \\ \quad 001001 \\ \quad 1001 \\ \quad \underline{0010} \\ \quad 001011 \\ \quad \quad \underline{1011} \\ \quad \quad 0000 \end{array} \quad \leftarrow \text{Remainder}$$

Canonical and Standard Forms

- We need to consider formal techniques for the simplification of Boolean functions.
 - Identical functions will have exactly the same canonical form.
 - Minterms and Maxterms
 - Sum-of-Minterms and Product-of- Maxterms
 - Product and Sum terms
 - Sum-of-Products (SOP) and Product-of-Sums (POS)

Definitions

- *Literal*: A variable or its complement
- *Product term*: literals connected by $\bullet$
- *Sum term*: literals connected by $+$
- *Minterm*: a product term in which all the variables appear exactly once, either complemented or uncomplemented
- *Maxterm*: a sum term in which all the variables appear exactly once, either complemented or uncomplemented

Truth Table notation for Minterms and Maxterms

- Minterms and Maxterms are easy to denote using a truth table.
- Example:
Assume 3 variables x, y, z
(order is fixed)

x	y	z	Minterm	Maxterm
0	0	0	<u>$x'y'z'$</u> = m_0	<u>$x+y+z$</u> = M_0
0	0	1	<u>$x'y'z$</u> = m_1	$x+y+z'$ = M_1
0	1	0	<u>$x'yz'$</u> = m_2	$x+y'+z$ = M_2
0	1	1	<u>$x'yz$</u> = m_3	$x+y'+z'$ = M_3
1	0	0	<u>$xy'z'$</u> = m_4	$x'+y+z$ = M_4
1	0	1	<u>$xy'z$</u> = m_5	$x'+y+z'$ = M_5
1	1	0	xyz' = m_6	$x'+y'+z$ = M_6
1	1	1	xyz = m_7	$x'+y'+z'$ = M_7

Canonical Forms (Unique)

- Any Boolean function $F()$ can be expressed as a *unique* **sum** of **minterms** and a unique **product** of **maxterms** (under a fixed variable ordering).
- In other words, every function $F()$ has two canonical forms:
 - Canonical Sum-Of-Products (sum of minterms)
 - Canonical Product-Of-Sums (product of maxterms)

Canonical Forms (cont.)

- Canonical Sum-Of-Products:
The minterms included are those m_j such that $F(\) = 1$ in row j of the truth table for $F(\)$.
- Canonical Product-Of-Sums:
The maxterms included are those M_j such that $F(\) = 0$ in row j of the truth table for $F(\)$.

Example

- Truth table for $f_1(a,b,c)$ at right
- The canonical sum-of-products form for f_1 is
$$f_1(a,b,c) = m_1 + m_2 + m_4 + m_6$$
$$= \underline{a'b'c} + \underline{a'bc'} + \underline{ab'c'} + \underline{abc'}$$
- The canonical product-of-sums form for f_1 is
$$f_1(a,b,c) = M_0 \cdot M_3 \cdot M_5 \cdot M_7$$
$$= \underline{(a+b+c)} \cdot \underline{(a+b'+c')} \cdot$$
$$\underline{(a'+b+c')} \cdot \underline{(a'+b'+c')}.$$
- Observe that: $\underline{m_j} = \underline{M_j'}$

a	b	c	f_1
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	1
1	1	1	0

Conversion Between Canonical Forms

- Replace Σ with Π (or *vice versa*) and replace those j 's that appeared in the original form with those that do not.

- Example:

$$\begin{aligned}f_1(a,b,c) &= a'b'c + a'bc' + ab'c' + abc' \\&= m_1 + m_2 + m_4 + m_6 \\&= \Sigma(1,2,4,6) \\&= \Pi(0,3,5,7) \\&= \\&= (a+b+c) \bullet (a+b'+c') \bullet (a'+b+c') \bullet (a'+b'+c')\end{aligned}$$

Conversion of SOP from standard to canonical form

- Expand *non-canonical* terms by inserting equivalent of 1 in each missing variable x :
 $(x + x') = 1$
- Remove duplicate minterms
- $f_1(a,b,c) = a'b'c + bc' + ac'$
 $= a'b'c + (a+a')bc' + a(b+b')c'$
 $= a'b'c + abc' + a'bc' + abc' + ab'c'$
 $= a'b'c + abc' + a'bc + ab'c'$

Conversion of POS from standard to canonical form

- Expand noncanonical terms by adding 0 in terms of missing variables (e.g., $xx' = 0$) and using the distributive law
- Remove duplicate maxterms
- $$\begin{aligned}f_1(a,b,c) &= (a+b+c) \cdot (b'+c') \cdot (a'+c') \\&= (a+b+c) \cdot (\textcolor{red}{a}a' + b' + c') \cdot (a' + \textcolor{red}{b}b' + c') \\&= (a+b+c) \cdot (a+b'+c') \cdot (\textcolor{orange}{a'} + \textcolor{orange}{b'} + \textcolor{orange}{c'}) \cdot \\&\quad (a' + b + c') \cdot (\textcolor{orange}{a'} + \textcolor{orange}{b'} + \textcolor{orange}{c'}) \\&= \\&= (a+b+c) \cdot (a+b'+c') \cdot (a'+b'+c') \cdot (a'+b+c')\end{aligned}$$

Boolean Algebra and Basic Gates

LOGIC GATES

Formal logic: In formal logic, a statement (proposition) is a declarative sentence that is either

true(1) or false (0).

It is easier to communicate with computers using formal logic.

- **Boolean variable:** Takes only two values – either

true (1) or false (0).

They are used as basic units of formal logic.

Boolean function and logic diagram

- **Boolean function:** Mapping from Boolean variables to a Boolean value.
- **Truth table:**
 - Represents relationship between a Boolean function and its binary variables.
 - It enumerates all possible combinations of arguments and the corresponding function values.

Boolean function and logic diagram

- **Boolean algebra:** Deals with binary variables and logic operations operating on those variables.
- **Logic diagram:** Composed of graphic symbols for logic gates. A simple circuit sketch that represents inputs and outputs of Boolean functions.

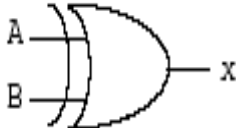
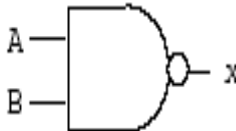
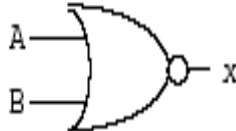
Gates

- Refer to the hardware to implement Boolean operators.
- The most basic gates are

Name	Graphic symbol	Algebraic function	Truth table															
Inverter	A  x	$x = A'$	<table><tr><th>A</th><th>x</th></tr><tr><td>0</td><td>1</td></tr><tr><td>1</td><td>0</td></tr></table>	A	x	0	1	1	0									
A	x																	
0	1																	
1	0																	
AND	A  B x	$x = AB$	<table><tr><th>A</th><th>B</th><th>x</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table> True if both are true.	A	B	x	0	0	0	0	1	0	1	0	0	1	1	1
A	B	x																
0	0	0																
0	1	0																
1	0	0																
1	1	1																
OR	A  B x	$x = A + B$	<table><tr><th>A</th><th>B</th><th>x</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table> True if either one is true.	A	B	x	0	0	0	0	1	1	1	0	1	1	1	1
A	B	x																
0	0	0																
0	1	1																
1	0	1																
1	1	1																

Boolean function and truth table

- Other common gates include:

Name	Graphic symbol	Algebraic function	Truth table															
Exclusive-OR (XOR)		$x = A \oplus B$ $= A'B + AB'$	<table><tr><th>A</th><th>B</th><th>x</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	A	B	x	0	0	0	0	1	1	1	0	1	1	1	0
A	B	x																
0	0	0																
0	1	1																
1	0	1																
1	1	0																
NAND		$x = (AB)'$	<table><tr><th>A</th><th>B</th><th>x</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	A	B	x	0	0	1	0	1	1	1	0	1	1	1	0
A	B	x																
0	0	1																
0	1	1																
1	0	1																
1	1	0																
NOR		$x = A + B$	<table><tr><th>A</th><th>B</th><th>x</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	A	B	x	0	0	1	0	1	0	1	0	0	1	1	0
A	B	x																
0	0	1																
0	1	0																
1	0	0																
1	1	0																

Parity check: True if only one is true.

Inversion of AND.

Inversion of OR.

BASIC IDENTITIES OF BOOLEAN ALGEBRA

- ***Postulate 1 (Definition)***: A Boolean algebra is a closed algebraic system containing a set K of two or more elements and the two operators $\cdot$ and $+$ which refer to logical AND and logical OR

Basic Identities of Boolean Algebra (Existence of 1 and 0 element)

$$(1) x + 0 = x$$

$$(2) x \cdot 0 = 0$$

$$(3) x + 1 = 1$$

$$(4) x \cdot 1 = x$$

$$(5) x + x = x$$

$$(6) x \cdot x = x$$

$$(7) x + x' = 1$$

$$(8) x \cdot x' = 0$$

Basic Identities of Boolean Algebra (Commutatively):

$$(9) \ x + y = y + x$$

$$(10) \ xy = yx$$

$$(11) \ x + (y + z) = (x + y) + z$$

$$(12) \ x(yz) = (xy)z$$

$$(13) \ x(y + z) = xy + xz$$

$$(14) \ x + yz = (x + y)(x + z)$$

$$(15) \ (x + y)' = x'y'$$

$$(16) \ (xy)' = x' + y'$$

$$(17) \ (x')' = x$$

Function Minimization using Boolean Algebra

- **Examples:**

$$(a) \ a + ab = a(1+b) = a$$

$$(b) \ a(a + b) = a.a \\ + ab = a + ab = a(1+b) = a.$$

$$(c) \ a + a'b = (a + a')(a + b) = 1(a + b) \\ = a + b$$

$$(d) \ a(a' + b) = a.a' + ab = 0 + ab = ab$$

DeMorgan's Theorem

$$(a) \ (a + b)' = a'b'$$

$$(b) \ (ab)' = a' + b'$$

Generalized DeMorgan's Theorem

$$(a) \ (a + b + \dots z)' = a'b' \dots z'$$

$$(b) \ (a.b \dots z)' = a' + b' + \dots z'$$

DeMorgan's Theorem

- $F = ab + c'd'$
- $F' = ??$

- $F = ab + c'd' + b'd$
- $F' = ??$

More DeMorgan's example

Show that: $(a(b + z(x + a')))' = a' + b' (z' + x')$

$$\begin{aligned}(a(b + z(x + a')))' &= a' + (b + z(x + a'))' \\&= a' + b' (z(x + a'))' \\&= a' + b' (z' + (x + a'))' \\&= a' + b' (z' + x'(a'))' \\&= a' + b' (z' + x'a) \\&= a' + b' z' + b' x' a \\&= (a' + b' x' a) + b' z' \\&= (a' + b' x')(a + a') + b' z' \\&= a' + b' x' + b' z' \\&= a' + b' (z' + x')\end{aligned}$$

Two Level implantation

- NAND-AND
- AND-NOR
- NOR-OR
- OR-NAND

NAND-AND

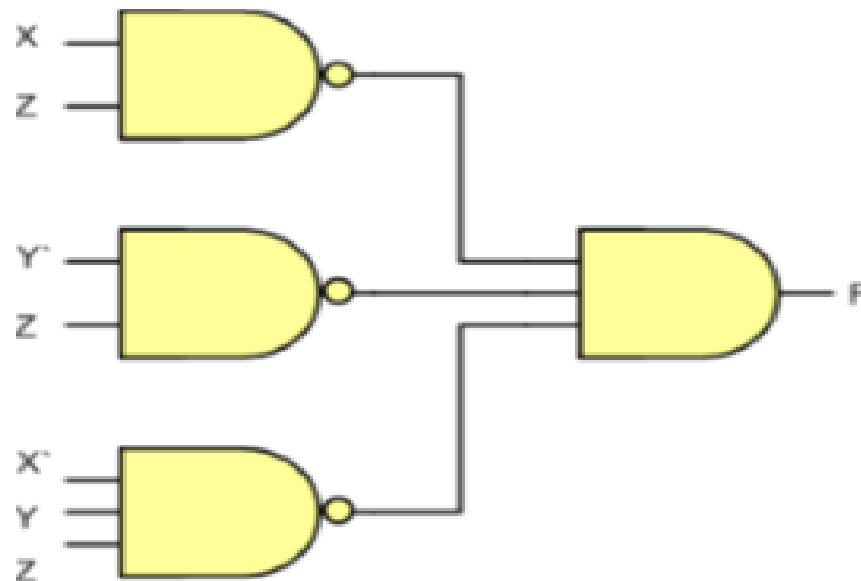
AND-NOR functions:

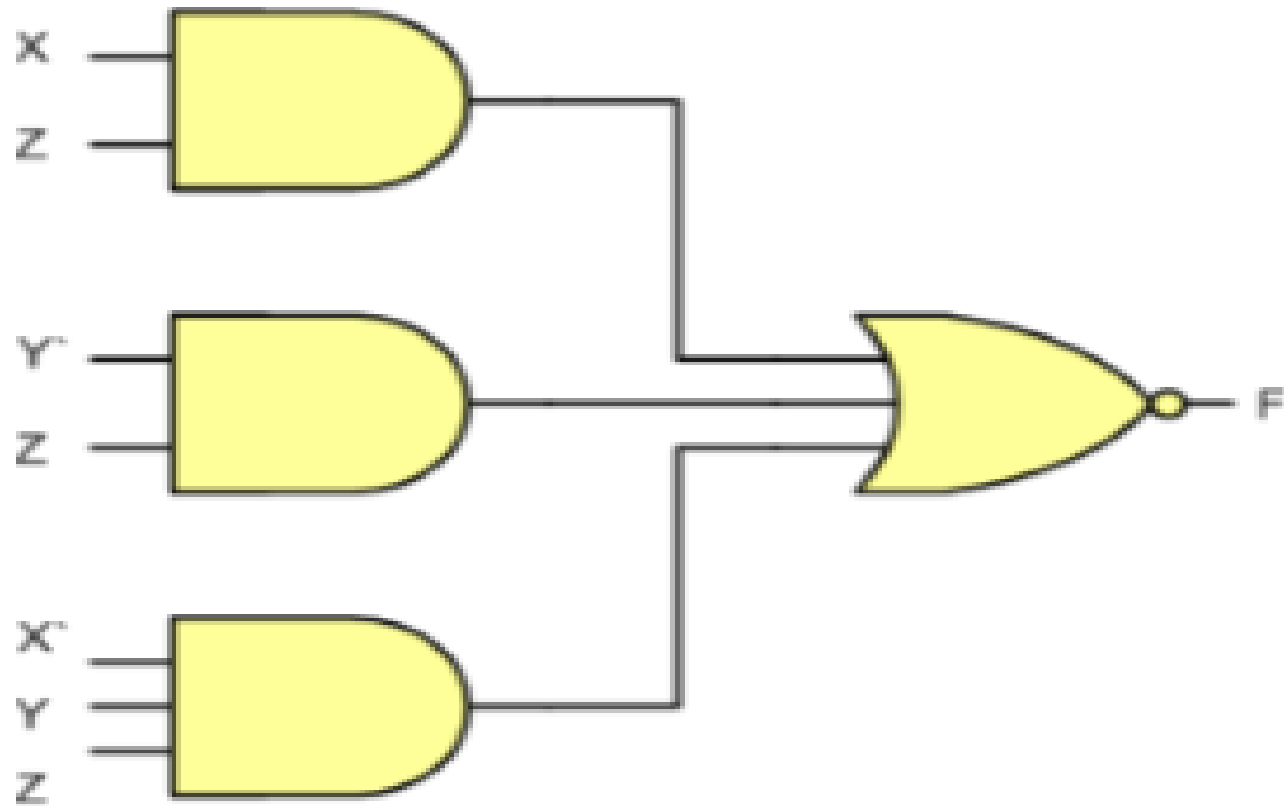
Example 3: Implement the following function

$$F = \overline{XZ} + \overline{YZ} + \overline{XYZ} \text{ or}$$

$$\overline{F} = XZ + YZ + XYZ$$

Since $\overline{F}$ is in SOP form, it can be implemented by using NAND-NAND circuit. By complementing the output we can get F , or by using **NAND-AND** circuit as shown in the figure.





It can also be implemented using AND-NOR circuit as it is equivalent to NAND- AND circuit

OR-NAND functions:

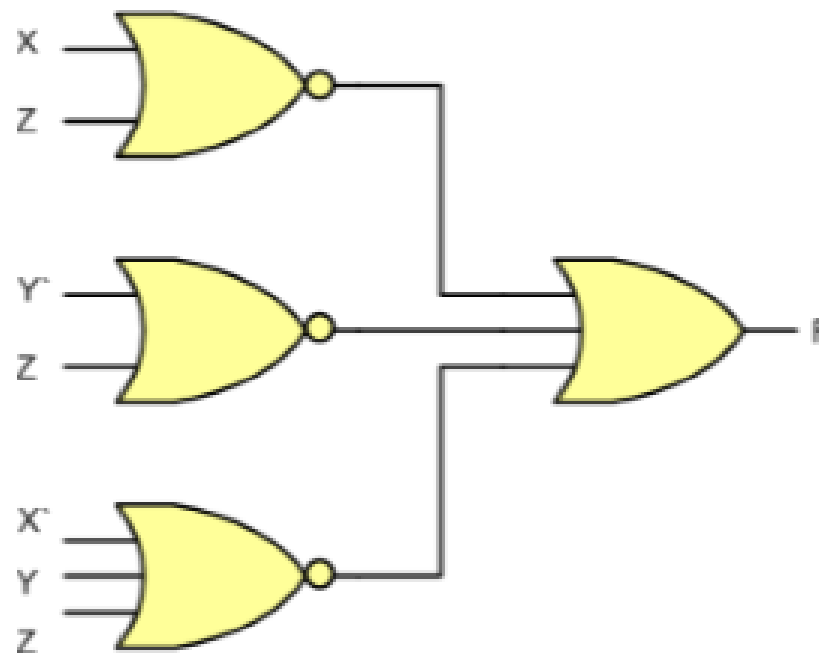
Example 4: Implement the following function

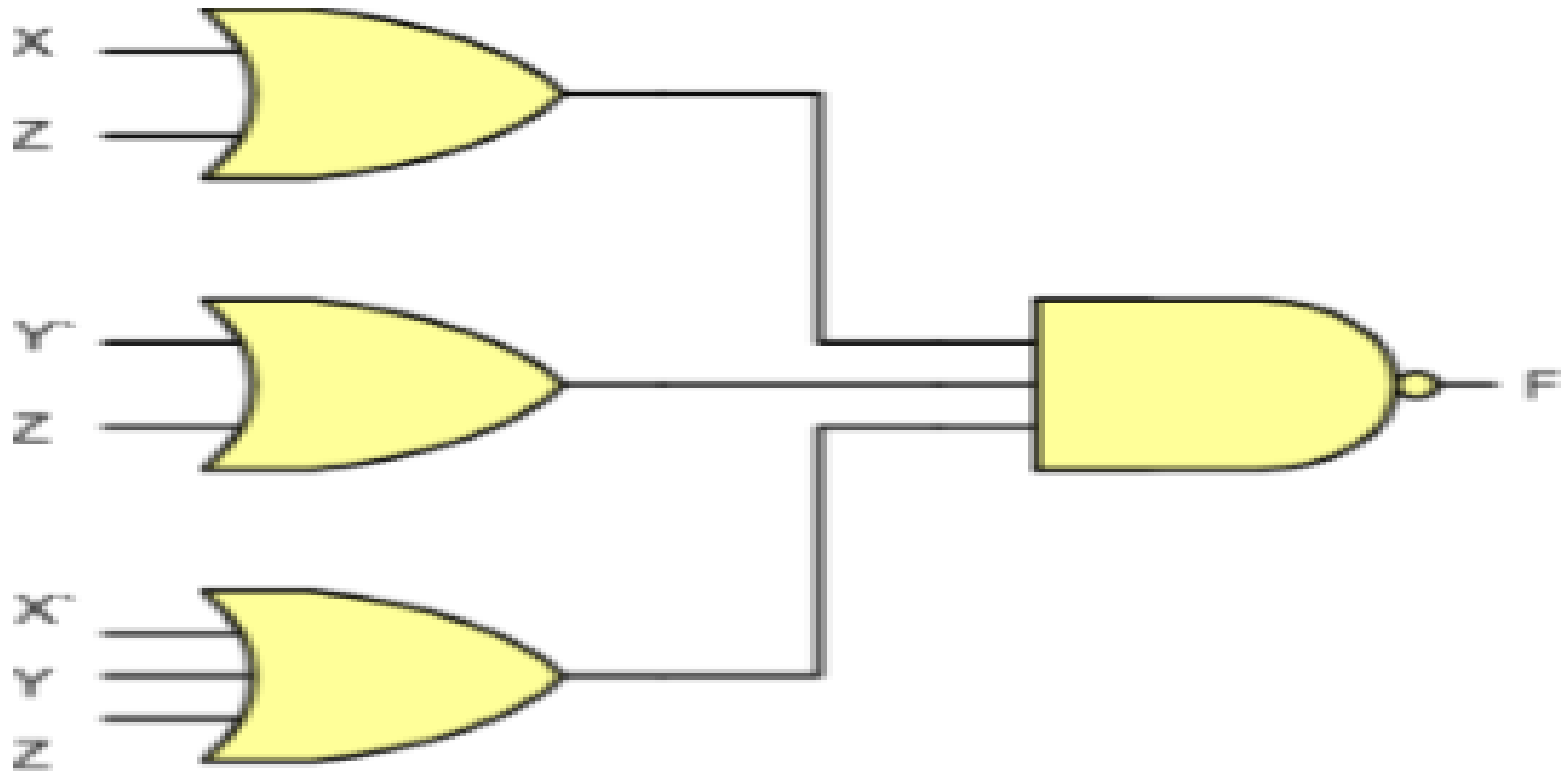
$$F = (X + Z) \cdot (\bar{Y} + Z) \cdot (\bar{X} + Y + Z) \text{ or}$$

$$\bar{F} = (X + Z)(\bar{Y} + Z)(\bar{X} + Y + Z)$$

Since $\bar{F}$ is in POS form, it can be implemented by using NOR-NOR circuit.

By complementing the output we can get F , or by using **NOR-OR** circuit as shown in the figure.





It can also be implemented using OR-NAND circuit as it is equivalent to NOR-OR circuit

Universal Gates

- The objectives of this lesson are to learn about:
 1. Universal gates - NAND and NOR.
 2. How to implement NOT, AND, and OR gate using NAND gates only.
 3. How to implement NOT, AND, and OR gate using NOR gates only.
 4. Equivalent gates.

NAND GATE

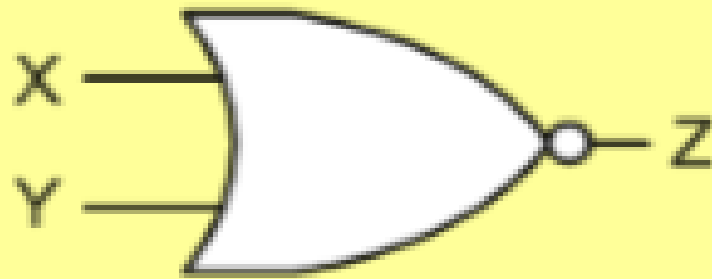
X	Y	NAND
0	0	1
0	1	1
1	0	1
1	1	0



$$Z = \overline{X \cdot Y}$$

NOR GATE

X	Y	NOR
0	0	1
0	1	0
1	0	0
1	1	0



$$Z = \overline{X + Y}$$

NAND AS A UNIVERSAL GATE

1. All NAND input pins connect to the input signal **A** gives an output **A'**.



2. One NAND input pin is connected to the input signal **A** while all other input pins are connected to logic **1**. The output will be **A'**.



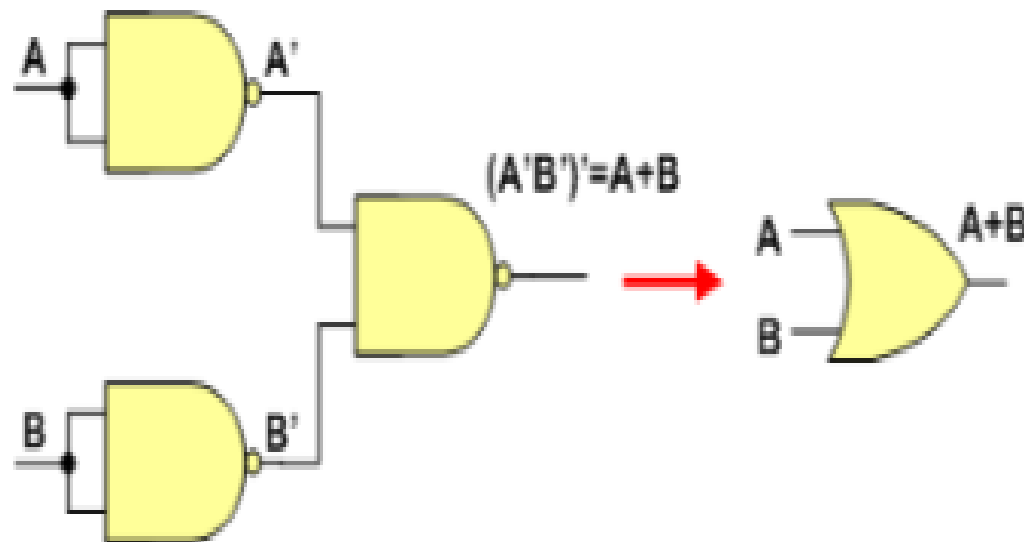
Implementing AND Using only NAND Gates

An **AND** gate can be replaced by NAND gates as shown in the figure (The AND is replaced by a NAND gate with its output complemented by a NAND gate inverter).



Implementing OR Using only NAND Gates

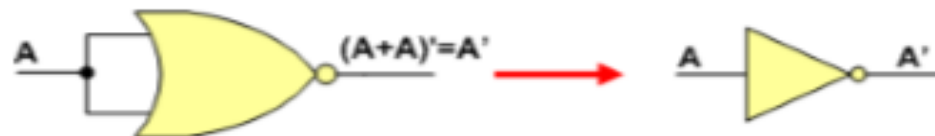
An OR gate can be replaced by NAND gates as shown in the figure (The OR gate is replaced by a NAND gate with all its inputs complemented by NAND gate inverters).



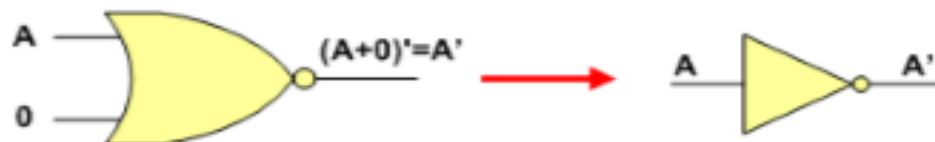
Thus, the NAND gate is a universal gate since it can implement the AND, OR and NOT functions.

NOR AS A UNIVERSAL GATE

1. All NOR input pins connect to the input signal **A** gives an output **A'**.



2. One NOR input pin is connected to the input signal **A** while all other input pins are connected to logic **0**. The output will be **A'**.



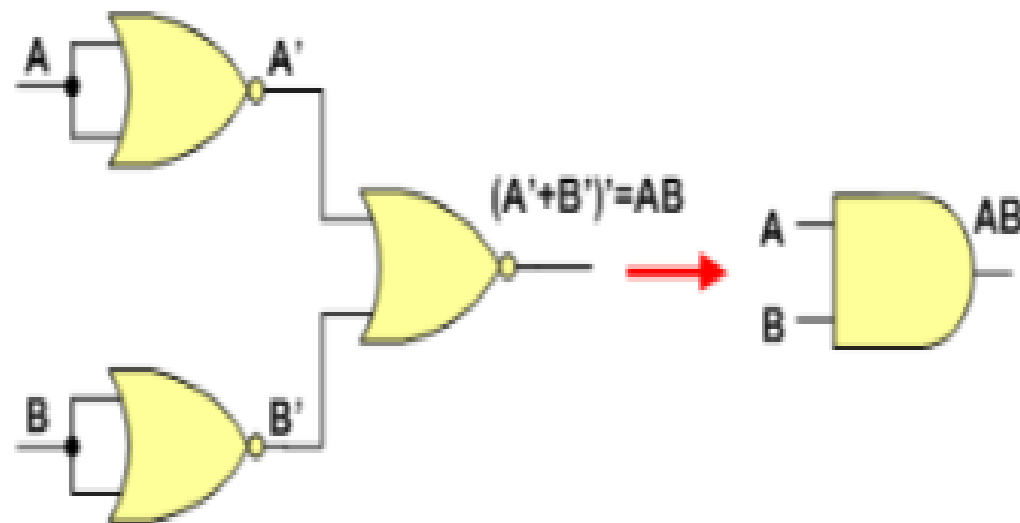
Implementing OR Using only NOR Gates

An **OR gate** can be replaced by NOR gates as shown in the figure (The OR is replaced by a NOR gate with its output complemented by a NOR gate inverter)



Implementing AND Using only NOR Gates

An AND gate can be replaced by NOR gates as shown in the figure (The AND gate is replaced by a NOR gate with all its inputs complemented by NOR gate inverters)



Thus, the NOR gate is a universal gate since it can implement the AND, OR and NOT functions.

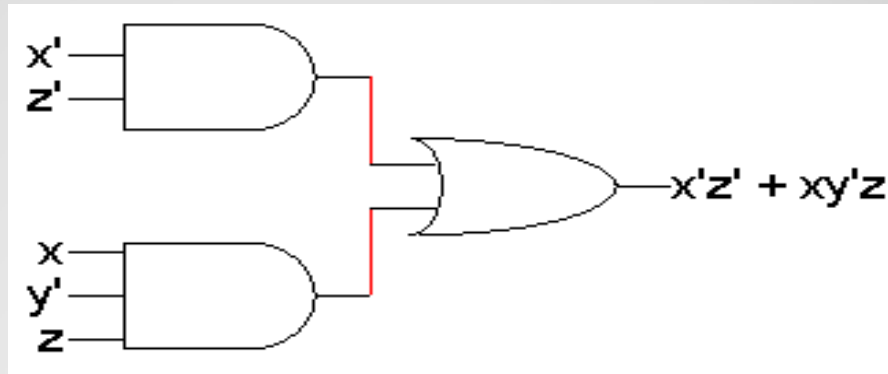
UNIT 2

Minimization and design of Combinational circuits

Karnaugh Maps for Simplification

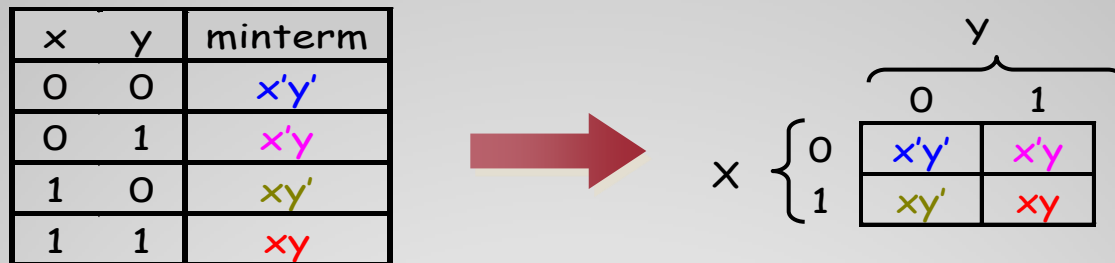
Karnaugh Maps

- Boolean algebra helps us simplify expressions and circuits
- Karnaugh Map: A graphical technique for simplifying a Boolean expression into either form:
 - minimal sum of products (MSP)
 - minimal product of sums (MPS)
- Goal of the simplification.
 - There are a minimal number of product/sum terms
 - Each term has a minimal number of literals
- Circuit-wise, this leads to a *minimal* two-level implementation

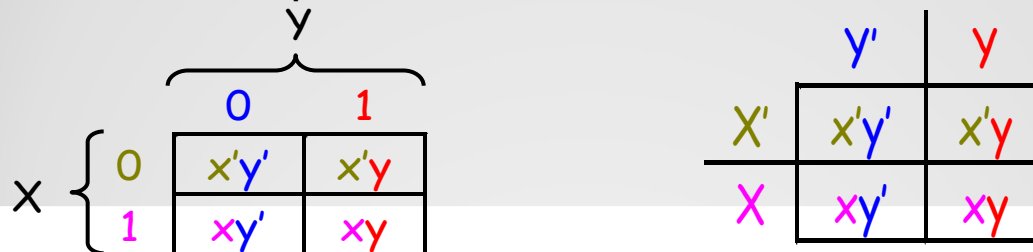


Re-arranging the Truth Table

- A two-variable function has four possible minterms. We can re-arrange these minterms into a **Karnaugh map**



- Now we can easily see which minterms contain common literals
 - Minterms on the left and right sides contain y' and y respectively
 - Minterms in the top and bottom rows contain x' and x respectively



Karnaugh Map Simplifications

- Imagine a two-variable sum of minterms:

$$x'y' + x'y$$

		y
	x'y'	x'y
x	xy'	xy

- Both of these Minterms appear in the top row of a Karnaugh map, which means that they both contain the literal x'

$$\begin{aligned}
 x'y' + x'y &= x'(y' + y) && [\text{Distributive}] \\
 &= x' \cdot 1 && [y + y' = 1] \\
 &= x' && [x \cdot 1 = x]
 \end{aligned}$$

- What happens if you simplify this expression using Boolean algebra?

More Two-Variable Examples

- Another example expression is $x'y + xy$
 - Both minterms appear in the right side, where y is uncomplemented
 - Thus, we can reduce $x'y + xy$ to just y

		y
	$x'y'$	$x'y$
x	xy'	xy

- How about $x'y' + x'y + xy$?
 - We have $x'y' + x'y$ in the top row, corresponding to x'
 - There's also $x'y + xy$ in the right side, corresponding to y
 - This whole expression can be reduced to $x' + y$

		y
	$x'y'$	$x'y$
x	xy'	xy

A Three-Variable Karnaugh Map

- For a three-variable expression with inputs x , y , z , the arrangement of minterms is more tricky:

		YZ			
		00	01	11	10
X	0	$x'y'z'$	$x'y'z$	$x'yz$	$x'yz'$
	1	$xy'z'$	$xy'z$	xyz	xyz'

		y			
		$x'y'z'$	$x'y'z$	$x'yz$	$x'yz'$
X		$xy'z'$	$xy'z$	xyz	xyz'

		YZ			
		00	01	11	10
X	0	m_0	m_1	m_3	m_2
	1	m_4	m_5	m_7	m_6

		y			
		m_0	m_1	m_3	m_2
X		m_4	m_5	m_7	m_6

- Another way to label the K-map (use whichever you like):

Why the funny ordering?

- With this ordering, any group of 2, 4 or 8 adjacent squares on the map contains common literals that can be factored out

				y
	$x'y'z'$	$x'y'z$	$x'yz$	$x'yz'$
x	$xy'z'$	$xy'z$	xyz	xyz'
				z

$$\begin{aligned}
 & x'y'z + x'yz \\
 &= x'z(y' + y) \\
 &= x'z \cdot 1 \\
 &= x'z
 \end{aligned}$$

- "Adjacency" includes wrapping around the left and right sides:

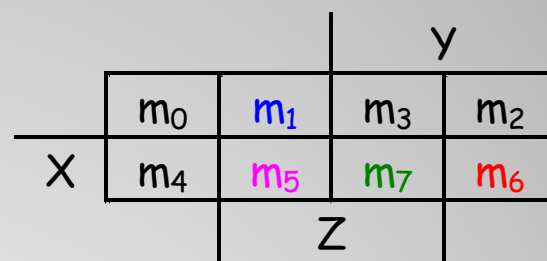
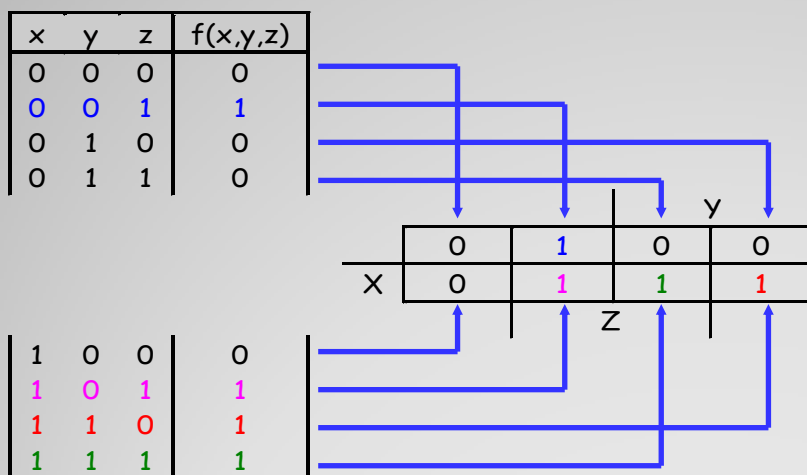
				y
	$x'y'z'$	$x'y'z$	$x'yz$	$x'yz'$
x	$xy'z'$	$xy'z$	xyz	xyz'
				z

$$\begin{aligned}
 & x'y'z' + xy'z' + x'yz' + xyz' \\
 &= z'(x'y' + xy' + x'y + xy) \\
 &= z'(y'(x' + x) + y(x' + x)) \\
 &= z'(y' + y) \\
 &= z'
 \end{aligned}$$

- We'll use this property of adjacent squares to do our simplifications.

K-maps From Truth Tables

- We can fill in the K-map directly from a truth table
 - The output in row i of the table goes into square m_i of the K-map
 - Remember that the rightmost columns of the K-map are “switched”



Reading the MSP from the K-map

- You can find the minimal SoP expression
 - Each rectangle corresponds to one product term
 - The product is determined by finding the common literals in that rectangle

	y			
	0	1	0	0
X	0	1	1	1
	z			

	y			
	$x'y'z'$	$x'y'z$	$x'yz$	$x'yz'$
X	$xy'z'$	$xy'z$	xyz	xyz'
	z			

$$F(x,y,z) = y'z + xy$$

Grouping the Minterms Together

- The most difficult step is grouping together all the 1s in the K-map
 - Make **rectangles** around groups of one, two, four or eight 1s
 - All of the 1s in the map should be included in at least one rectangle
 - Do *not* include any of the 0s
 - Each group corresponds to one product term

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

For the Simplest Result

- Make as few rectangles as possible, to minimize the number of products in the final expression.
- Make each rectangle as large as possible, to minimize the number of literals in each term.
- Rectangles can be overlapped, if that makes them larger.

K-map Simplification of SoP Expressions

- Let's consider simplifying $f(x,y,z) = xy + y'z + xz$
- You should convert the expression into a sum of minterms form,
 - The easiest way to do this is to make a truth table for the function, and then read off the minterms
 - You can either write out the literals or use the minterm shorthand
- Here is the truth table and sum of minterms for our example:

x	y	z	f(x,y,z)
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

$$\begin{aligned}f(x,y,z) &= x'y'z + xy'z + \\ &\quad xyz' + xyz \\ &= m_1 + m_5 + m_6 + m_7\end{aligned}$$

Unsimplifying Expressions

- You can also convert the expression to a sum of minterms with Boolean algebra
 - Apply the distributive law in reverse to add in missing variables.
 - Very few people actually do this, but it's occasionally useful.
$$\begin{aligned}xy + y'z + xz &= (xy \bullet 1) + (y'z \bullet 1) + (xz \bullet 1) \\&= (xy \bullet (z' + z)) + (y'z \bullet (x' + x)) + (xz \bullet (y' + y)) \\&= (xyz' + xyz) + (x'y'z + xy'z) + (xy'z + xyz) \\&= xyz' + xyz + x'y'z + xy'z \\&= m_1 + m_5 + m_6 + m_7\end{aligned}$$
- In both cases, we're actually "unsimplifying" our example expression
 - The resulting expression is larger than the original one!
 - But having all the individual minterms makes it easy to combine them together with the K-map

Making the Example K-map

- In our example, we can write $f(x,y,z)$ in two equivalent ways

$$f(x,y,z) = x'y'z + xy'z + xyz' + xyz$$

				y	
	x'y'z'	x'y'z	x'yz	x'yz'	
x	xy'z'	xy'z	xyz	xyz'	
		z			

$$f(x,y,z) = m_1 + m_5 + m_6 + m_7$$

		y		
	m_0	m_1	m_3	m_2
x	m_4	m_5	m_7	m_6
		z		

- In either case, the resulting K-map is shown below

			y	
	0	1	0	0
x	0	1	1	1
	z			

Practice K-map 1

- Simplify the sum of minterms $m_1 + m_3 + m_5 + m_6$

				y
x				
				z

				y
	m_0	m_1	m_3	m_2
x	m_4	m_5	m_7	m_6
				z

Solutions for Practice K-map 1

- Here is the filled in K-map, with all groups shown
 - The magenta and green groups overlap, which makes each of them as large as possible
 - Minterm m_6 is in a group all by its lonesome

	0	1	1	0
x	0	1	0	1

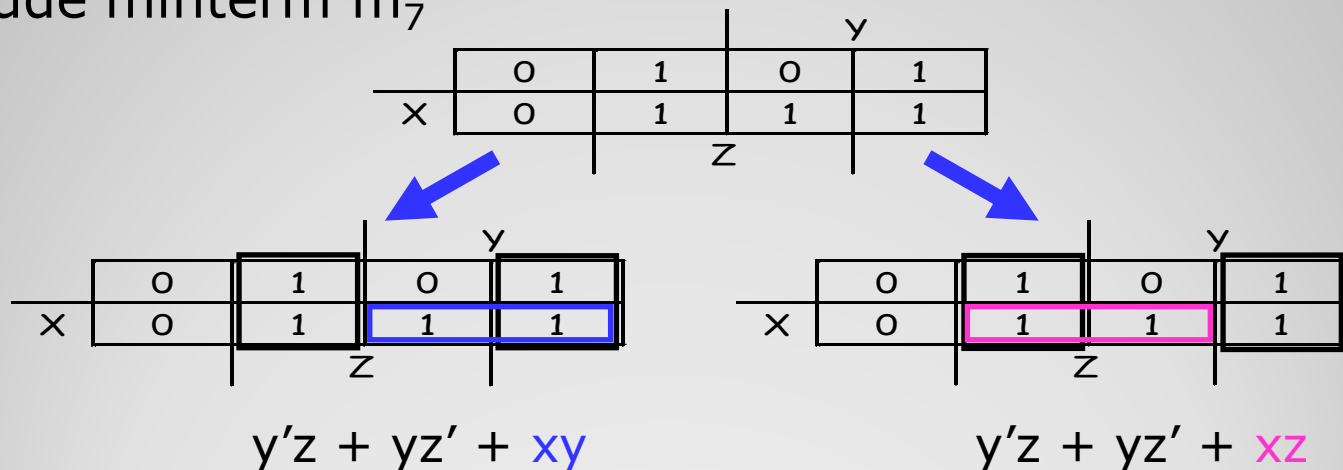
Diagram illustrating a 2x4 K-map for variables x, y, and z. The map is labeled with x (vertical axis) and z (horizontal axis). The cells contain values 0 or 1. The groups shown are:

- A green group (2x2 square) covering cells (0,1), (0,2), (1,1), and (1,2).
- A magenta group (2x1 rectangle) covering cells (0,1) and (1,1).
- A blue group (1x1 square) covering cell (1,3).

- The final MSP here is $x'z + y'z + xyz'$

K-maps can be tricky!

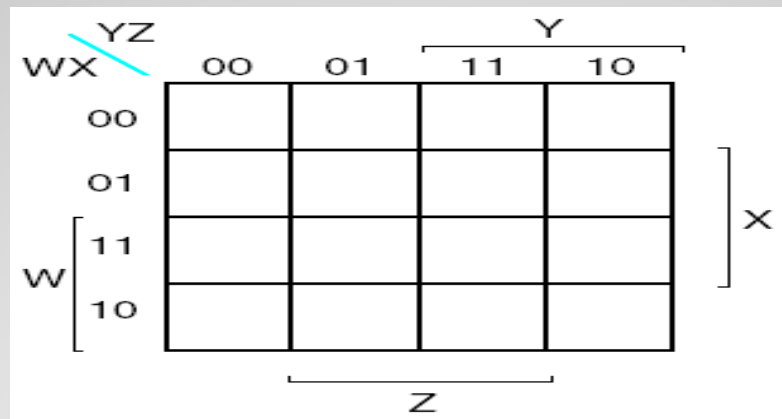
- There may not necessarily be a *unique* MSP. The K-map below yields two valid and equivalent MSPs, because there are two possible ways to include minterm m_7



- Remember that overlapping groups is possible, as shown above

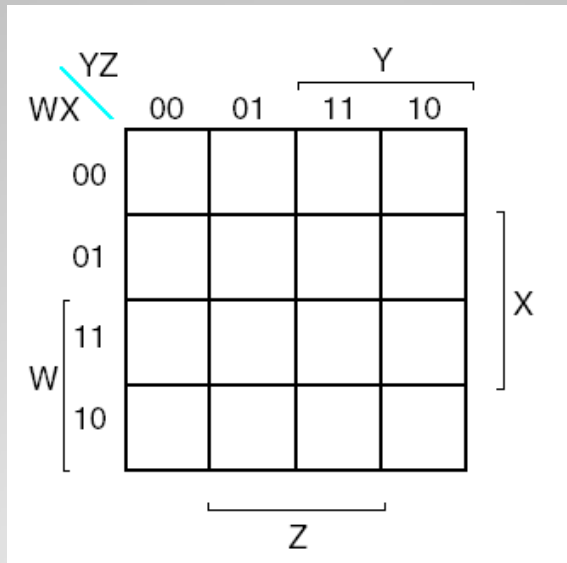
Four-variable K-maps – $f(W,X,Y,Z)$

- We can do four-variable expressions too!
 - The minterms in the third and fourth columns, *and* in the third and fourth rows, are switched around.
 - Again, this ensures that adjacent squares have common literals



- Grouping minterms is similar to the three-variable case, but:
 - You can have rectangular groups of 1, 2, 4, 8 or 16 minterms
 - You can wrap around *all four* sides

Four-variable K-maps



		y			
		$w'x'y'z'$	$w'x'y'z$	$w'x'yz$	$w'x'yz'$
		$w'xy'z'$	$w'xy'z$	$w'xyz$	$w'xyz'$
w		$wxy'z'$	$wxy'z$	$wxyz$	$wxyz'$
		$wx'y'z'$	$wx'y'z$	$wx'yz$	$wx'yz'$
		z			

		y			
		m_0	m_1	m_3	m_2
		m_4	m_5	m_7	m_6
w		m_{12}	m_{13}	m_{15}	m_{14}
		m_8	m_9	m_{11}	m_{10}
		z			

- The expression is already a sum of minterms, so here's the K-map:

		y				
		1	0	0	1	
		0	1	0	0	
		0	1	0	0	x
W		1	0	0	1	
		Z				

		y				
		m_0	m_1	m_3	m_2	
		m_4	m_5	m_7	m_6	
		m_{12}	m_{13}	m_{15}	m_{14}	x
W		m_8	m_9	m_{11}	m_{10}	
		Z				

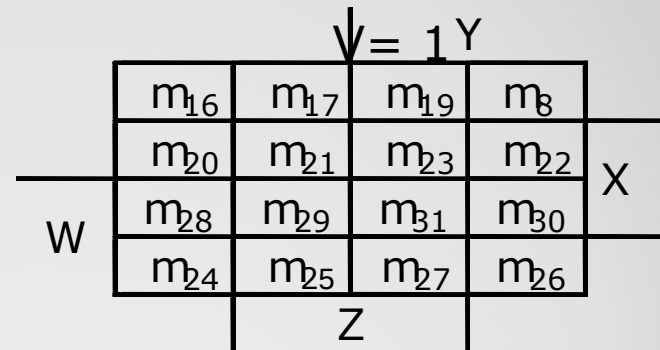
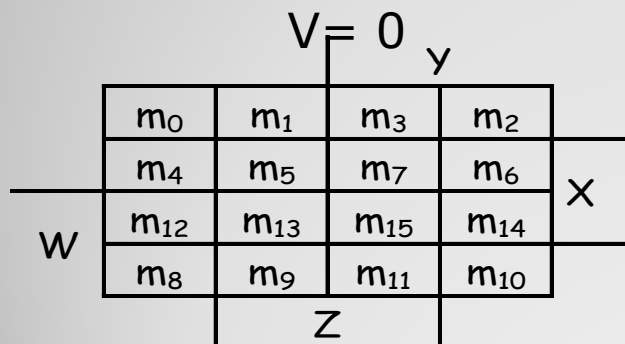
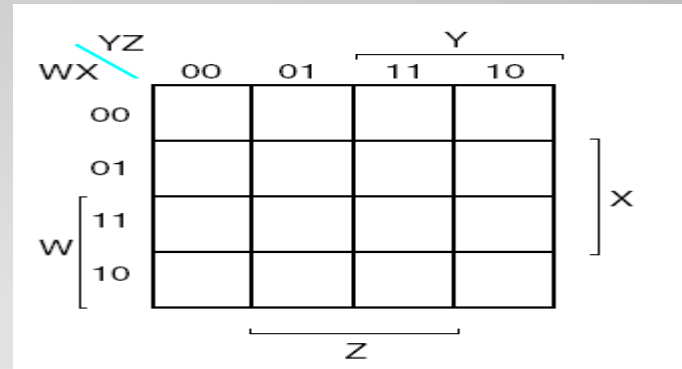
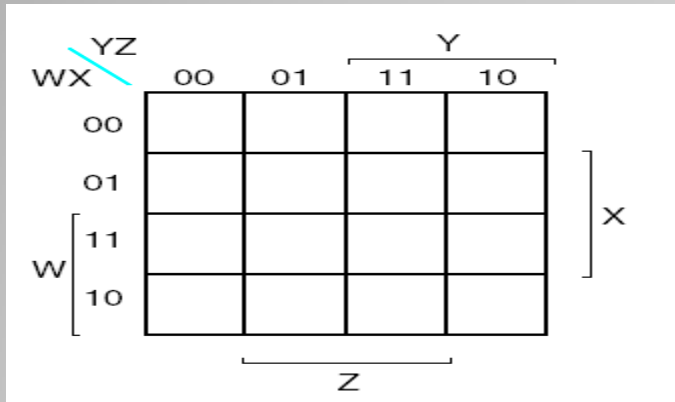
- We can make the following groups, resulting in the MSP $x'z' + xy'z$

		y				
		1	0	0	1	
		0	1	0	0	
		0	1	0	0	x
W		1	0	0	1	
		Z				

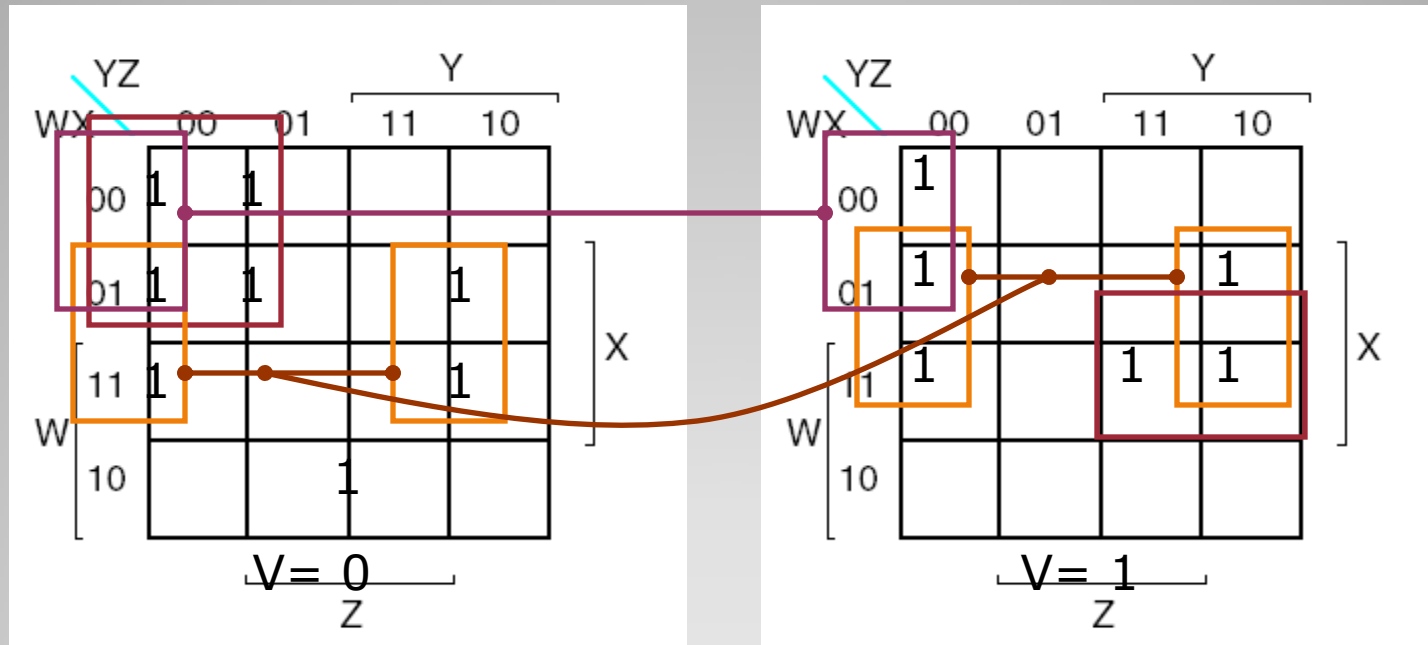
		y				
		$w'x'y'z'$	$w'x'y'z$	$w'x'yz$	$w'x'yz'$	
		$w'xy'z'$	$w'xy'z$	$w'xyz$	$w'xyz'$	
		$wxy'z'$	$wxy'z$	$wxyz$	$wxyz'$	x
W		$wx'y'z'$	$wx'y'z$	$wx'yz$	$wx'yz'$	
		Z				

Example: Simplify
 $m_0 + m_2 + m_5 + m_8 + m_{10} + m_{13}$

Five-variable K-maps - $f(V, W, X, Y, Z)$



Simplify $f(V,W,X,Y,Z) = \sum m(0,1,4,5,6,11,12,14,16,20,22,28,30,31)$



$$f = XZ'$$

$$\sum m(4,6,12,14,20,22,28,30)$$

$$+ V'W'Y' \quad \sum m(0,1,4,5)$$

$$+ W'Y'Z' \quad \sum m(0,4,16,20)$$

$$+ VWXY \quad \sum m(30,31)$$

$$+ V'WX'YZ \quad m11$$

PoS Optimization

- Maxterms are grouped to find minimal PoS expression

		yz			
		00	01	11	10
x	0	$x + y + z$	$x + y + z'$	$x + y' + z'$	$x + y' + z$
	1	$x' + y + z$	$x' + y + z'$	$x' + y' + z'$	$x' + y' + z$

- $F(W,X,Y,Z) = \prod M(0,1,2,4,5)$

0	$x+y+z$	$x+y+z'$	$x+y'+z'$	$x+y'+z$
1	$x'+y+z$	$x'+y+z'$	$x'+y'+z'$	$x'+y'+z$
	00	01	11	
	10			

$$F(W,X,Y,Z) = Y \cdot (X + Z)$$

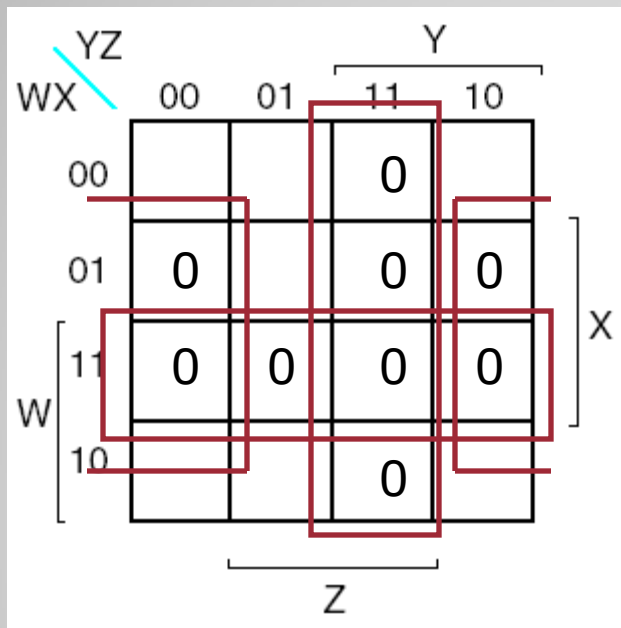
0	0	0	1	0
1	1	0	1	1
	00	01	11	
	10			

PoS Optimization

PoS Optimization from SoP

$$F(W,X,Y,Z) = \sum m(0,1,2,5,8,9,10)$$

$$= \prod M(3,4,6,7,11,12,13,14,15)$$



$$F(W,X,Y,Z) = (W' + X')(Y' + Z')(X' + Z)$$

Or,

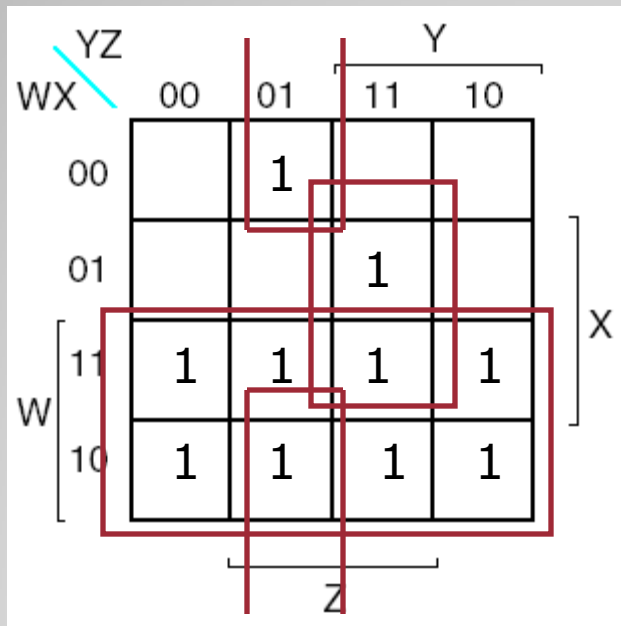
$$F(W,X,Y,Z) = X'Y' + X'Z' + W'Y'Z$$

Which one is the minimal one?

SoP Optimization from PoS

$$F(W,X,Y,Z) = \prod M(0,2,3,4,5,6)$$

$$= \sum m(1,7,8,9,10,11,12,13,14,15)$$



$$F(W,X,Y,Z) = W + XYZ + X'Y'Z$$

I don't care!

- You don't always need all 2^n input combinations in an n -variable function
 - If you can guarantee that certain input combinations never occur
 - If some outputs aren't used in the rest of the circuit
- We mark don't-care outputs in truth tables and K-maps with Xs.

x	y	z	$f(x,y,z)$
0	0	0	0
0	0	1	1
0	1	0	X
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	X
1	1	1	1

- Within a K-map, each X can be considered as either 0 or 1. You should pick the interpretation that allows for the most simplification.

Practice K-map

- Find a MSP for

$$f(w,x,y,z) = \sum m(0,2,4,5,8,14,15), \quad d(w,x,y,z) = \sum m(7,10,13)$$

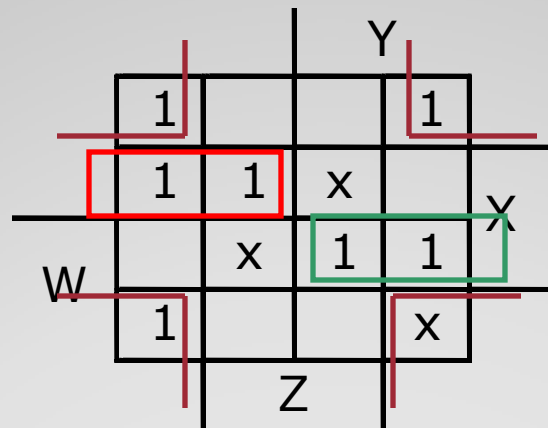
This notation means that input combinations $wxyz = 0111, 1010$ and 1101 (corresponding to minterms m_7, m_{10} and m_{13}) are unused.

	1	0	0	1	
	1	1	x	0	X
W	0	x	1	1	
	1	0	0	x	
	Z				

Solutions for Practice K-map

- Find a MSP for:

$$f(w,x,y,z) = \sum m(0,2,4,5,8,14,15), \quad d(w,x,y,z) = \sum m(7,10,13)$$



$$f(w,x,y,z) = x'z' + w'xy' + wxy$$

K-map Summary

- K-maps are an alternative to algebra for simplifying expressions
 - The result is a MSP/MPS, which leads to a minimal two-level circuit
 - It's easy to handle don't-care conditions
 - K-maps are really only good for manual simplification of small expressions...
- Things to keep in mind:
 - Remember the correct order of minterms/maxterms on the K-map
 - When grouping, you can wrap around all sides of the K-map, and your groups can overlap
 - Make as few rectangles as possible, but make each of them as large as possible. This leads to fewer, but simpler, product terms
 - There may be more than one valid solution

Tabular method notation

- Consider Support Set of f : $S = \{x_1, x_2, \dots, x_n\}$
- $x_i^{c_i}$ denotes:
 - x_i if $\bar{c}_i = '1'$
 - x_i if $c_i = '0'$
 - 1 if $c_i = '-'$
- If NO $c_i = -$, then we have a minterm
 - Can be Represented by Decimal Equivalent of c_i

EXAMPLE $(S = \{x_1, x_2, x_3, x_4\})$

$$x_1^1 x_2^0 x_3^0 x_4^1 = m_9, \text{ a minterm} \rightarrow c_1 c_2 c_3 c_4 = 1001 = 9$$

$$x_1^0 x_2^- x_3^- x_4^0 = \text{a 4-cube} \rightarrow 0--0$$

Cube merging

- Basic Operation in Tabulation Method
- 2 Cubes that Differ in a SINGLE c_i can be Merged into a Single Cube

EXAMPLE

$$\alpha = 1-01$$

$$\beta = 0-01$$

Merge α and β into γ

$$\gamma = --01$$

Merging is also called star operator and is a special case of

ab \ cd	cd			
	00	01	11	10
00		1		
01		1		
11		1		
10		1		

ab \ cd	cd			
	00	01	11	10
00		1		
01		1		
11		1		
10		1		

Tabulation Method

Input: f^{on} as a set of minterms

Output: f^{on} as a set of

1. All Essential Prime Implicants
2. As Few Prime Implicants as Possible

Finding as few Prime Implicants as Possible is an NP-Hard Problem!!!!

- Reduces to the “Set Covering” Problem for **Unate Functions**
Unate function – a constant or is represented by a SOP using either uncomplemented or complemented literals for each variable
- Reduces to the “Minimum Cost Assignment” Problem for Binate Functions (ex. EXOR)

This is 2-Level (SOP) Optimization (Minimization)

Tabulation Method

STEP 1:

- Convert Minterm List (specifying f^{on}) to Prime Implicant List

STEP 2:

- Choose All Essential Prime Implicants
- If all minterms are covered
 HALT
 Else
 GO To STEP 3

STEP 3:

- Formulate the Reduced Cover Table Omitting the rows/cols of EPI
- If Cover Table can be Reduced using Dominance Properties,
 Go To Step 2

- Else Must Solve the “Cyclic Cover” Problem
 - 1) Use Exact Method (exponentially complex)
 - 2) Use Heuristic Method (possibly non-optimal result)

NOTE: “Quine-McCluskey” Refers to Using a “Branch and Bound” Heuristic

NOTE: “Petrick’s Method” is Exact Technique – Generates all Solutions Allowing the Best to be Used

Tabular method –step 1

1. Partition Prime Implicants (or minterms)
According to Number of 1's
2. Check Adjacent Classes for Cube Merging
Building a New List
3. If Entry in New List Covers Entry in Current List
– Disregard Current List Entry
4. If Current List = New List
 HALT
 Else
 Current List $\leftarrow$ New List
 New List $\leftarrow$ NULL
 Go To Step 1

STEP 1 - EXAMPLE

$$f^{on} = \{m_0, m_1, m_2, m_3, m_5, m_8, m_{10}, m_{11}, m_{13}, m_{15}\} = \Sigma (0, 1, 2, 3, 5, 8, 10, 11, 13, 15)$$

Minterm	Cube			
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
8	1	0	0	0
3	0	0	1	1
5	0	1	0	1
10	1	0	1	0
11	1	0	1	1
13	1	1	0	1
15	1	1	1	1

STEP 1 - EXAMPLE

$$f^{on} = \{m_0, m_1, m_2, m_3, m_5, m_8, m_{10}, m_{11}, m_{13}, m_{15}\} = \Sigma (0, 1, 2, 3, 5, 8, 10, 11, 13, 15)$$

Minterm	Cube				
0	0	0	0	0	✓
1	0	0	0	1	✓
2	0	0	1	0	✓
8	1	0	0	0	✓
3	0	0	1	1	✓
5	0	1	0	1	✓
10	1	0	1	0	✓
11	1	0	1	1	✓
13	1	1	0	1	✓
15	1	1	1	1	✓

Minterm	Cube			
0,1	0	0	0	-
0,2	0	0	-	0
0,8	-	0	0	0
1,3	0	0	-	1
1,5	0	-	0	1
2,3	0	0	1	-
2,10	-	0	1	0
8,10	1	0	-	0
3,11	-	0	1	1
5,13	-	1	0	1
10,11	1	0	1	-
11,15	1	-	1	1
13,15	1	1	-	1

STEP 1 - EXAMPLE

$$f^{on} = \{m_0, m_1, m_2, m_3, m_5, m_8, m_{10}, m_{11}, m_{13}, m_{15}\} = \Sigma (0, 1, 2, 3, 5, 8, 10, 11, 13, 15)$$

Minterm	Cube				
0	0	0	0	0	✓
1	0	0	0	1	✓
2	0	0	1	0	✓
8	1	0	0	0	✓
3	0	0	1	1	✓
5	0	1	0	1	✓
10	1	0	1	0	✓
11	1	0	1	1	✓
13	1	1	0	1	✓
15	1	1	1	1	✓

Minterm	Cube				
0,1	0	0	0	-	✓
0,2	0	0	-	0	✓
0,8	-	0	0	0	✓
1,3	0	0	-	1	✓
1,5	0	-	0	1	
2,3	0	0	1	-	✓
2,10	-	0	1	0	✓
8,10	1	0	-	0	✓
3,11	-	0	1	1	✓
5,13	-	1	0	1	
10,11	1	0	1	-	✓
11,15	1	-	1	1	
13,15	1	1	-	1	

Minterm	Cube			
0,1,2,3	0	0	-	-
0,8,2,10	-	0	-	0
2,3,10,11	-	0	1	-

STEP 1 - EXAMPLE

$$f^{on} = \{m_0, m_1, m_2, m_3, m_5, m_8, m_{10}, m_{11}, m_{13}, m_{15}\} = \Sigma(0, 1, 2, 3, 5, 8, 10, 11, 13, 15)$$

Minterm	Cube		Minterm	Cube		Minterm	Cube	
0	0 0 0 0	✓	0,1	0 0 0 -	✓	0,1,2,3	0 0 - -	PI=A
1	0 0 0 1	✓	0,2	0 0 - 0	✓	0,8,2,10	- 0 - 0	PI=C
2	0 0 1 0	✓	0,8	- 0 0 0	✓	2,3,10,11	- 0 1 -	PI=B
8	1 0 0 0	✓	1,3	0 0 - 1	✓			
3	0 0 1 1	✓	1,5	0 - 0 1	PI=D			
5	0 1 0 1	✓	2,3	0 0 1 -	✓			
10	1 0 1 0	✓	2,10	- 0 1 0	✓			
11	1 0 1 1	✓	8,10	1 0 - 0	✓			
13	1 1 0 1	✓	3,11	- 0 1 1	✓			
15	1 1 1 1	✓	5,13	- 1 0 1	PI=E			
			10,11	1 0 1 -	✓			
			11,15	1 - 1 1	PI=F			
			13,15	1 1 - 1	PI=G			

$$f^{on} = \{A, B, C, D, E, F, G\} = \{00--, -01-, -0-0, 0-01, -101, 1-11, 11-1\}$$

STEP 2 – Construct Cover Table

- PIs Along Vertical Axis (in order of # of literals)
- Minterms Along Horizontal Axis

[illegible]

STEP 2 – Finding the Minimum Cover

- Extract All **Essential Prime** Implicants, EPI
- EPIs are the PI for which a Single x Appears in a Column

	0	1	2	3	5	8	10	11	13	15
A	x	x	x	x						
B			x	x			x	x		
C	x		x			x	x			
D		x			x					
E					x				x	
F								x		x
G									x	x

- C is an EPI so: $f_{on} = \{C, \dots\}$
- Row C and Columns 0, 2, 8, and 10 can be Eliminated Giving Reduced Cover Table
- Examine Reduced Table for New EPIs

STEP 2 – Reduced Table

Distinguished Column

	0	1	2	3	5	8	10	11	13	15
A	x	x	x	x						
B			x	x			x	x		
C	x		x			x	x			
D		x			x					
E					x				x	
F								x		x
G									x	x

Essential row



	1	3	5	11	13	15
A	x	x				
B		x		x		
D	x		x			
E			x		x	
F				x		x
G					x	x

- The Row of an EPI is an *Essential row*
- The Column of the Single x in the *Essential Row* is a *Distinguished Column*

Row and Column Dominance

- If Row P has x 's Everywhere Row Q Does
Then Q Dominates P if P has fewer x 's
- If Column i has x 's Everywhere j Does
Then j Dominates i if i has fewer x 's
- If Row P is equal to Row Q and Row Q does not cost more than Row P , eliminate Row P , or if Row P is dominated by Row Q and Row Q Does not cost more than Row P , eliminate Row P
- If Column i is equal to Column j , eliminate Column i or if Column i dominates Column j , eliminate Column i

STEP 3 – The Reduced Cover Table

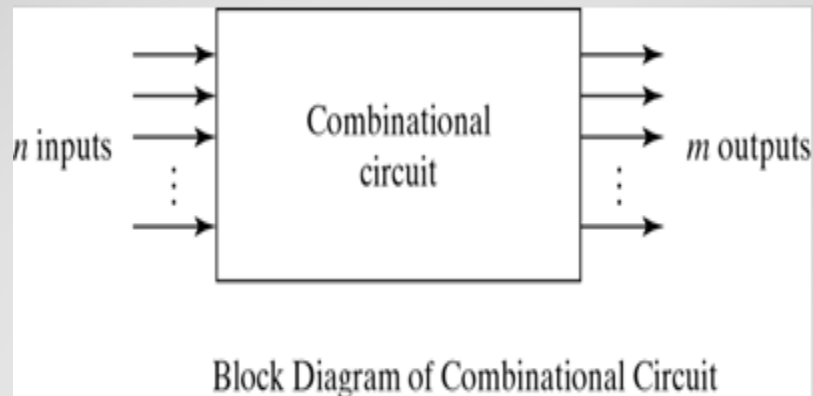
- Initially, Columns 0, 2, 8 and 10 Removed

	1	3	5	11	13	15
A	x	x				
B		x		x		
D	x		x			
E			x		x	
F				x		x
G					x	x

- No EPIs are Present
- No Row Dominance Exists
- No Column Dominance Exists
- This is *Cyclic Cover* Table
- Must Solve Exactly OR Use a Heuristic

Combinational Logic

- Logic circuits for digital systems may be combinational or sequential.
- A combinational circuit consists of input variables, logic gates, and output variables.



Analysis procedure

- To obtain the output Boolean functions from a logic diagram, proceed as follows:
 1. Label all gate outputs that are a function of input variables with arbitrary symbols. Determine the Boolean functions for each gate output.
 2. Label the gates that are a function of input variables and previously labeled gates with other arbitrary symbols. Find the Boolean functions for these gates.

Analysis procedure...

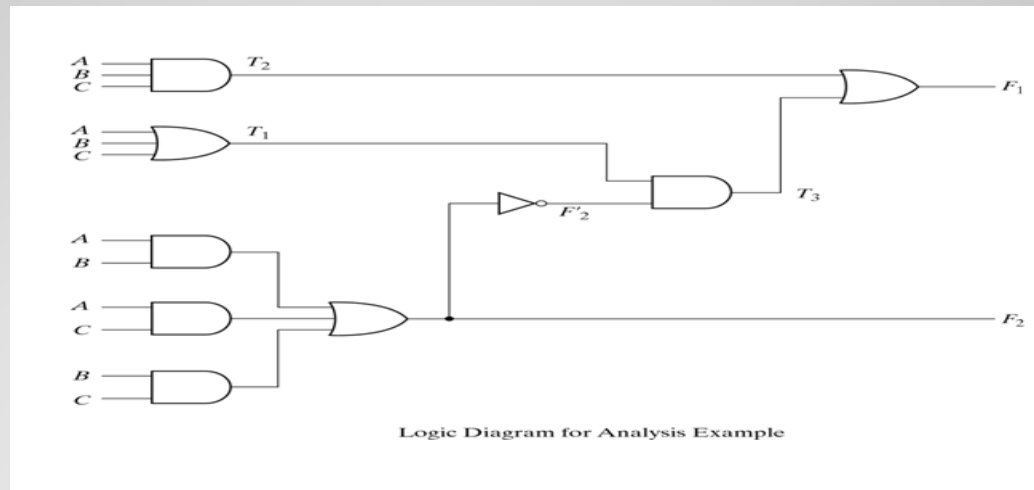
3. Repeat the process outlined in step 2 until the outputs of the circuit are obtained.
4. By repeated substitution of previously defined functions, obtain the output Boolean functions in terms of input variables.

Example

$$F_2 = AB + AC + BC; \quad T_1 = A + B + C; \quad T_2 = ABC; \quad T_3 = F_2'T_1;$$

$$F_1 = T_3 + T_2$$

$$F_1 = T_3 + T_2 = F_2'T_1 + ABC = A'BC' + A'B'C + AB'C' + ABC$$



Derive truth table from logic diagram

- We can derive the truth table in below Table by using the circuit of above Fig.

<i>A</i>	<i>B</i>	<i>C</i>	<i>F₂</i>	<i>F₂</i>	<i>T₁</i>	<i>T₂</i>	<i>T₃</i>	<i>F₁</i>
0	0	0	0	1	0	0	0	0
0	0	1	0	1	1	0	1	1
0	1	0	0	1	1	0	1	1
0	1	1	1	0	1	0	0	0
1	0	0	0	1	1	0	1	1
1	0	1	1	0	1	0	0	0
1	1	0	1	0	1	0	0	0
1	1	1	1	0	1	1	0	1

Design procedure

1. Table 4-2 is a Code-Conversion example, first, we can list the relation of the BCD and Excess-3 codes in the truth table.

Table 4-2

Truth Table for Code-Conversion Example

Input BCD				Output Excess-3 Code			
A	B	C	D	w	x	y	z
0	0	0	0	0	0	1	1
0	0	0	1	0	1	0	0
0	0	1	0	0	1	0	1
0	0	1	1	0	1	1	0
0	1	0	0	0	1	1	1
0	1	0	1	1	0	0	0
0	1	1	0	1	0	0	1
0	1	1	1	1	0	1	0
1	0	0	0	1	0	1	1
1	0	0	1	1	1	0	0

Karnaugh map

- For each symbol of the Excess-3 code, we use 1's to draw the map for simplifying Boolean function.

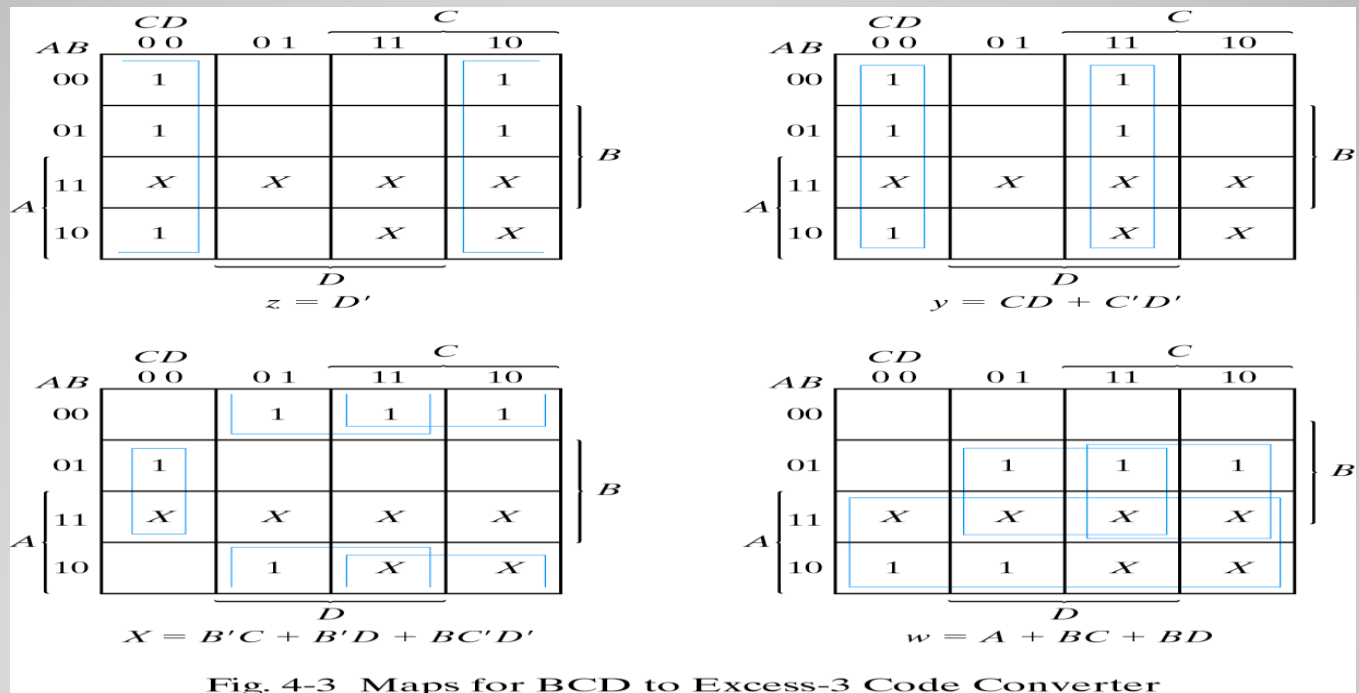


Fig. 4-3 Maps for BCD to Excess-3 Code Converter

Circuit implementation

$$z = D'; \quad y = CD + C'D' = CD + (C + D)'$$

$$x = B'C + B'D + BC'D' = B'(C + D) + B(C + D)'$$

$$w = A + BC + BD = A + B(C + D)$$

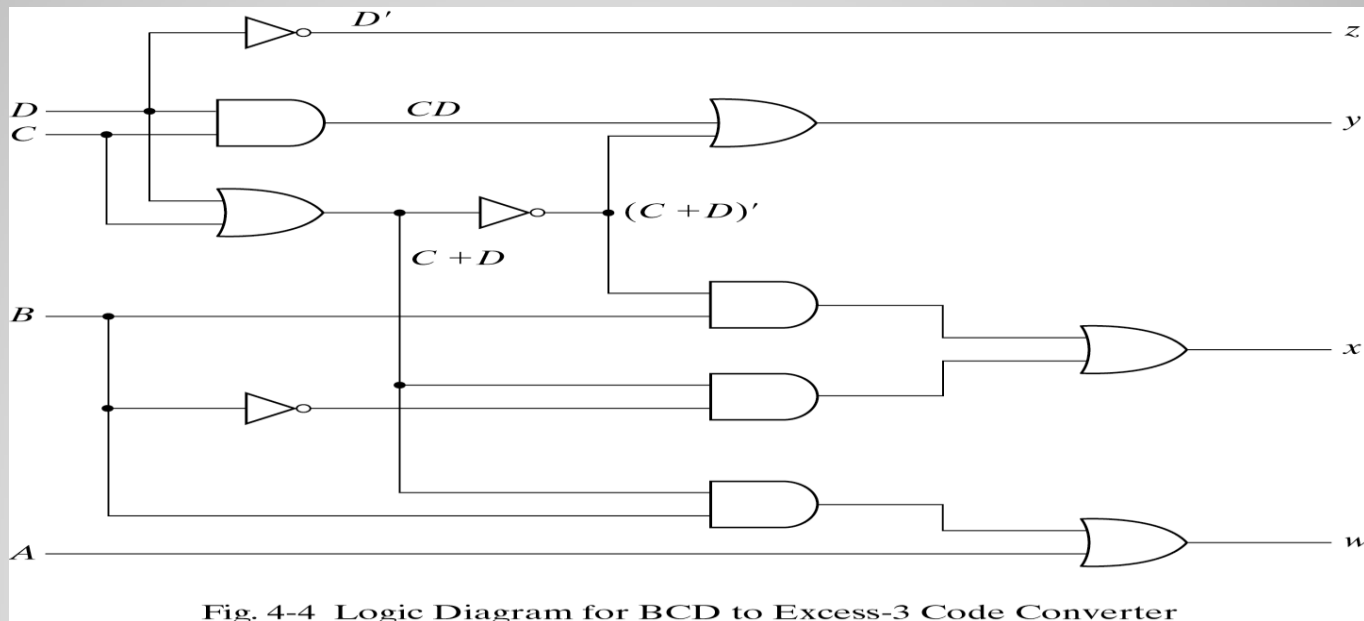


Fig. 4-4 Logic Diagram for BCD to Excess-3 Code Converter

Binary Adder-Subtractor

- A combinational circuit that performs the addition of two bits is called a **half adder**.
- The truth table for the half adder is listed below:

Table 4-3
Half Adder

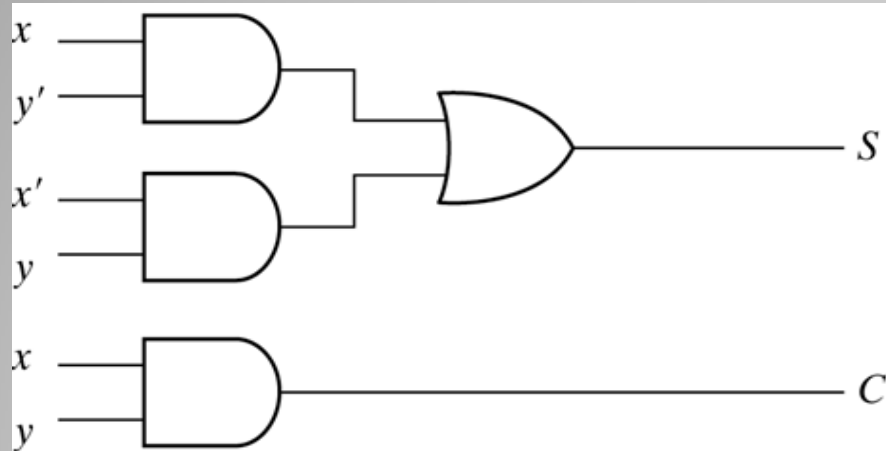
<i>x</i>	<i>y</i>	<i>C</i>	<i>S</i>
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

S: Sum
C: Carry

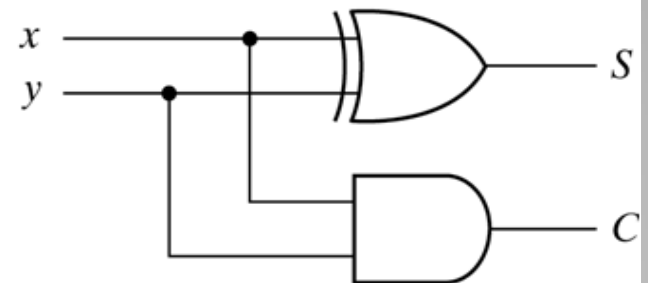
$$S = x'y + xy'$$

$$C = xy$$

Implementation of Half-Adder



$$(a) \begin{aligned} S &= xy' + x'y \\ C &= xy \end{aligned}$$



$$(b) \begin{aligned} S &= x \oplus y \\ C &= xy \end{aligned}$$

Implementation of Half-Adder

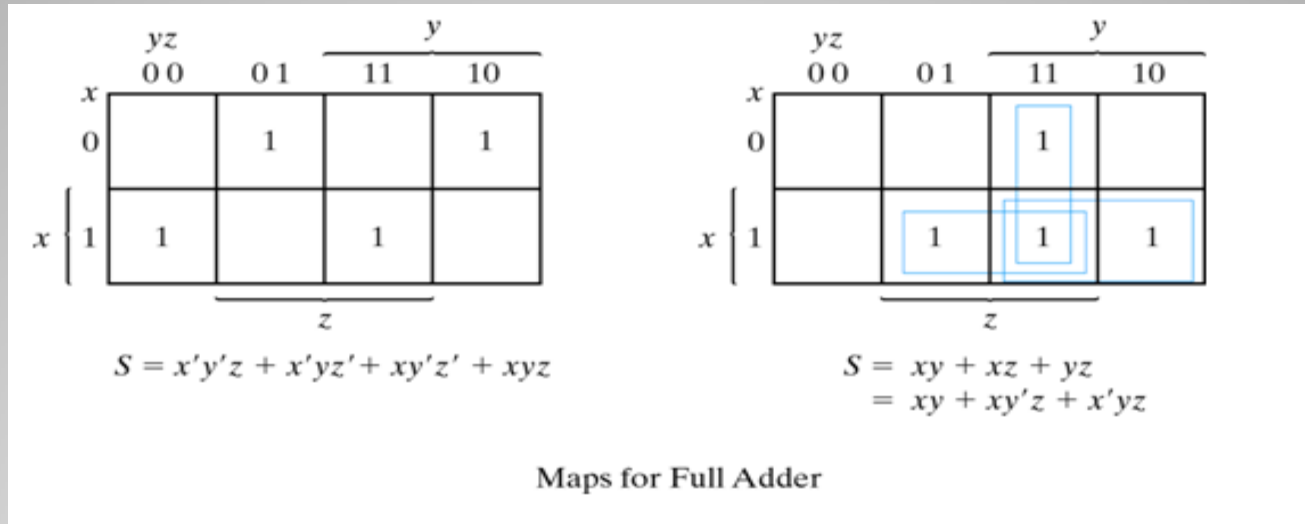
Full-Adder

- One that performs the addition of three bits (two significant bits and a previous carry) is a **full adder**.

Table
Full Adder

<i>x</i>	<i>y</i>	<i>z</i>	<i>C</i>	<i>S</i>
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

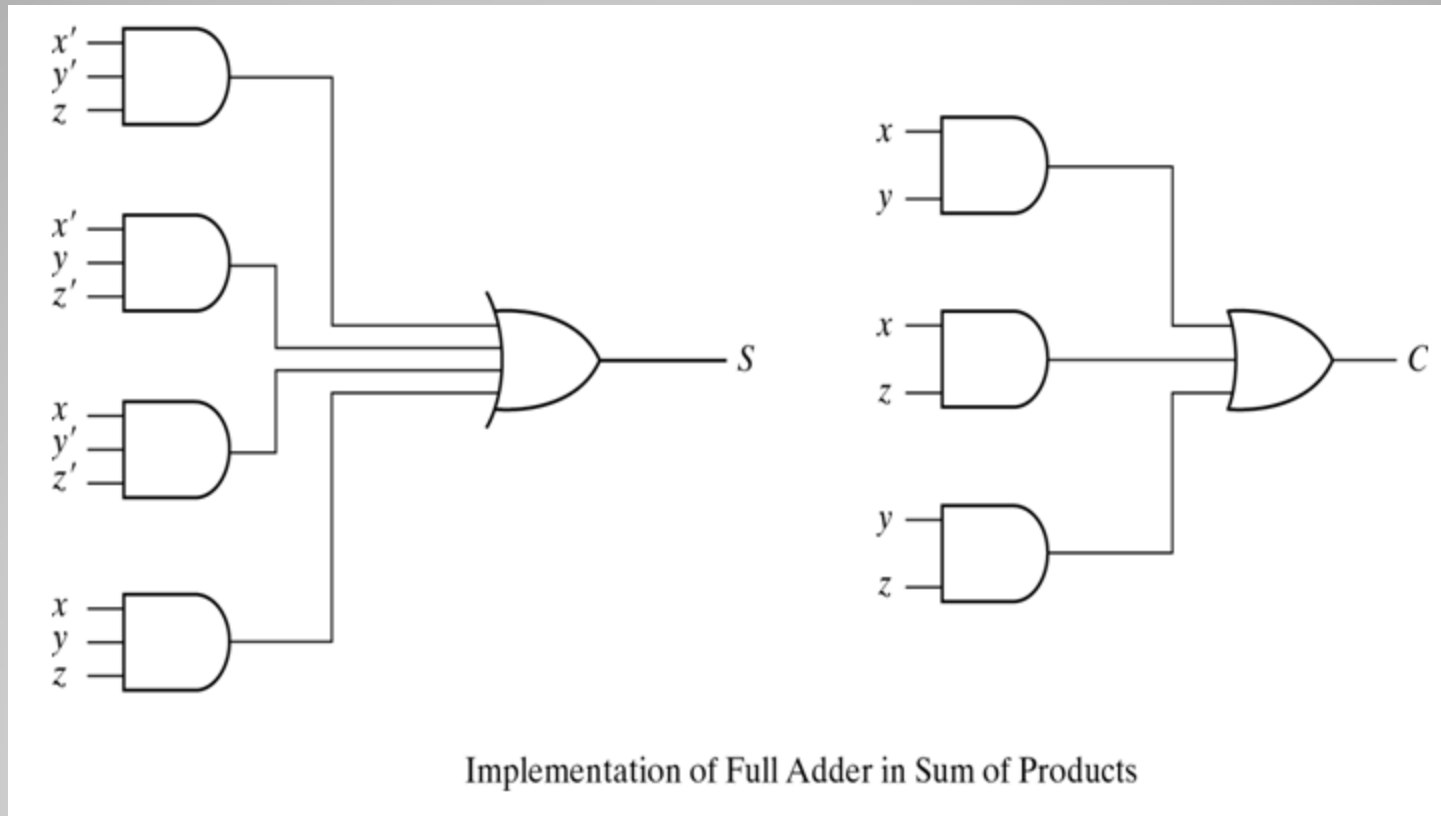
Simplified Expressions



$$S = x'y'z + x'yz' + xy'z' + xyz$$

$$C = xy + xz + yz$$

Full adder implemented in SOP



Another implementation

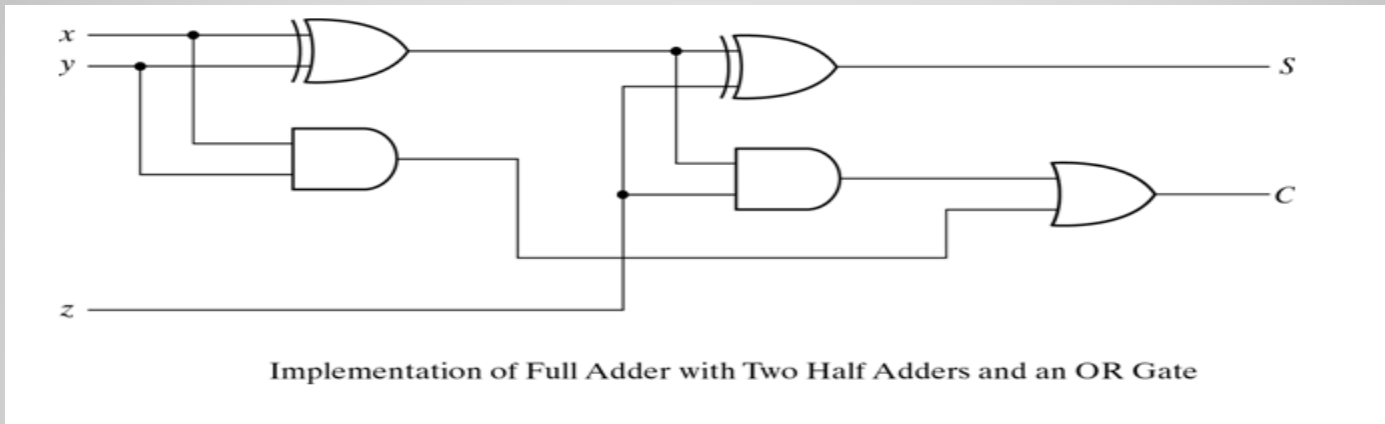
- Full-adder can also be implemented with two half adders and one OR gate (Carry Look-Ahead adder).

$$S = z \oplus (x \oplus y)$$

$$= z'(xy' + x'y) + z(xy' + x'y)'$$

$$= xy'z' + x'yz' + xyz + x'y'z$$

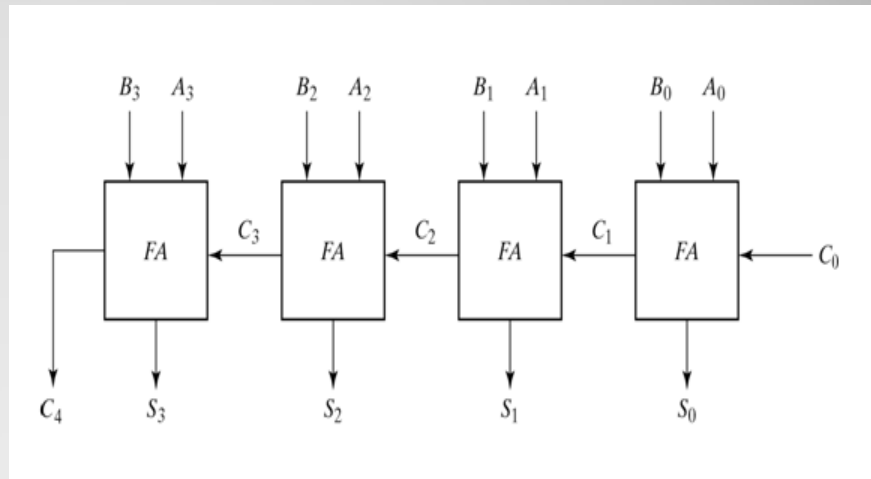
$$C = z(xy' + x'y) + xy = xy'z + x'yz + xy$$



Binary adder

- This is also called **Ripple Carry Adder**, because of the construction with full adders are connected in cascade.

<i>Subscript i:</i>	3	2	1	0	
Input carry	0	1	1	0	C_i
Augend	1	0	1	1	A_i
Addend	0	0	1	1	B_i
Sum	1	1	1	0	S_i
Output carry	0	0	1	1	C_{i+1}

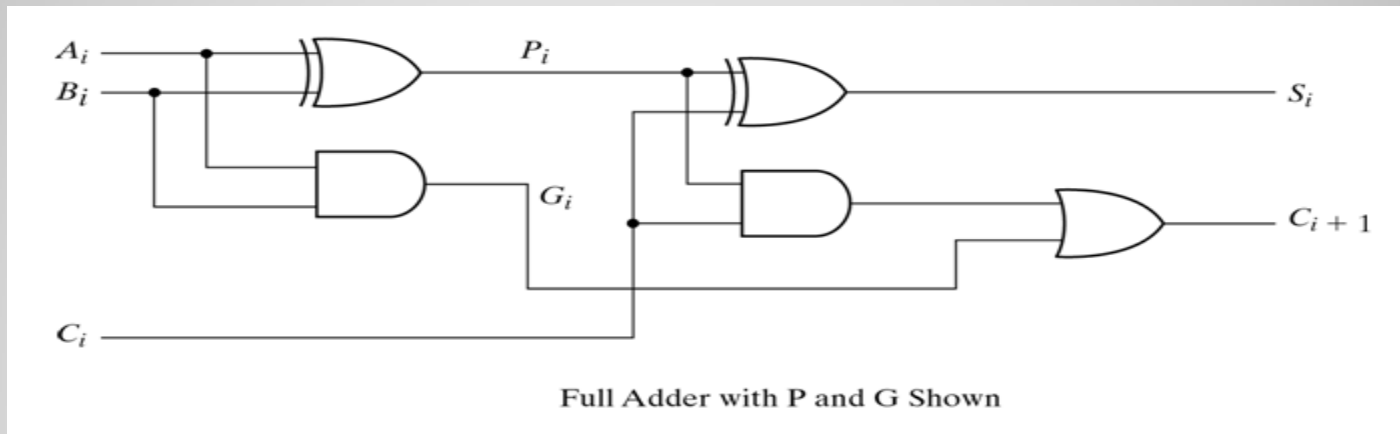


Carry Propagation

- Fig.4-9 causes a **unstable** factor on **carry bit**, and produces a **longest propagation delay**.
- The signal from C_i to the output carry C_{i+1} , **propagates through an AND and OR gates**, so, for an n-bit RCA, there are **$2n$** gate levels for the carry to propagate from input to output.

Carry Propagation

- Because the propagation delay will affect the output signals on different time, so the signals are given enough time to get the precise and stable outputs.
- The most widely used technique employs the principle of carry look-ahead to improve the speed of the algorithm.



Boolean functions

$$P_i = A_i \oplus B_i \quad \text{steady state value}$$

$$G_i = A_i B_i \quad \text{steady state value}$$

Output sum and carry

$$S_i = P_i \oplus C_i$$

$$C_{i+1} = G_i + P_i C_i$$

G_i : carry generate P_i : carry propagate

C_0 = input carry

$$C_1 = G_0 + P_0 C_0$$

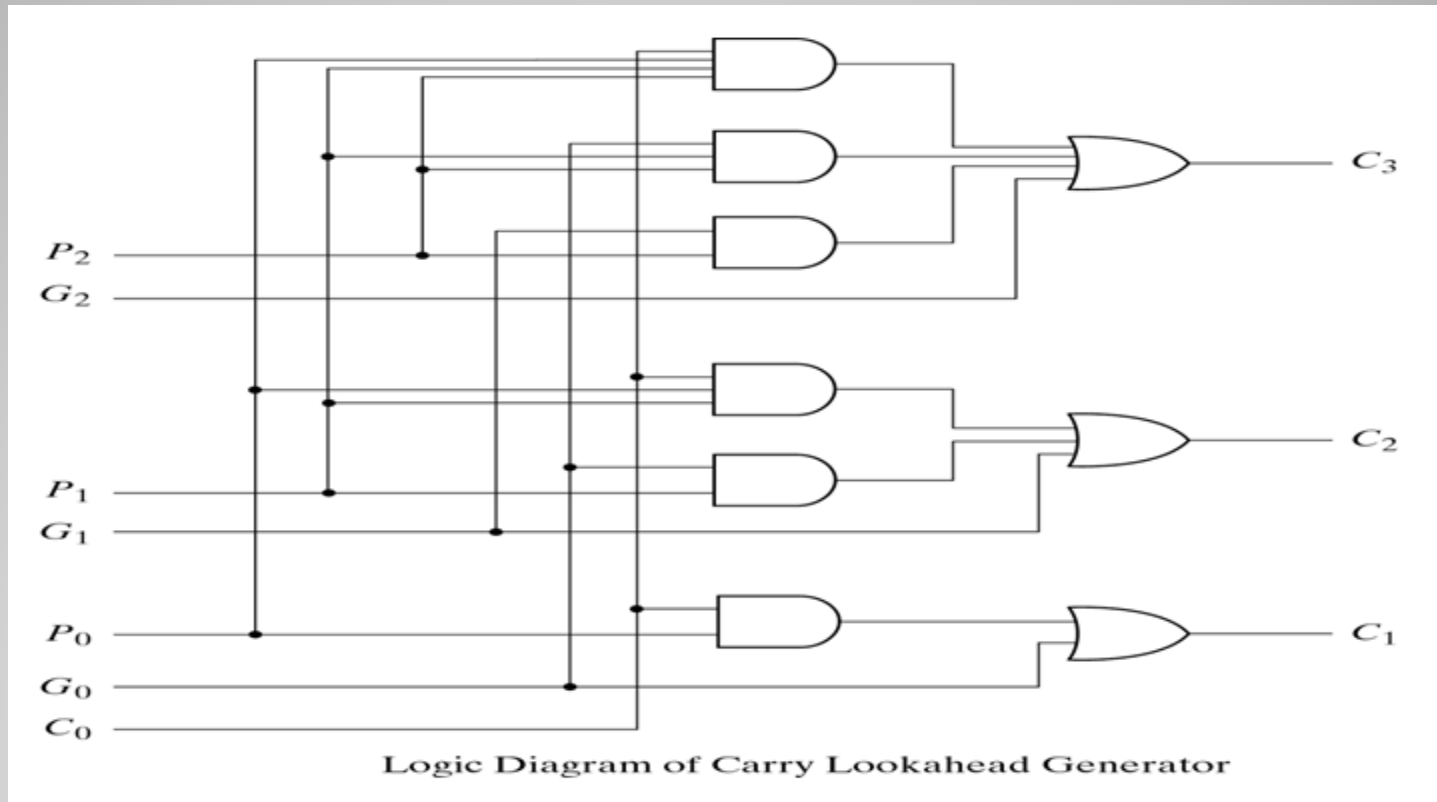
$$C_2 = G_1 + P_1 C_1 = G_1 + P_1 G_0 + P_1 P_0 C_0$$

$$C_3 = G_2 + P_2 C_2 = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_0$$

- C_3 does not have to wait for C_2 and C_1 to propagate.

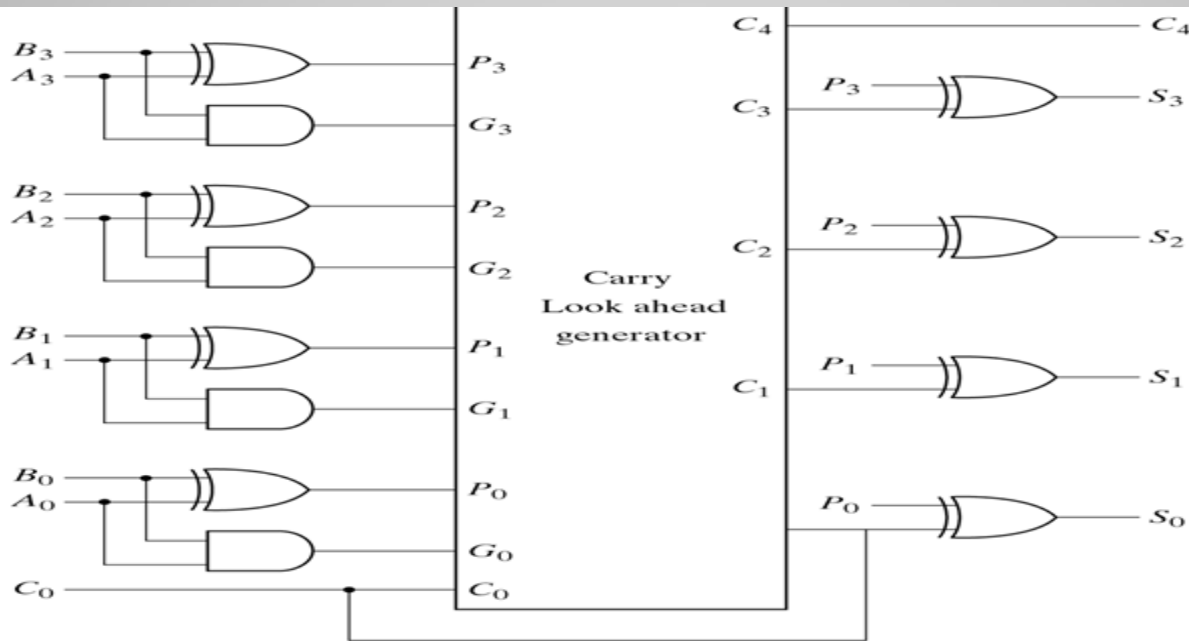
Logic diagram of carry look-ahead generator

- C_3 is propagated at the same time as C_2 and C_1 .



4-bit adder with carry lookahead

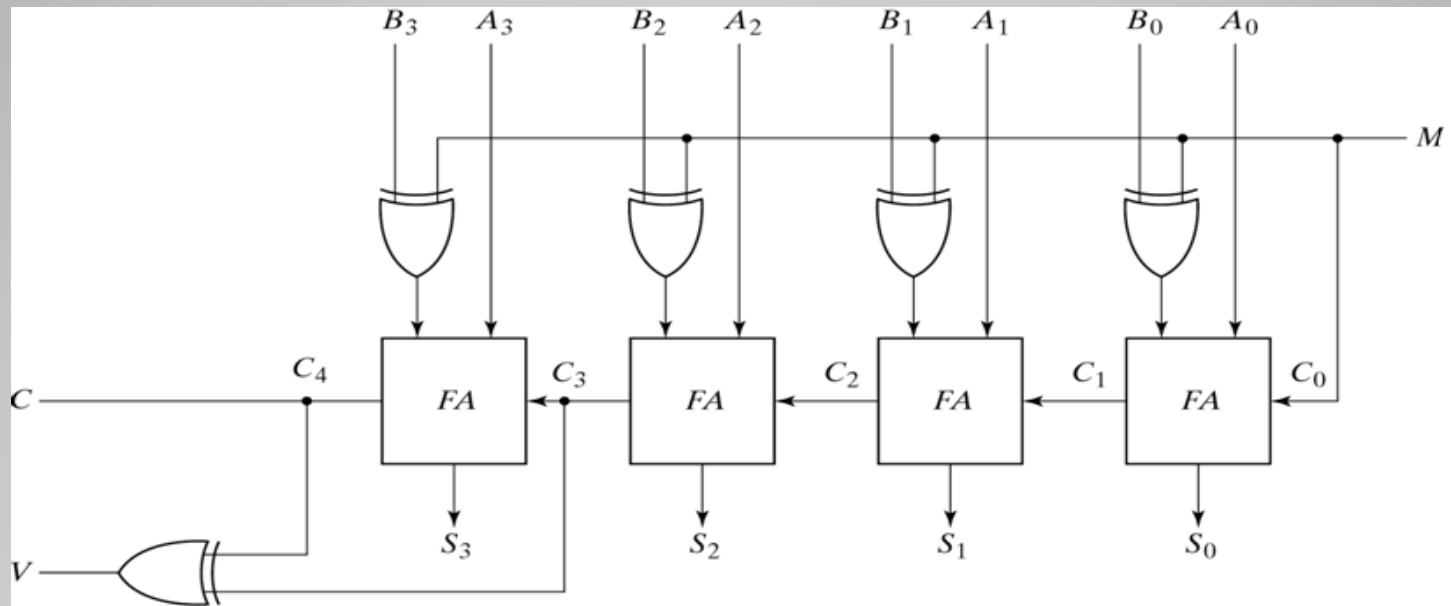
- Delay time of n-bit CLAA = XOR + (AND + OR) + XOR



4-Bit Adder with Carry Lookahead

Binary subtractor

$M = 1 \rightarrow$ subtractor ; $M = 0 \rightarrow$ adder



4-Bit Adder Subtractor

Overflow

- It is worth noting in above figure that binary numbers in the signed-complement system are added and subtracted by the same basic addition and subtraction rules as unsigned numbers.
- Overflow is a problem in digital computers because the number of bits that hold the number is finite and a result that contains $n+1$ bits cannot be accommodated.

Overflow on signed and unsigned

- When two unsigned numbers are added, an overflow is detected from the end carry out of the MSB position.
- When two signed numbers are added, the sign bit is treated as part of the number and the end carry does not indicate an overflow.
- An overflow can't occur after an addition if one number is positive and the other is negative.
- An overflow may occur if the two numbers added are both positive or both negative.

4-5 Decimal adder

BCD adder can't exceed 9 on each input digit. K is the carry.

Derivation of BCD Adder

Binary Sum					BCD Sum					Decimal
K	Z ₈	Z ₄	Z ₂	Z ₁	C	S ₈	S ₄	S ₂	S ₁	
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	2
0	0	0	1	1	0	0	0	1	1	3
0	0	1	0	0	0	0	1	0	0	4
0	0	1	0	1	0	0	1	0	1	5
0	0	1	1	0	0	0	1	1	0	6
0	0	1	1	1	0	0	1	1	1	7
0	1	0	0	0	0	1	0	0	0	8
0	1	0	0	1	0	1	0	0	1	9
0	1	0	1	0	1	0	0	0	0	10
0	1	0	1	1	1	0	0	0	1	11
0	1	1	0	0	1	0	0	1	0	12
0	1	1	0	1	1	0	0	1	1	13
0	1	1	1	0	1	0	1	0	0	14
0	1	1	1	1	1	0	1	0	1	15
1	0	0	0	0	1	0	1	1	0	16
1	0	0	0	1	1	0	1	1	1	17
1	0	0	1	0	1	1	0	0	0	18
1	0	0	1	1	1	1	0	0	1	19

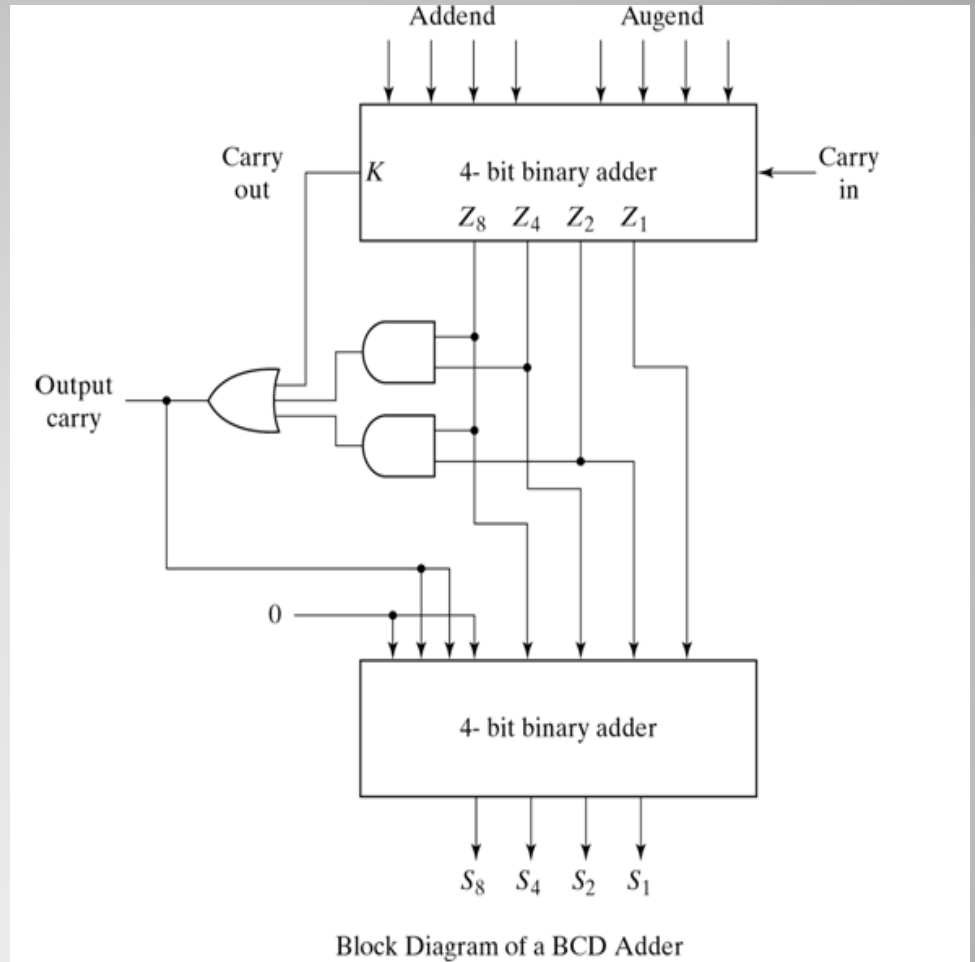
Rules of BCD adder

- When the binary sum is greater than 1001, we obtain a non-valid BCD representation.
- The addition of binary 6(0110) to the binary sum converts it to the correct BCD representation and also produces an output carry as required.
- To distinguish them from binary 1000 and 1001, which also have a 1 in position Z_8 , we specify further that either Z_4 or Z_2 must have a 1.

$$C = K + Z_8Z_4 + Z_8Z_2$$

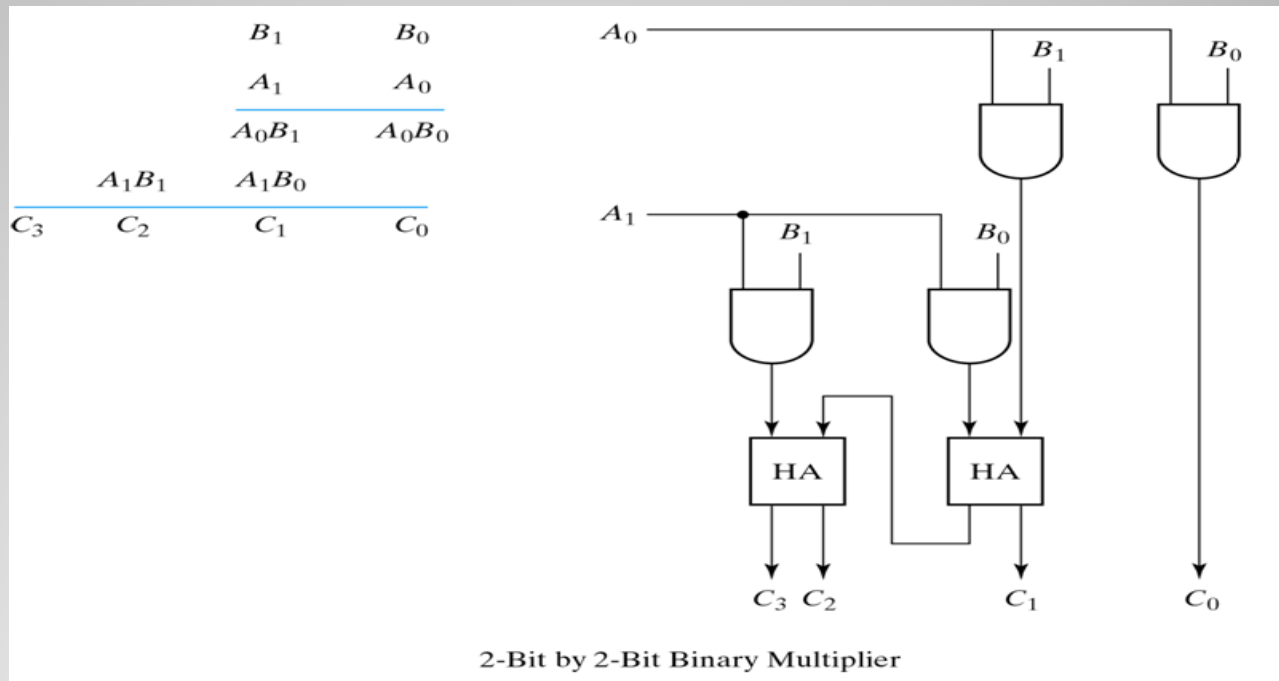
Implementation of BCD adder

- A decimal parallel adder that adds n decimal digits needs n BCD adder stages.
- The output carry from one stage must be connected to the input carry of the next higher-order stage.



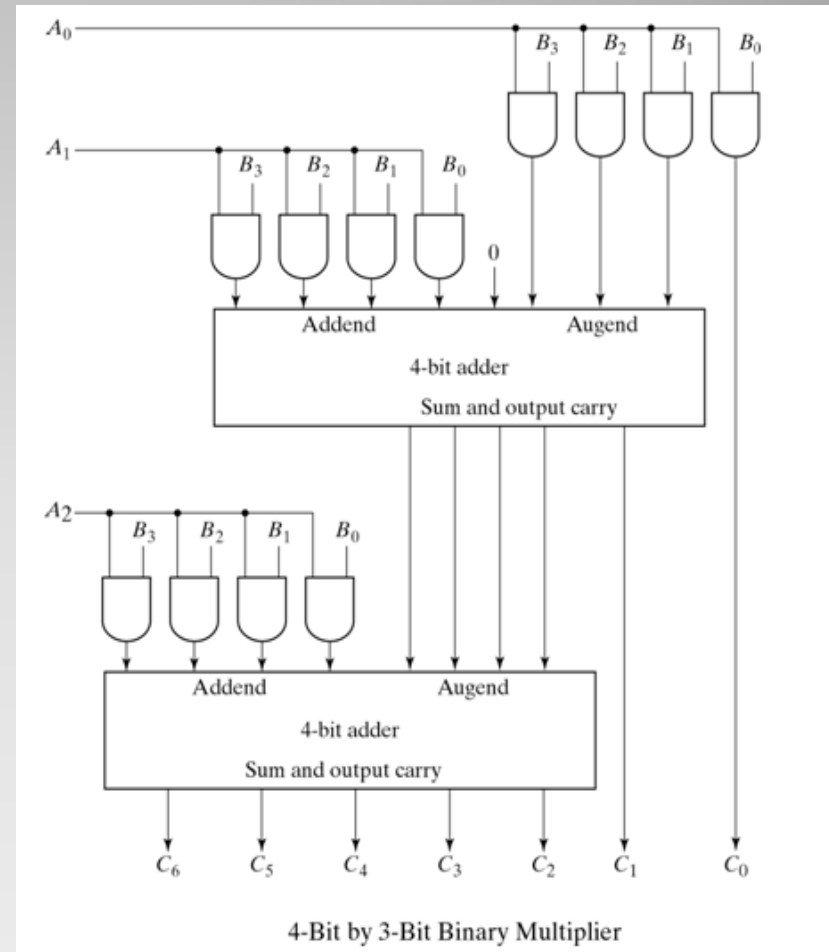
4-6. Binary multiplier

- Usually there are **more bits** in the partial products and it is necessary to use **full adders** to produce the sum of the partial products.



4-bit by 3-bit binary multiplier

- For J multiplier bits and K multiplicand bits we need $(J \times K)$ AND gates and $(J - 1)$ K-bit adders to produce a product of $J+K$ bits.
- $K=4$ and $J=3$, we need 12 AND gates and two 4-bit adders.



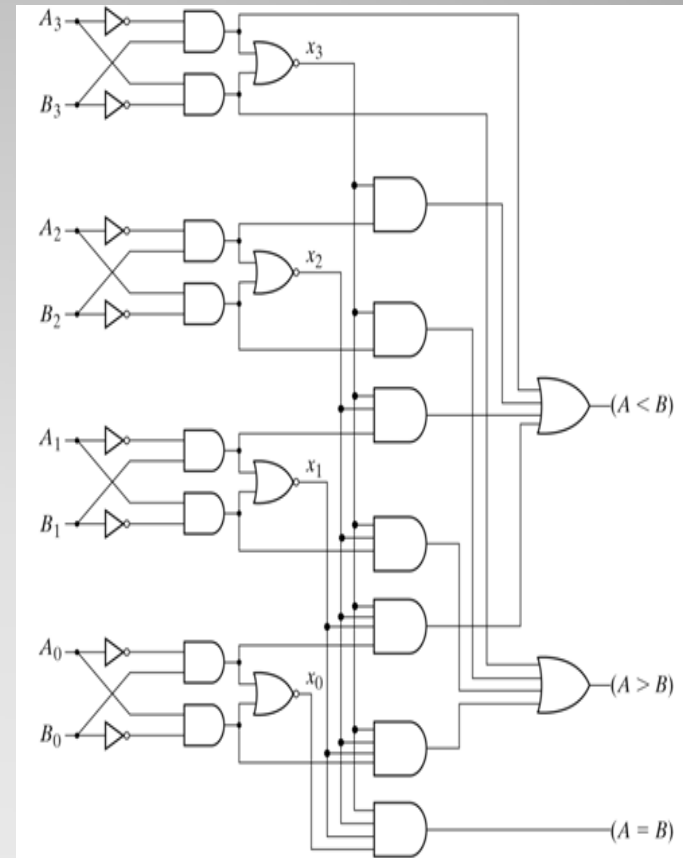
7. Magnitude comparator

- The equality relation of each pair of bits can be expressed logically with an exclusive-NOR function as:

$$A = A_3A_2A_1A_0 ; B = B_3B_2B_1B_0$$

$$x_i = A_iB_i + A_i'B_i' \quad \text{for } i = 0, 1, 2, 3$$

$$(A = B) = x_3x_2x_1x_0$$



4-Bit Magnitude Comparator

Magnitude comparator

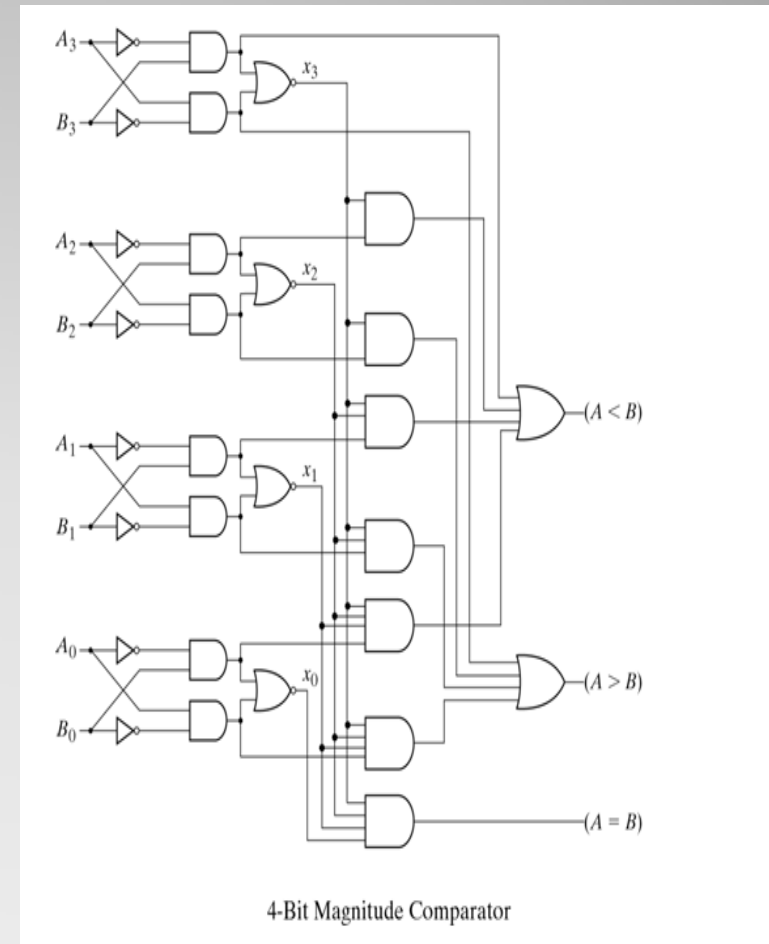
- We inspect the relative magnitudes of pairs of MSB. If equal, we compare the next lower significant pair of digits until a pair of unequal digits is reached.
- If the corresponding digit of A is 1 and that of B is 0, we conclude that $A > B$.

$(A > B) =$

$$A_3 B'_3 + x_3 A_2 B'_2 + x_3 x_2 A_1 B'_1 + x_3 x_2 x_1 A_0 B'_0$$

$(A < B) =$

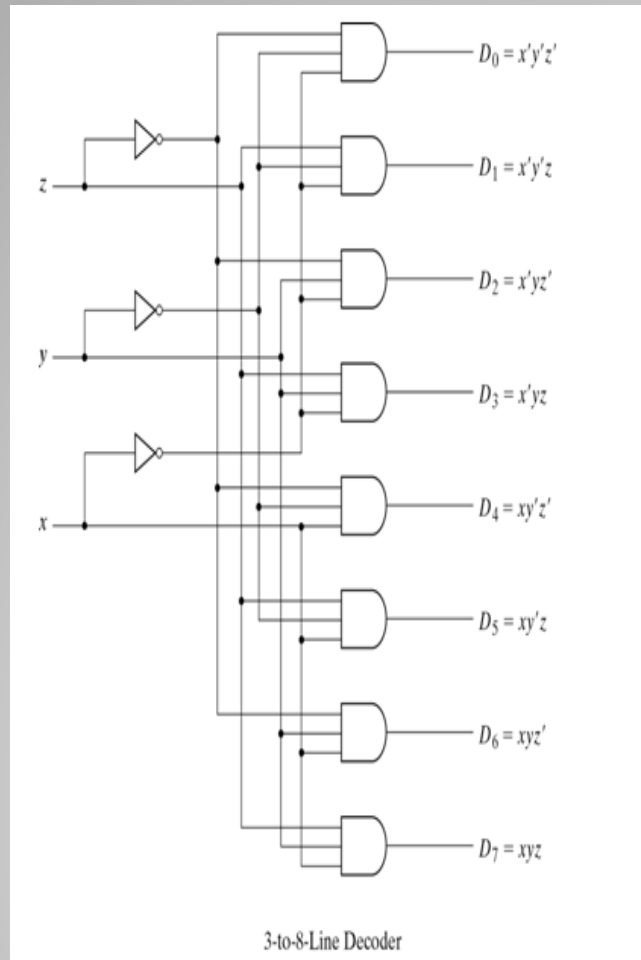
$$A'_3 B_3 + x_3 A'_2 B_2 + x_3 x_2 A'_1 B_1 + x_3 x_2 x_1 A'_0 B_0$$



Decoders

- The decoder is called n-to-m-line decoder, where $m \leq 2^n$.
- the decoder is also used in conjunction with other code converters such as a BCD-to-seven segment decoder.
- 3-to-8 line decoder: For each possible input combination, there are seven outputs that are equal to 0 and only one that is equal to 1.

Implementation and truth table

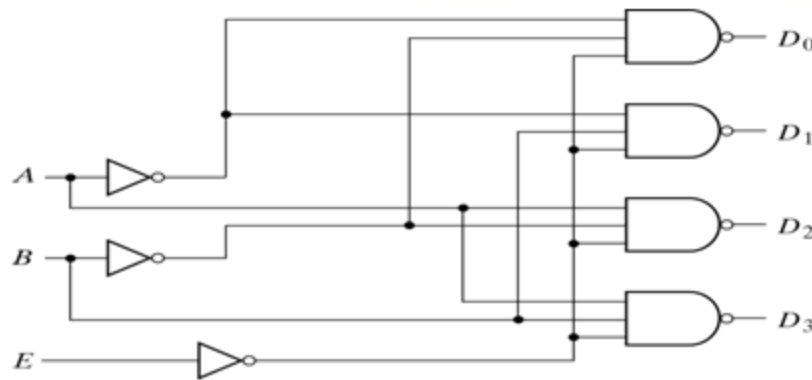


Truth Table of a 3-to-8-Line Decoder

Inputs			Outputs							
x	y	z	D ₀	D ₁	D ₂	D ₃	D ₄	D ₅	D ₆	D ₇
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

Decoder with enable input

- Some decoders are constructed with NAND gates, it becomes more economical to generate the decoder minterms in their complemented form.
- As indicated by the truth table , only one output can be equal to 0 at any given time, all other outputs are equal to 1.



(a) Logic diagram

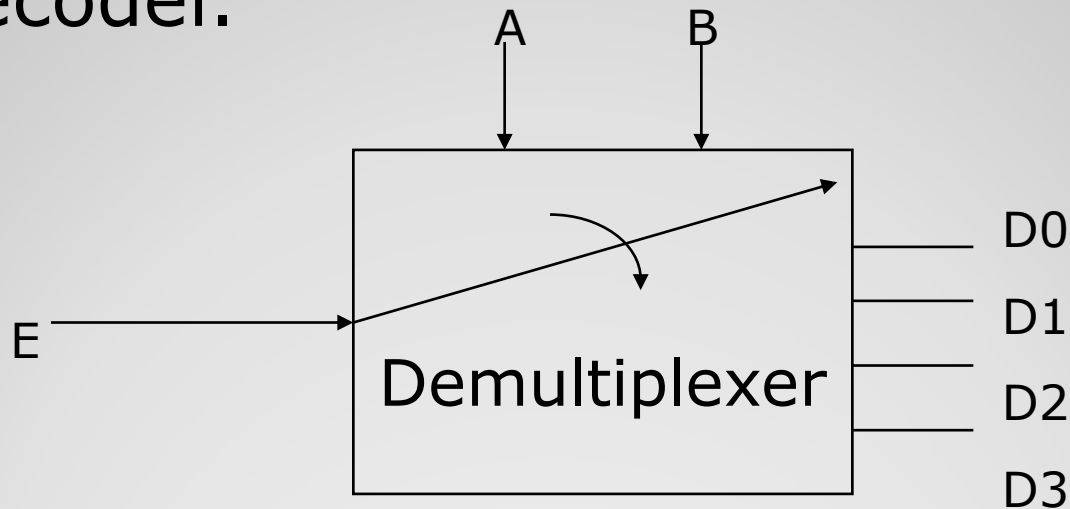
<i>E</i>	<i>A</i>	<i>B</i>	<i>D</i> ₀	<i>D</i> ₁	<i>D</i> ₂	<i>D</i> ₃
1	<i>X</i>	<i>X</i>	1	1	1	1
0	0	0	0	1	1	1
0	0	1	1	0	1	1
0	1	0	1	1	0	1
0	1	1	1	1	1	0

(b) Truth table

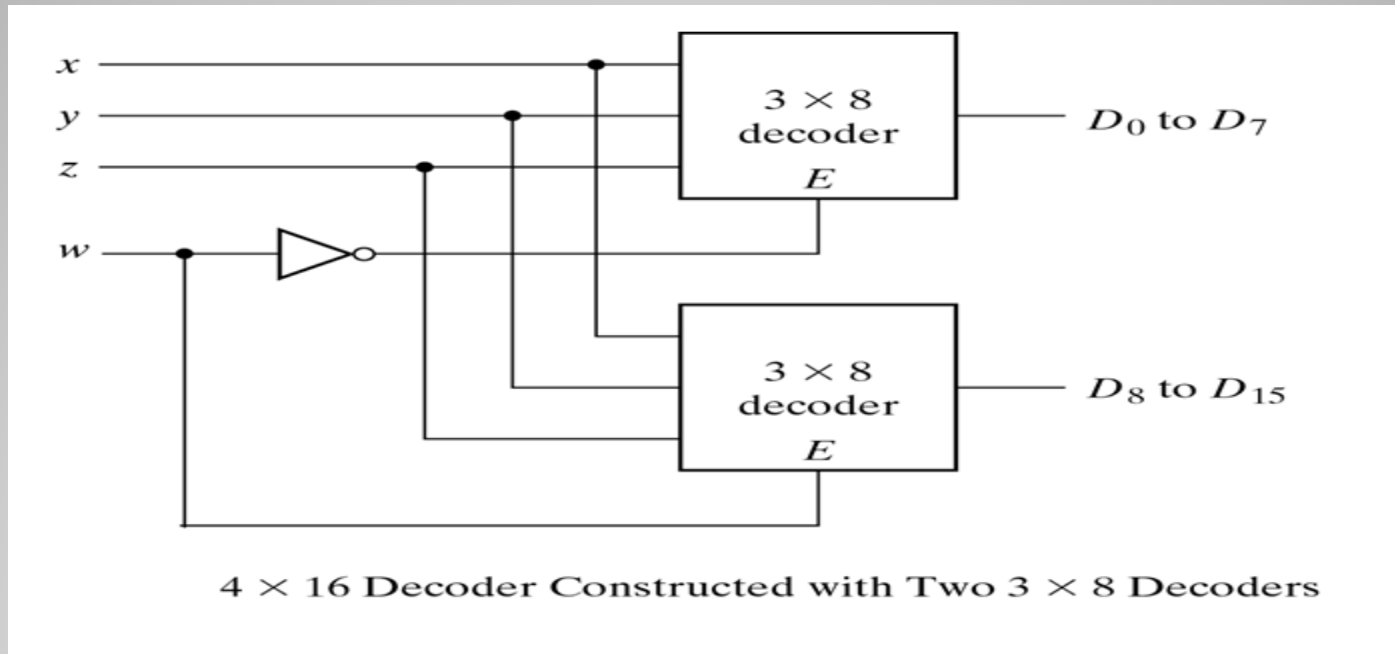
2-to-4-Line Decoder with Enable Input

Demultiplexer

- A decoder with an enable input is referred to as a decoder/demultiplexer.
- The truth table of demultiplexer is the same with decoder.



3-to-8 decoder with enable implement the 4-to-16 decoder

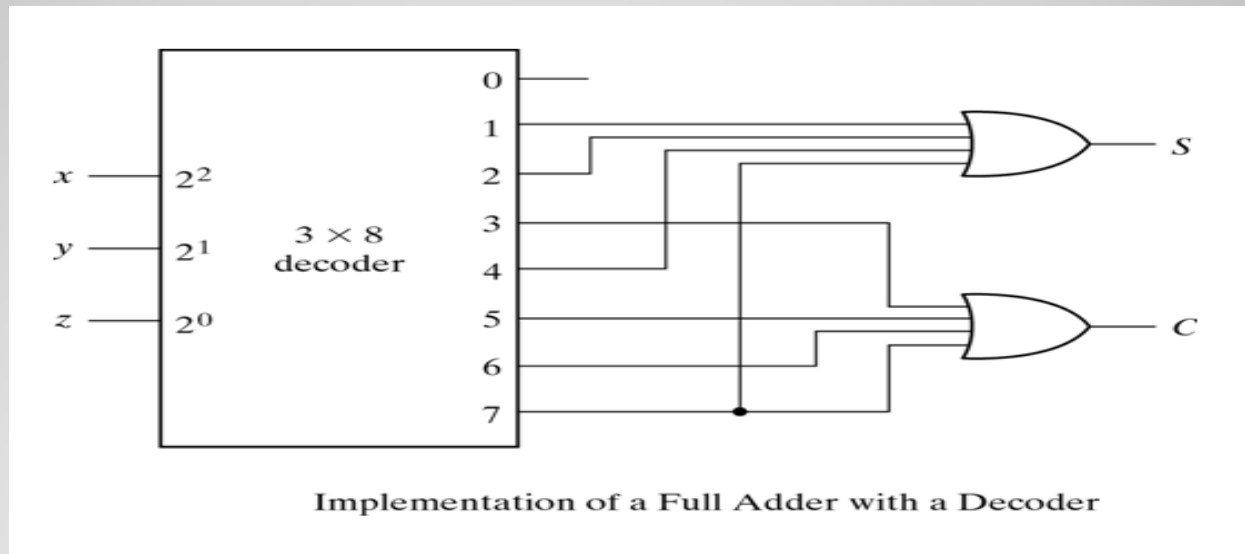


Implementation of a Full Adder with a Decoder

- From table 4-4, we obtain the functions for the combinational circuit in sum of minterms:

$$S(x, y, z) = \Sigma(1, 2, 4, 7)$$

$$C(x, y, z) = \Sigma(3, 5, 6, 7)$$



Encoders

- An encoder is the inverse operation of a decoder.
- We can derive the Boolean functions by table 4-7

$$z = D_1 + D_3 + D_5 + D_7$$

$$y = D_2 + D_3 + D_6 + D_7$$

$$x = D_4 + D_5 + D_6 + D_7$$

Truth Table of Octal-to-Binary Encoder

Inputs								Outputs		
D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7	x	y	z
1	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	0	0	1	0	0	0	0	0	1	1
0	0	0	0	1	0	0	0	1	0	0
0	0	0	0	0	1	0	0	1	0	1
0	0	0	0	0	0	1	0	1	1	0
0	0	0	0	0	0	0	1	1	1	1

Priority encoder

- If two inputs are active simultaneously, the output produces an undefined combination. We can establish an input priority to ensure that only one input is encoded.
- Another ambiguity in the octal-to-binary encoder is that an output with all 0's is generated when all the inputs are 0; the output is the same as when D_0 is equal to 1.
- The discrepancy tables on Table 4-7 and Table 4-8 can resolve aforesaid condition by providing one more output to indicate that at least one input is equal to 1.

Priority encoder

$V=0 \rightarrow$ no valid inputs

$V=1 \rightarrow$ valid inputs

X's in output columns represent

don't-care conditions

X's in the input columns are useful for representing a truth

table in condensed form.

Instead of listing all 16 minterms of four variables.

Truth Table of a Priority Encoder

Inputs				Outputs		
D_0	D_1	D_2	D_3	x	y	V
0	0	0	0	X	X	0
1	0	0	0	0	0	1
X	1	0	0	0	1	1
X	X	1	0	1	0	1
X	X	X	1	1	1	1

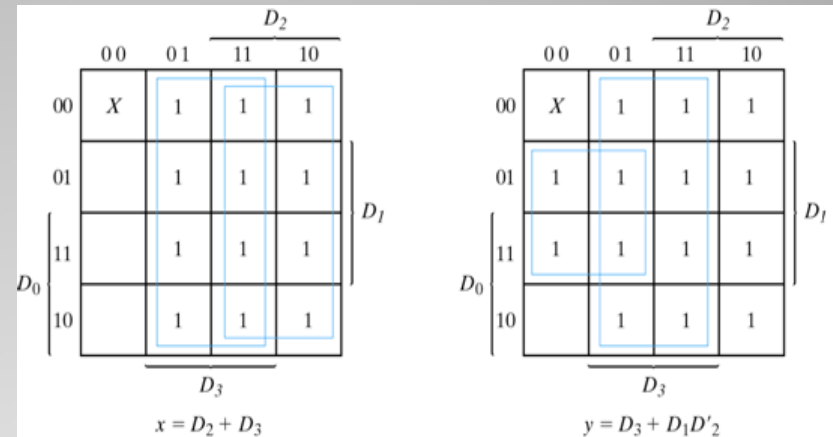
4-input priority encoder

- Implementation of table 4-8

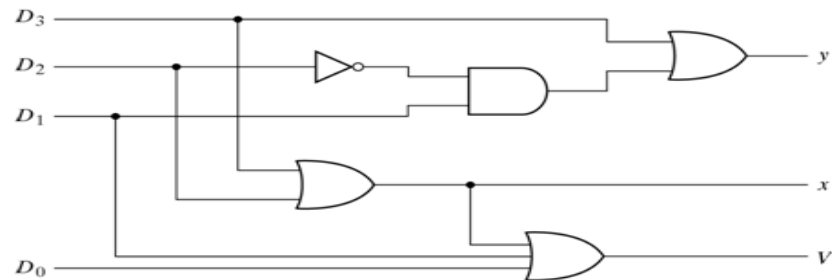
$$x = D_2 + D_3$$

$$y = D_3 + D_1 D'_2$$

$$V = D_0 + D_1 + D_2 + D_3$$



Maps for a Priority Encoder



4-Input Priority Encoder

Multiplexers

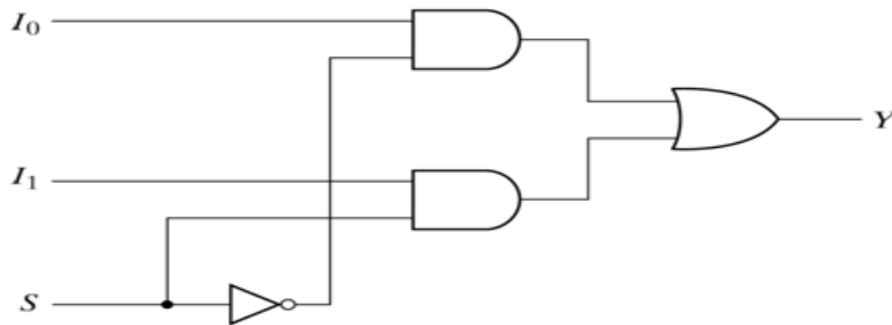
$S = 0, Y = I_0$

$S = 1, Y = I_1$

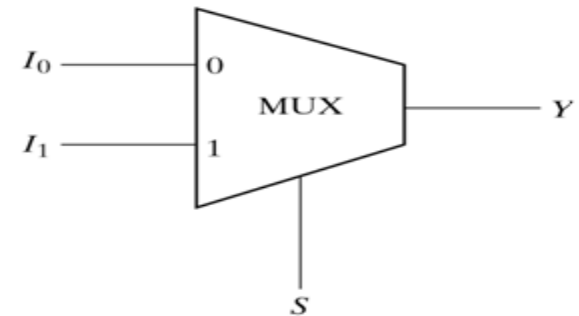
Truth Table $\rightarrow$

S	Y
0	I_0
1	I_1

$$Y = S'I_0 + SI_1$$



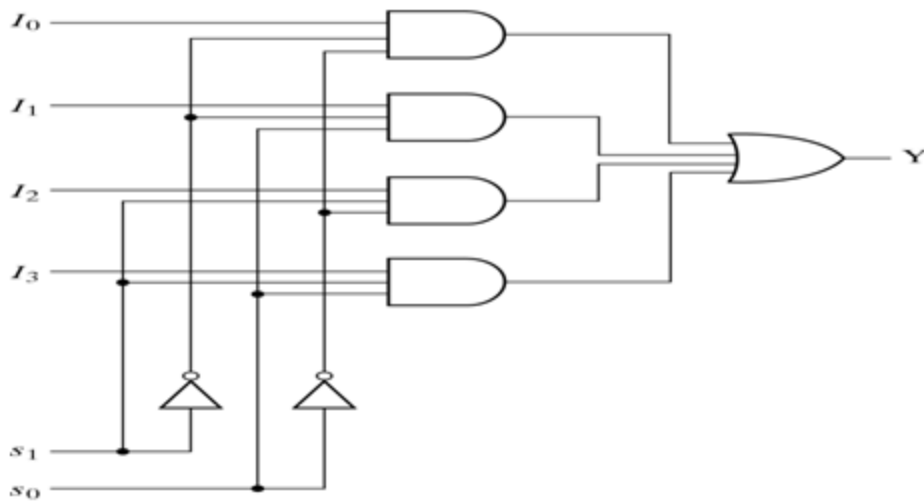
(a) Logic diagram



(b) Block diagram

2-to-1-Line Multiplexer

4-to-1 Line Multiplexer



(a) Logic diagram

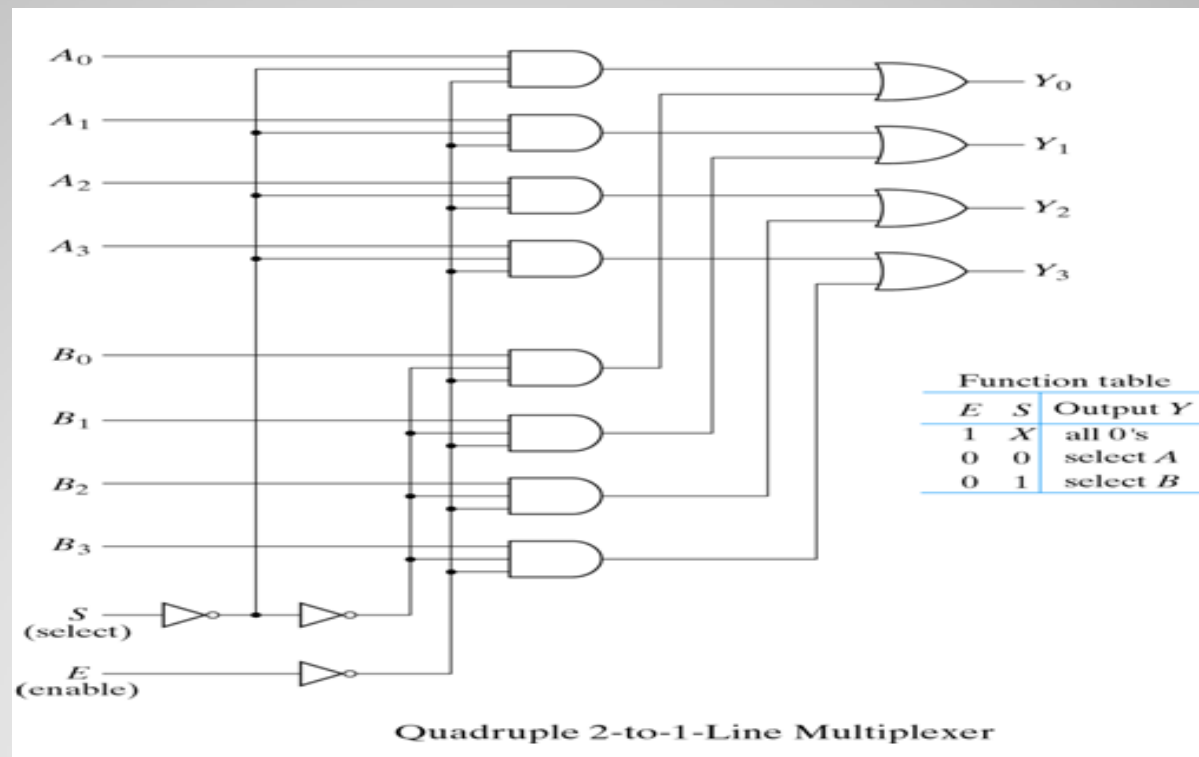
s_1	s_0	Y
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

(b) Function table

4-to-1-Line Multiplexer

Quadruple 2-to-1 Line Multiplexer

- Multiplexer circuits can be combined with common selection inputs to provide multiple-bit selection logic. Compare with Fig4-24.



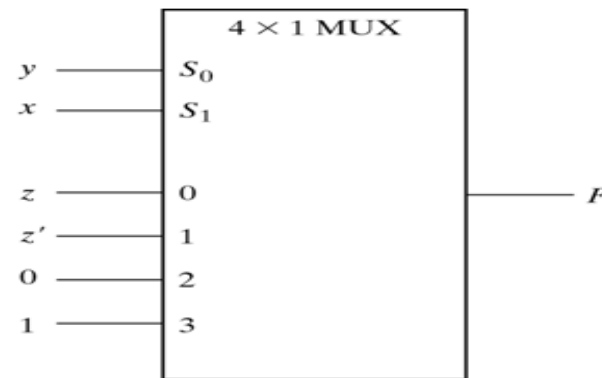
Boolean function implementation

- A more efficient method for implementing a Boolean function of n variables with a multiplexer that has $n-1$ selection inputs.

$$F(x, y, z) = \Sigma(1,2,6,7)$$

x	y	z	F	
0	0	0	0	
0	0	1	1	$F = z$
0	1	0	1	
0	1	1	0	$F = z'$
1	0	0	0	$F = 0$
1	0	1	0	
1	1	0	1	
1	1	1	1	$F = 1$

(a) Truth table



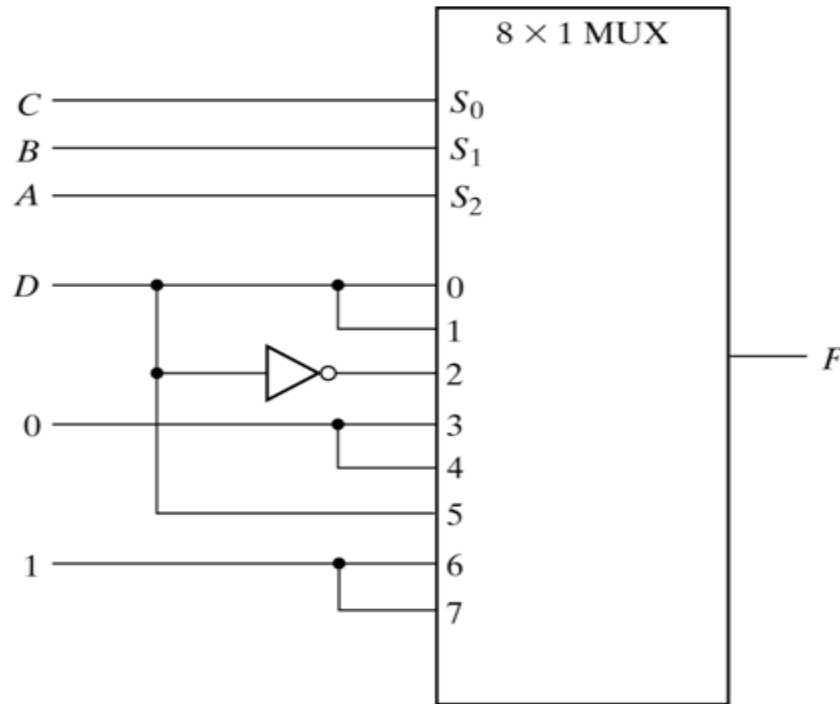
(b) Multiplexer implementation

Implementing a Boolean Function with a Multiplexer

4-input function with a multiplexer

$$F(A, B, C, D) = \Sigma(1, 3, 4, 11, 12, 13, 14, 15)$$

A	B	C	D	F	
0	0	0	0	0	$F = D$
0	0	0	1	1	
0	0	1	0	0	$F = D$
0	0	1	1	1	
0	1	0	0	1	$F = D'$
0	1	0	1	0	
0	1	1	0	0	$F = 0$
0	1	1	1	0	
1	0	0	0	0	$F = 0$
1	0	0	1	0	
1	0	1	0	0	$F = D$
1	0	1	1	1	
1	1	0	0	1	$F = 1$
1	1	0	1	1	
1	1	1	0	1	$F = 1$
1	1	1	1	1	



Implementing a 4-Input Function with a Multiplexer

Combinational Logic Design

- A process with 5 steps
 - Specification
 - Formulation
 - Optimization
 - Technology mapping
 - Verification
- 1st three steps and last best illustrated by example

Functional Blocks

- Fundamental circuits that are the base building blocks of most larger digital circuits
- They are reusable and are common to many systems.
- Examples of functional logic circuits
 - Decoders
 - Encoders
 - Code converters
 - Multiplexers

Where they are used

- Multiplexers
 - Selectors for routing data to the processor, memory, I/O
 - Multiplexers route the data to the correct bus or port.
- Decoders
 - are used for selecting things like a bank of memory and then the address within the bank. This is also the function needed to 'decode' the instruction to determine the operation to perform.
- Encoders
 - are used in various components such as keyboards.

BCD-to-Excess-3 Code converter

- BCD is a code for the decimal digits 0-9
- Excess-3 is also a code for the decimal digits

Decimal Digit	Input BCD	Output Excess-3
0	0 0 0 0	0 0 1 1
1	0 0 0 1	0 1 0 0
2	0 0 1 0	0 1 0 1
3	0 0 1 1	0 1 1 0
4	0 1 0 0	0 1 1 1
5	0 1 0 1	1 0 0 0
6	0 1 1 0	1 0 0 1
7	0 1 1 1	1 0 1 0
8	1 0 0 0	1 0 1 1
9	1 0 0 1	1 1 0 0

Specification of BCD-to-Excess3

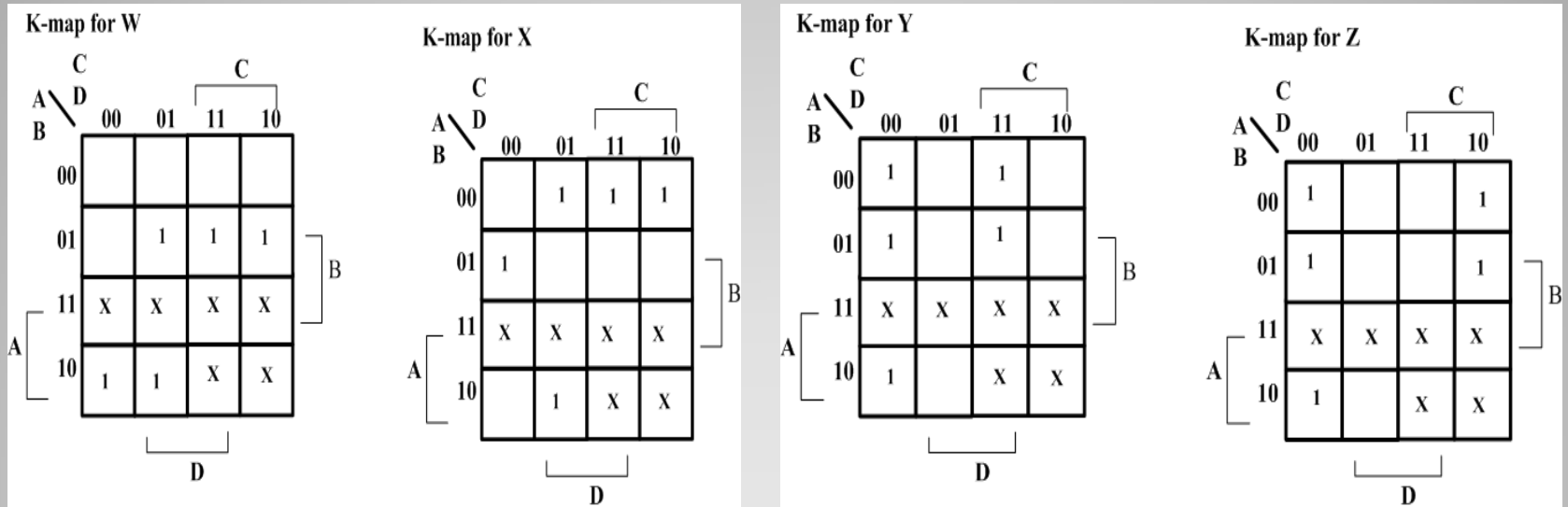
- Inputs: a BCD input, A,B,C,D with A as the most significant bit and D as the least significant bit.
- Outputs: an Excess-3 output W,X,Y,Z that corresponds to the BCD input.
- Internal operation – circuit to do the conversion in combinational logic.

Formulation of BCD-to-Excess-3

- Excess-3 code is easily formed by adding a binary 3 to the binary or BCD for the digit.
- There are 16 possible inputs for both BCD and Excess-3.
- It can be assumed that only valid BCD inputs will appear so the six combinations not used can be treated as don't cares.

Optimization – BCD-to-Excess-3

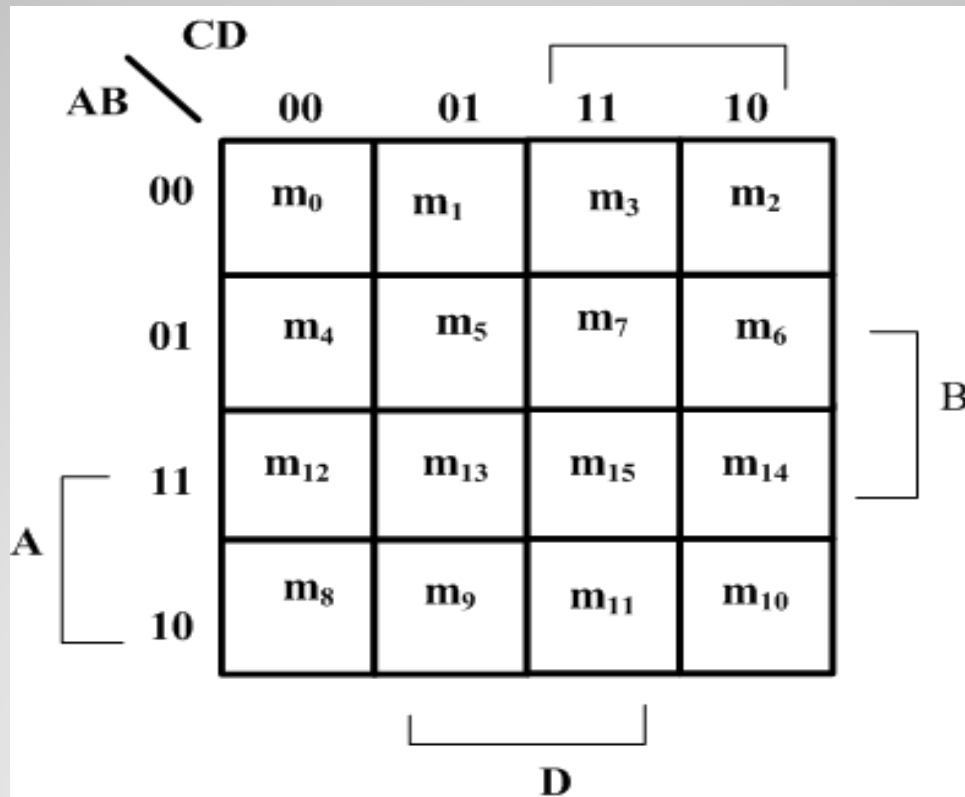
- Lay out K-maps for each output, W X Y Z



- A step in the digital circuit design process.

Placing 1 on K-maps

- Where are the Minterms located on a K-Map?



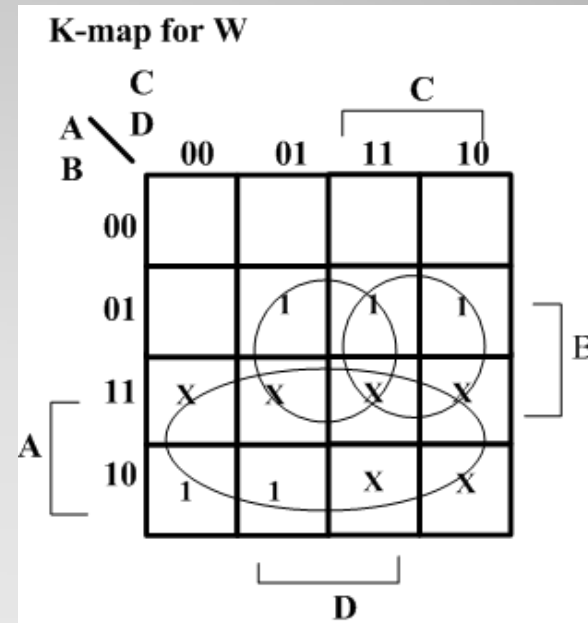
Expressions for W X Y Z

- $W(A,B,C,D) = \Sigma m(5,6,7,8,9)$
 $+d(10,11,12,13,14,15)$
- $X(A,B,C,D) = \Sigma m(1,2,3,4,9)$
 $+d(10,11,12,13,14,15)$
- $Y(A,B,C,D) = \Sigma m(0,3,4,7,8)$
 $+d(10,11,12,13,14,15)$
- $Z(A,B,C,D) = \Sigma m(0,2,4,6,8)$
 $+d(10,11,12,13,14,15)$

Minimize K-Maps

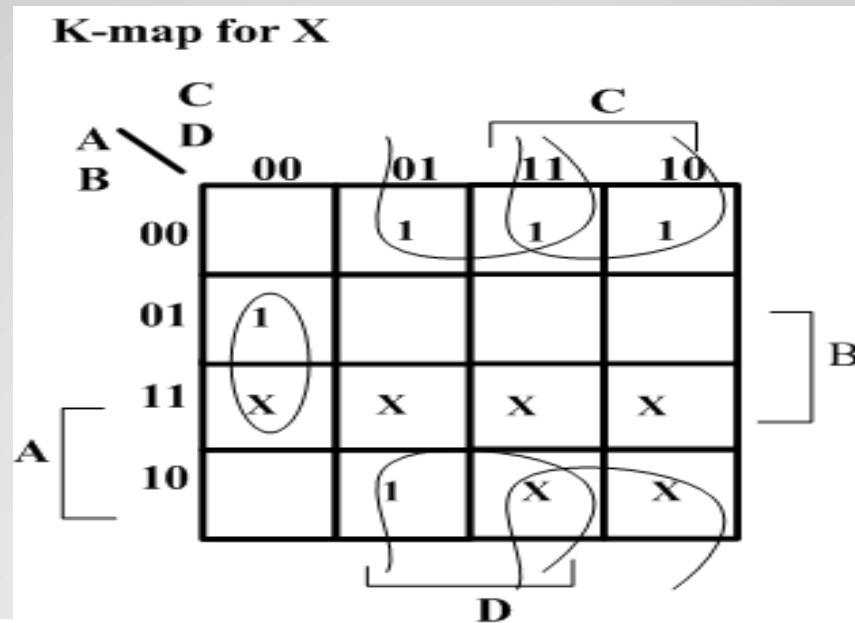
- W minimization

- Find $W = A + BC +$



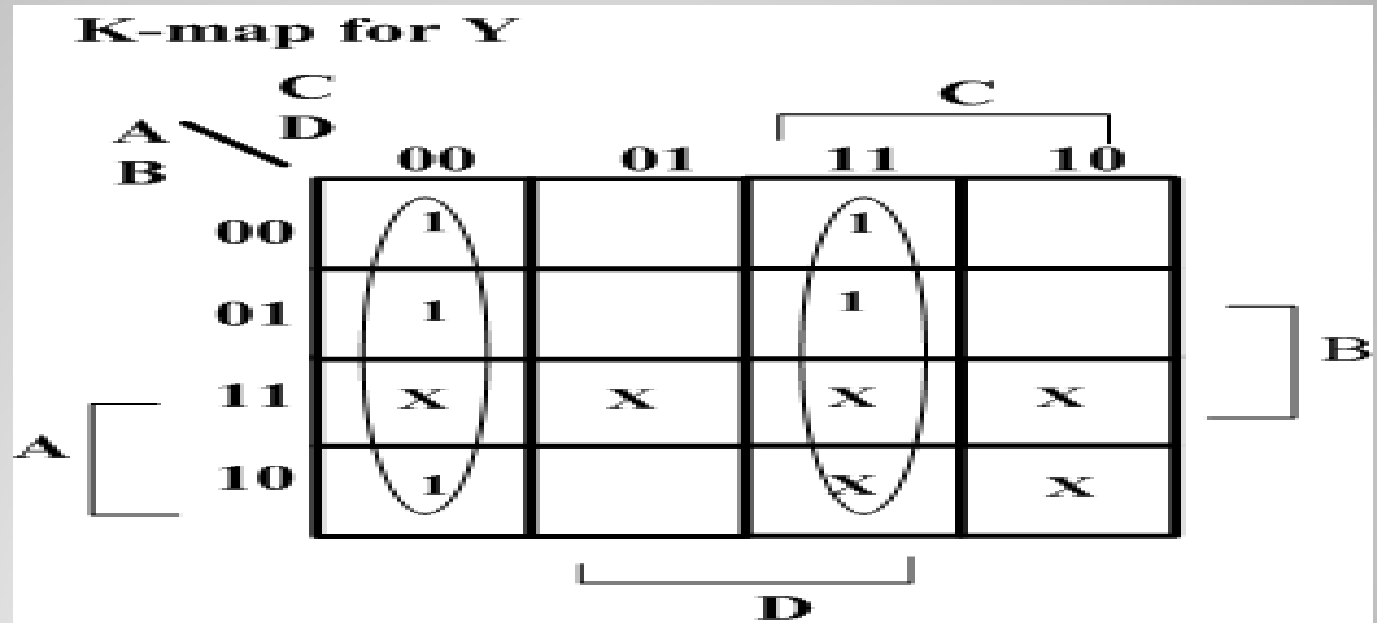
Minimize K-Maps

- X minimization
- Find $X = BC'D' + B'C + B'D$



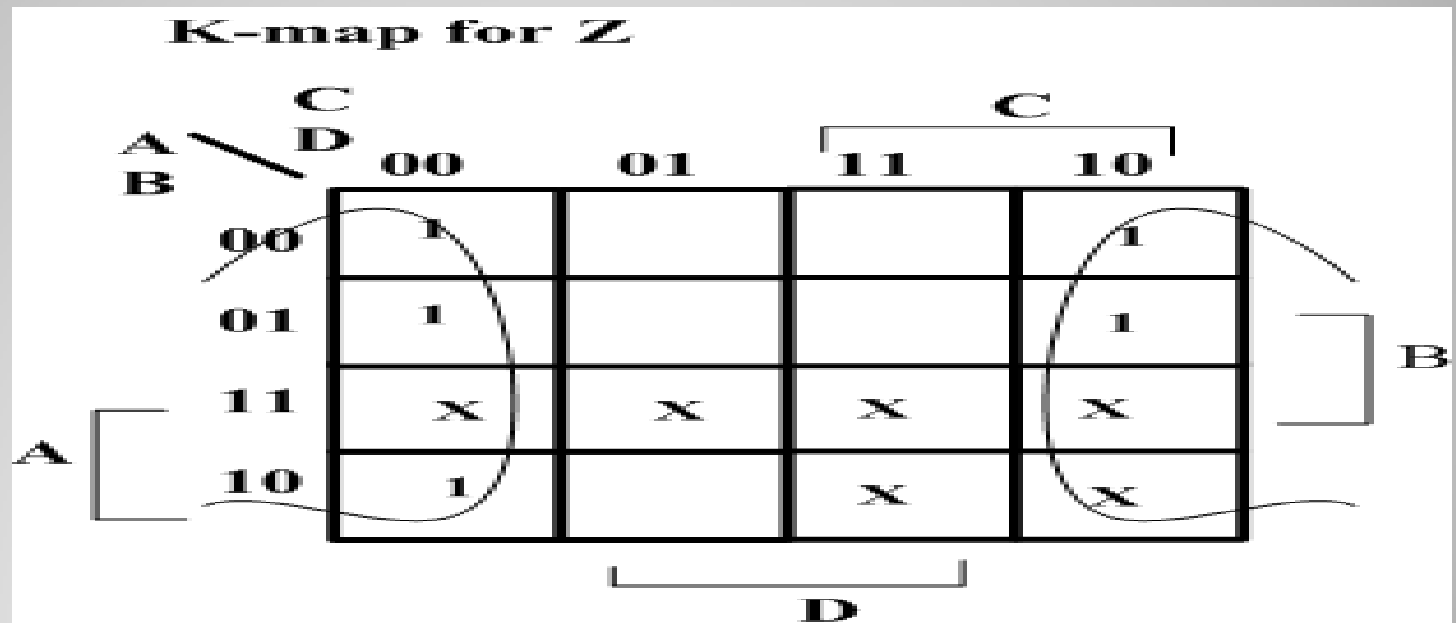
Minimize K-Maps

- Y minimization
- Find $Y = CD + C'D'$



Minimize K-Maps

- Z minimization
- Find $Z = D'$

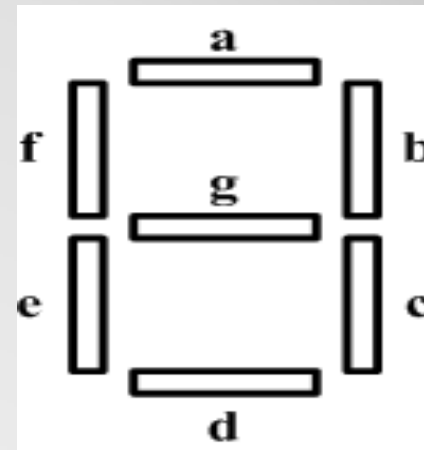


BCD-to-Seven-Segment Decoder

- Specification
 - Digital readouts on many digital products often use LED seven-segment displays.
 - Each digit is created by lighting the appropriate segments. The segments are labeled a,b,c,d,e,f,g
 - The decoder takes a BCD input and outputs the correct code for the seven-segment display.

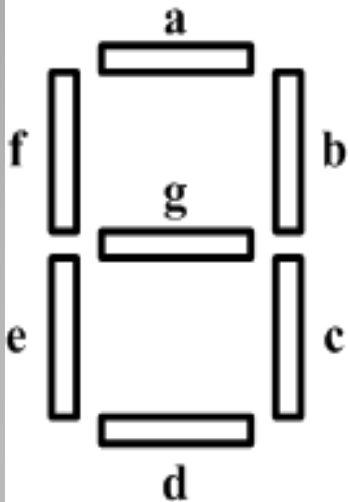
Specification

- Input: A 4-bit binary value that is a BCD coded input.
- Outputs: 7 bits, a through g for each of the segments of the display.
- Operation: Decode the input to activate the correct segments.



Formulation

- Construct a truth table



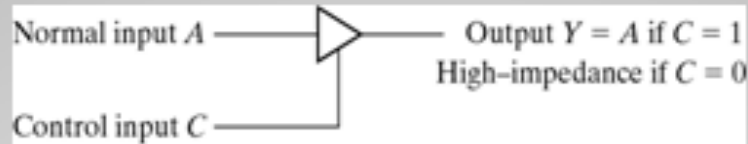
Decimal Digit	Input BCD	Seven-Segment Decoder Outputs						
		a	b	c	d	e	f	g
0	0 0 0 0	1	1	1	1	1	1	0
1	0 0 0 1	0	1	1	0	0	0	0
2	0 0 1 0	1	1	0	1	1	0	1
3	0 0 1 1	1	1	1	1	0	0	1
4	0 1 0 0	1	0	1	1	0	1	1
5	0 1 0 1	1	0	1	1	0	1	1
6	0 1 1 0	1	0	1	1	1	1	1
7	0 1 1 1	1	1	1	0	0	0	0
8	1 0 0 0	1	1	1	1	1	1	1
9	1 0 0 1	1	1	1	1	0	1	1
All other inputs		0	0	0	0	0	0	0

Optimization

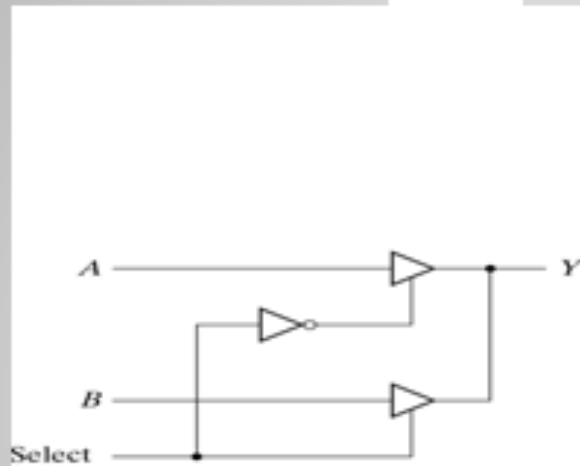
- Create a K-map for each output and get
 - $A = A'C + A'BD + B'C'D' + AB'C'$
 - $B = A'B' + A'C'D' + A'CD + AB'C'$
 - $C = A'B + A'D + B'C'D' + AB'C'$
 - $D = A'CD' + A'B'C + B'C'D' + AB'C' + A'BC'D$
 - $E = A'CD' + B'C'D'$
 - $F = A'BC' + A'C'D' + A'BD' + AB'C'$
 - $G = A'CD' + A'B'C + A'BC' + AB'C'$

Three-State Gates

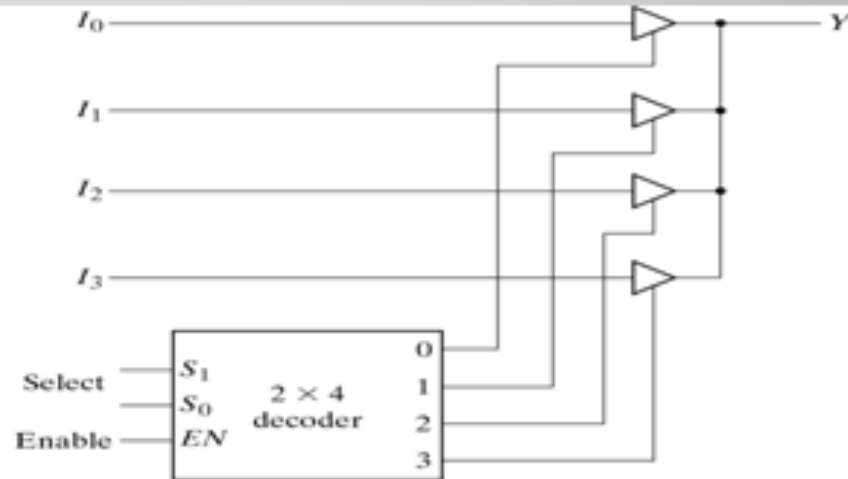
- A multiplexer can be constructed with three-state gates.



Graphic Symbol for a Three-State Buffer



(a) 2-to-1- line mux



(b) 4 - to - 1 line mux

Multiplexers with Three-State Gates

Three state gates

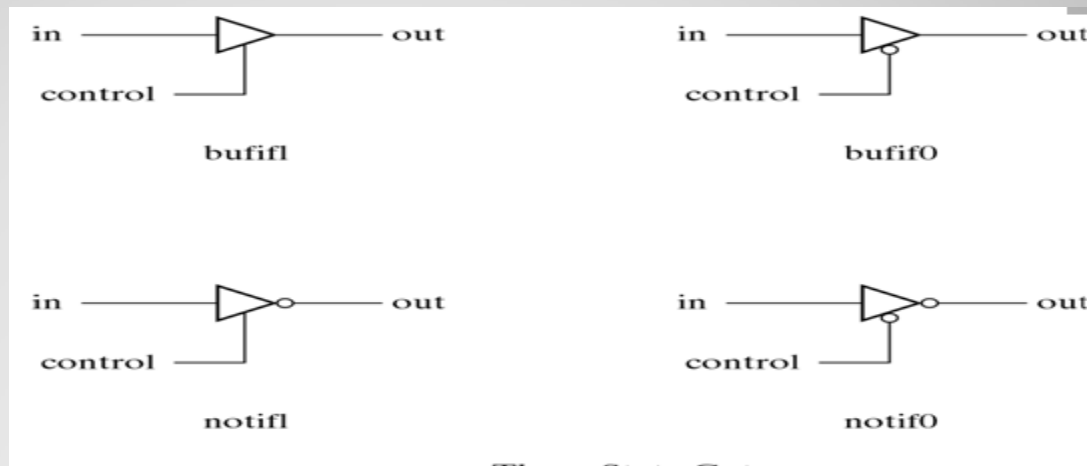
Gates statement: gate name(output, input, control)

>> `bufif1(OUT, A, control);`

A = OUT when control = 1, OUT = z when control = 0;

>> `notif0(Y, B, enable);`

Y = B' when enable = 0, Y = z when enable = 1;





Timing Hazards

Timing Hazards

- In a real logic circuit there is a delay in an input change to the corresponding output change.
- Because of circuit delays, the transient behavior of a logic circuit may differ from what is predicted by a steady-state analysis.
- In particular, a circuit's output may produce a short pulse, often called a glitch.
- A hazard is said to exist when a circuit has the possibility of producing such a glitch.

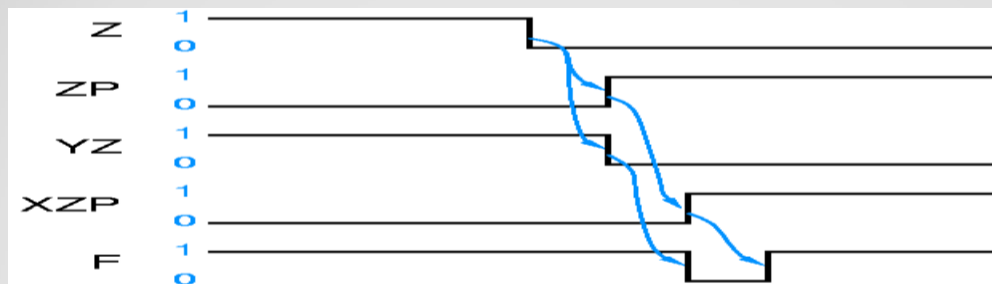
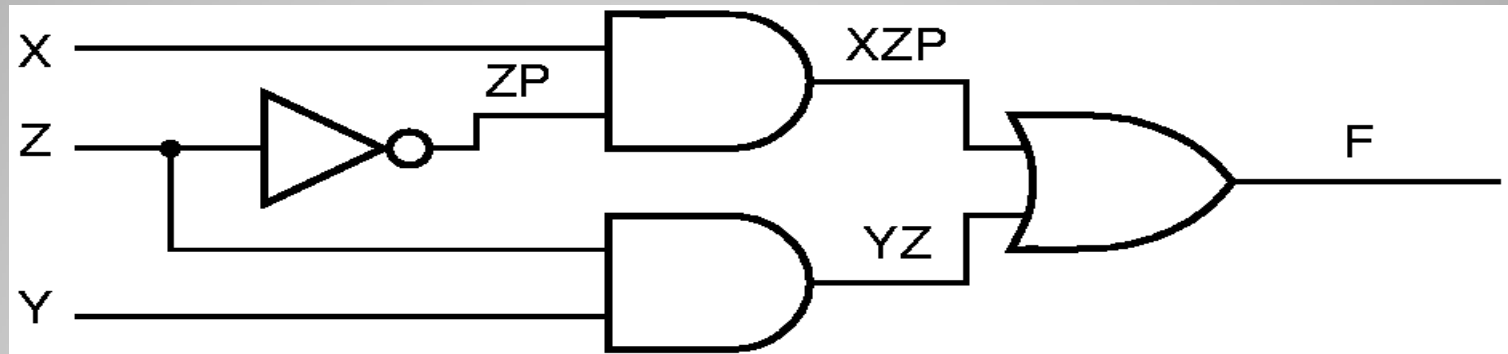
Timing Hazards

- Whether or not the glitch actually occurs depends on the exact delays and other electrical characteristics of the circuit.
- Since such parameters are difficult to control in production circuits, a logic designer must be prepared to eliminate hazards.
- Two kinds of hazards:
 - Static, and
 - Dynamic

Static -1 Hazard

- A **static -1** hazard is the possibility of a circuit's output producing a 0 glitch when we would expect the output to remain at a nice steady 1 based on a static analysis.
- A static -1 hazard is a pair of input combinations that:
 - a) differ in only one input variable and
 - b) both give a 1 output;such that it is possible for a momentary 0 output to occur during a transition in the differing input variable.

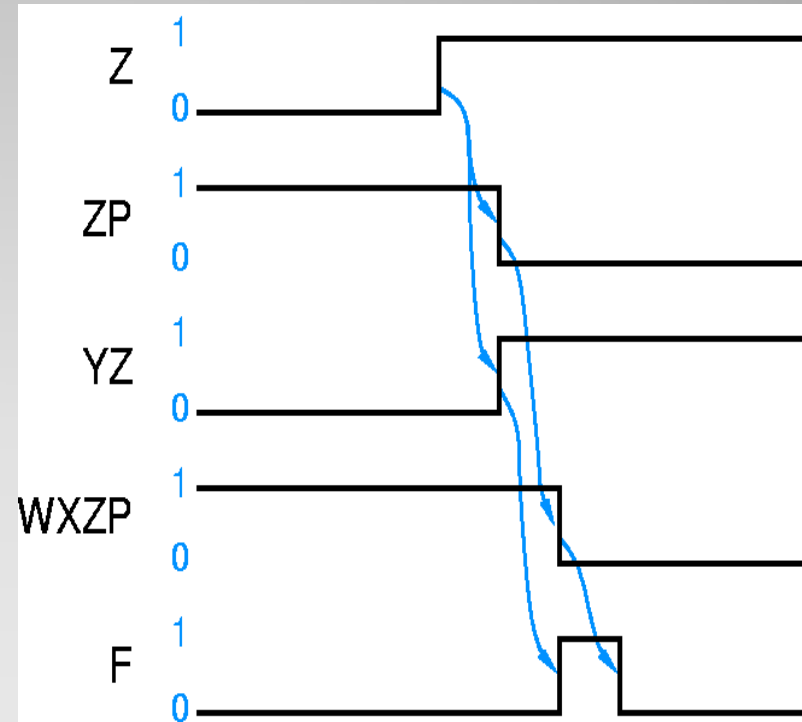
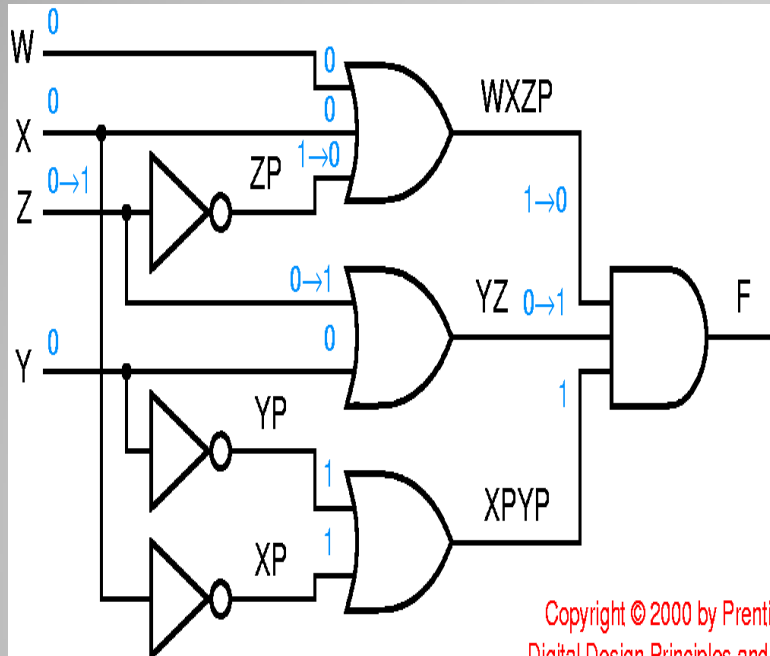
A static -1 Hazard



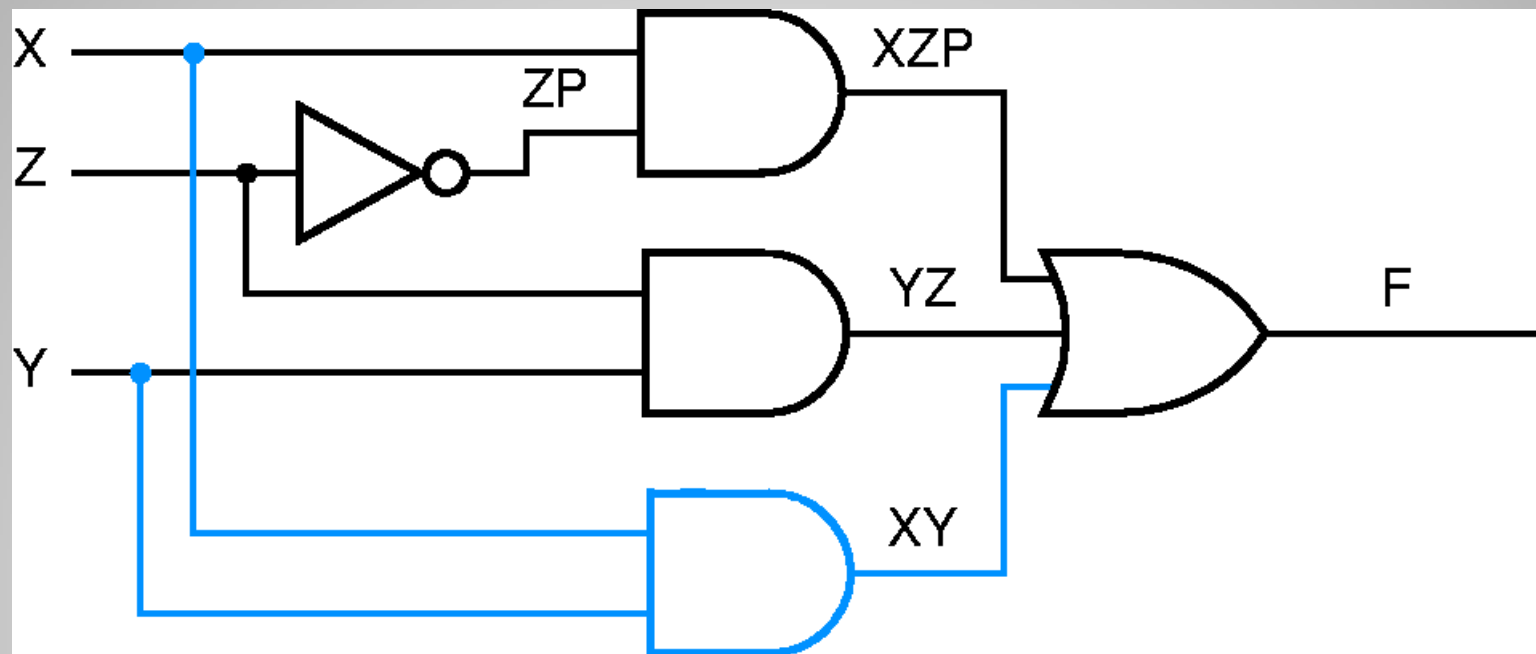
Static -0 Hazard

- A **static -0** hazard is the possibility of a circuit's output producing a 1 glitch when we would expect the output to remain at a nice steady 0 based on a static analysis.
- A static -0 hazard is a pair of input combinations that:
 - a) differ in only one input variable and
 - b) both give a 0 output;such that it is possible for a momentary 1 output to occur during a transition in the differing input variable.

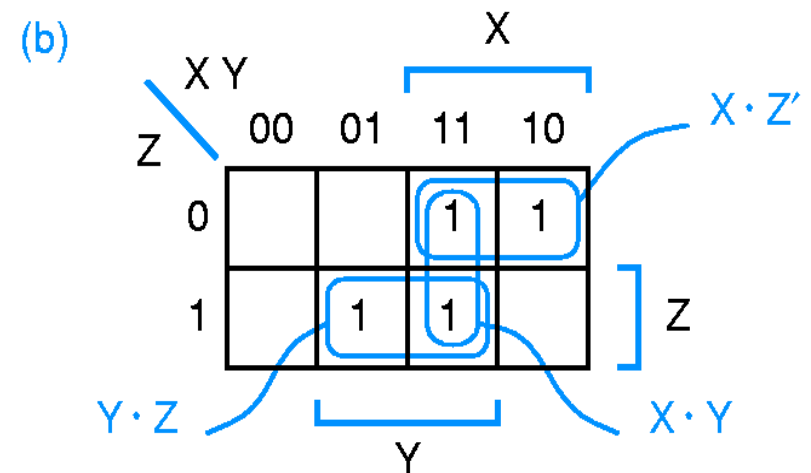
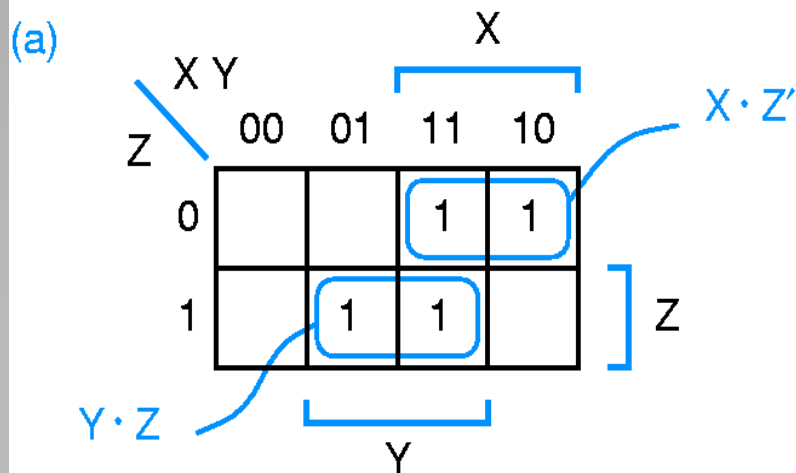
A static - 0 hazard



A static - 1 hazard elimination



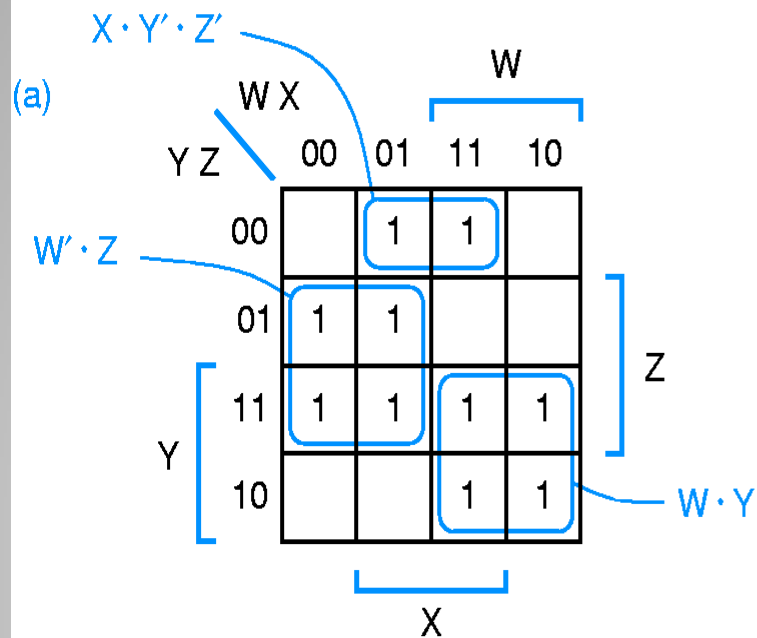
A static - 1 hazard elimination using Karnaugh map



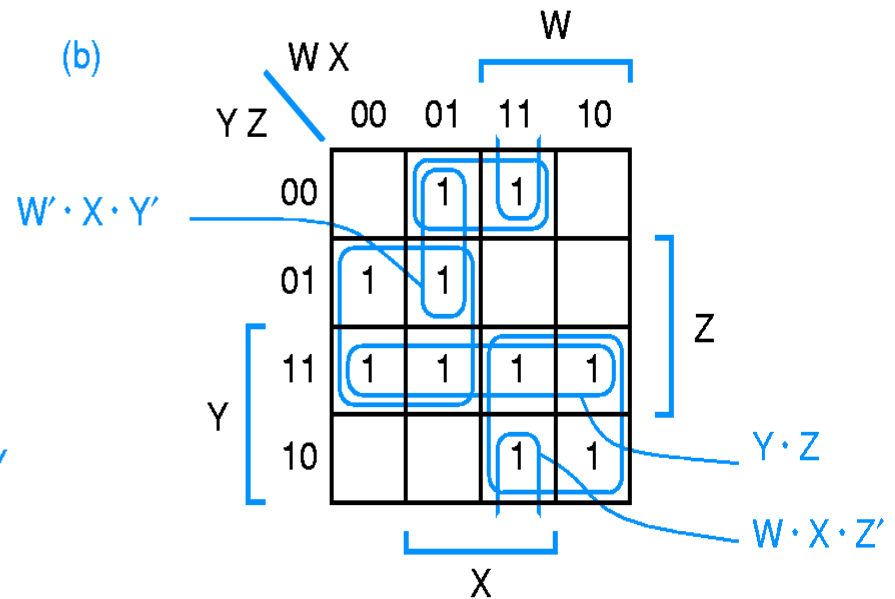
Elimination of Static Hazards

- The extra product term is the consensus of the two original terms.
- In general, we must add consensus terms to eliminate hazards.

A static - 1 hazard elimination using Karnaugh map

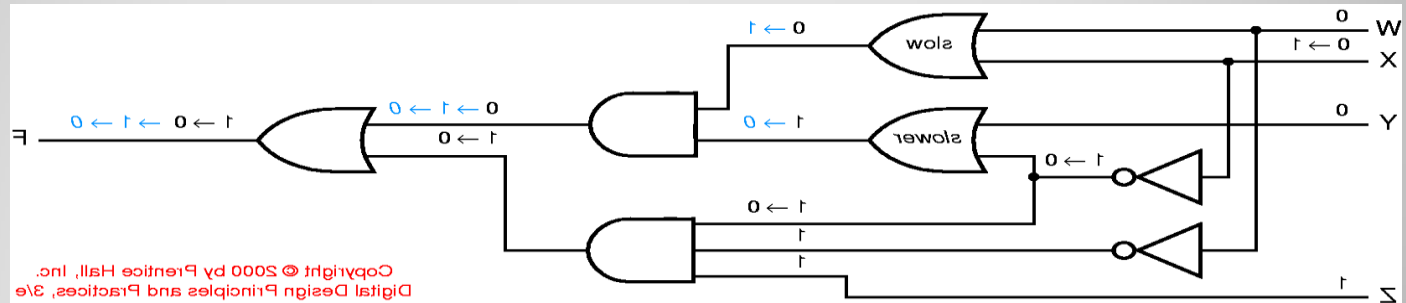


$$F = X \cdot Y' \cdot Z' + W' \cdot Z + W \cdot Y$$



$$F = X \cdot Y' \cdot Z' + W' \cdot Z + W \cdot Y + W' \cdot X \cdot Y' + Y \cdot Z + W \cdot X \cdot Z'$$

- A dynamic hazard is the possibility of an output changing more than once as the result of a single input transition.
- Multiple output transitions can occur if there are multiple paths with different delays from the changing input to the changing output.



Designing hazard-free circuits

- Only a few situations, such as the design of feedback sequential circuits, require hazard-free combinational circuits.
- If cost is not a problem, then a to obtain a hazard-free realization is to use the **complete sum** - the sum of **all prime implicants**.
- In a synchronous system, all of the inputs to a combinational circuit are changed at a particular time, and the outputs are not "looked at" until they have had time to settle to a steady-state value.

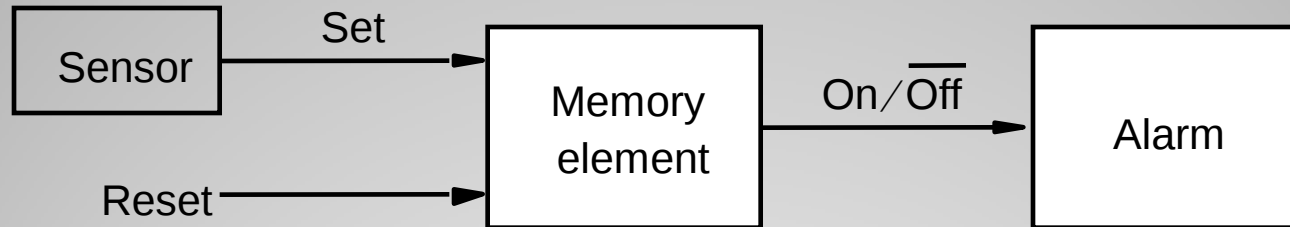
UNIT 3

Sequential machines fundamentals

Objectives

- In this chapter you will learn about:
 - Logic circuits that can store information
 - Flip-flops, which store a single bit
 - Registers, which store multiple bits
 - Shift registers, which shift the contents of a register
 - Counters of various types

Motivation: Control of an Alarm System

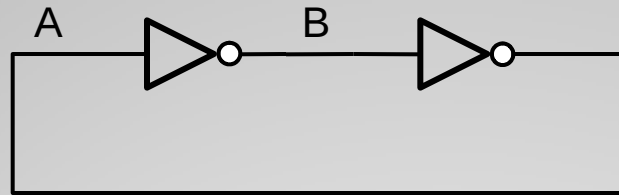


- Alarm turned on when On/Off = 1
- Alarm turned off when On/Off = 0
- Once triggered, alarm stays on until manually reset
- The circuit requires a memory element

The Basic Latch

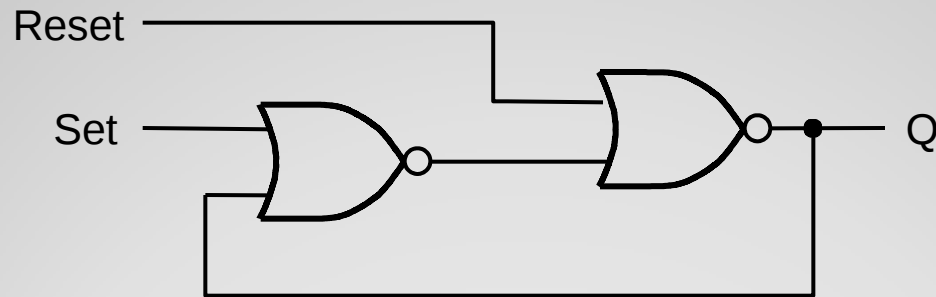
- **Basic latch** is a feedback connection of two NOR gates or two NAND gates
- It can store one bit of information
- It can be set to 1 using the *S* input and reset to 0 using the *R* input.

A Simple Memory Element



- A feedback loop with even number of inverters
- If $A = 0$, $B = 1$ or when $A = 1$, $B = 0$
- This circuit is not useful due to the lack of a mechanism for changing its state

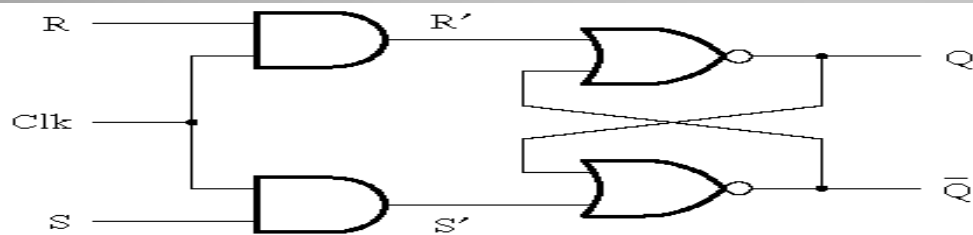
A Memory Element with NOR Gates



The Gated Latch

- **Gated latch** is a basic latch that includes input gating and a control signal
- The latch retains its existing state when the control input is equal to 0
- Its state may be changed when the control signal is equal to 1. In our discussion we referred to the control input as the clock
- We consider two types of gated latches:
 - **Gated SR latch** uses the S and R inputs to set the latch to 1 or reset it to 0, respectively.
 - **Gated D latch** uses the D input to force the latch into a state that has the same logic value as the D input.

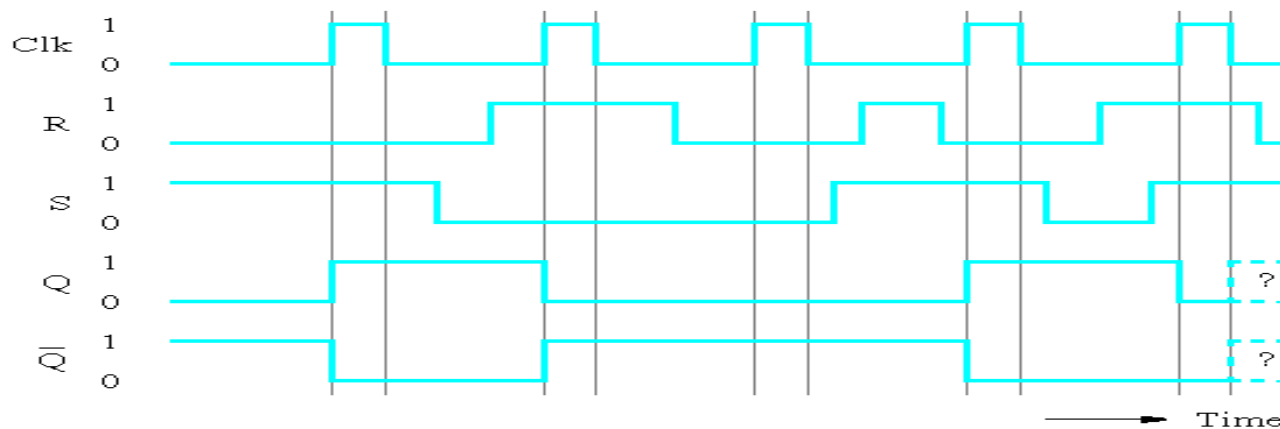
Gated S/R Latch



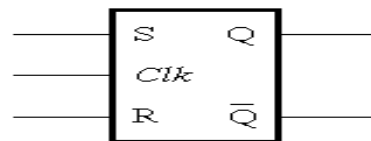
(a) Circuit

Clk	S	R	$Q(t+1)$
0	x	x	$Q(t)$ (no change)
1	0	0	$Q(t)$ (no change)
1	0	1	0
1	1	0	1
1	1	1	x

(b) Characteristic table

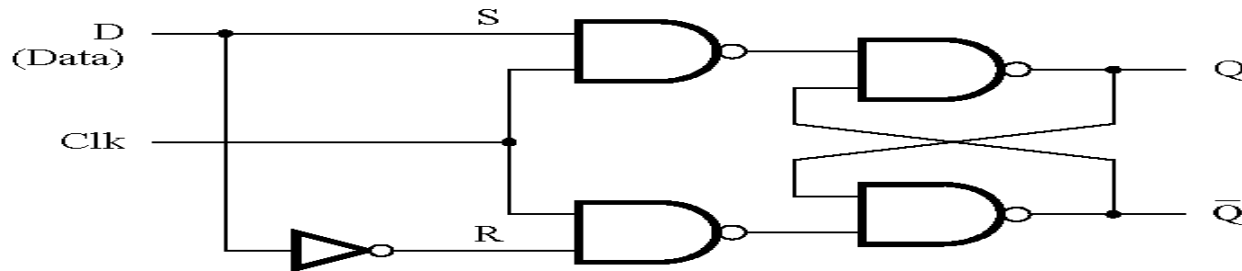


(c) Timing diagram



(d) Graphical symbol

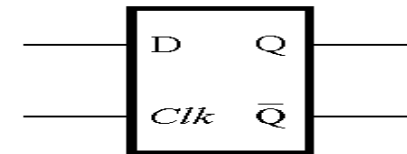
Gated D Latch



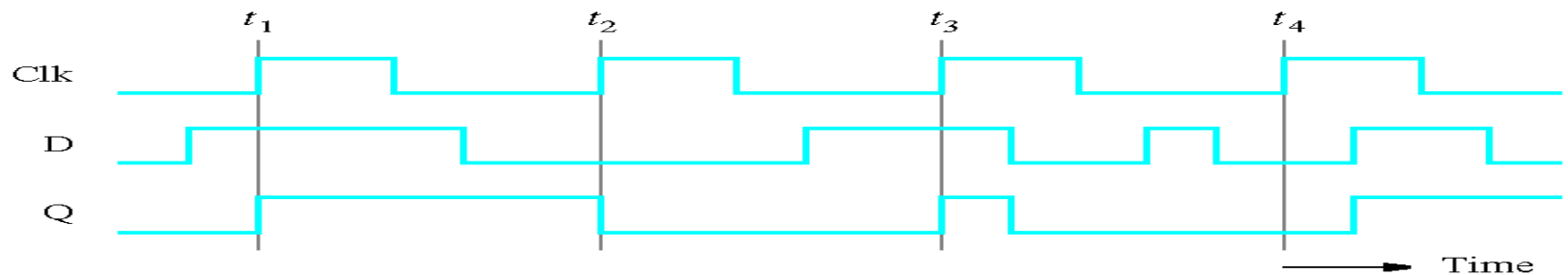
(a) Circuit

Clk	D	$Q(t+1)$
0	x	$Q(t)$
1	0	0
1	1	1

(b) Characteristic table



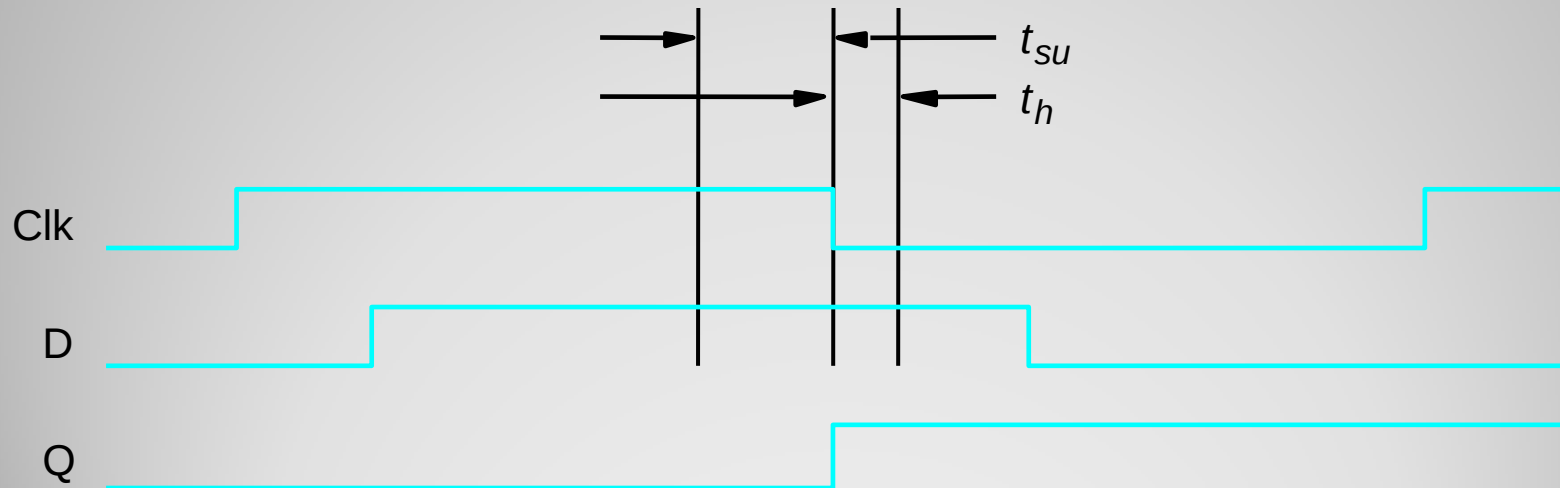
(c) Graphical symbol



(d) Timing diagram

Setup and Hold Times

- Setup Time t_{su}
 - The minimum time that the input signal must be stable prior to the edge of the clock signal.
- Hold Time t_h
 - The minimum time that the input signal must be stable after the edge of the clock signal.



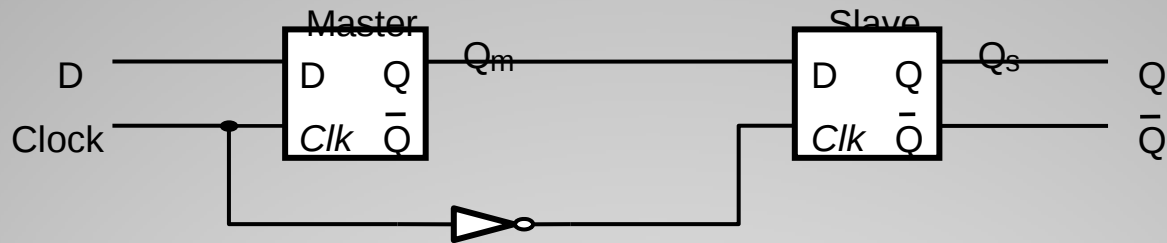
Flip-Flops

- A **flip-flop** is a storage element based on the gated latch principle
- It can have its output state changed only on the edge of the controlling clock signal

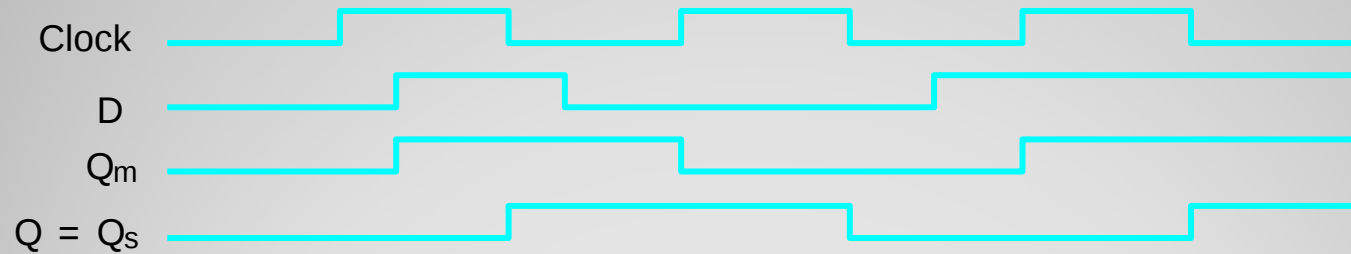
Flip-Flops

- We consider two types:
 - **Edge-triggered flip-flop** is affected only by the input values present when the active edge of the clock occurs
 - **Master-slave flip-flop** is built with two gated latches
 - The master stage is active during half of the clock cycle, and the slave stage is active during the other half.
 - The output value of the flip-flop changes on the edge of the clock that activates the transfer into the slave stage.

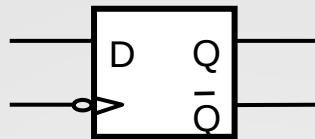
Master-Slave D Flip-Flop



(a) Circuit

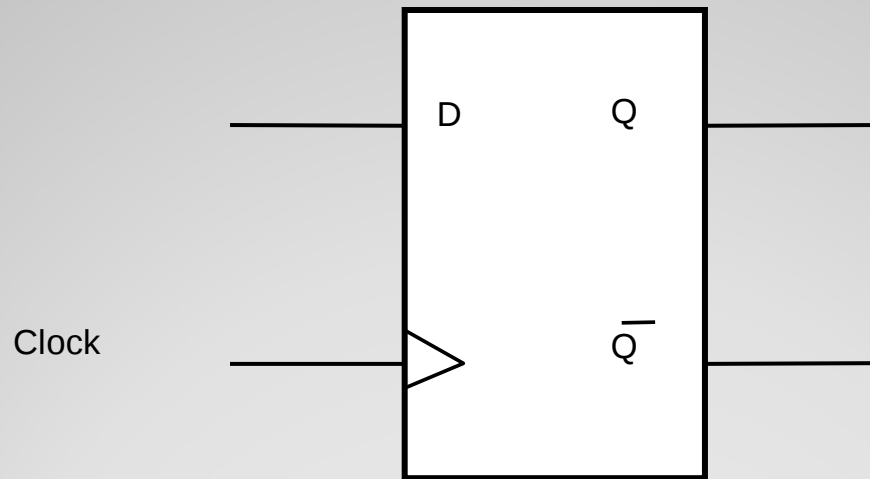


(b) Timing diagram



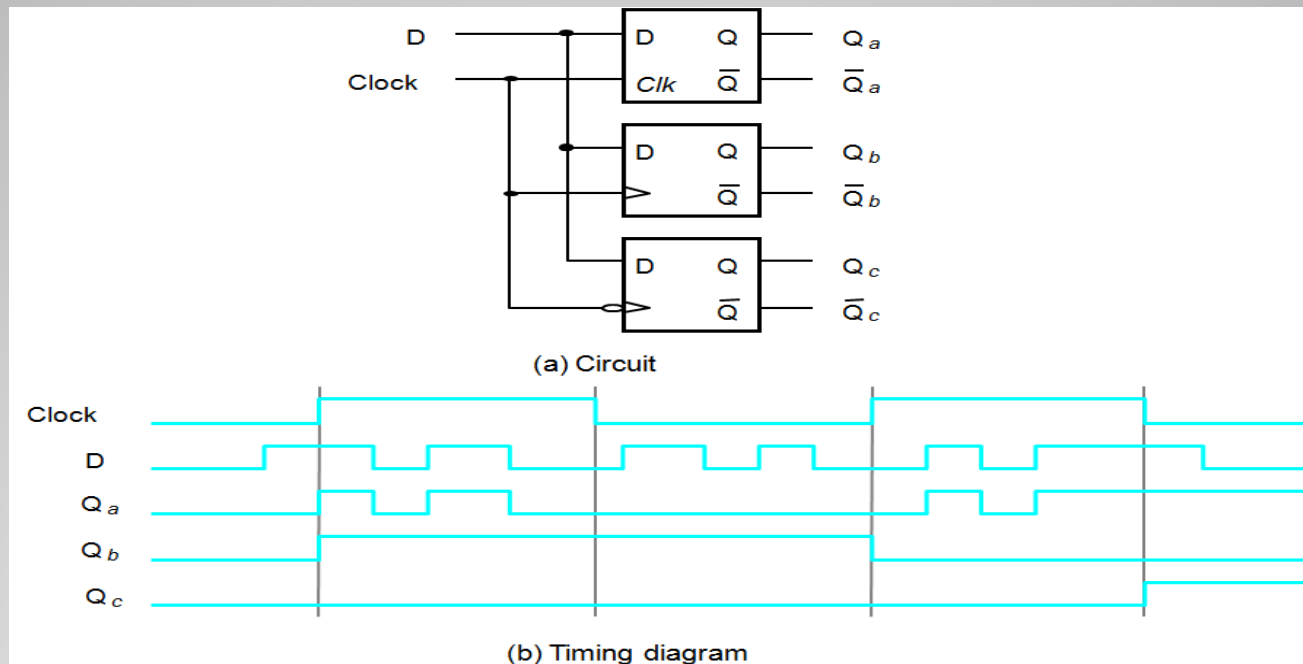
(c) Graphical symbol

A Positive-Edge-Triggered D Flip-Flop



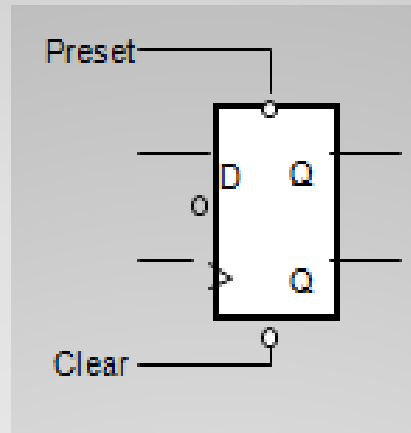
Graphical symbol

Comparison of Level-Sensitive and Edge-Triggered D Storage Elements

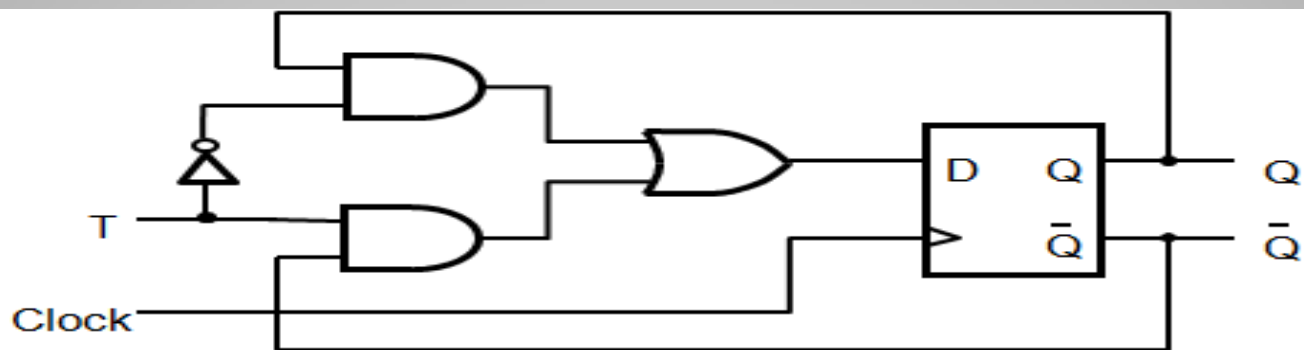


Comparison of Level-Sensitive and
Edge-Triggered D Storage Elements

Master-Slave D Flip-Flop with *Clear and Preset*



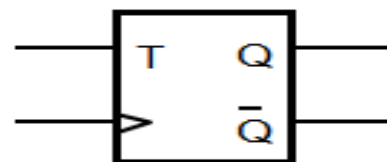
T Flip-Flop



(a) Circuit

T	$Q(t+1)$
0	$Q(t)$
1	$\bar{Q}(t)$

(b) Characteristic table

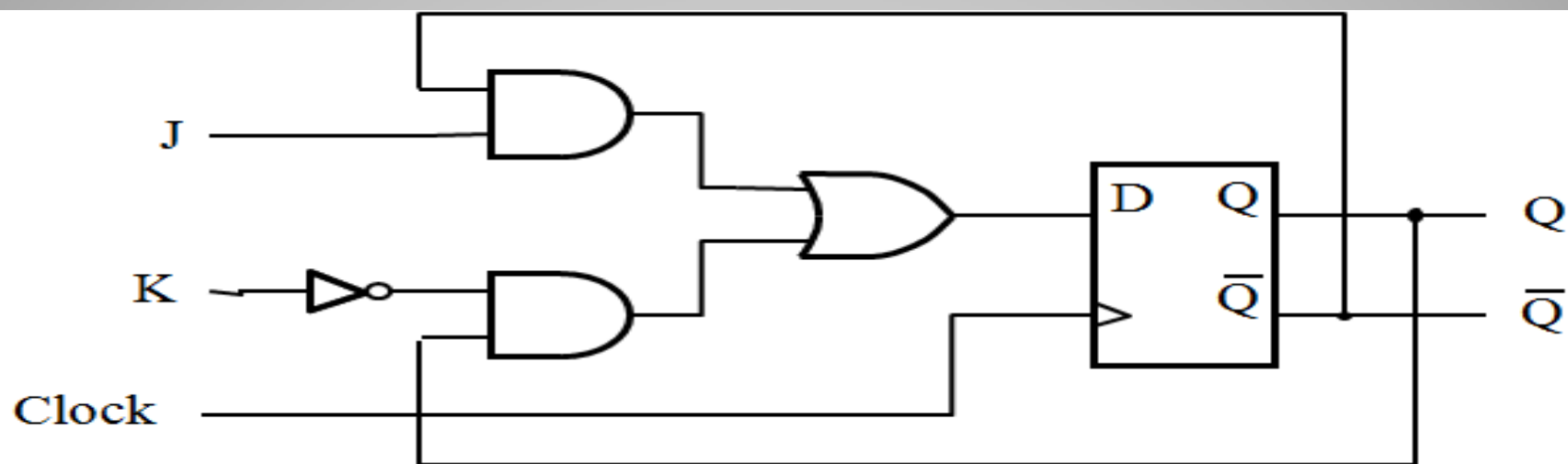


(c) Graphical symbol



(d) Timing diagram

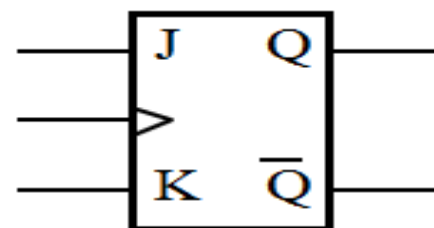
JK Flip-Flop



(a) Circuit

J	K	$Q(t+1)$
0	0	$Q(t)$
0	1	0
1	0	1
1	1	$\bar{Q}(t)$

(b) Characteristic table

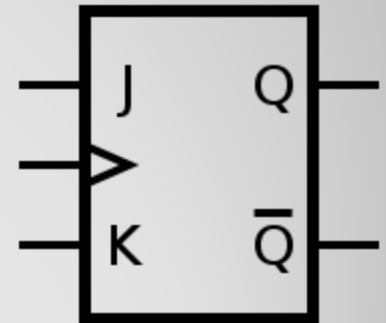
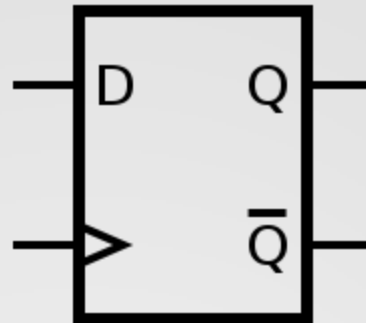
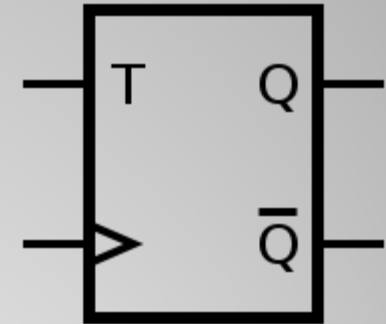
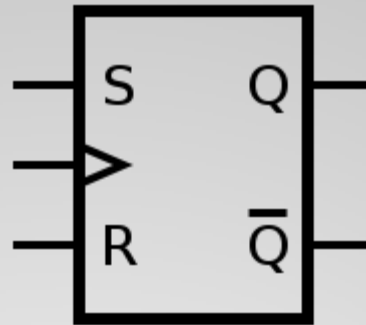


(c) Graphical symbol

Flip-flop excitation tables

Types of Flip-flops

- SR flip-flop (Set, Reset)
- T flip-flop (Toggle)
- D flip-flop (Delay)
- JK flip-flop



Excitation Tables

Previous State -> Present State	S	R
0 -> 0	0	X
0 -> 1	1	0
1 -> 0	0	1
1 -> 1	X	0

Previous State -> Present State	T
0 -> 0	0
0 -> 1	1
1 -> 0	1
1 -> 1	0

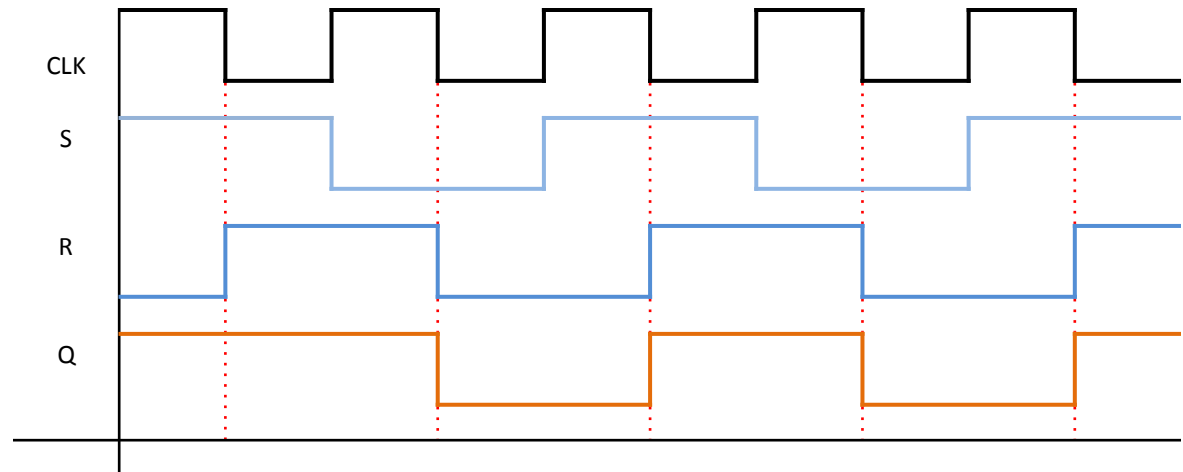
Excitation Tables

Previous State -> Present State	D
0 -> 0	0
0 -> 1	1
1 -> 0	0
1 -> 1	1

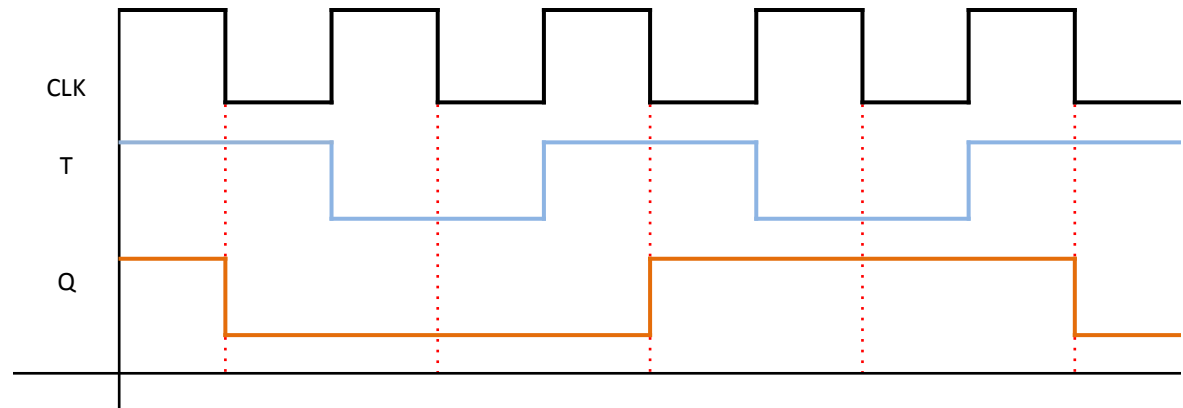
Previous State -> Present State	J	K
0 -> 0	0	X
0 -> 1	1	X
1 -> 0	X	1
1 -> 1	X	0

Timing Diagrams

	S	R
0->0	0	X
0->1	1	0
1->0	0	1
1->1	X	0

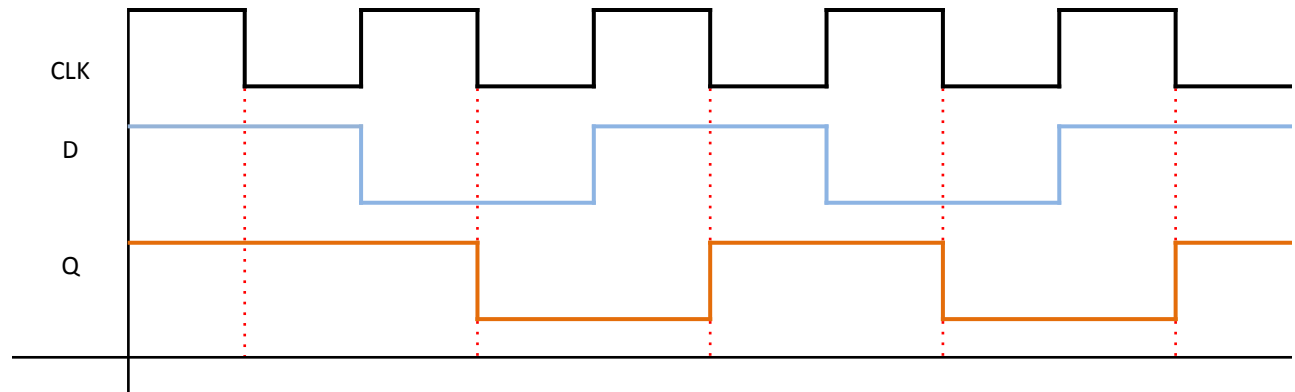


	T
0->0	0
0->1	1
1->0	1
1->1	0

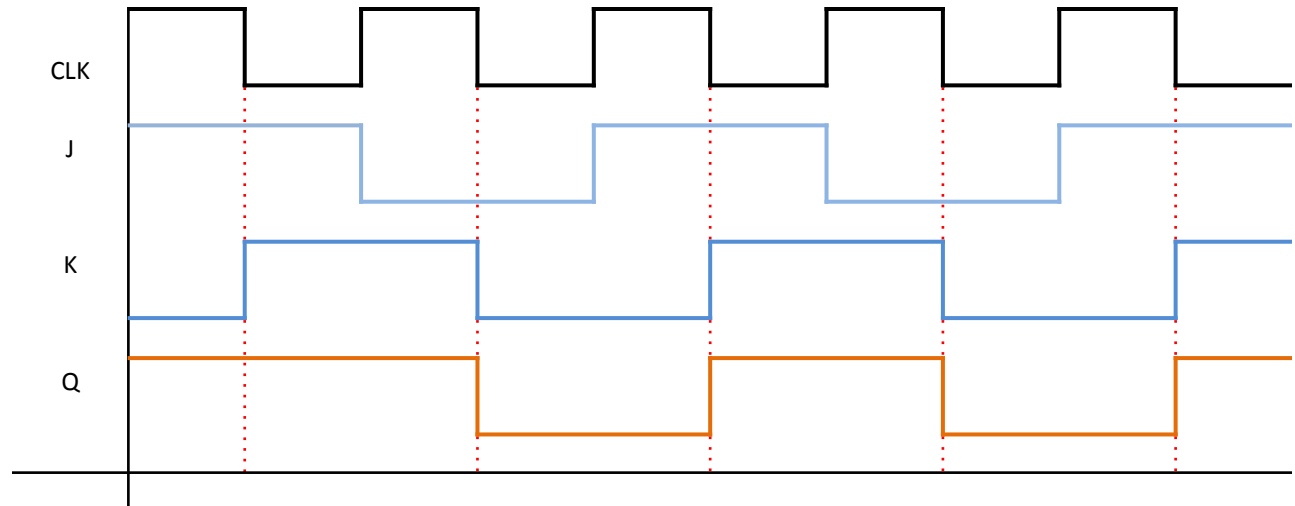


Timing Diagrams

	D
0->0	0
0->1	1
1->0	0
1->1	1



	J	K
0->0	0	X
0->1	1	X
1->0	X	1
1->1	X	0

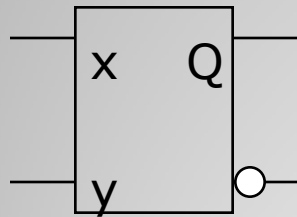


Conversions of flipflops

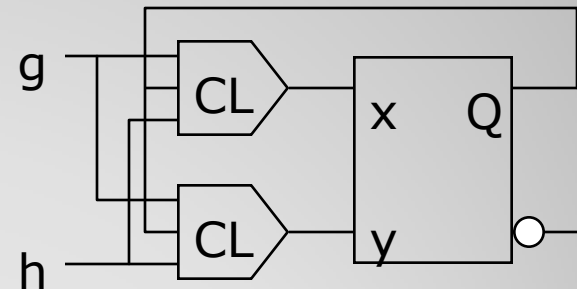
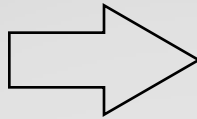
Procedure uses excitation tables

Method: to realize a type **A** flipflop using a type **B** flipflop:

1. Start with the K-map or state-table for the A-flipflop.
2. Express B-flipflop inputs as a function of the inputs and present state of A-flipflop such that the required state transitions of A-flipflop are realized.



Type B



Type A

1. Find $Q^+ = f(g,h,Q)$ for type A (using type A state-table)
2. Compute $x = f_1(g,h,Q)$ and $y = f_2(g,h,Q)$ to realize Q^+ .

Example: Use JK-FF to realize D-FF

- 1) Start transition table for D-FF
- 2) Create K-maps to express J and K as functions of inputs (D, Q)
- 3) Fill in K-maps with appropriate values for J and K to cause the same state transition as in the D-FF transition table

D	Q	Q ⁺	J	K
0	0	0	0	X
0	1	0	X	1
1	0	1	1	X
1	1	1	X	0

State-Table

e.g.
 when $D=Q=0$, then $Q^+=0$
 the same transition $Q \rightarrow Q^+$
 is realized with $J=0, K=X$

Q	Q ⁺	R	S	J	K	T	D
0	0	X	0	0	X	0	0
0	1	0	1	1	X	1	1
1	0	1	0	X	1	1	0
1	1	0	X	X	0	0	1

		D	
		0	1
Q	0	0	1
	1	X	X

$$J = D$$

		D	
		0	1
Q	0	X	X
1	1	1	0

$$K = \overline{D}$$

Example: Implement JK-FF using a D-FF

J	K	Q	Q+	D	T
0	0	0	0	0	0
0	0	1	1	1	0
0	1	0	0	0	0
0	1	1	0	0	1
1	0	0	1	1	1
1	0	1	1	1	0
1	1	0	1	1	1
1	1	1	0	0	1

JK \ Q	J			
	00	01	11	10
0	0	0	1	1
1	1	0	0	1

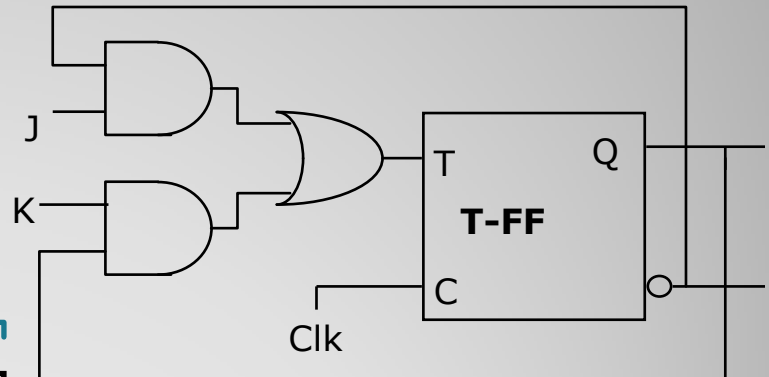
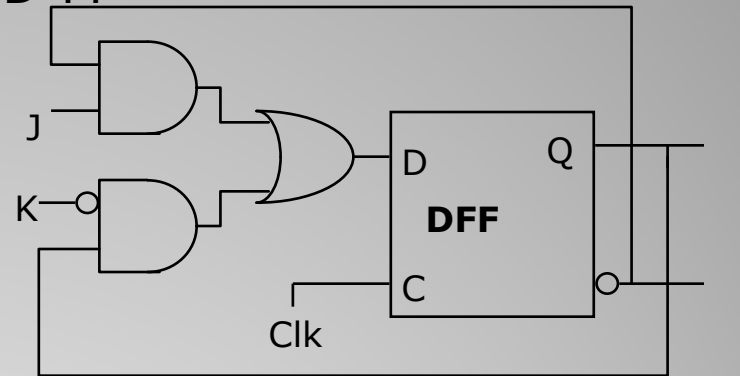
K

$$d = jQ + Kq$$

JK \ Q	J			
	00	01	11	10
0	0	0	1	1
1	0	1	1	0

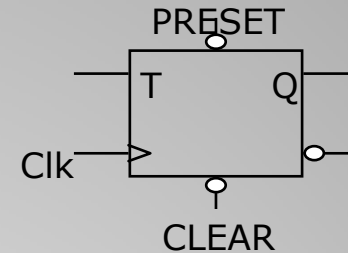
K

$$t = jQ + Kq$$



Asynchronous inputs

PRESET and CLEAR:
asynchronous, level-sensitive inputs
used to initialize a flipflop.

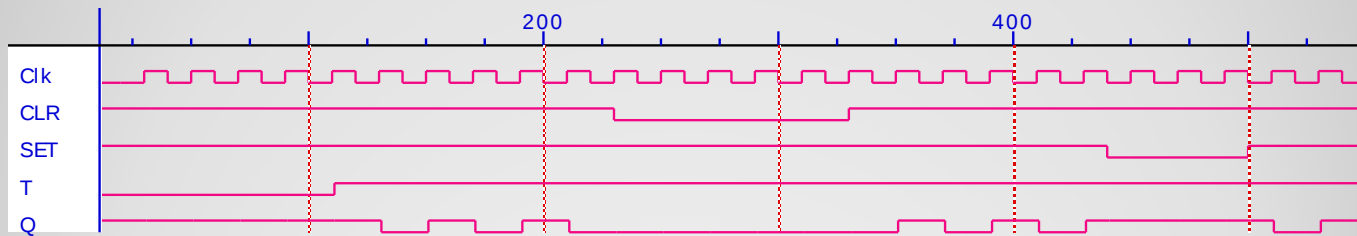
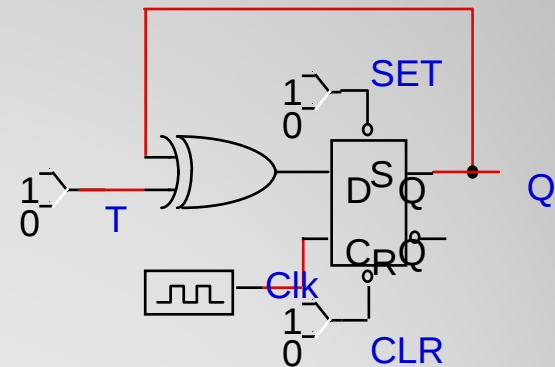


PRESET, CLEAR: active low inputs

$\text{PRESET} = 0 \rightarrow Q = 1$

$\text{CLEAR} = 0 \rightarrow Q = 0$

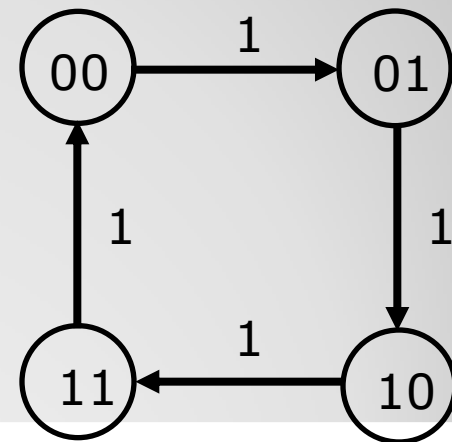
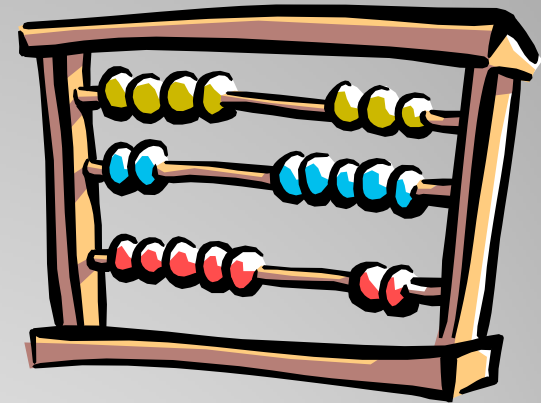
LogicWorks Simulation



Counters

- Counters are a specific type of sequential circuit.
- Like registers, the state, or the flip-flop values themselves, serves as the "output."
- The output value increases by one on each clock cycle.
- After the largest value, the output "wraps around" back to 0.
- Using two bits, we'd get something like this:

Present State		Next State	
A	B	A	B
0	0	0	1
0	1	1	0
1	0	1	1
1	1	0	0



Benefits of counters

- Counters can act as simple clocks to keep track of “time.”
- You may need to record how many times something has happened.
 - How many bits have been sent or received?
 - How many steps have been performed in some computation?
- All processors contain a **program counter**, or **PC**.
 - Programs consist of a list of instructions that are to be executed one after another (for the most part).
 - The PC keeps track of the instruction currently being executed.
 - The PC increments once on each clock cycle, and the next program instruction is then executed.

A slightly fancier counter

- Let's try to design a slightly different two-bit counter:
 - Again, the counter outputs will be 00, 01, 10 and 11.
 - Now, there is a single input, X. When $X=0$, the counter value should *increment* on each clock cycle. But when $X=1$, the value should *decrement* on successive cycles.
- We'll need two flip-flops again. Here are the four possible states:

00

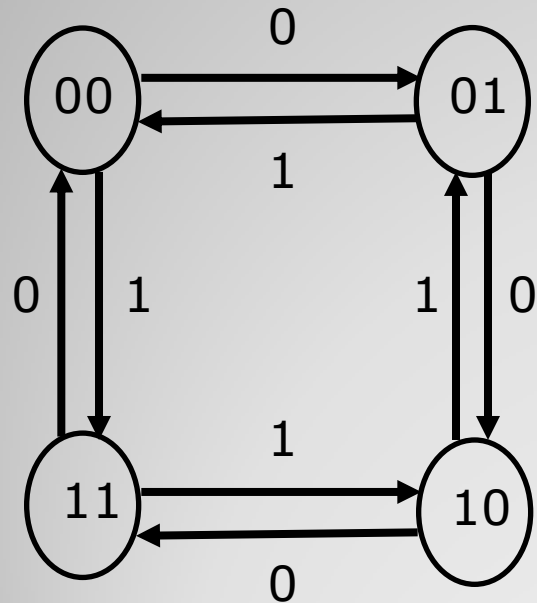
01

11

10

The complete state diagram and table

- Here's the complete state diagram and state table for this circuit.



Present State		Inputs X	Next State	
Q ₁	Q ₀		Q ₁	Q ₀
0	0	0	0	1
0	0	1	1	1
0	1	0	1	0
0	1	1	0	0
1	0	0	1	1
1	0	1	0	1
1	1	0	0	0
1	1	1	1	0

D flip-flop inputs

- If we use D flip-flops, then the D inputs will just be the same as the desired next states.
- Equations for the D flip-flop inputs are shown at the right.
- Why does $D_0 = Q_0'$ make sense?

Present State		Inputs X	Next State	
Q_1	Q_0		Q_1	Q_0
0	0	0	0	1
0	0	1	1	1
0	1	0	1	0
0	1	1	0	0
1	0	0	1	1
1	0	1	0	1
1	1	0	0	0
1	1	1	1	0

			Q_0	
	0	1	0	1
Q_1	1	0	1	0
		X		

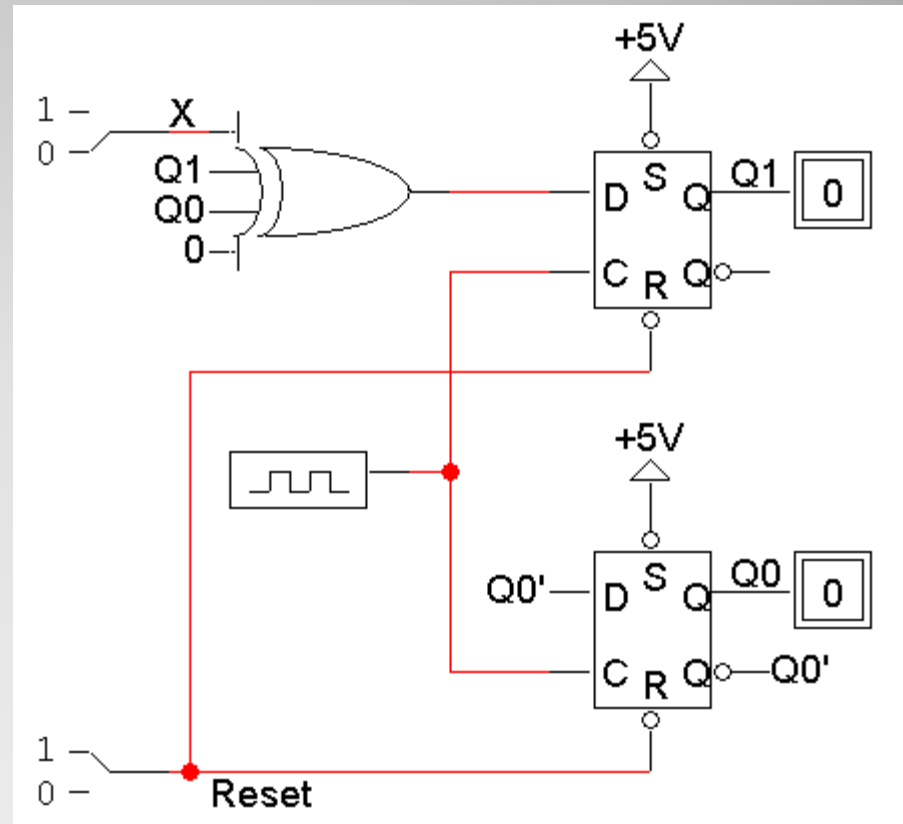
$$D_1 = Q_1 \oplus Q_0 \oplus X$$

			Q_0	
	1	1	0	0
Q_1	1	1	0	0
		X		

$$D_0 = Q_0'$$

The counter in Logic Works

- Here are some **D Flip Flop** devices from LogicWorks.
- They have both normal and complemented outputs, so we can access $Q0'$ directly without using an inverter. ($Q1'$ is not needed in this example.)
- This circuit counts normally when **Reset** = 1. But when Reset is 0, the flip-flop outputs are cleared to 00 immediately.
- There is no three-input XOR gate in LogicWorks so we've used a four-input version instead, with one of the inputs connected to 0.



JK flip-flop inputs

- If we use JK flip-flops instead, then we have to compute the JK inputs for each flip-flop.
- Look at the present and desired next state, and use the excitation table on the right.

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

Present State		Inputs	Next State		Flip flop inputs			
Q ₁	Q ₀	X	Q ₁	Q ₀	J ₁	K ₁	J ₀	K ₀
0	0	0	0	1	0	x	1	x
0	0	1	1	1	1	x	1	x
0	1	0	1	0	1	x	x	1
0	1	1	0	0	0	x	x	1
1	0	0	1	1	x	0	1	x
1	0	1	0	1	x	1	1	x
1	1	0	0	0	x	1	x	1
1	1	1	1	0	x	0	x	1

JK flip-flop input equations

Present State		Inputs	Next State		Flip flop inputs			
Q_1	Q_0		Q_1	Q_0	J_1	K_1	J_0	K_0
0	0	0	0	1	0	x	1	x
0	0	1	1	1	1	x	1	x
0	1	0	1	0	1	x	x	1
0	1	1	0	0	0	x	x	1
1	0	0	1	1	x	0	1	x
1	0	1	0	1	x	1	1	x
1	1	0	0	0	x	1	x	1
1	1	1	1	0	x	0	x	1

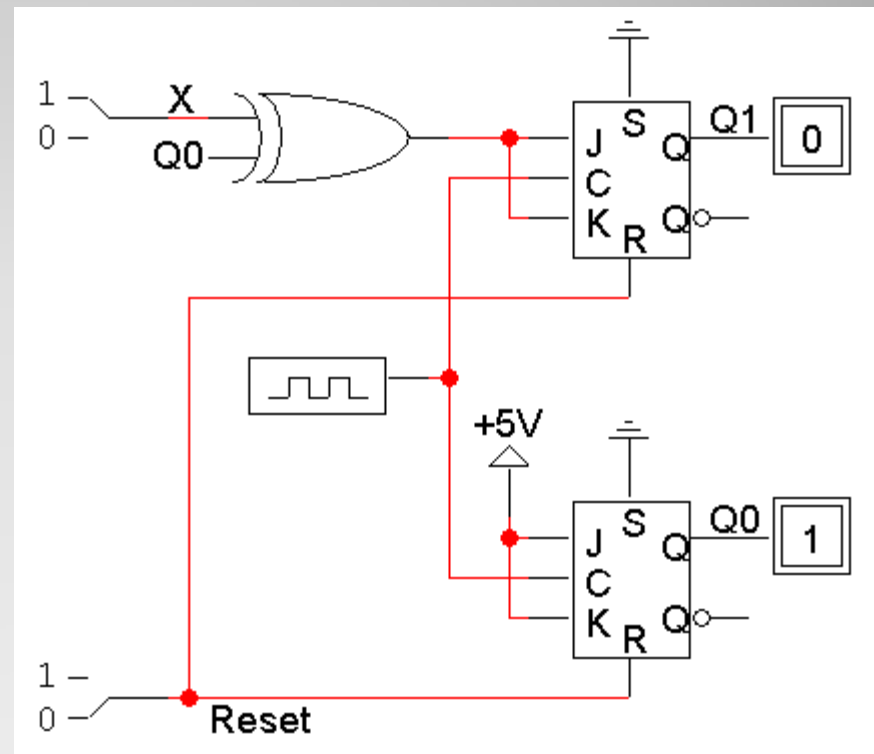
- We can then find equations for all four flip-flop inputs, in terms of the present state and inputs. Here, it turns out $J_1 = K_1$ and $J_0 = K_0$.

$$J_1 = K_1 = Q_0' X + Q_0 X'$$

$$J_0 = K_0 = 1$$

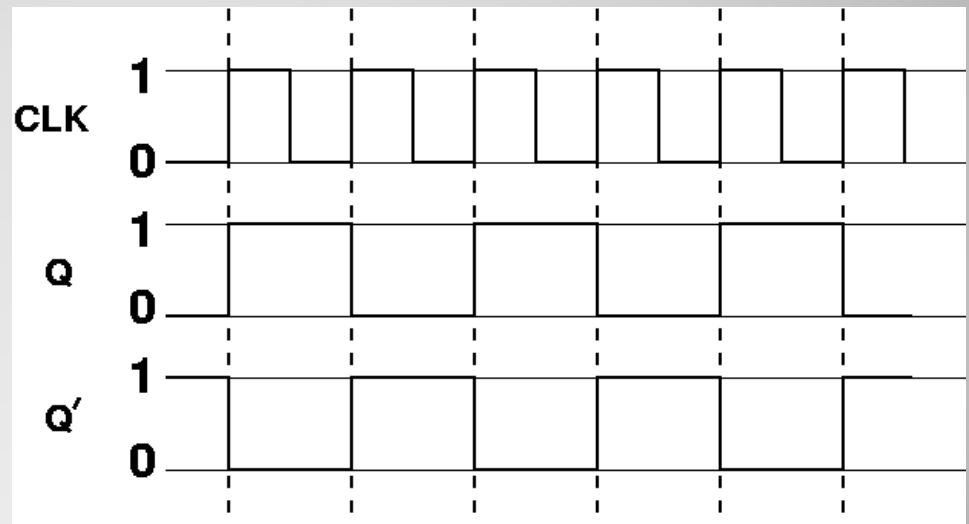
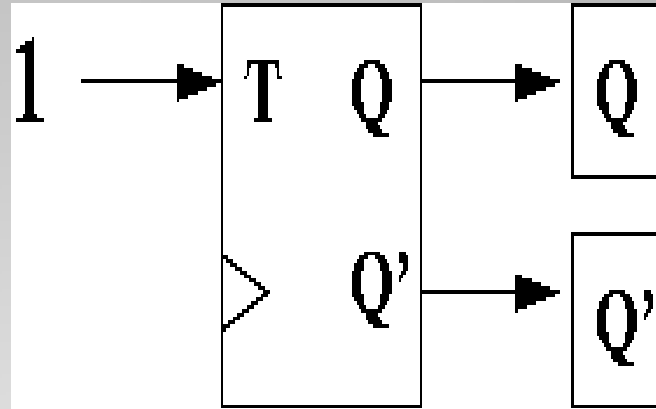
The counter in Logic Works again

- Here is the counter again, but using **JK Flip Flop n.i. RS** devices instead.
- The direct inputs R and S are non-inverted, or active-high.
- So this version of the circuit counts normally when **Reset = 0**, but initializes to 00 when Reset is 1.



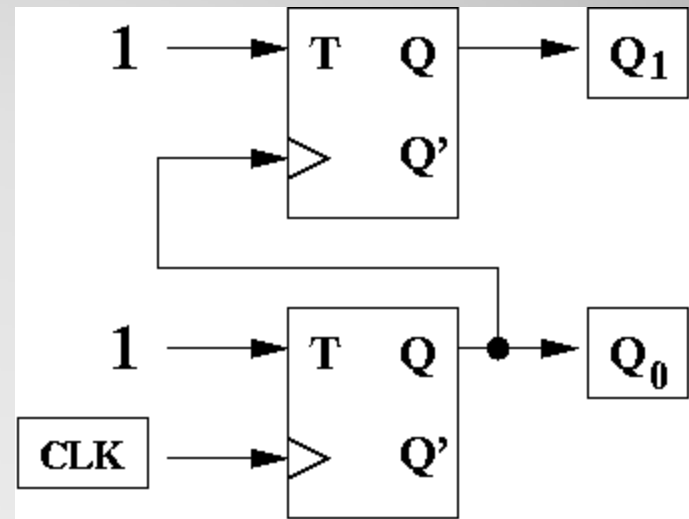
Asynchronous Counters

- This counter is called *asynchronous* because not all flip flops are hooked to the same clock.
- Look at the waveform of the output, **Q**, in the timing diagram. It resembles a clock as well. If the period of the clock is T , then what is the period of **Q**, the output of the flip flop? It's $2T$!
- We have a way to create a clock that runs twice as slow. We feed the clock into a T flip flop, where T is hardwired to 1. The output will be a clock whose period is twice as long.



Asynchronous counters

If the clock has period T . **Q0** has period $2T$. **Q1** period is $4T$
With n flip flops the period is 2^n .

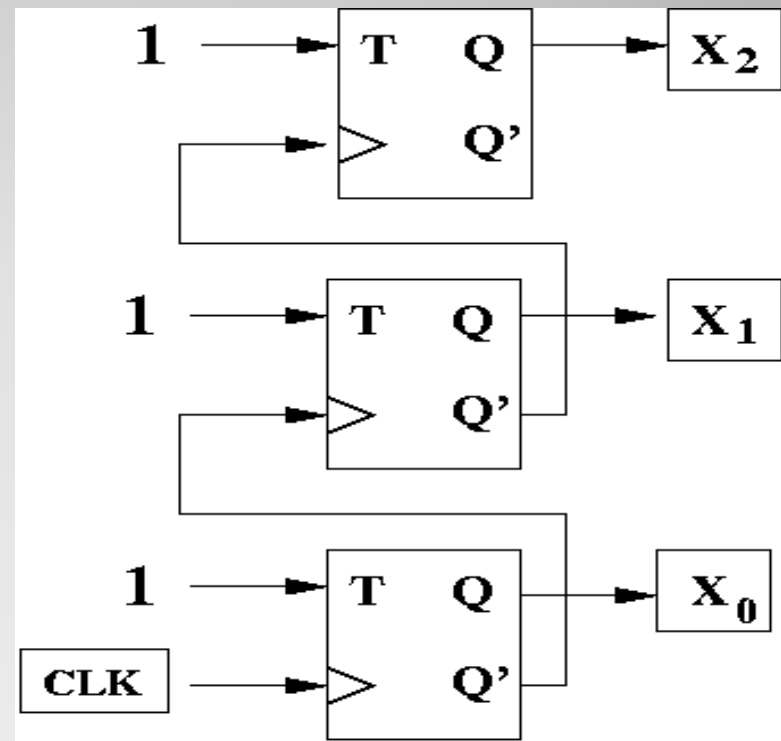
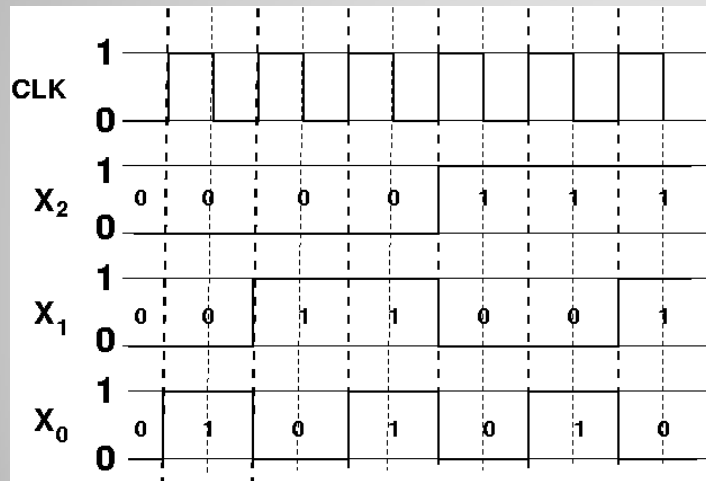


UNIT 4

Registers, Counters, State Reduction

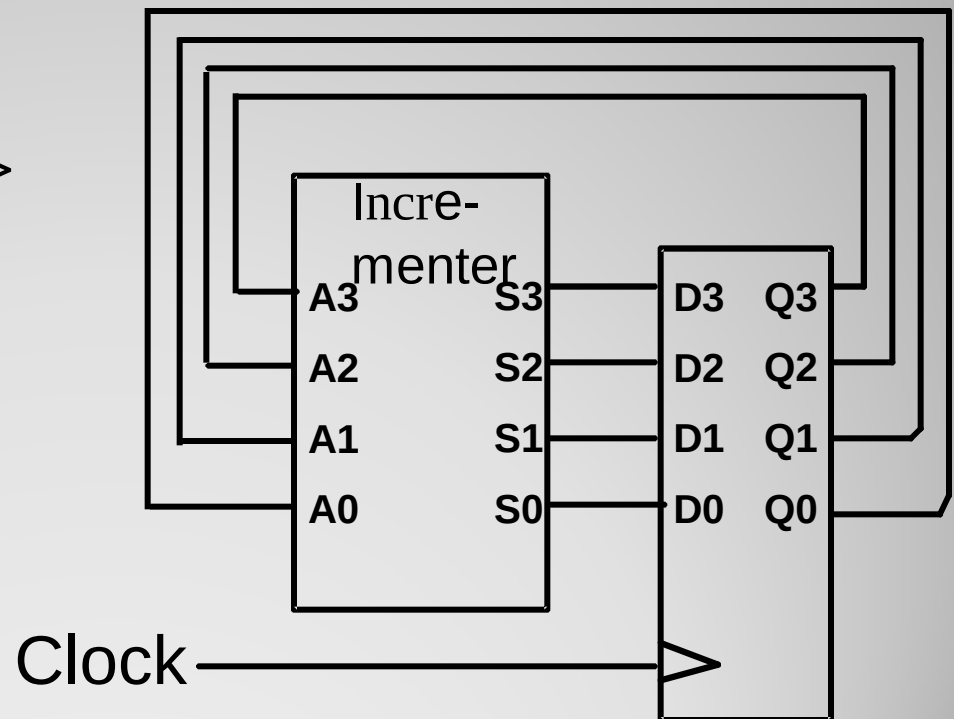
3 bit asynchronous "ripple" counter using T flip flops

- This is called as a *ripple counter* due to the way the FFs respond one after another in a kind of rippling effect.



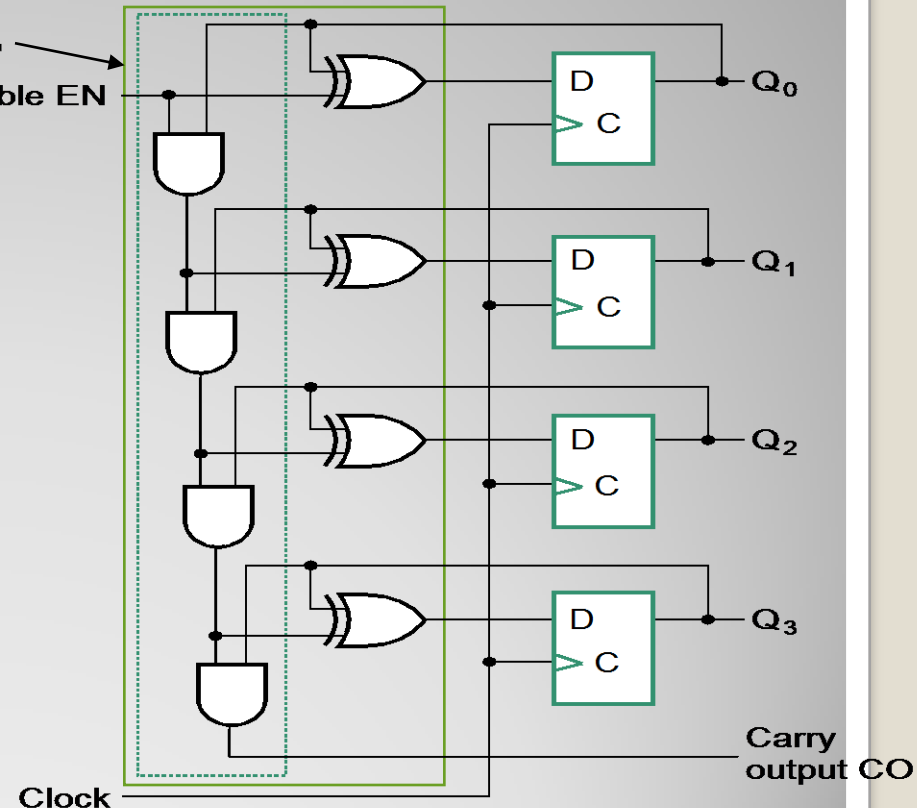
Synchronous Counters

- To eliminate the "ripple" effects, use a common clock for each flip-flop and a combinational circuit to generate the next state.
- For an up-counter, use an incrementer =>



Synchronous Counters (continued)

- Internal details =>
- Internal Logic
 - XOR complements each bit
 - AND chain causes complement of a bit if all bits toward LSB from it equal 1
- Count Enable
 - Forces all outputs of AND chain to 0 to "hold" the state
- Carry Out
 - Added as part of incrementer
 - Connect to Count Enable of additional 4-bit counters to form larger counters



(a) Logic Diagram-Serial Gating

Design Example: Synchronous BCD

- Use the sequential logic model to design a synchronous BCD counter with D flip-flops
- State Table =>
- Input combinations 1010 through 1111 are don't cares

Current State				Next State			
Q8	Q4	Q2	Q1	Q8	Q4	Q2	Q1
0	0	0	0	0	0	0	1
0	0	0	1	0	0	1	0
0	0	1	0	0	0	1	1
0	0	1	1	0	1	0	0
0	1	0	0	0	1	0	1
0	1	0	1	0	1	1	0
0	1	1	0	0	1	1	1
0	1	1	1	1	0	0	0
1	0	0	0	1	0	0	1
1	0	0	1	0	0	0	0

Synchronous BCD (continued)

- Use K-Maps to two-level optimize the next state equations and manipulate into forms containing XOR gates:

$$D1 = Q1'$$

- $D2 = Q2 + Q1Q8'$
 $D4 = Q4 + Q1Q2$
 $D8 = Q8 + (Q1Q8 + Q1Q2Q4)$

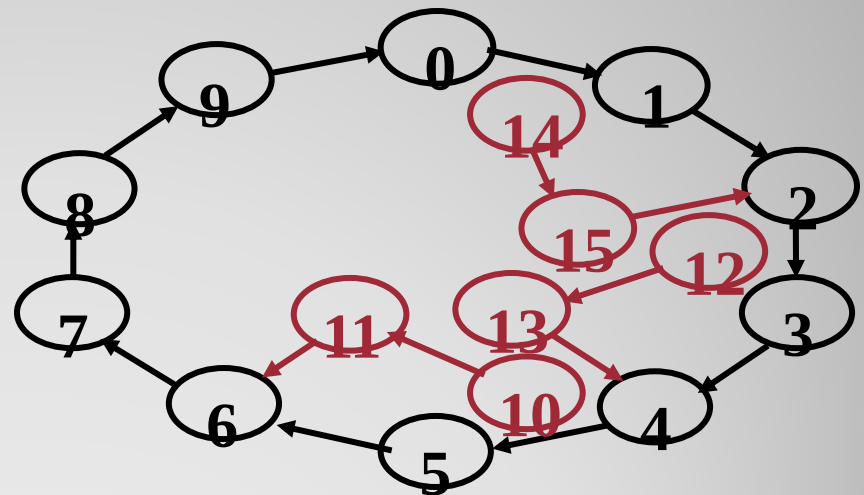
- $Y = Q1Q8$

- The logic diagram can be drawn from these equations
 - An asynchronous or synchronous reset should be added
- What happens if the counter is perturbed by a power disturbance or other interference and it enters a state other than 0000 through 1001?

Synchronous BCD (continued)

- Find the actual values of the six next states for the don't care combinations from the equations
- Find the overall state diagram to assess behavior for the don't care states (states in decimal)

Present State	Next State
Q8 Q4 Q2 Q1	Q8 Q4 Q2 Q1
1 0 1 0	1 0 1 1
1 0 1 1	0 1 1 0
1 1 0 0	1 1 0 1
1 1 0 1	0 1 0 0
1 1 1 0	1 1 1 1
1 1 1 1	0 0 1 0



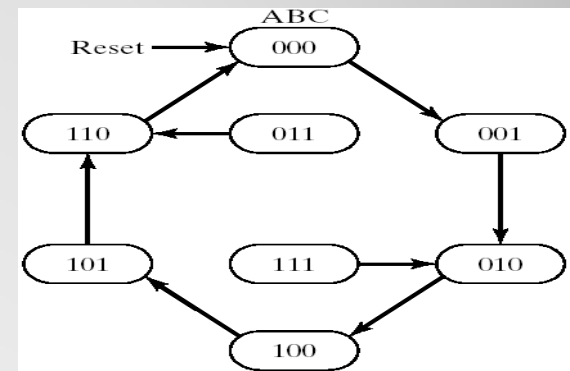
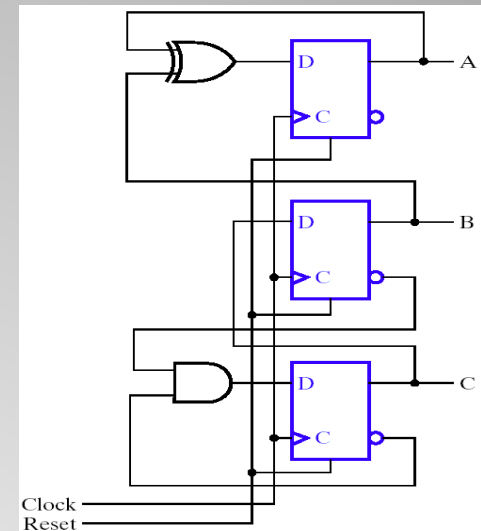
Synchronous BCD (continued)

- For the BCD counter design, if an invalid state is entered, return to a valid state occurs within two clock cycles
- Is this adequate?!

Counting an arbitrary sequence

□ **TABLE 7-10**
State Table and Flip-Flop Inputs for Counter

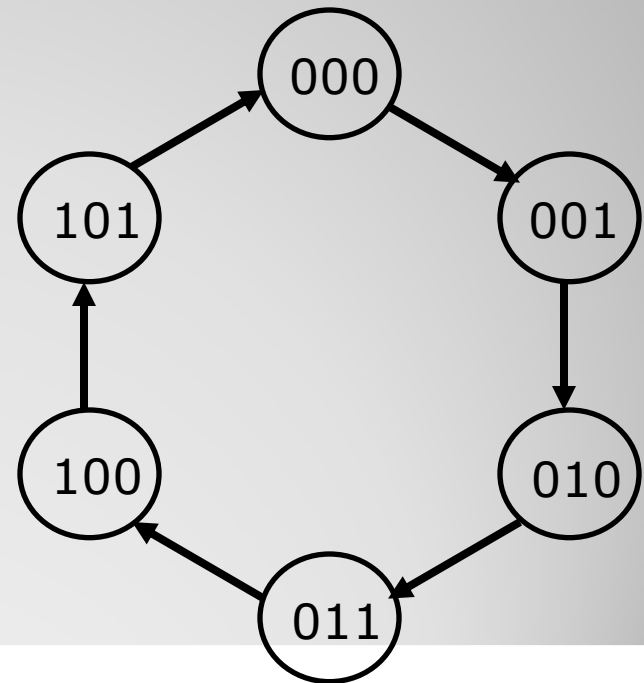
Present State			Next State		
A	B	C	DA = A(t+1)	DB = B(t+1)	DC = C(t+1)
0	0	0	0	0	1
0	0	1	0	1	0
0	1	0	1	0	0
1	0	0	1	0	1
1	0	1	1	1	0
1	1	0	0	0	0



Unused states

- The examples shown so far have all had 2^n states, and used n flip-flops. But sometimes you may have unused, leftover states.
- For example, here is a state table and diagram for a counter that repeatedly counts from 0 (000) to 5 (101).
- What should we put in the table for the two unused states?

Present State			Next State		
Q_2	Q_1	Q_0	Q_2	Q_1	Q_0
0	0	0	0	0	1
0	0	1	0	1	0
0	1	0	0	1	1
0	1	1	1	0	0
1	0	0	1	0	1
1	0	1	0	0	0
1	1	0	?	?	?
1	1	1	?	?	?

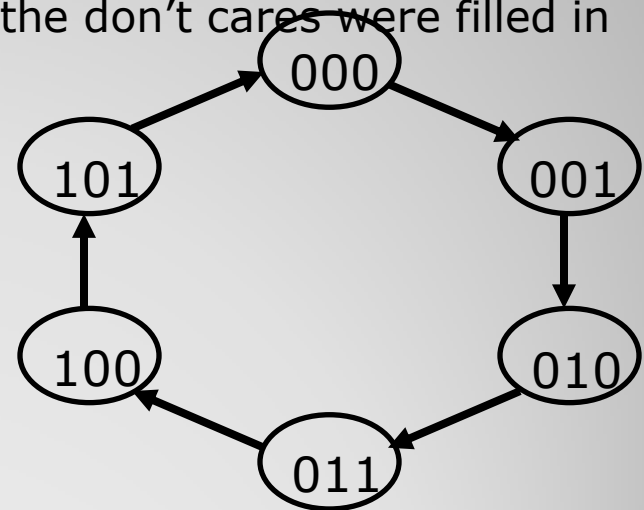


Unused states can be don't cares...

- To get the *simplest* possible circuit, you can fill in don't cares for the next states. This will also result in don't cares for the flip-flop inputs, which can simplify the hardware.

Present State			Next State		
Q_2	Q_1	Q_0	Q_2	Q_1	Q_0
0	0	0	0	0	1
0	0	1	0	1	0
0	1	0	0	1	1
0	1	1	1	0	0
1	0	0	1	0	1
1	0	1	0	0	0
1	1	0	x	x	x
1	1	1	x	x	x

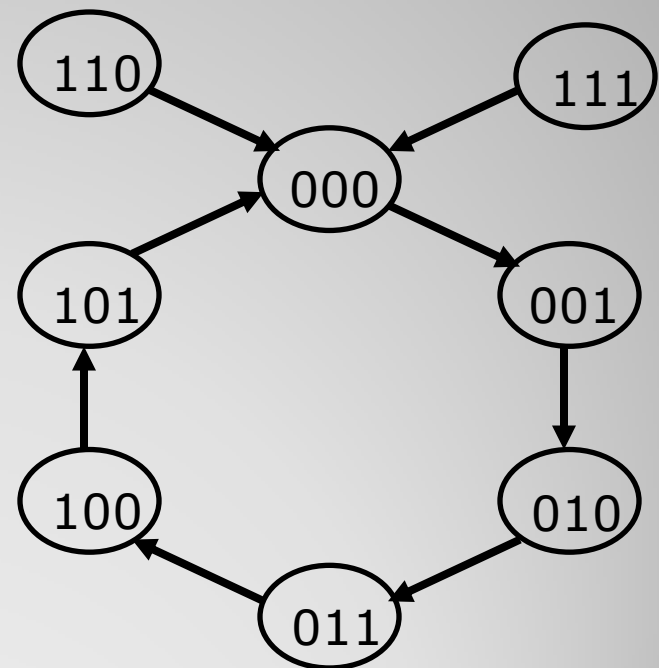
In one of the unused states (110 or 111),
what the don't cares were filled in



...or maybe you *do* care

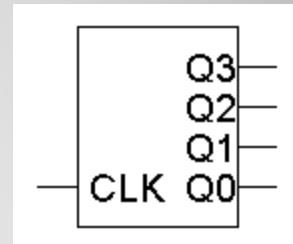
- To get the *safest* possible circuit, you can explicitly fill in next states for the unused states 110 and 111.
- This guarantees that even if the circuit somehow enters an unused state, it will eventually end up in a valid state.
- This is called a **self-starting counter**.

Present State			Next State		
Q_2	Q_1	Q_0	Q_2	Q_1	Q_0
0	0	0	0	0	1
0	0	1	0	1	0
0	1	0	0	1	1
0	1	1	1	0	0
1	0	0	1	0	1
1	0	1	0	0	0
1	1	0	0	0	0
1	1	1	0	0	0



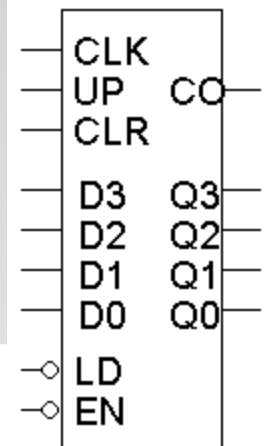
Logic Works counters

- There are a couple of different counters available in LogicWorks.
- The simplest one, the **Counter-4 Min**, just increments once on each clock cycle.
 - This is a four-bit counter, with values ranging from 0000 to 1111.
 - The only “input” is the clock signal.



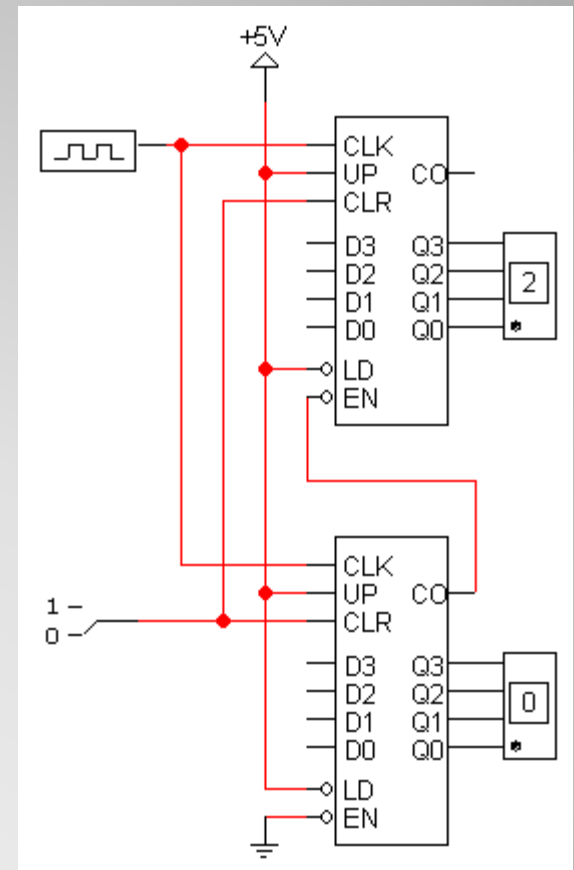
More complex counters

- More complex counters are also possible. The full-featured LogicWorks **Counter-4** device below has several functions.
 - It can increment or decrement, by setting the **UP** input to 1 or 0.
 - You can immediately (asynchronously) clear the counter to 0000 by setting **CLR** = 1.
 - You can specify the counter's next output by setting **D₃-D₀** to any four-bit value and clearing **LD**.
 - The active-low **EN** input enables or disables the counter.
 - When the counter is disabled, it continues to output the same value without incrementing, decrementing, loading, or clearing.
 - The "counter out" **CO** is normally 1, but becomes 0 when the counter reaches its maximum value, 1111.



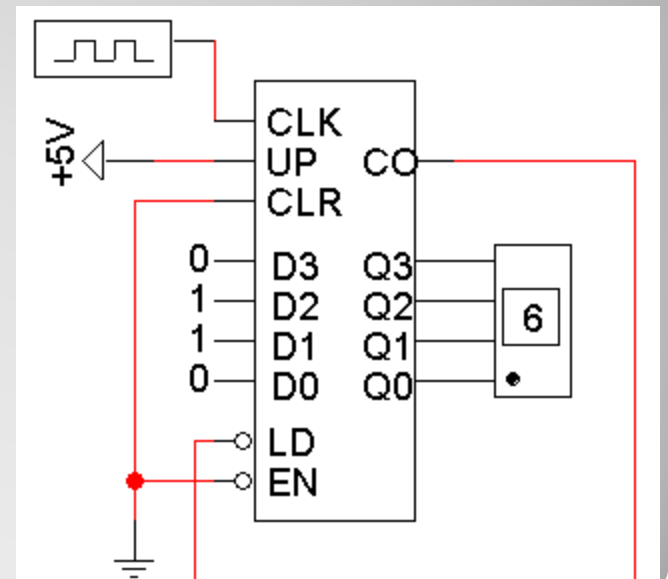
An 8-bit counter

- As you might expect by now, we can use these general counters to build other counters.
- Here is an 8-bit counter made from two 4-bit counters.
 - The bottom device represents the least significant four bits, while the top counter represents the most significant four bits.
 - When the bottom counter reaches 1111 (i.e., when $CO = 0$), it enables the top counter for one cycle.
- Other implementation notes:
 - The counters share clock and clear signals.



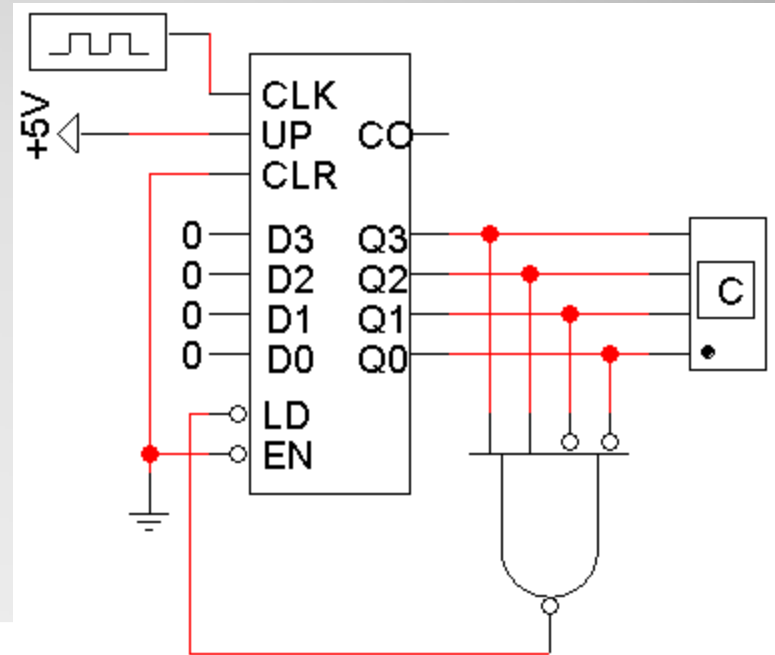
A restricted 4-bit counter

- We can also make a counter that “starts” at some value besides 0000.
- In the diagram below, when $CO=0$ the LD signal forces the next state to be loaded from D_3 - D_0 .
- The result is this counter wraps from 1111 to 0110 (instead of 0000).



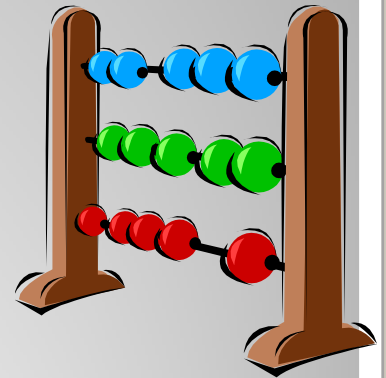
Another restricted counter

- We can also make a circuit that counts up to only 1100, instead of 1111.
- Here, when the counter value reaches 1100, the NAND gate forces the counter to load, so the next state becomes 0000.



Summary of Counters

- Counters serve many purposes in sequential logic design.
- There are lots of variations on the basic counter.
 - Some can increment or decrement.
 - An enable signal can be added.
 - The counter's value may be explicitly set.
- There are also several ways to make counters.
 - You can follow the sequential design principles to build counters from scratch.
 - You could also modify or combine existing counter devices.



Sequential Circuit Design

Creating a sequential circuit to address a design need.

Sequential Circuit Design

- Steps in the design process for sequential circuits
- State Diagrams and State Tables
- Examples

Sequential Circuit Design Process

- Steps in Design of a Sequential Circuit
 - 1. Specification – A description of the sequential circuit. Should include a detailing of the inputs, the outputs, and the operation. Possibly assumes that you have knowledge of digital system basics.
 - 2. Formulation: Generate a state diagram and/or a state table from the statement of the problem.
 - 3. State Assignment: From a state table assign binary codes to the states.
 - 4. Flip-flop Input Equation Generation: Select the type of flip-flop for the circuit and generate the needed input for the required state transitions

Sequential Circuit Design Process

2

- 5. Output Equation Generation: Derive output logic equations for generation of the output from the inputs and current state.
- 6. Optimization: Optimize the input and output equations. Today, CAD systems are typically used for this in real systems.
- 7. Technology Mapping: Generate a logic diagram of the circuit using ANDs, ORs, Inverters, and F/Fs.
- 8. Verification: Use a HDL to verify the design.

Mealy and Moore

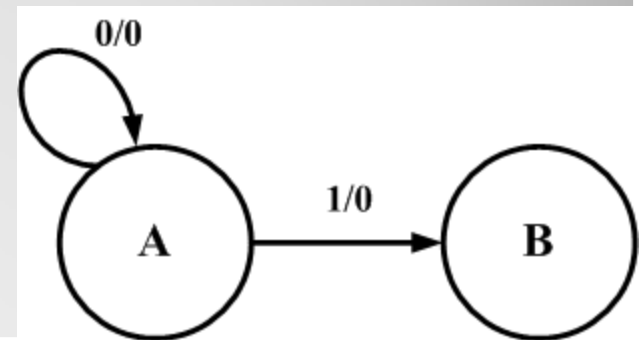
- Sequential machines are typically classified as either a Mealy machine or a Moore machine implementation.
- Moore machine: The outputs of the circuit depend only upon the current state of the circuit.
- Mealy machine: The outputs of the circuit depend upon both the current state of the circuit and the inputs.

An example to go through the steps

- The specification: The circuit will have one input, X , and one output, Z . The output Z will be 0 except when the input sequence 1101 are the last 4 inputs received on X . In that case it will be a 1.

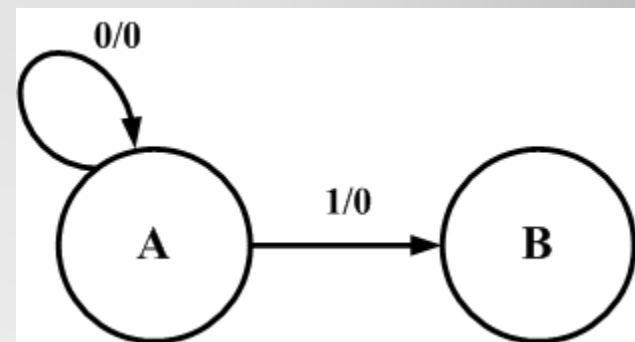
Generation of a state diagram

- Create states and meaning for them.
 - State A – the last input was a 0 and previous inputs unknown. Can also be the reset state.
 - State B – the last input was a 1 and the previous input was a 0. The start of a new sequence possibly.
- Capture this in a state diagram



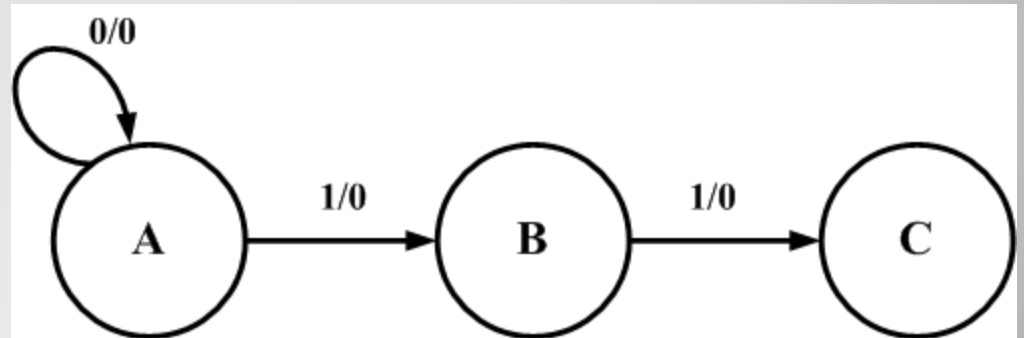
Notes on State diagrams

- Capture this in a state diagram
 - Circles represent the states
 - Lines and arcs represent the transition between state.
 - The notation Input/Output on the line or arc specifies the input that causes this transition and the output for this change of state.



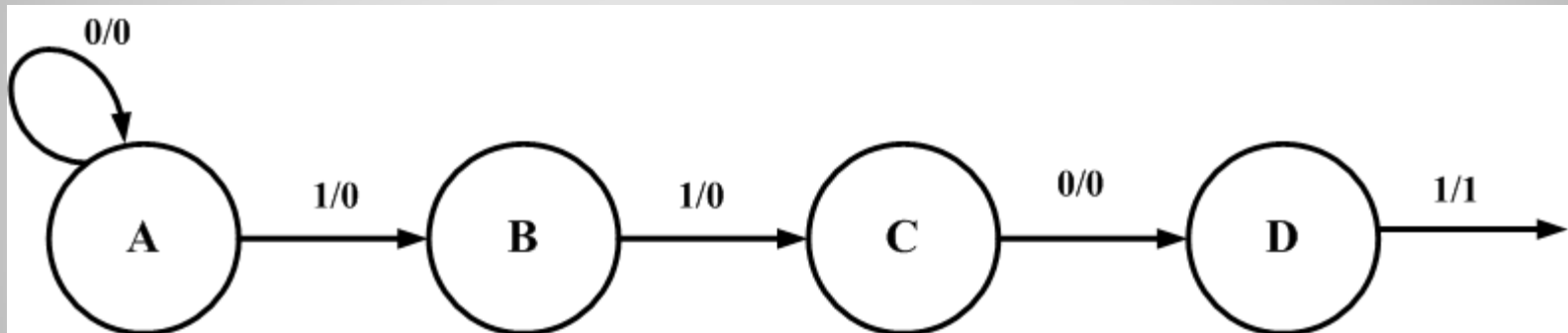
Continue to build up the diagram

- Add a state C
 - State C – Have detected the input sequence 11 which is the start of the sequence.



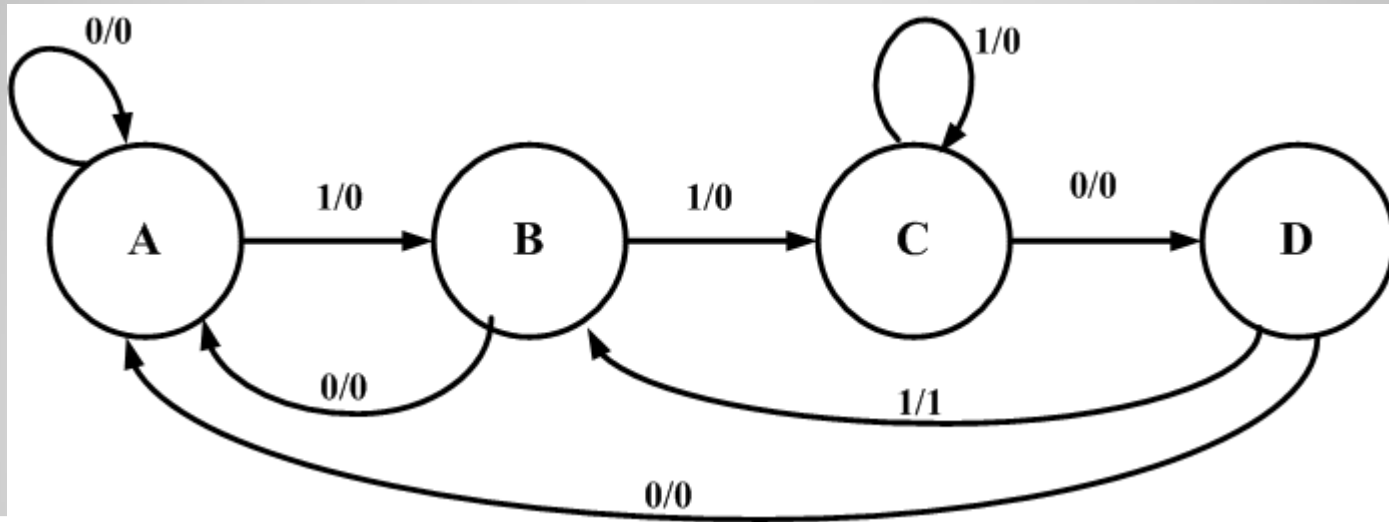
Continue.....

- Add a state D
 - State D – have detected the 3rd input in the start of a sequence, a 0, now having 110. From State D, if the next input is a 1 the sequence has been detected and a 1 is output.



Add remaining transitions

- The previous diagram was incomplete.
- In each state the next input could be a 0 or a 1. This must be included.



Now generate a state table

- The state table
- This can be done directly from the state diagram.
- Now need to do a state assignment

Present State	Next State		Output	
	X=0	X=1	X=0	X=1
A	A	B	0	0
B	A	C	0	0
C	D	C	0	0
D	A	B	0	1

Select a state assignment

- Will select a gray encoding
- For this state A will be encoded 00, state B 01, state C 11 and state D 10

Present State	Next State		Output	
	X = 0	X = 1	X = 0	X = 1
00	00	01	0	0
01	00	11	0	0
11	10	11	0	0
10	00	01	0	1

Flip-flop input equations

- Generate the equations for the flip-flop inputs
- Generate the D_0 equation

Q_0Q_1		00	01	11	10
X	0			1	
	1		1	1	

$D_0 = Q_0 Q_1 + X Q_1$

- Generate the D_1 equation

Q_0Q_1		00	01	11	10
X	0				
	1	1	1	1	1

$D_1 = X$

The output equation

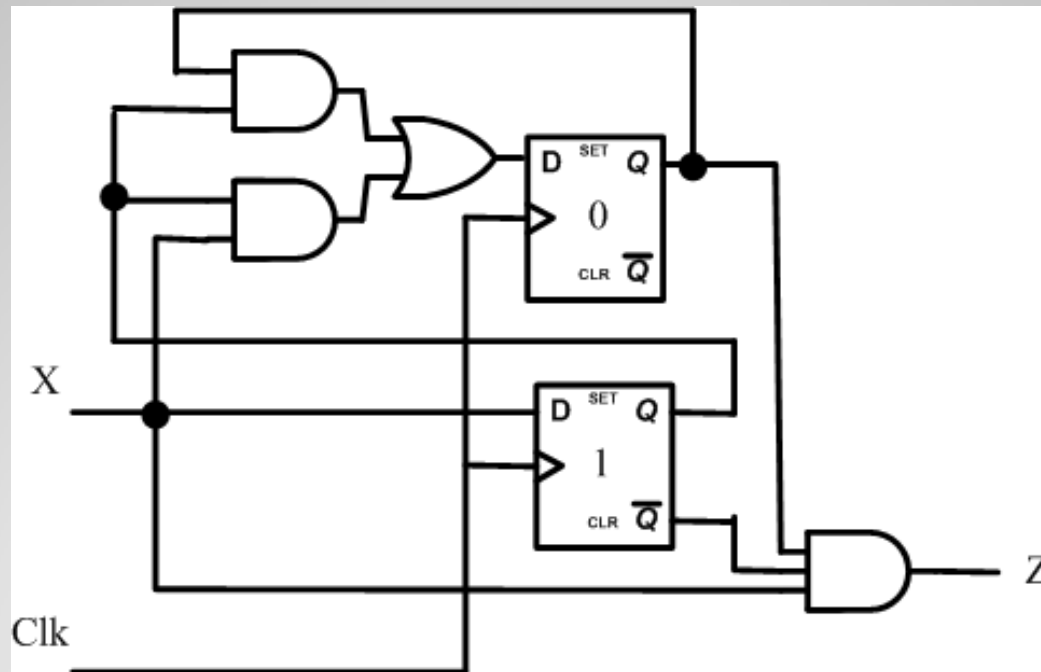
- The next step is to generate the equation for the output Z and what is needed to generate it.
- Create a K-map from the truth table.

		Q_0Q_1			
X	0				
	1				1

$$Z = X Q_0 \overline{Q_1}$$

Now map to a circuit

- The circuit has 2 D type F/Fs



UNIT 5

State Minimization for Completely Specified Machines and Incomplete FSM

STGs may contain **redundant** states, i.e. states whose function can be accomplished by other states.

State minimization is the transformation of a given machine into an **equivalent** machine with no redundant states.

State Minimization: Completely Specified Machines

Two states, s_i and s_j of machine M are *distinguishable* if and only if there exists a finite input *sequence* which when applied to M causes different output sequences depending on whether M started in s_i or s_j .

Such a sequence is called a *distinguishing sequence* for (s_i, s_j) .

If there exists a distinguishing sequence of length k for (s_i, s_j) , they are said to be *k-distinguishable*.

State Minimization: Completely Specified Machines

PS	NS, z	
	x=0	x=1
A	E, 0	D, 1
B	F, 0	D, 0
C	E, 0	B, 1
D	F, 0	B, 0
E	C, 0	F, 1
F	B, 0	C, 0

Example:

- states A and B are **1-distinguishable**, since a 1 input applied to A yields an output 1, versus an output 0 from B.
- states A and E are **3-distinguishable**, since input sequence 111 applied to A yields output 100, versus an output 101 from E.

Completely Specified Machines

States s_i and s_j ($s_i \sim s_j$) are said to be equivalent iff no distinguishing sequence exists for (s_i, s_j) .

If $s_i \sim s_j$ and $s_j \sim s_k$, then $s_i \sim s_k$. So state equivalence is an **equivalence relation** (i.e. it is a reflexive, symmetric and transitive relation).

An equivalence relation partitions the elements of a set into **equivalence classes**.

Property: If $s_i \sim s_j$, their corresponding X-successors, for all inputs X, are also equivalent.

Procedure: Group states of M so that two states are in the same group iff they are equivalent (forms a partition of the states).

Completely Specified Machines

PS	NS, z	
	x=0	x=1
A	E, 0	D, 1
B	F, 0	D, 0
C	E, 0	B, 1
D	F, 0	B, 0
E	C, 0	F, 1
F	B, 0	C, 0

P_i : partition using distinguishing sequences of length i .

Partition:

$$P_0 = (A \ B \ C \ D \ E \ F)$$

$$P_1 = (A \ C \ E)(B \ D \ F)$$

$$P_2 = (A \ C \ E)(B \ D)(F)$$

$$P_3 = (A \ C)(E)(B \ D)(F)$$

$$P_4 = (A \ C)(E)(B \ D)(F)$$

Distinguishing Sequence:

$$x = 1$$

$$x = 1; x = 1$$

$$x = 1; x = 1; x = 1$$

Algorithm terminates when $P_k = P_{k+1}$

Completely Specified Machines

Outline of state minimization procedure:

- All states equivalent to each other form an equivalence class. These may be combined into one state in the reduced (quotient) machine.
- Start an initial partition of a single block. Iteratively refine this partition by separating the 1-distinguishable states, 2-distinguishable states and so on.
- To obtain P_{k+1} , for each block B_i of P_k , create one block of states that not 1-distinguishable within B_i , and create different blocks states that are 1-distinguishable within B_i .

Completely Specified Machines

Theorem: The equivalence partition is **unique**.

Theorem: If two states, s_i and s_j , of machine M are distinguishable, then they are $(n-1)$ -distinguishable, where n is the number of states in M .

Definition: Two machines, M_1 and M_2 , are **equivalent** ($M_1 \sim M_2$) iff, for every state in M_1 there is a corresponding equivalent state in M_2 and vice versa.

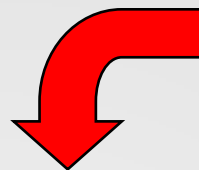
Theorem. For every machine M there is a minimum machine $M_{red} \sim M$.
 M_{red} is **unique** up to isomorphism.

Completely Specified Machines

Reduced machine obtained from previous example:

$$P_4 = (A \ C)(E)(B \ D)(F) \\ = \alpha \ \beta \ \gamma \ \delta$$

PS	NS, z	
	x=0	x=1
α	$\beta, 0$	$\gamma, 1$
β	$\alpha, 0$	$\delta, 1$
γ	$\delta, 0$	$\gamma, 0$
δ	$\gamma, 0$	$\alpha,$
0		



PS	NS, z	
	x=0	x=1
A	E, 0	D,
1		
B	F, 0	D,
0		
C	E, 0	B,
1		
D	F, 0	B,
0		
E	C, 0	F, 1
F	B, 0	C,
0		

State Minimization of CSMs: Complexity

Algorithm $\text{DFA} \sim \text{DFA}_{\min}$

Input: A finite automaton $M = (Q, \Sigma, \delta, q_0, F)$ with no unreachable states.

Output: A minimum finite automaton $M' = (Q', \Sigma, \delta', q'_0, F')$.

Method:

1. $t := 2$; $Q_0 := \{ \text{undefined} \}$; $Q_1 := F$; $Q_2 := Q \setminus F$.
 2. while there is $0 < i \leq t$, $a \in \Sigma$ with $\delta(Q_i, a) \subseteq Q_j$, for all $j \leq t$
 - do (a) Choose such an i, a , and $j \leq t$ with $\delta(Q_i, a) \cap Q_j \neq \emptyset$.
 - (b) $Q_{t+1} := \{q \in Q_i \mid \delta(q, a) \in Q_j\}$;
 $Q_i := Q_i \setminus Q_{t+1}$;
 $t := t + 1$.
- end.

State Minimization of CSMs: Complexity

3. (* Denote
 $[q]$ the equivalence class of state q ,
 and $\{Q_i\}$ the set of all
 equivalence classes.

*)

$$Q' := \{Q_1, Q_2, \dots, Q_t\}.$$

$$q'_0 := [q_0].$$

$$F' := \{ [q] \in Q' \mid q \in F \}.$$

$$\delta'([q], a) := [\delta(q, a)] \text{ for all } q \in Q, a \in \Sigma.$$

State Minimization of CSMs: Complexity

Standard implementation: $O(kn^2)$, where $n = |Q|$ and $k = |\Sigma|$

Modification of the body of the while loop:

1. Choose such an $i, a \in \Sigma$, and choose $j_1, j_2 \leq t$ with $j_1 \neq j_2$, $\delta(Q_i, a) \cap Q_{j_1} \neq \emptyset$, and $\delta(Q_i, a) \cap Q_{j_2} \neq \emptyset$.
2. **If** $|\{q \in Q_i \mid \delta(q, a) \in Q_{j_1}\}| \leq |\{q \in Q_i \mid \delta(q, a) \in Q_{j_2}\}|$
 then $Q_{t+1} := \{q \in Q_i \mid \delta(q, a) \in Q_{j_1}\}$
 else $Q_{t+1} := \{q \in Q_i \mid \delta(q, a) \in Q_{j_2}\}$ **fi**;
 $Q_i := Q_i \setminus Q_{t+1}$;
 $t := t + 1$.
 (i.e. put smallest set in $t + 1$)

State Minimization of CSMs: Complexity

Note: $|Q_{t+1}| \leq 1/2|Q_i|$. Therefore, for all $q \in Q$, the **name** of the class which contains a given state q changes at most $\log(n)$ times.

Goal: Develop an implementation such that all computations can be assigned to transitions containing a state for which the name of the corresponding class is changed.

State Minimization of CSMs: BDD Implementation

X and Y are spaces of all states:

$$E_0(x,y) = \prod_{i=1}^{|S|} (x_i \sim y_i) \quad (\text{initially all states are equivalent})$$

$$E_{j+1}(x,y) = E_j(x,y) \wedge \forall i \exists (o,z,w) [T(x,i,z,o) \wedge T(y,i,w,o) \wedge E_j(z,w)]$$

(i.e. states x,y continue to be equivalent if they are
 j - equivalent and for all inputs the next states are
 j - equivalent)

State Minimization: Incompletely Specified Machines

Statement of the problem: given an incompletely specified machine M , find a machine M' such that:

- on any input sequence, M' produces the same outputs as M , whenever M is specified.
- there does not exist a machine M'' with fewer states than M' which has the same property.

State Minimization: Incompletely Specified Machines

Machine M :

PS	NS, z	
	x=0	x=1
s1	s3, 0	s2, 0
s2	s2, -	s3, 0
s3	s3, 1	s2, 0

Attempt to **reduce** this case to usual state minimization of **completely specified machines**.

- **Brute Force Method:** Force the don't cares to **all** their possible values and choose the smallest of the completely specified machines so obtained.

In this example, it means to state minimize two completely specified machines obtained from M , by setting the don't care to either 0 and 1.

State Minimization: Incompletely Specified Machines

Suppose that the - is set to be a 0.

Machine M' :

PS	NS, z	
	x=0	x=1
s1	s3, 0	s2, 0
s2	s2, 0	s3, 0
s3	s3, 1	s2, 0

States s1 and s2 are equivalent if s3 and s2 are equivalent, but s3 and s2 assert different outputs under input 0, so s1 and s2 are not equivalent.

States s1 and s3 are not equivalent either.

So this completely specified machine cannot be reduced further (3 states is the minimum).

Incompletely Specified Machines

Suppose that the - is set to be a 1.

Machine M'' :

PS	NS, z	
	x=0	x=1
s1	s3, 0	s2, 0
s2	s2, 1	s3, 0
s3	s3, 1	s2, 0

States s1 is incompatible with both s2 and s3.

States s3 and s2 are equivalent.

So number of states is reduced from 3 to 2.

Machine M''_{red} :

PS	NS, z	
	x=0	x=1
A	A, 1	A, 0
B	B, 0	A, 0

State Minimization: Incompletely Specified Machines

Can this always be done?

Machine *M*:

PS	NS, z	
	x=0	x=1
s1	s3, 0	s2, 0
s2	s2, -	s1, 0
s3	s1, 1	s2, 0

State Minimization: Incompletely Specified Machines

Machine M_2 :

PS	NS, z	
	x=0	x=1
s1	s3, 0	s2, 0
s2	s2, 0	s1, 0
s3	s1, 1	s2, 0

Machine M_3 :

PS	NS, z	
	x=0	x=1
s1	s3, 0	s2, 0
s2	s2, 1	s1, 0
s3	s1, 1	s2, 0

Machine M_2 and M_3 are formed by filling in the unspecified entry in M with 0 and 1, respectively.

Both machines M_2 and M_3 cannot be reduced.

Conclusion?: M cannot be minimized further!

But is it a correct conclusion?

State Minimization: Incompletely Specified Machines

Note: that we want to 'merge' two states when, for any input sequence, they generate the same output sequence, but only where both outputs are specified.

Definition: A set of states is compatible if they agree on the outputs where they are all **specified**.

Machine M'' :

PS	NS, z	
	x=0	x=1
s1	s3, 0	s2, 0
s2	s2, -	s1, 0
s3	s1, 1	s2, 0

In this case we have two compatible sets: $A = (s1, s2)$ and $B = (s3, s2)$. A reduced machine M_{red} can be built as follows.

Machine M_{red} :

PS	NS, z	
	x=0	x=1
A	B, 0	A, 0
B	A, 1	A, 0

Incompletely Specified Machines

Can we simply look for a set of compatibles of minimum cardinality, such that every original state is in at least one compatible?

No. To build a reduced machine we must be able to send compatibles into compatibles. So choosing a given compatible may imply that some other compatibles must be chosen too.

Incompletely Specified Machines

PS	NS, z			
	I1	I2	I3	I4
s1	s3,0	s1,-	-	-
s2	s6,-	s2,0	s1,-	-
s3	-,1	-, -	s4,0	-
s4	s1,0	-, -	-	s5,1
s5	-, -	s5,-	s2,1	s1,1
s6	-, -	s2,1	s6,-	s4,1

A set of compatibles that cover all states is: (s3s6), (s4s6), (s1s6), (s4s5), (s2s5).

But (s3s6) requires (s4s6),

(s4s6) requires (s4s5), (s4s5) requires (s1s5),

(s1s6) requires (s1s2), (s1s2) requires (s3s6),

(s2s5) requires (s1s2).

So, this selection of compatibles requires too many other compatibles...

Incompletely Specified Machines

PS	NS, z			
	I1	I2	I3	I4
s1	s3,0	s1,-	-	-
s2	s6,-	s2,0	s1,-	-
s3	-,1	-, -	s4,0	-
s4	s1,0	-, -	-	s5,1
s5	-, -	s5,-	s2,1	s1,1
s6	-, -	s2,1	s6,-	s4,1

Another set of compatibles that covers all states is $(s_1s_2s_5)$, (s_3s_6) , (s_4s_5) .

But

(s1s2s5) requires (s3s6) (s3s6) requires (s4s6) (s4s6) requires (s4s5) (s4s5) requires (s1s5).

So must select also (s4s6) and (s1s5).

Selection of minimum set is a binate covering problem !!!

Incompletely Specified Machines

More formally:

When a next state is unspecified, the future behavior of the machine is unpredictable. This suggests the definition of admissible input sequence.

Definition. An input sequence is *admissible*, for a starting state of a machine if no unspecified next state is encountered, except possibly at the final step.

Definition. State s_i of machine M_1 is said to *cover*, or *contain*, state s_j of M_2 provided

1. every input sequence admissible to s_j is also admissible to s_i , and
2. its application to both M_1 and M_2 (initially is s_i and s_j , respectively) results in identical output sequences whenever the outputs of M_2 are specified.

State Minimization: Incompletely Specified Machines

Definition. Machine M_1 is said to cover machine M_2 iff
for every state s_j in M_2 , there is a
corresponding state s_i in M_1 such that s_i covers
 s_j .

The problem of state minimization for an
incompletely specified machine M is:

*find a machine M' which covers M such that
for any other machine M'' covering M , the
number of states of M' does not exceed the
number of states of M'' .*

Short summary of rest of section

- Definition of compatible states
- Method to compute when two states are incompatible
- Definition of maximal compatible sets
 - A set is compatible if all pairs in the set are compatible
- Definition of prime compatibles
- Solve Quine-McCluskey type problem
 - Generate all prime compatibles
 - Solve binate covering problem

Algorithmic State Machines

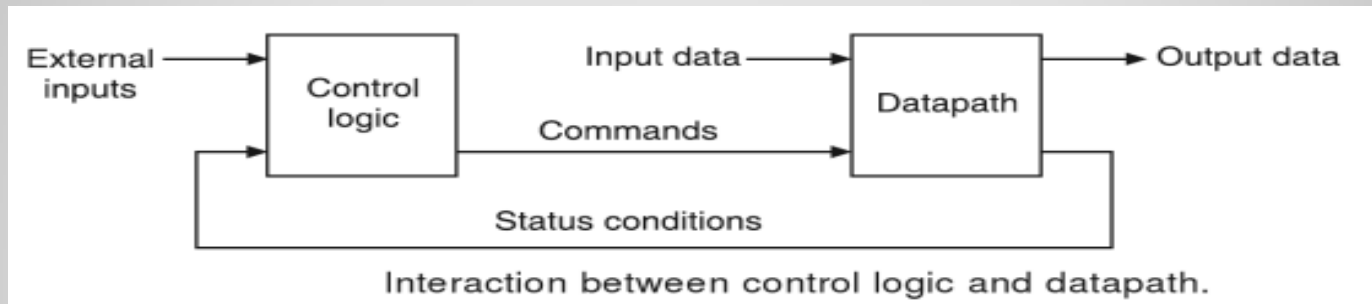
- ASM chart
- Salient features of the ASM
- Examples of system design

Binary multiplier

Weighing Machine

Introduction

- The binary information stored in the digital system can be classified as either data or control information
- The data information is manipulated by performing arithmetic, logic, shift and other data processing tasks
- The control information provides the command signals that controls the various operations on the data in order to accomplish the desired data processing task
- Design a digital system we have to design two subsystems data path subsystem and control subsystem



ASM CHART

- A special flow chart that has been developed specifically to define digital hardware algorithms is called ASM chart.
- A hardware algorithm is a step by step procedure to implement the desire task

What is the Difference b/n conventional flow chart and ASM chart

- conventional flow chart describes the sequence of procedural steps and decision paths for an algorithm without concern for their time relationship
- An ASM chart describes the sequence of events as well as the timing relationship b/n the states of sequential controller and the events that occur while going from one state to the next

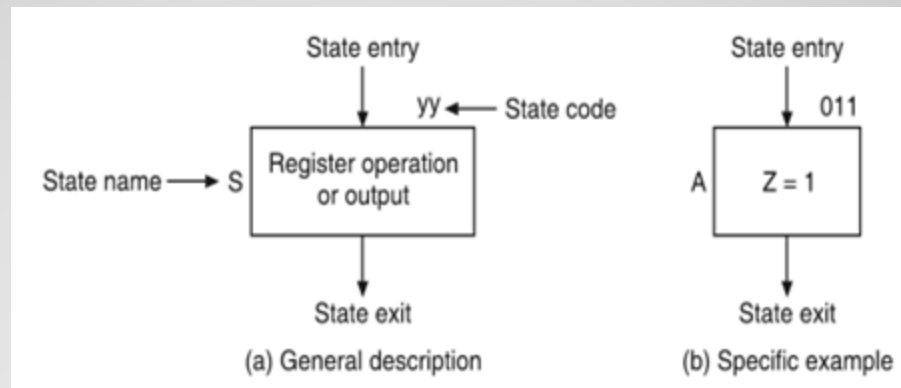
1. State box: A state of a clocked sequential circuit is represented by a rectangle called *state box*. It is equivalent to a node in the state diagram or a row in the state table. The name of the state is written to the left of the box. The binary code assigned to the state is indicated outside on the top right-side of the box. A list of unconditional outputs if any associated with the state are written within the box.

2. Decision box: The decision box or condition box is represented by a diamond-shaped symbol with one input and two or more output paths. The output branches are true and false branches. The decision box describes the effect of an input on the control subsystem. A Boolean variable or input or expression written inside the diamond indicates a condition which is evaluated to determine which branch to take.

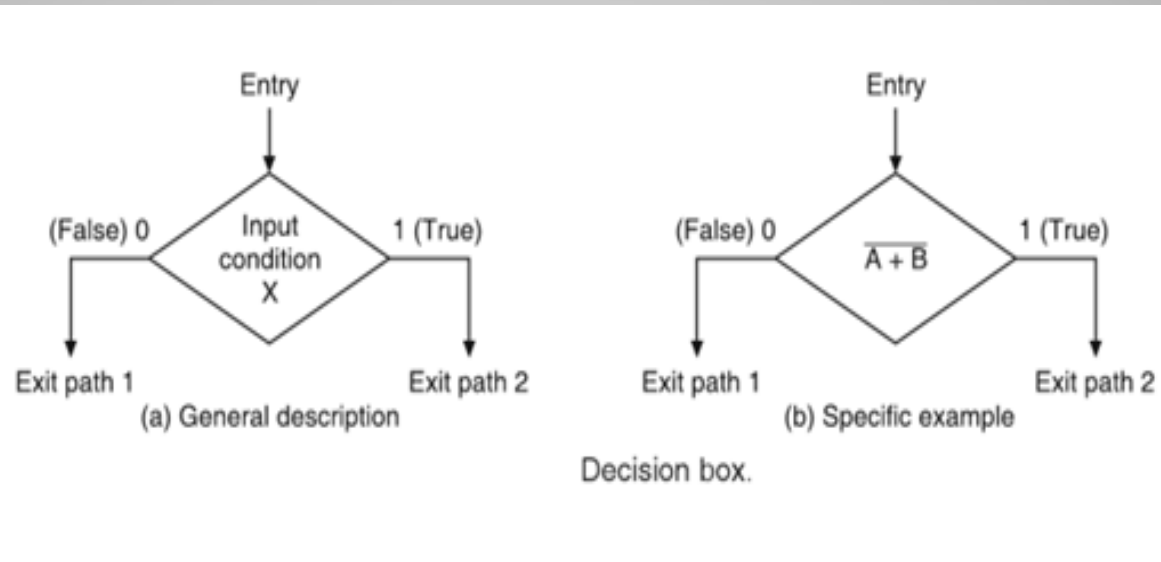
ASM consists of

1. State box
2. Decision box
3. Conditional box

State box

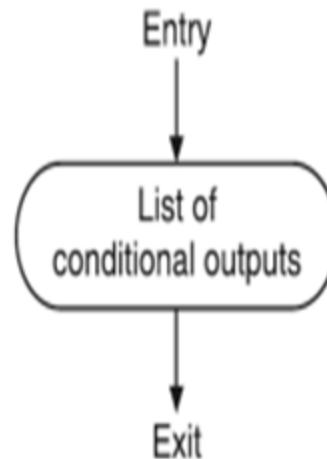


Decision box



Decision box.

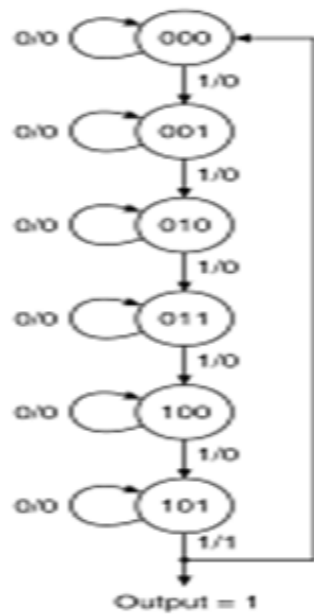
3. Conditional output box: The conditional output box is represented by a rectangle with rounded corners or by an oval with one input line and one output line. The outputs that depend on both the state of the system and the inputs are indicated inside the box.



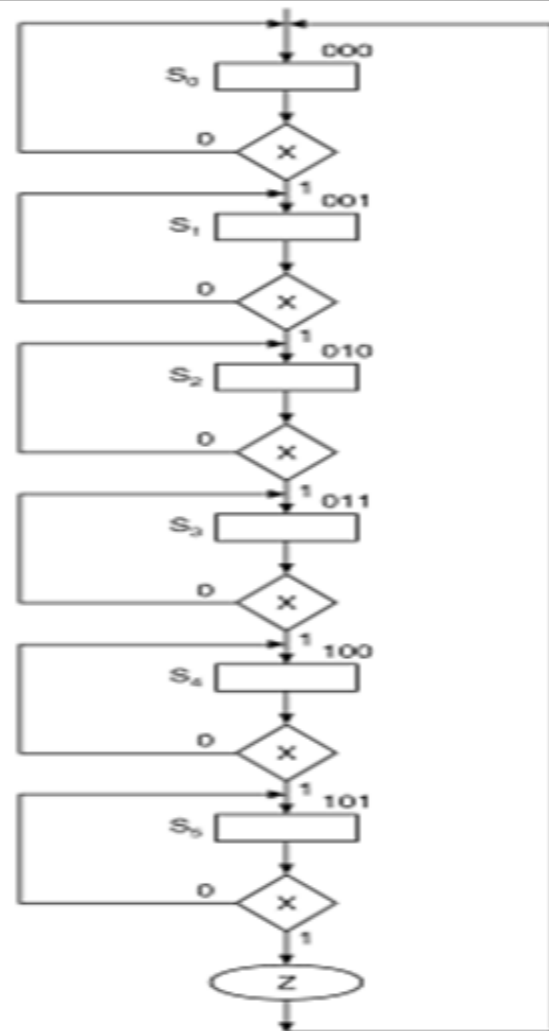
Conditional output box.

SALIENT FEATURES OF ASM CHARTS

1. An ASM chart describes the sequence of events as well as the timing relationship between the states of a sequential controller and the events that occur while going from one state to the next.
2. An ASM chart contains one or more interconnected ASM blocks.
3. Each ASM block contains exactly one state box together with the decision boxes and conditional output boxes associated with that state.
4. Every block in an ASM chart specifies the operations that are to be performed during one common clock pulse.
5. An ASM block has exactly one entrance path and one or more exit paths represented by the structure of the decision boxes.
6. A path through an ASM block from entrance to exit is referred to as a link path.
7. The operations specified within the state and conditional output boxes in the block are performed in the datapath subsystem.
8. Internal feedback within an ASM block is not permitted. Even so, following a decision box or conditional output boxes, the machine may reenter the same state.
9. Each block in the ASM chart describes the state of the system during one clock pulse interval. When a digital system enters the state associated with a given ASM block, the outputs indicated within the state box become true. The conditions associated with the decision boxes are evaluated to determine which path or paths to be followed to enter the next ASM block.



(a) State diagram



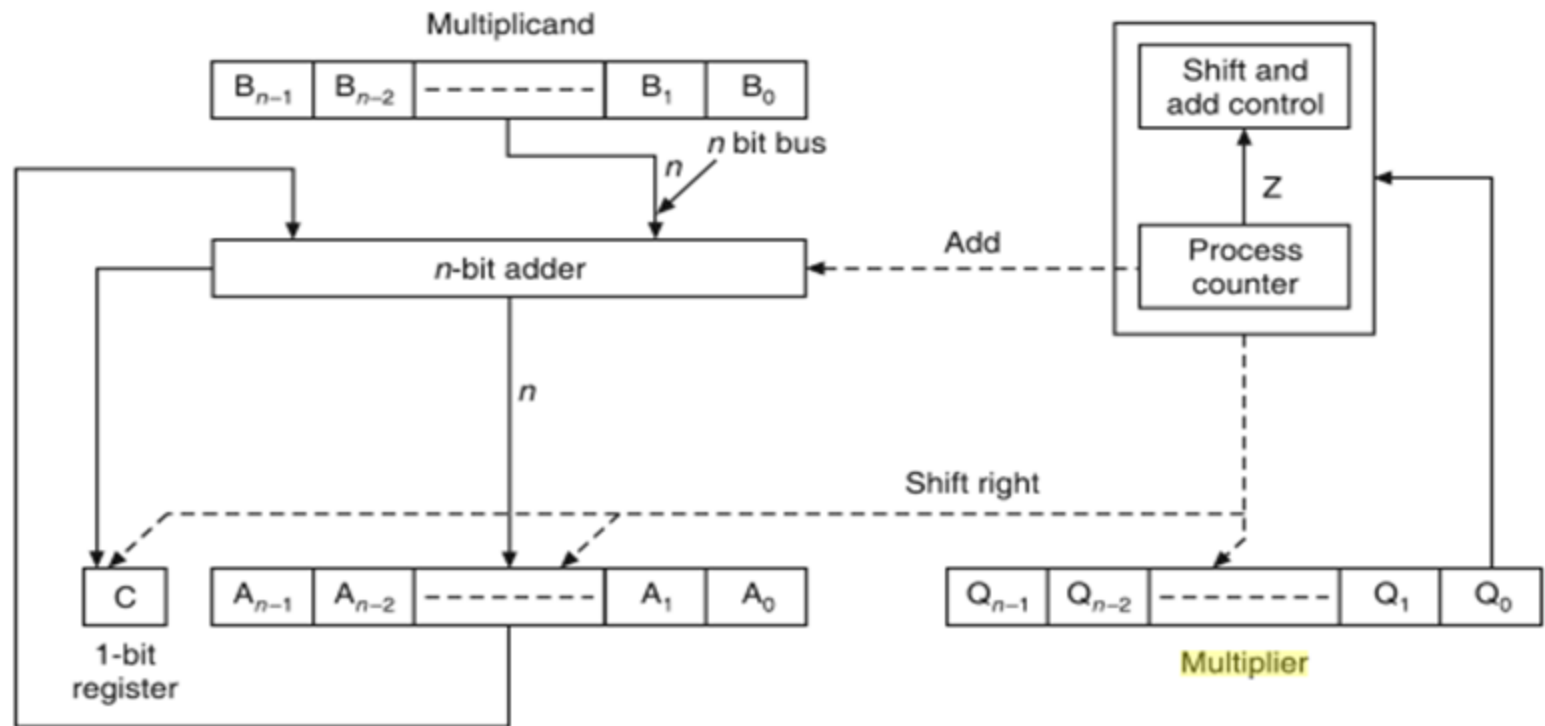
(b) ASM chart

State diagram and ASM chart for mod-6 counter.

BINARY MULTIPLIER

1 1 0 1	←	13_{10} ... Multiplicand
1 0 1 0	←	10_{10} ... Multiplier
0 0 0 0	←	Partial product 1
1 1 0 1	←	Partial product 2
0 0 0 0	←	Partial product 3
1 1 0 1	←	Partial product 4
1 0 0 0 0 0 1 0	←	130_{10} ... Product

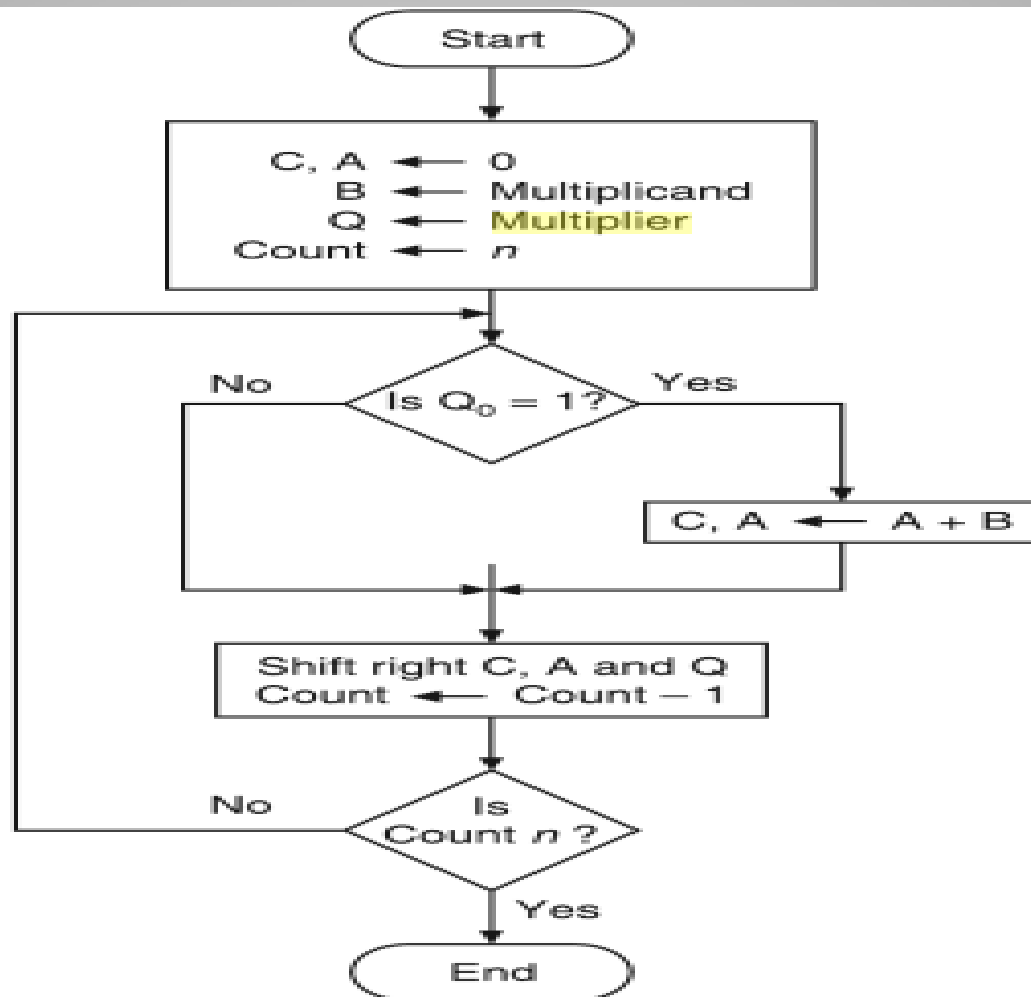
Data path subsystem for binary multiplier



Datapath subsystem for binary multiplier.

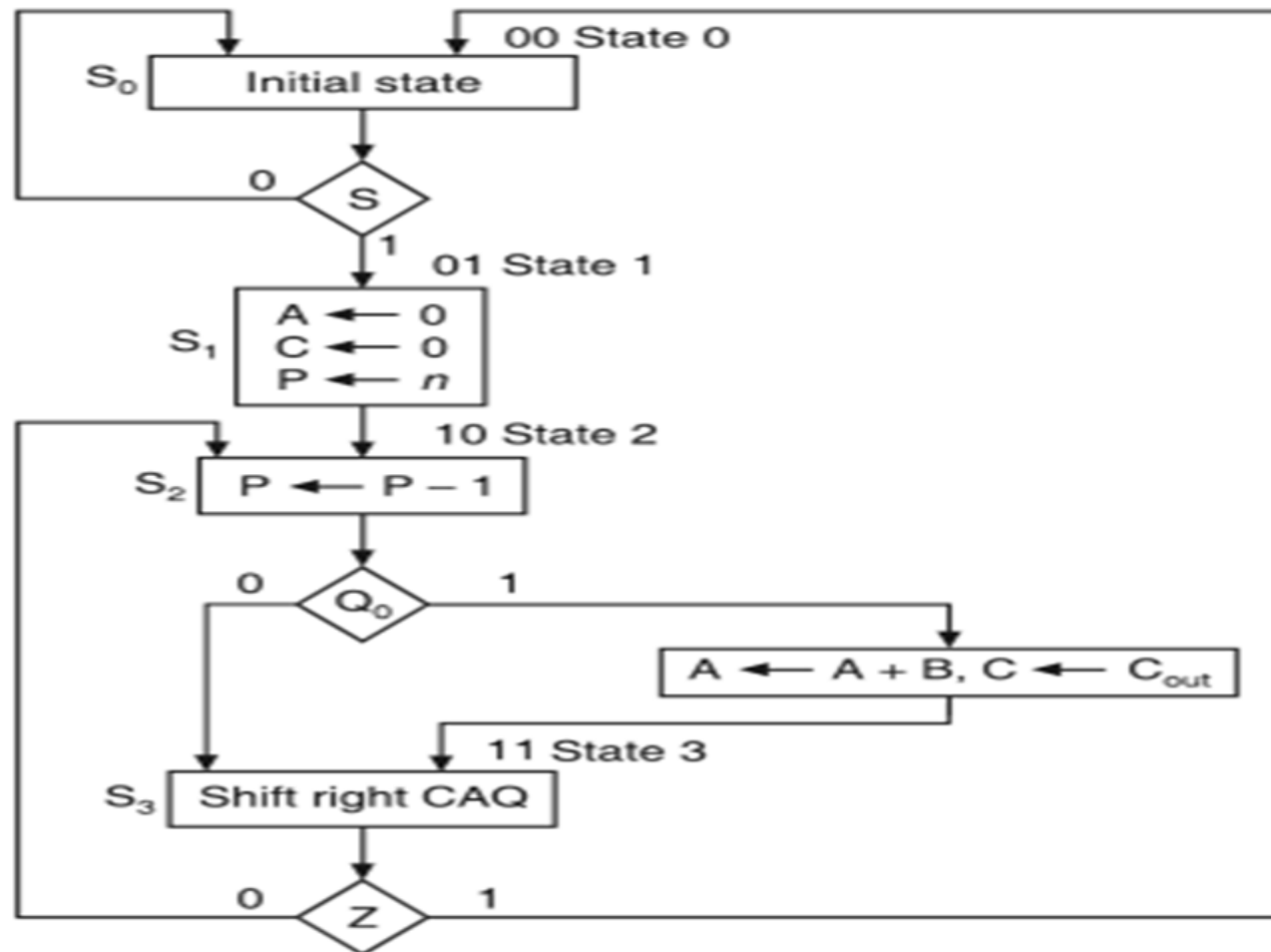
Multiplication Operation Steps

1. Bit 0 of multiplier operand (Q_0 of Q register) is checked.
2. If bit 0 (Q_0) is one then multiplicand and partial product are added and all bits of C, A and Q registers are shifted to the right one bit, so that the C bit goes into A_{n-1} , A_0 goes into Q_{n-1} , and Q_0 is lost. If bit 0 (Q_0) is 0, then no addition is performed, only shift operation is carried out.
3. Steps 1 and 2 are repeated n times to get the desired result in the A and Q registers.



B	C	A	Q	Components	Count P
1 1 0 1	0	0 0 0 0	1 0 1 0	B ← Multiplicand Q ← Multiplier A ← 0, C ← 0, P ← n	100 (4)
1 1 0 1	0	0 0 0 0	1 0 1 0	P ← P - 1 Q ₀ = 0	011 (3)
	0	0 0 0 0	0 1 0 1	C A Q shifted right	
1 1 0 1	0	1 1 0 1	0 1 0 1	P ← P - 1 Q ₀ = 1, A ← A + B	010 (2)
	0	0 1 1 0	1 0 1 0	C A Q shifted right	
1 1 0 1	0	0 1 1 0	1 0 1 0	P ← P - 1 Q ₀ = 0,	001 (1)
	0	0 0 1 1	0 1 0 1	C A Q shifted right	
1 1 0 1	1	0 0 0 0	0 1 0 1	P ← P - 1 Q ₀ = 1, A ← A + B	000 (0)
	0	1 0 0 0	0 0 1 0	C A Q shifted right	

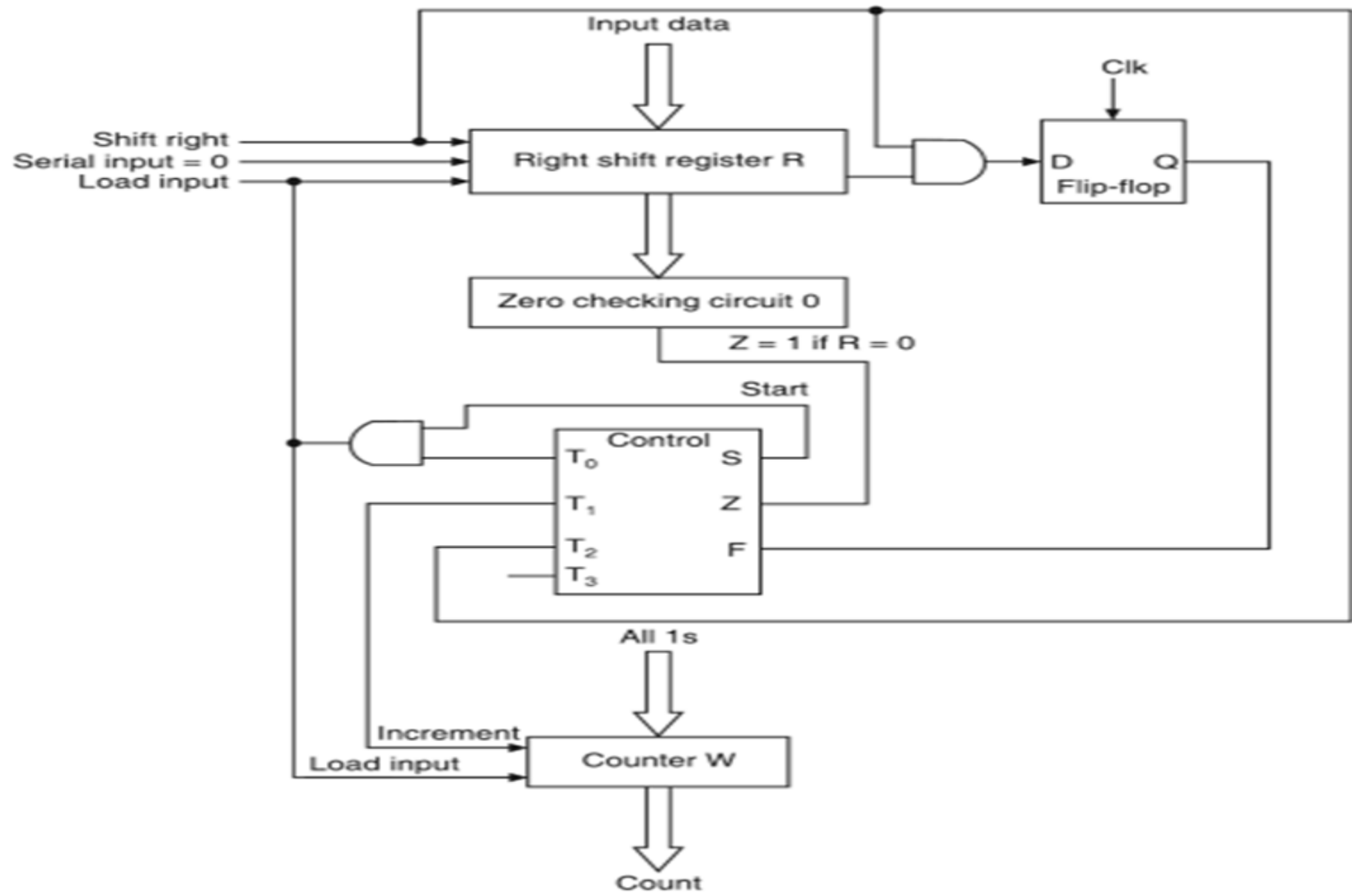
Flow chart for multiplication in a computer.



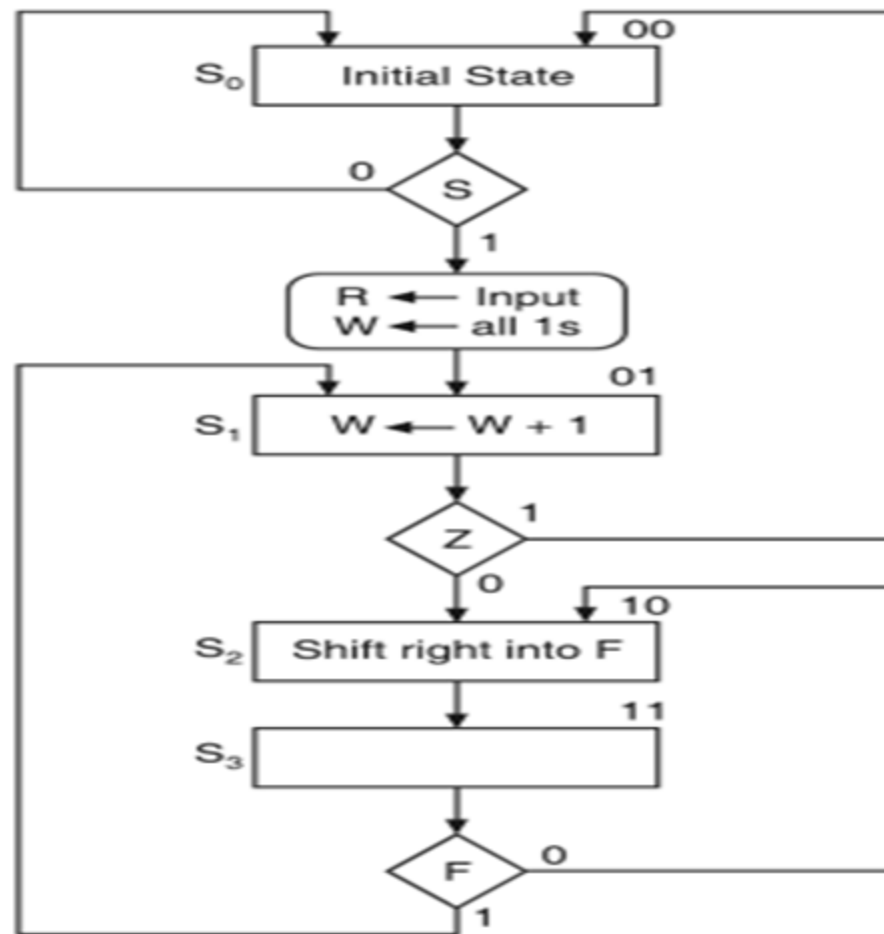
ASM chart for a **binary multiplier**.

ASM FOR WEIGHING MACHINE

In the algorithm for tabular minimization of Boolean expressions, we have to arrange the minterms in the ascending order of their weights. This is only one of the many situations when we have to examine the 1s of a given binary word. The weight of a binary number is defined as the number of 1s present in its binary representation.



Datapath subsystem for weighing machine.



ASM chart for weighing machine.

State S_0 : Initially the weighing machine is in state S_0 . The weighing process starts when start (S) signal becomes 1. While in state S_0 , if S is 1, the clock pulse causes three jobs to be done simultaneously:

1. Binary number is loaded into register R.
2. W register is set to all 1s.
3. The machine is transferred to state S_1 .

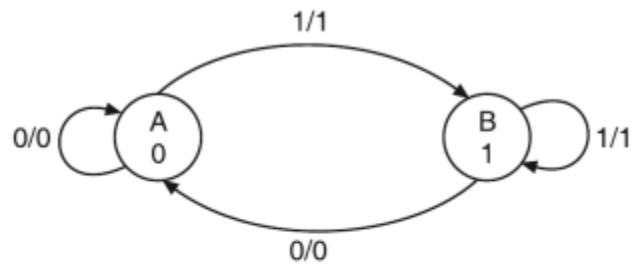
State S_1 : While in state S_1 , the clock pulse causes two jobs to be done simultaneously:

1. Counter W is incremented by 1 (in the first round, all 1s become all 0s).
2. If Z is 0, the machine goes to the state S_2 ; if Z is 1, the machine goes to state S_0 .

State S_2 : In this state, register R is shifted right by 1 bit so that LSB goes into F and MSB is loaded with 0.

State S_3 : In this state, the value of F is checked. If it is 0, the machine is transferred to the state S_2 , otherwise the machine is transferred to state S_1 . Thus, when $F = 1$, W is incremented.

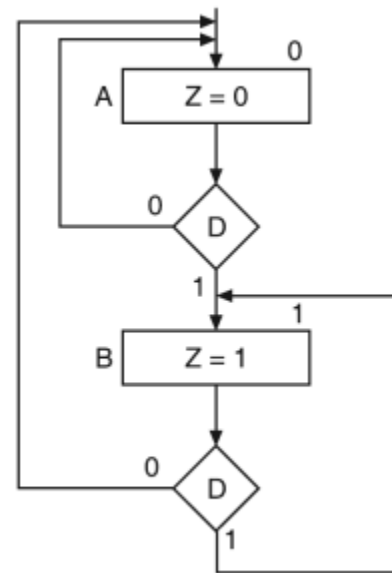
All the operations occur in coincidence with the clock pulse while in the corresponding state. Also notice that the register R should eventually contain all 0s when the last 1 is shifted into it.



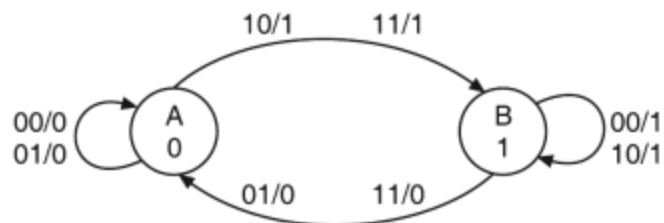
(a) State diagram

PS	NS, O/P	
	Input D	
	D = 0	D = 1
A	A, 0	B, 1
B	A, 0	B, 1

(b) State table



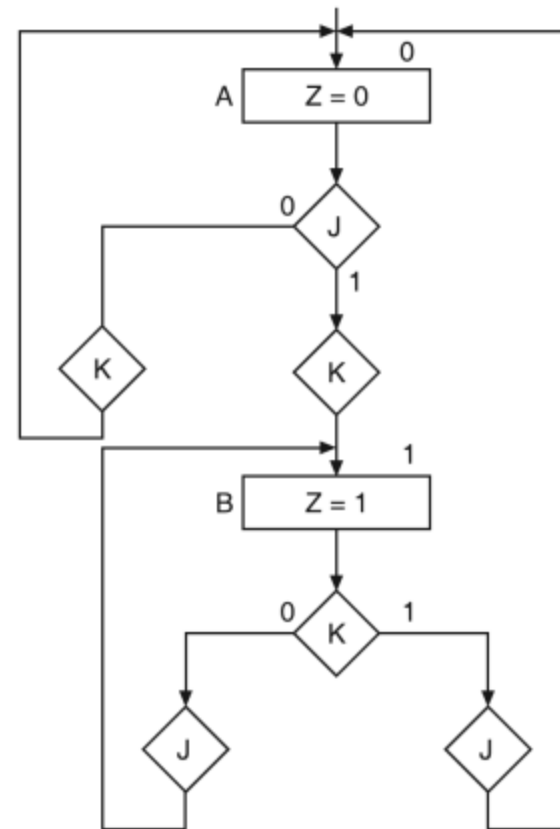
(c) ASM chart



(a) State diagram

PS	NS, O/P Input J-K			
	00	01	10	11
A	A, 0	A, 0	B, 1	B, 1
B	B, 1	A, 0	B, 1	A, 0

(b) State table



(c) ASM chart

Thank you