
Logic and Computer Design Fundamentals

Chapter 1 – Digital Systems and Information

Charles Kime & Thomas Kaminski

© 2008 Pearson Education, Inc.

(Hyperlinks are active in View Show mode)

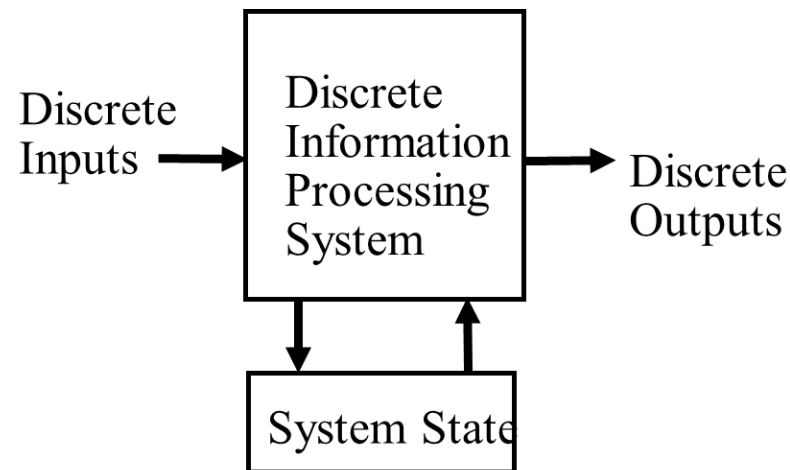
Updated thoroughly by Dr. Waleed Dweik

Overview

- Digital Systems, Computers, and Beyond
- Information Representation
- Number Systems [binary, octal and hexadecimal]
- Base Conversion
- Decimal Codes [BCD (binary coded decimal)]
- Alphanumeric Codes
- Parity Bit
- Gray Codes

DIGITAL & COMPUTER SYSTEMS - Digital System

- Takes a set of discrete information inputs and discrete internal information (system state) and generates a set of discrete information outputs.
- Digits (Latin word for fingers) : Discrete numeric elements
- Logic : Circuits that operate on a set of two elements with values 0 (False), 1 (True)
- **Computers are digital logic circuits**



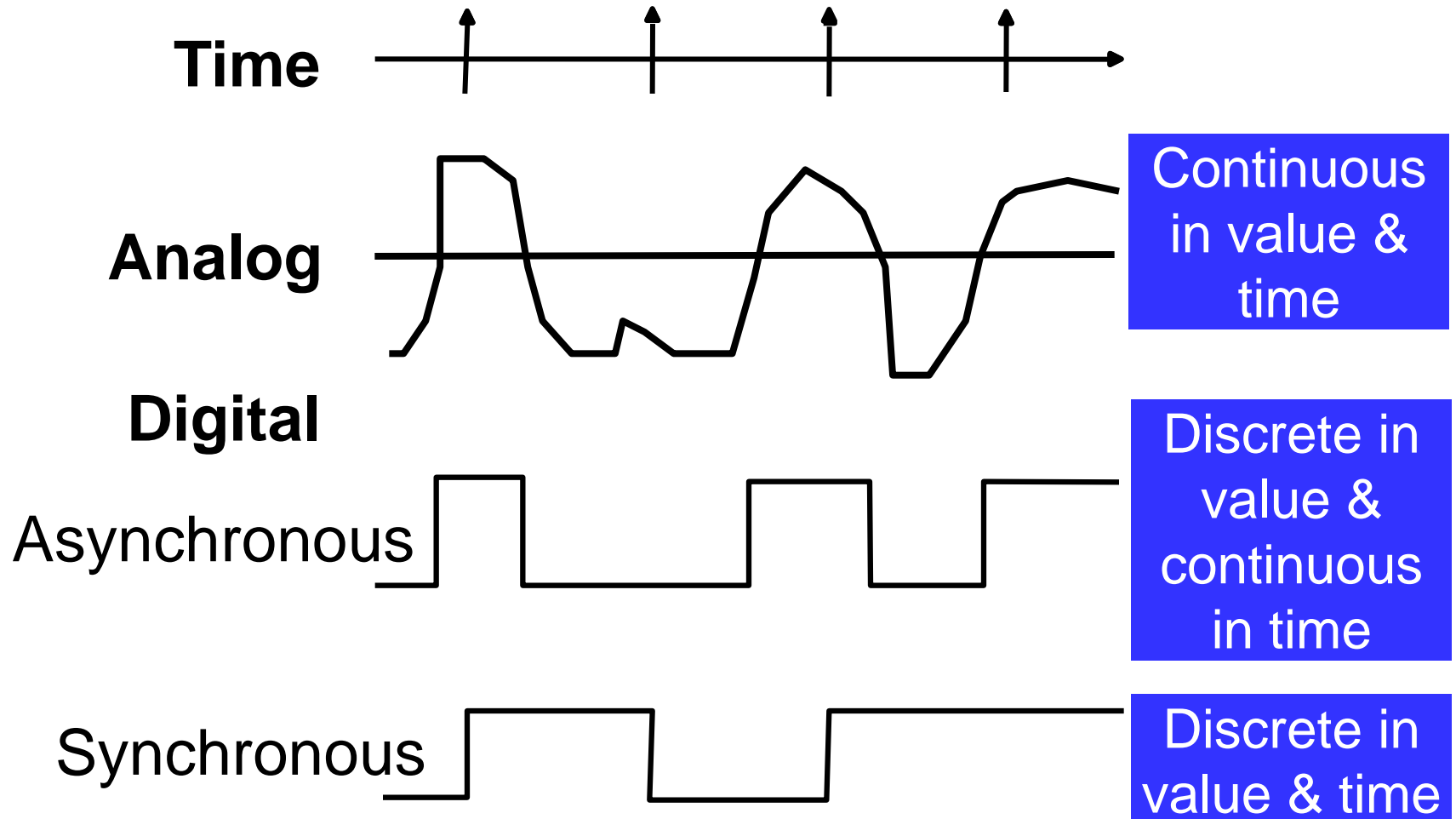
Types of Digital Systems

- No state present
 - Combinational Logic System
 - Output = Function(Input)
- State present
 - Synchronous Sequential System: State updated at discrete times
 - Asynchronous Sequential System: State updated at any time
 - State = Function (State, Input)
 - Output = Function (State) or Function (State, Input)

↙
Moore

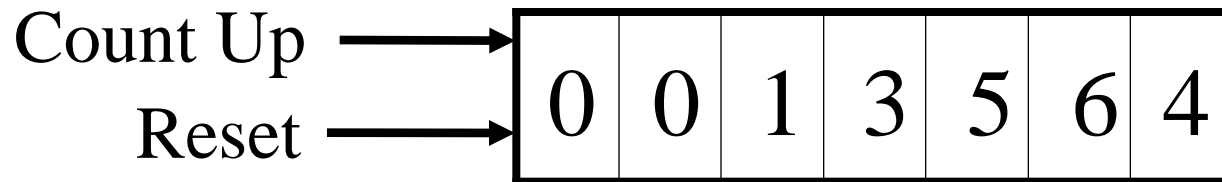
↘
Mealy

Signal Examples Over Time



Digital System Example

A Digital Counter (e. g., odometer):



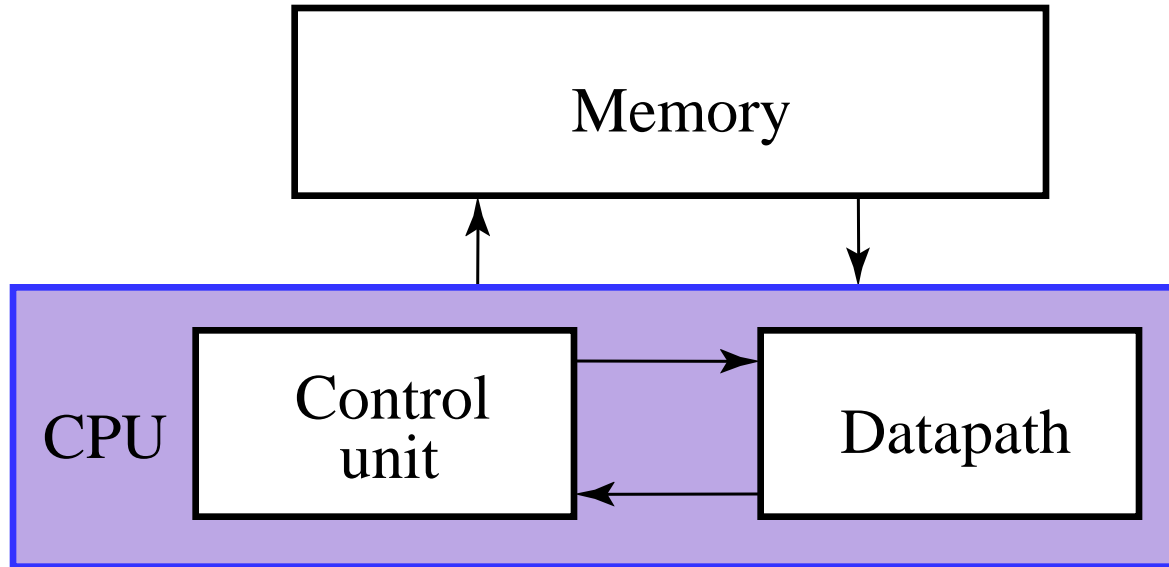
Inputs: Count Up, Reset

Outputs: Visual Display

State: "Value" of stored digits

Synchronous or Asynchronous?

Digital Computer Example



Inputs: keyboard,
mouse, wireless,
microphone

Outputs: LCD
screen, wireless,
speakers

**Synchronous or
Asynchronous?**

And Beyond – Embedded Systems

- Computers as integral parts of other products
- Examples of embedded computers
 - Microcomputers
 - Microcontrollers
 - Digital signal processors
- Examples of embedded systems applications

Cell phones	Dishwashers
Automobiles	Flat Panel TVs
Video games	Global Positioning Systems
Copiers	

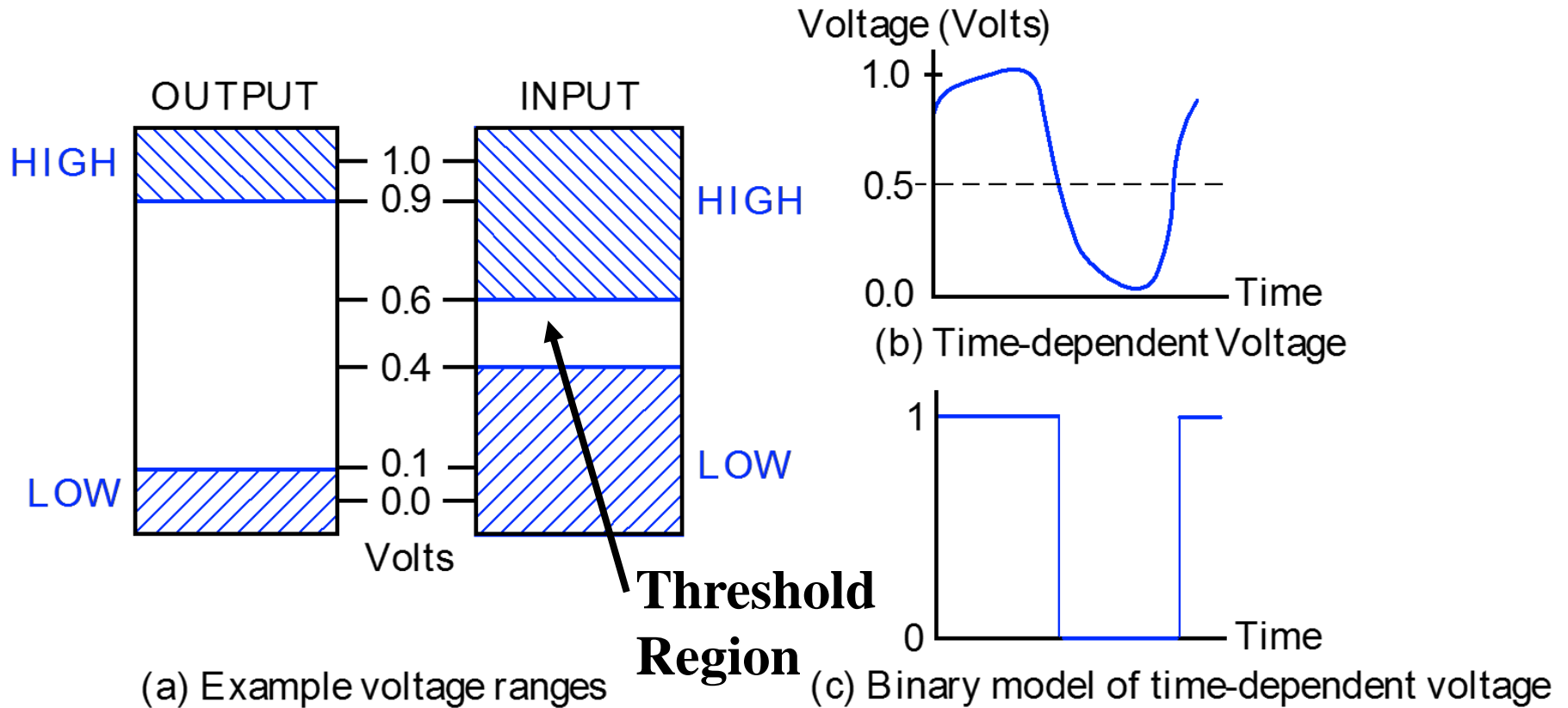
INFORMATION REPRESENTATION - Signals

- Information variables represented by physical quantities.
- For digital systems, the variables take on discrete values.
- Two level, or binary values are the most prevalent values in digital systems.
 - Binary systems have higher immunity to noise.
- Binary values are represented abstractly by:
 - digits 0 and 1
 - words (symbols) False (F) and True (T)
 - words (symbols) Low (L) and High (H)
 - and words On and Off.
- Binary values are represented by values or ranges of values of physical quantities.

Binary Values: Other Physical Quantities

- What are other physical quantities represent 0 and 1?
 - CPU → Voltage
 - Disk → Magnetic Field Direction
 - CD → Surface Pits/Light
 - Dynamic RAM → Electrical Charge stored in capacitors

Signal Example – Physical Quantity: Voltage



NUMBER SYSTEMS – Representation

- Positive radix, positional number systems
- A number with radix r is represented by a string of digits:
 $A_{n-1}A_{n-2} \dots A_1A_0 \cdot A_{-1}A_{-2} \dots A_{-m+1}A_{-m}$

in which $0 \leq A_i < r$ and \cdot is the *radix point*

- i represents the position of the coefficient
- r^i represents the weight by which the coefficient is multiplied
- A_{n-1} is the most significant digit (MSD) and A_{-m} is the least significant digit (LSD)
- The string of digits represents the power series:

$$(Number)_r = \left(\sum_{i=0}^{n-1} A_i r^i \right) + \left(\sum_{j=-m}^{-1} A_j r^j \right)$$

Integer Portion

Fraction Portion

Number Systems – Examples

	General	Decimal	Binary
Radix (Base)	r	10	2
Digits	0 => r - 1	0 => 9	0 => 1
Powers of Radix	0	r^0	1
	1	r^1	2
	2	r^2	4
	3	r^3	8
	4	r^4	16
	5	r^5	32
	-1	r^{-1}	0.5
	-2	r^{-2}	0.25
	-3	r^{-3}	0.125
	-4	r^{-4}	0.0625
-5	r^{-5}	0.03125	

Example

- $(403)_5 = 4 \times 5^2 + 0 \times 5^1 + 3 \times 5^0 = (103)_{10}$

- $(103)_{10} = 1 \times 10^2 + 0 \times 10^1 + 3 \times 10^0 = 103$

BASE CONVERSION - Positive Powers of 2

- Useful for Base Conversion

Exponent	Value
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128
8	256
9	512
10	1024

Exponent	Value
11	2,048
12	4,096
13	8,192
14	16,384
15	32,768
16	65,536
17	131,072
18	262,144
19	524,288
20	1,048,576
21	2,097,152

Special Powers of 2

- 2^{10} (1024) is Kilo, denoted "K"
- 2^{20} (1,048,576) is Mega, denoted "M"
- 2^{30} (1,073, 741,824) is Giga, denoted "G"
- 2^{40} (1,099,511,627,776) is Tera, denoted "T"

Commonly Occurring Bases

Name	Radix	Digits
Binary	2	0,1
Octal	8	0,1,2,3,4,5,6,7
Decimal	10	0,1,2,3,4,5,6,7,8,9
Hexadecimal	16	0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F

- The six letters A, B, C, D, E, and F represent the digits for values 10, 11, 12, 13, 14, 15 (given in decimal), respectively, in hexadecimal. Alternatively, a, b, c, d, e, f can be used.

Binary System

- $r = 2$
- Digits = $\{0, 1\}$
- Every binary digit is called a bit
- When a bit is equal to zero, it does not contribute to the value of the number
- Example:
 - $(10011.101)_2 = (1 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0) + (1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3})$
 - $(10011.101)_2 = (16 + 2 + 1) + \left(\frac{1}{2} + \frac{1}{8}\right) = (19.625)_{10}$

Octal System

- $r = 8$
- Digits = $\{0, 1, 2, 3, 4, 5, 6, 7\}$
- Every digit is represented by 3-bits \rightarrow More compact than binary
- Example:
 - $(127.4)_8 = (1 \times 8^2 + 2 \times 8^1 + 7 \times 8^0) + (4 \times 8^{-1})$
 - $(127.4)_8 = (64 + 16 + 7) + \left(\frac{1}{2}\right) = (87.5)_{10}$

Hexadecimal System

- $r = 16$
- Digits = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F}
- Every digit is represented by 4-bits
- Example:
 - $(B65F)_{16} = (11 \times 16^3 + 6 \times 16^2 + 5 \times 16^1 + 15 \times 16^0)$
 - $(B65F)_{16} = (46687)_{10}$

Numbers in Different Bases

- **Good idea to memorize!**

Decimal (Base 10)	Binary (Base 2)	Octal (Base 8)	Hexadecimal (Base 16)
00	0000	00	00
01	0001	01	01
02	0010	02	02
03	0011	03	03
04	00100	04	04
05	00101	05	05
06	00110	06	06
07	00111	07	07
08	01000	10	08
09	01001	11	09
10	01010	12	0A
11	01011	13	0B
12	01100	14	0C
13	01101	15	0D
14	01110	16	0E
15	01111	17	0F
16	10000	20	10

Converting from any Base (r) to Decimal

$$(Number)_r = \left(\sum_{i=0}^{n-1} A_i r^i \right) + \left(\sum_{j=-m}^{-1} A_j r^j \right)$$

Integer Portion **Fraction Portion**

- **Example: Convert 11010_2 to N_{10} :**

Conversion from Decimal to Base (r)

- Convert the Integer Part
- Convert the Fraction Part
- Join the two results with a radix point

Conversion Details

■ ***To Convert the Integral Part:***

- Repeatedly divide the number by the new radix and save the remainders until the quotient is zero
- The digits for the new radix are the remainders in reverse order of their computation
- If the new radix is > 10 , then convert all remainders > 10 to digits A, B, ...

■ ***To Convert the Fractional Part:***

- Repeatedly multiply the fraction by the new radix and save the integer digits of the results until the fraction is zero or you reached the required number of fractional digits
- The digits for the new radix are the integer digits in order of their computation
- If the new radix is > 10 , then convert all integers > 10 to digits A, B, ...

Example: Convert 46.6875_{10} To Base 2

- Convert 46 to Base 2:

$$(46)_{10} = (101110)_2$$

Division	Quotient	Remainder	
46/2	23	0	↑ LSD MSD
23/2	11	1	
11/2	5	1	
5/2	2	1	
2/2	1	0	
1/2	0	1	

- Convert 0.6875 to Base 2:

$$(0.6875)_{10} = (0.1011)_2$$

Multiplication	Answer	
0.6875*2	1.375	↓ MSD LSD
0.375*2	0.75	
0.75*2	1.5	
0.5*2	1.0	

- Join the results together with the radix point:

$$(46.6875)_{10} = (101110.1011)_2$$

Example: Convert 153.513_{10} To Base 8

- Convert 153 to Base 8:

$$(153)_{10} = (231)_8$$

Division	Quotient	Remainder	
153/8	19	1	↑ LSD
19/8	2	3	
2/8	0	2	

- Convert 0.513 to Base 8: (*Up to 3 digits*)

- Truncate:

$$(0.513)_{10} = (0.406)_8$$

- Round:

$$(0.513)_{10} = (0.407)_8$$


Multiplication	Answer	
0.513*8	4.104	↓ MSD
0.104*8	0.832	
0.832*8	6.656	
0.656*8	5.248	

- Join the results together with the radix point:

$$(153.513)_{10} = (231.407)_8$$

Example: Convert 423_{10} To Base 16

Division	Quotient	Remainder	
423/16	26	7	LSD
26/16	1	10	
1/16	0	1	MSD



$$(423)_{10} = (1A7)_{16}$$

Converting Decimal to Binary: Alternative Method

- Subtract the largest power of 2 that gives a positive remainder and record the power
- Repeat, subtracting from the prior remainder and recording the power, until the remainder is zero
- Place 1's in the positions in the binary result corresponding to the powers recorded; in all other positions place 0's

Example: Convert 46.6875_{10} To Base 2 Using Alternative Method

- Convert 46 to Base 2:

$$(46)_{10} = (101110)_2$$

Subtract	Remainder	Power
46-32	14	5
14-8	6	3
6-4	2	2
2-2	0	1

- Convert 0.6875 to Base 2:

$$(0.6875)_{10} = (0.1011)_2$$

Subtract	Remainder	Power
0.6875-0.5	0.1875	-1
0.1875-0.125	0.0625	-3
0.0625-0.0625	0	-4

- Join the results together with the radix point:

$$(46.6875)_{10} = (101110.1011)_2$$

- Easier way to do it:

Power	6	5	4	3	2	1	0	.	-1	-2	-3	-4
	0	1	0	1	1	1	0	.	1	0	1	1

Additional Issue - Fractional Part

- Note that in this conversion, the fractional part can become 0 as a result of the repeated multiplications
- In general, it may take many bits to get this to happen or it may never happen
- Example Problem: Convert 0.65_{10} to N_2
 - $0.65 = 0.1010011001001 \dots$
 - The fractional part begins repeating every 4 steps yielding repeating 1001 forever!
- **Solution: Specify number of bits to right of radix point and round or truncate to this number**

Checking the Conversion

- To convert back, sum the digits times their respective powers of r
- From the prior conversion of 46.6875_{10}

$$\begin{aligned}101110_2 &= 1 \cdot 32 + 0 \cdot 16 + 1 \cdot 8 + 1 \cdot 4 + 1 \cdot 2 + 0 \cdot 1 \\ &= 32 + 8 + 4 + 2 \\ &= 46\end{aligned}$$

$$\begin{aligned}0.1011_2 &= 1/2 + 1/8 + 1/16 \\ &= 0.5000 + 0.1250 + 0.0625 \\ &= 0.6875\end{aligned}$$

Octal (Hexadecimal) to Binary and Back: Method1

- Octal (Hexadecimal) to Binary:
 1. Convert octal (hexadecimal) to decimal (Slide 23)
 2. Convert decimal to binary (Slide 24 or Slide 29)

- Binary to Octal (Hexadecimal):
 1. Convert binary to decimal (Slide 23)
 2. Convert decimal to octal (hexadecimal) (Slide 24)

Octal (Hexadecimal) to Binary and Back: Method2 (Easier)

- Octal (Hexadecimal) to Binary:
 - **Restate** the octal (hexadecimal) as three (four) binary digits starting at the radix point and going both ways
- Binary to Octal (Hexadecimal):
 - **Group** the binary digits into three (four) bit groups starting at the radix point and going both ways, padding with zeros as needed
 - Convert each group of three (four) bits to an octal (hexadecimal) digit

Octal	0	1	2	3	4	5	6	7
Binary	000	001	010	011	100	101	110	111

Hexadecimal	0	1	2	3	4	5	6	7
Binary	0000	0001	0010	0011	0100	0101	0110	0111
Hexadecimal	8	9	A	B	C	D	E	F
Binary	1000	1001	1010	1011	1100	1101	1110	1111

Examples

- $(673.12)_8 = (110\ 111\ 011 . 001\ 010)_2$

- $(3A6.C)_{16} = (0011\ 1010\ 0110 . 1100)_2$

- $(10110001101011.1111000001)_2 = (?)_8$

$$(10/110/001/101/011.111/100/000/1)_2 = (26153.7404)_8$$

- $(10110001101011.1111000001)_2 = (?)_{16}$

$$(10/1100/0110/1011.1111/0000/01)_2 = (2C6B.F04)_{16}$$

Octal to Hexadecimal via Binary

- Convert octal to binary
- Use groups of four bits and convert to hexadecimal digits
- Example: Octal to Binary to Hexadecimal

$$\begin{array}{c} (635.177)_8 \\ \downarrow \\ (110\ 011\ 101\ .\ 001\ 111\ 111)_2 \\ \downarrow \\ (1/1001/1101\ .\ 0011/1111/1)_2 \\ \downarrow \\ (19D.3F8)_{16} \end{array}$$

One last Conversion Example

- Given that $(365)_r = (194)_{10}$, compute the value of r ?

$$3 \times r^2 + 6 \times r^1 + 5 \times r^0 = 194$$

$$3r^2 + 6r + 5 = 194$$

$$3r^2 + 6r - 189 = 0$$

$$r^2 + 2r - 63 = 0$$

$$(r - 7)(r + 9) = 0$$

$$r = 7$$

Binary Numbers and Binary Coding

- Flexibility of representation
 - Within constraints below, can assign any binary combination (called a code word) to any data as long as data is uniquely encoded

- Information Types
 - Numeric
 - Must represent range of data needed
 - Very desirable to represent data such that simple, straightforward computation for common arithmetic operations permitted
 - Tight relation to binary numbers

 - Non-numeric
 - Greater flexibility since arithmetic operations not applied
 - Not tied to binary numbers

Non-numeric Binary Codes

- Given n binary digits (called bits), a binary code is a mapping from a set of represented elements to a subset of the 2^n binary numbers.
- Example: A binary code for the seven colors of the rainbow
- Code 100 is not used

Color	Binary Number
Red	000
Orange	001
Yellow	010
Green	011
Blue	101
Indigo	110
Violet	111

Number of Bits Required

- Given M elements to be represented by a binary code, the minimum number of bits, n , needed, satisfies the following relationships:

$$2^n \geq M > 2^{n-1}$$

$n = \lceil \log_2 M \rceil$, where $\lceil x \rceil$ is called the *ceiling function*, is the integer greater than or equal to x .

- Example: How many bits are required to represent decimal digits with a binary code?

$$M = 10$$

$$n = \lceil \log_2 10 \rceil = \lceil 3.33 \rceil = 4$$

Number of Elements Represented

- Given n digits in radix r , there are r^n distinct elements that can be represented.
- But, you can represent m elements, $m \leq r^n$

Examples:

- You can represent 4 elements in radix $r = 2$ with $n = 2$ digits: (00, 01, 10, 11).
- You can represent 4 elements in radix $r = 2$ with $n = 4$ digits: (0001, 0010, 0100, 1000).
- This second code is called a "one hot" code.

DECIMAL CODES - Binary Codes for Decimal Digits

- There are over 8,000 ways that you can chose 10 elements from the 16 binary numbers of 4 bits. A few are useful:

Decimal	8, 4, 2, 1	Excess 3	8, 4, -2, -1	Gray
0	0000	0011	0000	0000
1	0001	0100	0111	0001
2	0010	0101	0110	0011
3	0011	0110	0101	0010
4	0100	0111	0100	0110
5	0101	1000	1011	1110
6	0110	1001	1010	1010
7	0111	1010	1001	1011
8	1000	1011	1000	1001
9	1001	1100	1111	1000

Binary Coded Decimal (BCD)

- Numeric code
- The BCD code is the 8, 4, 2, 1 code
- 8, 4, 2, and 1 are weights → BCD is a *weighted* code
- This code is the simplest, most intuitive binary code for decimal digits and uses the same powers of 2 as a binary number, ***but only encodes the first ten values from 0 to 9***
- Example: $1001 (9) = 1000 (8) + 0001 (1)$
- How many “invalid” code words are there?
 - Answer: 6
- What are the “invalid” code words?
 - Answer: 1010, 1011, 1100, 1101, 1110, 1111

Warning: Conversion or Coding?

- Do NOT mix up *conversion* of a decimal number to a binary number with *coding* a decimal number with a BINARY CODE.
- $13_{10} = 1101_2$ (This is conversion)
- $13 \Leftrightarrow 0001|0011$ (This is coding)

Excess 3 Code and 8, 4, -2, -1 Code

- What interesting property is common to these two codes?
 - Answer: Both codes have the property that the codes for 0 and 9, 1 and 8, etc. can be obtained from each other by replacing the 0's with the 1's and vice-versa. Such a code is sometimes called a *complement code*.

Decimal	Excess 3	8, 4, -2, -1
0	0011	0000
1	0100	0111
2	0101	0110
3	0110	0101
4	0111	0100
5	1000	1011
6	1001	1010
7	1010	1001
8	1011	1000
9	1100	1111

ALPHANUMERIC CODES - ASCII Character Codes

- Non-numeric code
- ASCII stands for American Standard Code for Information Interchange (Refer to Table 1-5 in the text)
- This code is a popular code used to represent information sent as character-based data. It uses 7-bits (i.e. 128 characters) to represent:
 - 95 Graphic printing characters
 - 33 Non-printing characters

ASCII Code Table

Least Significant
ASCII Code Chart

Most Significant

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2		!	"	#	\$	%	&	'	()	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{		}	~	DEL

ASCII Character Codes

- Graphic printing characters
 - 26 upper case letters (A-Z)
 - 26 lower case letters (a-z)
 - 10 numerals (0-9)
 - 33 special characters (e.g. %, @, \$)
- Non-printing characters
 - Format effectors: used for text format (e.g. BS = Backspace, CR = carriage return)
 - Information separators: used to separate the data into paragraphs and pages (e.g. RS = record separator, FS = file separator)
 - Communication control characters (e.g. STX and ETX start and end text areas).

ASCII Properties

- ASCII has some interesting properties:
 - Digits 0 to 9 span Hexadecimal values 30_{16} to 39_{16}
 - Upper case A-Z span 41_{16} to $5A_{16}$
 - Lower case a-z span 61_{16} to $7A_{16}$
 - Lower to upper case translation (and vice versa) occurs by flipping bit 6

UNICODE

- UNICODE extends ASCII to 65,536 universal characters codes:
 - Non-numeric
 - For encoding characters in world languages
 - Available in many modern applications
 - 2 byte (16-bit) code words

PARITY BIT Error-Detection Codes

- Non-numeric
- **Redundancy** (e.g. extra information), in the form of extra bits, can be incorporated into binary code words to detect and correct errors
- A simple form of redundancy is **parity**, an extra bit appended onto the code word to make the number of 1's odd or even. Parity can detect all single-bit errors and some multiple-bit errors
- A code word has **even parity** if the number of 1's in the code word is even
- A code word has **odd parity** if the number of 1's in the code word is odd

4-Bit Parity Code Example

- Fill in the even and odd parity bits:

Even Parity Message	Odd Parity Message
000 <u>0</u>	000 <u>1</u>
001 <u>1</u>	001 <u>0</u>
010 <u>1</u>	010 <u>0</u>
011 <u>0</u>	011 <u>1</u>
100 <u>1</u>	100 <u>0</u>
101 <u>0</u>	101 <u>1</u>
110 <u>0</u>	110 <u>1</u>
111 <u>1</u>	111 <u>0</u>

- The code word "1111" has even parity and the code word "1110" has odd parity. Both can be used to represent the same 3-bit data

Logic and Computer Design Fundamentals

Chapter 2 – Combinational Logic Circuits

Part 1 – Gate Circuits and Boolean Equations

Charles Kime & Thomas Kaminski

© 2008 Pearson Education, Inc.

(Hyperlinks are active in View Show mode)

Updated by Dr. Waleed Dweik

Combinational Logic Circuits

- Digital (logic) circuits are hardware components that manipulate binary information.
- Integrated circuits: transistors and interconnections.
 - Basic circuits is referred to as *logic gates*
 - The outputs of gates are applied to the inputs of other gates to form a digital circuit
- Combinational? Later...

Overview

- **Part 1 – Gate Circuits and Boolean Equations**
 - Binary Logic and Gates
 - Boolean Algebra
 - Standard Forms

- **Part 2 – Circuit Optimization**
 - Two-Level Optimization
 - Map Manipulation
 - Practical Optimization (Espresso)
 - Multi-Level Circuit Optimization

- **Part 3 – Additional Gates and Circuits**
 - Other Gate Types
 - Exclusive-OR Operator and Gates
 - High-Impedance Outputs

Binary Logic and Gates

- **Binary variables** take on one of two values
- **Logical operators** operate on binary values and binary variables
- Basic logical operators are the logic functions **AND**, **OR** and **NOT**
- **Logic gates** implement logic functions
- **Boolean Algebra**: a useful mathematical system for specifying and transforming logic functions
- We study Boolean algebra as a foundation for designing and analyzing digital systems!

Binary Variables

- Recall that the two binary values have different names:
 - True/False
 - On/Off
 - Yes/No
 - 1/0
- We use 1 and 0 to denote the two values
- Variable identifier examples:
 - A, B, y, z, or X_1 for now
 - RESET, START_IT, or ADD1 later

Logical Operations

- The three basic logical operations are:
 - AND
 - OR
 - NOT
- AND is denoted by a dot (\cdot) or (\wedge)
- OR is denoted by a plus ($+$) or (\vee)
- NOT is denoted by an over-bar ($\bar{\quad}$), a single quote mark ($'$) after, or (\sim) before the variable

Notation Examples

- Examples:
 - $Z = X \cdot Y = XY = X \wedge Y$: is read “Z is equal to X AND Y”
 - $Z = 1$ if and only if $X = 1$ and $Y = 1$; otherwise, $Z = 0$
 - $Z = X + Y = X \vee Y$: is read “Z is equal to X OR Y”
 - $Z = 1$ if (only $X = 1$) or if (only $Y = 1$) or if ($X = 1$ and $Y = 1$)
 - $Z = \bar{X} = X' = \sim X$: is read “Z is equal to NOT X”
 - $Z = 1$ if $X = 0$; otherwise, $Z = 0$
- Notice the difference between arithmetic addition and logical OR:
 - The statement:
 $1 + 1 = 2$ (read “one plus one equals two”)
is not the same as
 $1 + 1 = 1$ (read “1 or 1 equals 1”)

Operator Definitions

- Operations are defined on the values "0" and "1" for each operator:

AND
$0 \cdot 0 = 0$
$0 \cdot 1 = 0$
$1 \cdot 0 = 0$
$1 \cdot 1 = 1$

OR
$0 + 0 = 0$
$0 + 1 = 1$
$1 + 0 = 1$
$1 + 1 = 1$

NOT
$\bar{0} = 1$
$\bar{1} = 0$

Truth Tables

- **Truth table** - a tabular listing of the values of a function for all possible combinations of values on its arguments
- Example: Truth tables for the basic logic operations:

AND		
Inputs		Output
X	Y	$Z = X \cdot Y$
0	0	0
0	1	0
1	0	0
1	1	1

OR		
Inputs		Output
X	Y	$Z = X + Y$
0	0	0
0	1	1
1	0	1
1	1	1

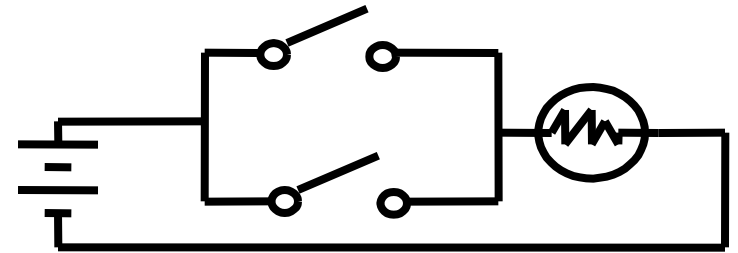
NOT	
Inputs	Output
X	$Z = \bar{X}$
0	1
1	0

Logic Function Implementation

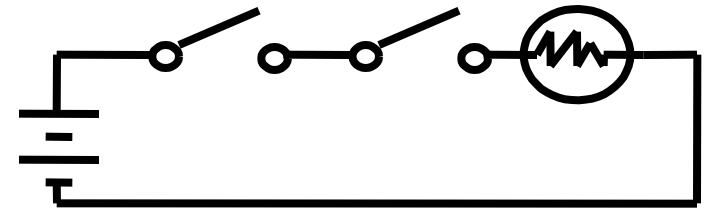
■ Using Switches

- For inputs:
 - logic 1 is switch closed
 - logic 0 is switch open
- For outputs:
 - logic 1 is light on
 - logic 0 is light off
- NOT uses a switch such that:
 - logic 1 is switch open
 - logic 0 is switch closed

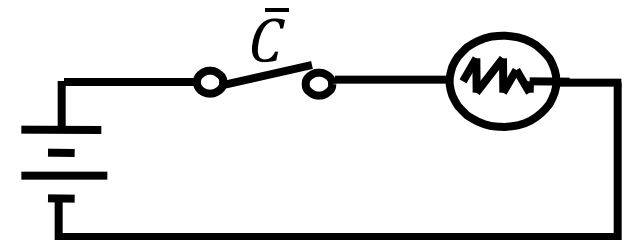
Switches in parallel => OR



Switches in series => AND

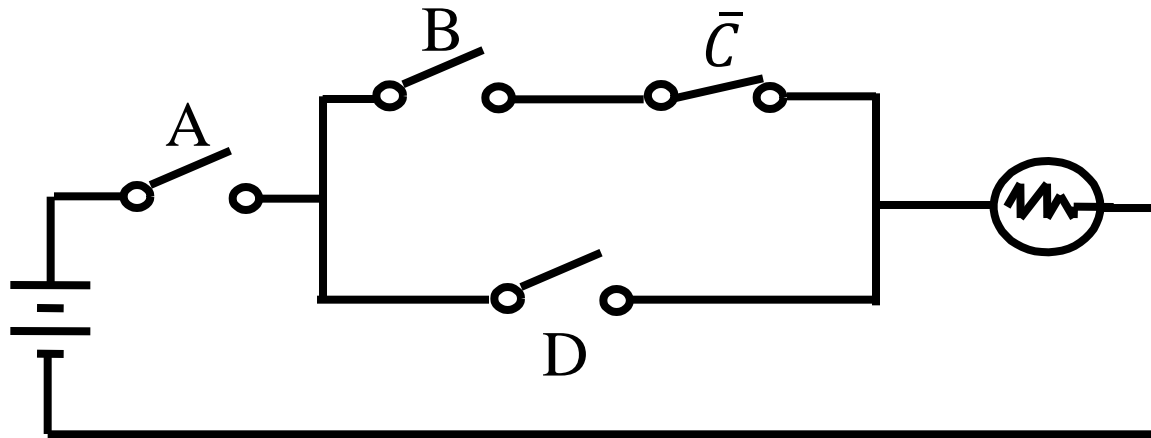


Normally-closed switch => NOT



Logic Function Implementation (Continued)

- Example: Logic Using Switches



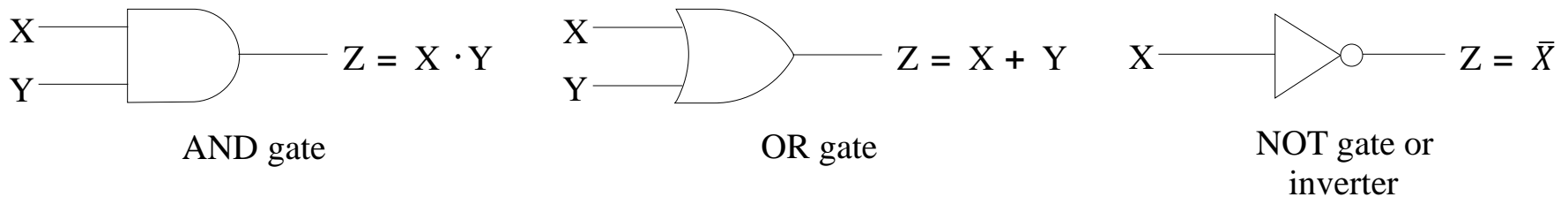
- Light is **ON** ($L = 1$) for $L(A, B, C, D) = A \cdot (B\bar{C} + D) = AB\bar{C} + AD$ and **OFF** ($L = 0$), otherwise.
- Useful model for relay circuits and for CMOS gate circuits, the foundation of current digital logic technology

Logic Gates

- In the earliest computers, switches were opened and closed by magnetic fields produced by energizing coils in *relays*. The switches in turn opened and closed the current paths
- Later, *vacuum tubes* that open and close current paths electronically replaced relays
- Today, *transistors* are used as electronic switches that open and close current paths
- Optional: Chapter 6 – Part 1: The Design Space

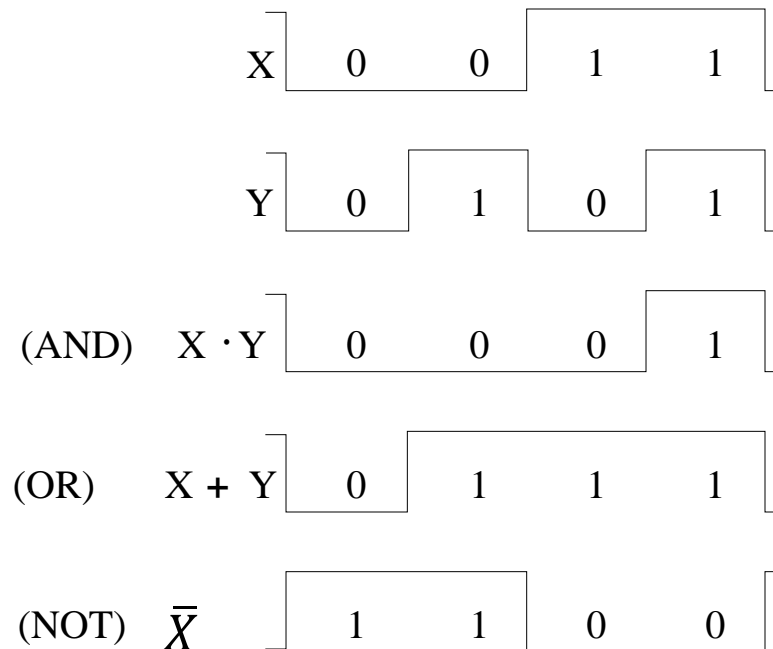
Logic Gate Symbols and Behavior

- Logic gates have special symbols:



(a) Graphic symbols

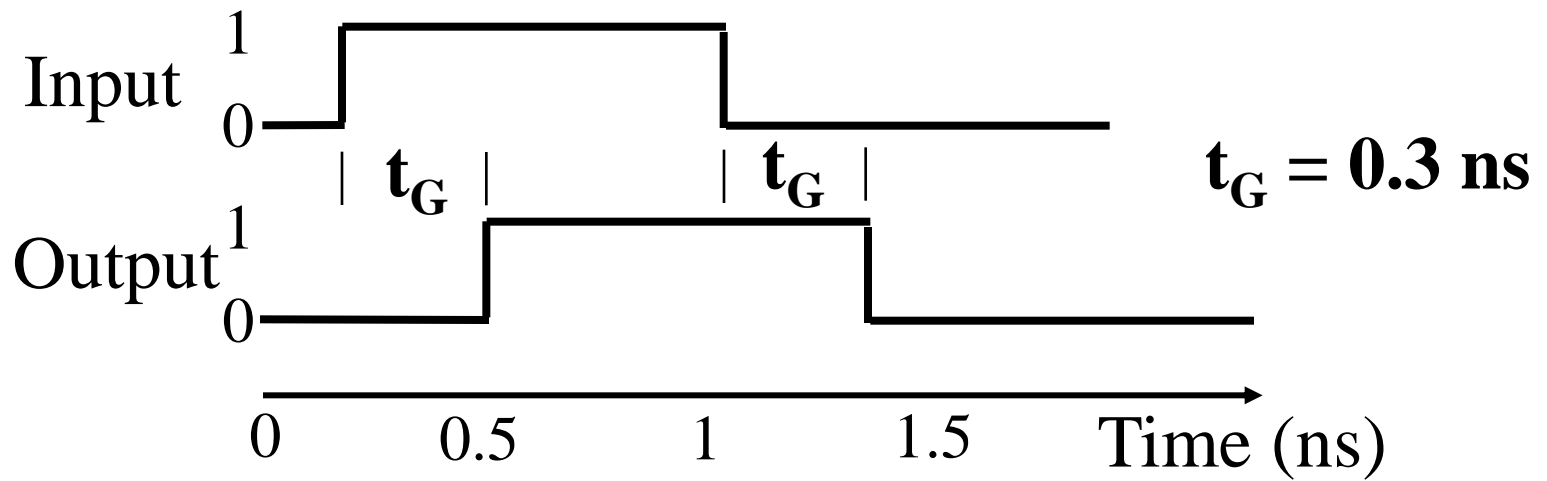
- And waveform behavior in time as follows:



(b) Timing diagram

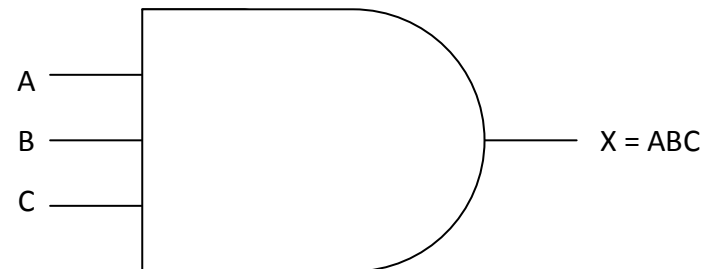
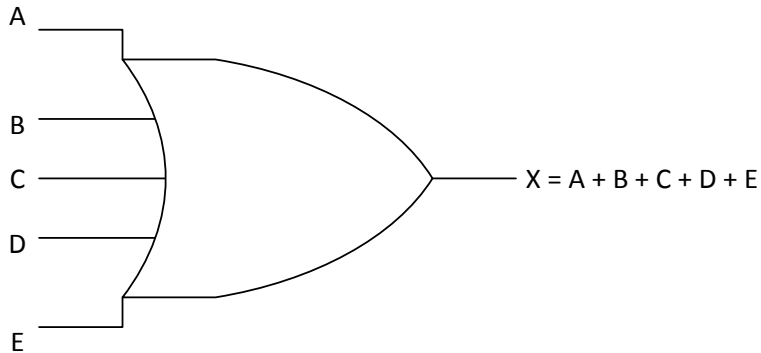
Gate Delay

- In actual physical gates, if one or more input changes causes the output to change, the output change does not occur instantaneously
- The delay between an input change(s) and the resulting output change is the *gate delay* denoted by t_G :



Logic Gates: Inputs and Outputs

- NOT (inverter)
 - Always one input and one output
- AND and OR gates
 - Always one output
 - Two or more inputs



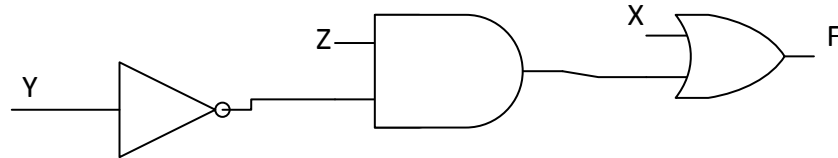
Boolean Algebra

- An algebra dealing with binary variables and logic operations
 - Variables are designated by letters of the alphabet
 - Basic logic operations: AND, OR, and NOT
- ***A Boolean expression*** is an algebraic expression formed by using binary variables, constants 0 and 1, the logic operation symbols, and parentheses
 - E.g.: $X \cdot 1$, $A + B + C$, $(A + B)(C + D)$
- ***A Boolean function*** consists of a binary variable identifying the function followed by equals sign and a Boolean expression
 - E.g.: $F = A + B + C$, $L(D, X, A) = DX + \bar{A}$

Logic Diagrams and Expressions

1. Equation: $F = X + \bar{Y}Z$

2. Logic Diagram:



3. Truth Table:

- Boolean equations, truth tables and logic diagrams describe the same function!
- Truth tables are unique; expressions and logic diagrams are not. This gives flexibility in implementing functions.

X	Y	Z	F
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

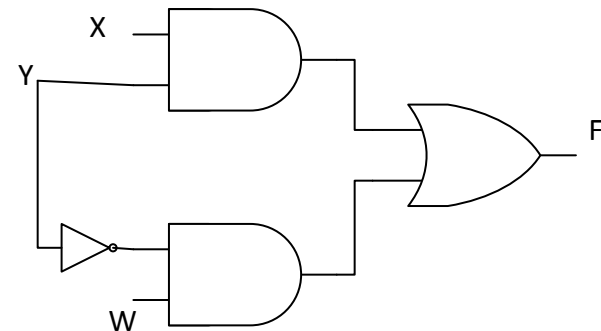
Example

- Draw the logic diagram and the truth table of the following Boolean function: $F(W, X, Y) = XY + W\bar{Y}$

- Logic Diagram:

- Truth Table:

W	X	Y	F
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1



- This example represents a ***Single Output Function***

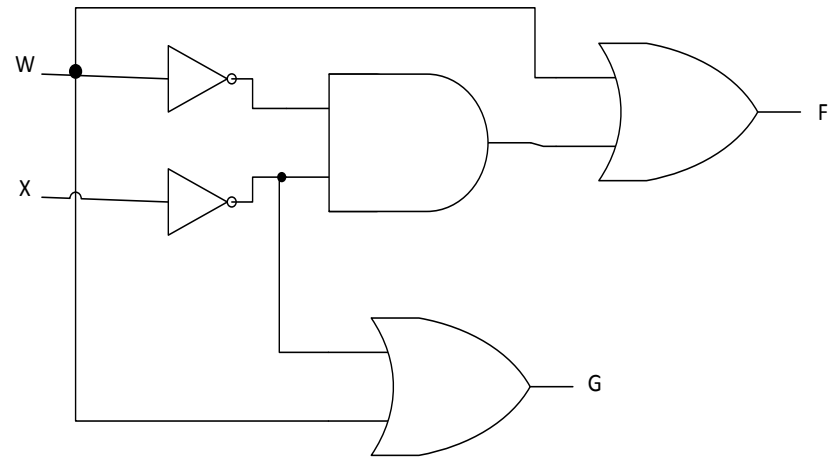
Example

- Draw the logic diagram and the truth table of the following Boolean functions: $F(W, X) = \bar{W}\bar{X} + W$, $G(W, X) = W + \bar{X}$

- Logic Diagram:

- Truth Table:

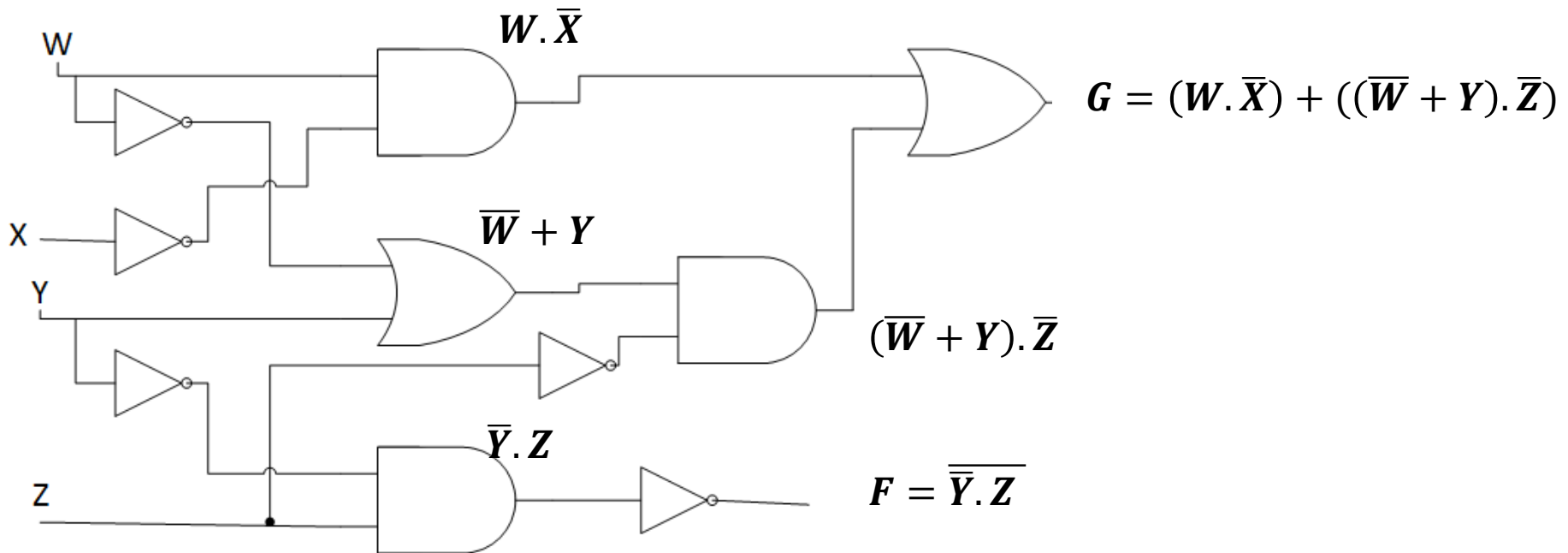
W	X	F	G
0	0	1	1
0	1	0	0
1	0	1	1
1	1	1	1



- This example represents a ***Multiple Output Function***

Example:

- Given the following logic diagram, write the corresponding Boolean equation:



- Logic circuits of this type are called combinational logic circuits since the variables are combined by logical operations

Basic Identities of Boolean Algebra

1. $X + 0 = X$	2. $X \cdot 1 = X$	<i>Existence of 0 and 1</i>
3. $X + 1 = 1$	4. $X \cdot 0 = 0$	
5. $X + X = X$	6. $X \cdot X = X$	<i>Idempotence</i>
7. $X + \bar{X} = 1$	8. $X \cdot \bar{X} = 0$	<i>Existence of complement</i>
9. $\bar{\bar{X}} = X$		<i>Involution</i>
10. $X + Y = Y + X$	11. $XY = YX$	<i>Commutative Laws</i>
12. $(X + Y) + Z = X + (Y + Z)$	13. $(XY)Z = X(YZ)$	<i>Associative Laws</i>
14. $X(Y + Z) = XY + XZ$	15. $X + YZ = (X + Y)(X + Z)$	<i>Distributive Laws</i>
16. $\overline{X + Y} = \bar{X} \cdot \bar{Y}$	17. $\overline{X \cdot Y} = \bar{X} + \bar{Y}$	<i>DeMorgan's Laws</i>

Some Properties of Identities & the Algebra

- If the meaning is unambiguous, we leave out the symbol “.”
- The identities above are organized into pairs
 - The *dual* of an algebraic expression is obtained by interchanging (+) and (\cdot) and interchanging 0's and 1's
 - The identities appear in *dual* pairs. When there is only one identity on a line the identity is *self-dual*, i. e., the dual expression = the original expression.

Some Properties of Identities & the Algebra (Continued)

- Unless it happens to be self-dual, the dual of an expression does not equal the expression itself
- Examples:
 - $F = (A + \bar{C}) \cdot B + 0$
 - $Dual\ F = (A \cdot \bar{C}) + B \cdot 1 = A \cdot \bar{C} + B$
 - $G = XY + \overline{(W + Z)}$
 - $Dual\ G = (X + Y) \cdot \overline{WZ} = (X + Y) \cdot (\bar{W} + \bar{Z})$
 - $H = AB + AC + BC$
 - $Dual\ H = (A + B)(A + C)(B + C) = (A + BC)(B + C) = AB + AC + BC$
- Are any of these functions self-dual?
 - Yes, H is self-dual

Boolean Operator Precedence

- The order of evaluation in a Boolean expression is:
 1. Parentheses
 2. NOT
 3. AND
 4. OR

- Consequence: Parentheses appear around OR expressions

- Examples:
 - $F = A(B + C)(C + \bar{D})$
 - $F = \sim AB = \bar{A}B$
 - $F = AB + C$
 - $F = A(B + C)$

Useful Boolean Theorems

<i>Theorem</i>	<i>Dual</i>	<i>Name</i>
$x \cdot y + \bar{x} \cdot y = y$	$(x + y)(\bar{x} + y) = y$	Minimization
$x + x \cdot y = x$	$x \cdot (x + y) = x$	Absorption
$x + \bar{x} \cdot y = x + y$	$x \cdot (\bar{x} + y) = x \cdot y$	Simplification
$x \cdot y + \bar{x} \cdot z + y \cdot z = x \cdot y + \bar{x} \cdot z$		Consensus
$(x + y)(\bar{x} + z)(y + z) = (x + y)(\bar{x} + z)$		

Example 1: Boolean Algebraic Proof

- $A + A \cdot B = A$ (Absorption Theorem)

Proof Steps	Justification (identity or theorem)
$A + A \cdot B$	
$= A \cdot 1 + A \cdot B$	$X = X \cdot 1$
$= A \cdot (1 + B)$	<i>Distributive Law</i>
$= A \cdot 1$	$1 + X = 1$
$= A$	$X \cdot 1 = X$

- Our primary reason for doing proofs is to learn:
 - Careful and efficient use of the identities and theorems of Boolean algebra
 - How to choose the appropriate identity or theorem to apply to make forward progress, irrespective of the application

Example 2: Boolean Algebraic Proofs

- $AB + \bar{A}C + BC = AB + \bar{A}C$ (Consensus Theorem)

Proof Steps	Justification (identity or theorem)
$AB + \bar{A}C + BC$	
$= AB + \bar{A}C + 1.BC$	$1.X = X$
$= AB + \bar{A}C + (A + \bar{A}).BC$	$X + \bar{X} = 1$
$= AB + \bar{A}C + ABC + \bar{A}BC$	<i>Distributive Law</i>
$= AB + ABC + \bar{A}C + \bar{A}BC$	<i>Commutative Law</i>
$= AB.1 + AB.C + \bar{A}C.1 + \bar{A}C.B$	$X.1 = X$ and <i>Commutative Law</i>
$= AB(1 + C) + \bar{A}C(1 + B)$	<i>Distributive Law</i>
$= AB.1 + \bar{A}C.1$	$1 + X = 1$
$= AB + \bar{A}C$	$X.1 = X$

Proof of Simplification

- $A + \bar{A}.B = A + B$ (Simplification Theorem)

Proof Steps	Justification (identity or theorem)
$A + \bar{A}.B$	
$= (A + \bar{A})(A + B)$	<i>Distributive Law</i>
$= 1.(A + B)$	$X + \bar{X} = 1$
$= A + B$	$X.1 = X$

- $A.(\bar{A} + B) = AB$ (Simplification Theorem)

Proof Steps	Justification (identity or theorem)
$A.(\bar{A} + B)$	
$= (A.\bar{A}) + (A.B)$	<i>Distributive Law</i>
$= 0 + AB$	$X.\bar{X} = 0$
$= AB$	$X + 0 = X$

Proof of Minimization

- $A.B + \bar{A}.B = B$ (Minimization Theorem)

Proof Steps	Justification (identity or theorem)
$A.B + \bar{A}.B$	
$= B(A + \bar{A})$	<i>Distributive Law</i>
$= B.1$	$X + \bar{X} = 1$
$= B$	$X.1 = X$

- $(A + B)(\bar{A} + B) = B$ (Minimization Theorem)

Proof Steps	Justification (identity or theorem)
$(A + B)(\bar{A} + B)$	
$= B + (A.\bar{A})$	<i>Distributive Law</i>
$= B + 0$	$X.\bar{X} = 0$
$= B$	$X + 0 = X$

Proof of DeMorgan's Laws (1)

- $\overline{X + Y} = \bar{X} \cdot \bar{Y}$ (DeMorgan's Law)
 - We will show that, $\bar{X} \cdot \bar{Y}$, satisfies the definition of the complement of $(X + Y)$, defined as $\overline{X + Y}$ by DeMorgan's Law.
 - To show this, we need to show that $A + A' = 1$ and $A \cdot A' = 0$ with $A = X + Y$ and $A' = \bar{X} \cdot \bar{Y}$. This proves that $\bar{X} \cdot \bar{Y} = \overline{X + Y}$.
- Part 1: Show $X + Y + \bar{X} \cdot \bar{Y} = 1$

Proof Steps	Justification (identity or theorem)
$(X + Y) + \bar{X} \cdot \bar{Y}$	
$= (X + Y + \bar{X})(X + Y + \bar{Y})$	<i>Distributive Law</i>
$= (1 + Y)(X + 1)$	$X + \bar{X} = 1$
$= 1 \cdot 1$	$X + 1 = 1$
$= 1$	$X \cdot 1 = X$

Proof of DeMorgan's Laws (2)

- Part 2: Show $(X + Y).X'.Y' = 0$

Proof Steps	Justification (identity or theorem)
$(X + Y).X'.Y'$	
$= (X.X'.Y') + (Y.X'.Y')$	<i>Distributive Law</i>
$= (0.Y') + (X'.0)$	$X.\bar{X} = 0$
$= 0 + 0$	$X.0 = 0$
$= 0$	$X + 0 = X$

- Based on the above two parts, $X'.Y' = \overline{X + Y}$
- The second DeMorgans' law is proved by duality
- Note that DeMorgan's law, given as an identity is not an axiom in the sense that it can be proved using the other identities.

Example 3: Boolean Algebraic Proofs

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

Proof Steps	Justification (identity or theorem)
$\overline{(X + Y)}Z + X\bar{Y}$	
$= X'Y'Z + X.Y'$	<i>DeMorgan's law</i>
$= Y'(X'Z + X)$	<i>Distributive law</i>
$= Y'(X + X'Z)$	<i>Commutative law</i>
$= Y'(X + Z)$	<i>Simplification Theorem</i>

Boolean Function Evaluation

- $F_1 = xy\bar{z}$
- $F_2 = x + \bar{y}z$
- $F_3 = \bar{x}\bar{y}\bar{z} + \bar{x}yz + x\bar{y}$
- $F_4 = x\bar{y} + \bar{x}z$

x	y	z	F_1	F_2	F_3	F_4
0	0	0	0	0	1	0
0	0	1	0	1	0	1
0	1	0	0	0	0	0
0	1	1	0	0	1	1
1	0	0	0	1	1	1
1	0	1	0	1	1	1
1	1	0	1	1	0	0
1	1	1	0	1	0	0

Expression Simplification

- An application of Boolean algebra
- Simplify to contain the smallest number of literals (complemented and uncomplemented variables)
- Example: Simplify the following Boolean expression
 - $AB + A'CD + A'BD + A'CD' + ABCD$

Simplification Steps	Justification (identity or theorem)
$AB + A'CD + A'BD + A'CD' + ABCD$	
$= AB + ABCD + A'CD + A'CD' + A'BD$	<i>Commutative law</i>
$= AB(1 + CD) + A'C(D + D') + A'BD$	<i>Distributive law</i>
$= AB.1 + A'C.1 + A'BD$	$1 + X = 1$ and $X + X' = 1$
$= AB + A'C + A'BD$	$X.1 = X$
$= AB + A'BD + A'C$	<i>Commutative law</i>
$= B(A + A'D) + A'C$	<i>Distributive law</i>
$= B(A + D) + A'C \rightarrow 5 \text{ Literals}$	<i>Simplification Theorem</i>

Complementing Functions

- Use DeMorgan's Theorem to complement a function:
 1. Interchange AND and OR operators
 2. Complement each constant value and literal

- Example: Complement $F = x'yz' + xy'z'$

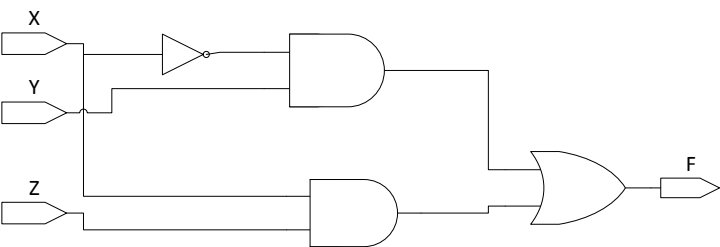
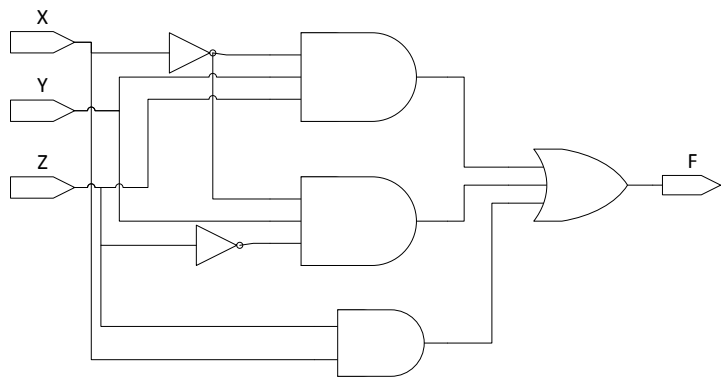
$$F' = (x + y' + z)(x' + y + z)$$

- Example: Complement $G = (a' + bc)d' + e$

$$G' = (a(b' + c') + d).e'$$

Example

- Simplify the following:
 - $F = X'YZ + X'YZ' + XZ$



Simplification Steps	(identity or theorem)
$X'YZ + X'YZ' + XZ$	
$= X'Y(Z + Z') + XZ$	<i>Distributive law</i>
$= X'Y.1 + XZ$	$X + X' = 1$
$= X'Y + XZ$	$X.1 = X$

x	y	z	$X'YZ + X'YZ' + XZ$	$X'Y + XZ$
0	0	0	0	0
0	0	1	0	0
0	1	0	1	1
0	1	1	1	1
1	0	0	0	0
1	0	1	1	1
1	1	0	0	0
1	1	1	1	1
			3 terms and 8 literals	2 terms and 4 literals

Example

- Show that $F = x'y' + xy' + x'y + xy = 1$

- Solution1: Truth Table

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

- Solution2: Boolean Algebra

Proof Steps	(identity or theorem)
$x'y' + xy' + x'y + xy$	
$= y'(x' + x) + y(x' + x)$	<i>Distributive law</i>
$= y'.1 + y.1$	$X + X' = 1$
$= y' + y$	$X.1 = X$
$= 1$	$X + X' = 1$

Examples

- Show that $ABC + A'C' + AC' = AB + C'$ using Boolean algebra.

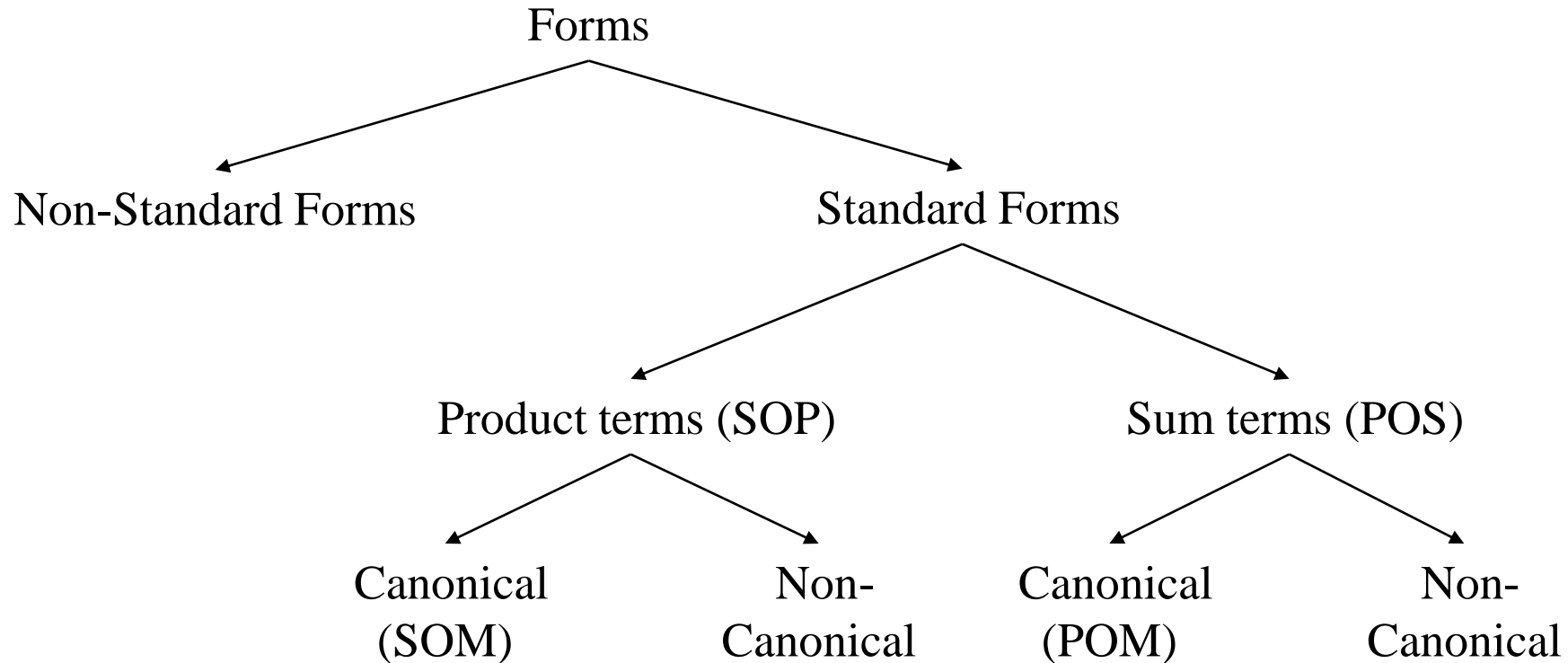
Proof Steps	(identity or theorem)
$ABC + A'C' + AC'$	
$= ABC + C'(A' + A)$	<i>Distributive law</i>
$= ABC + C'.1$	$X + X' = 1$
$= ABC + C'$	$X.1 = X$
$= (AB + C')(C + C')$	<i>Distributive law</i>
$= (AB + C').1$	$X + X' = 1$
$= AB + C'$	$X.1 = X$

- Find the dual and the complement of $f = wx + y'z.0 + w'z$
 - $Dual(f) = (w + x)(y' + z + 1)(w' + z)$
 - $f' = (w' + x')(y + z' + 1)(w + z')$

Overview – Canonical Forms

- What are Canonical Forms?
- Minterms and Maxterms
- Index Representation of Minterms and Maxterms
- Sum-of-Minterm (SOM) Representations
- Product-of-Maxterm (POM) Representations
- Representation of Complements of Functions
- Conversions between Representations

Boolean Representation Forms



Canonical Forms

- It is useful to specify Boolean functions in a form that:
 - Allows comparison for equality
 - Has a correspondence to the truth tables
 - Facilitates simplification
- Canonical Forms in common usage:
 - Sum of Minterms (SOM)
 - Product of Maxterms (POM)

Minterms

- ***Minterms*** are AND terms with ***every variable*** present in either true or complemented form
- Given that each binary variable may appear normal (e.g., x) or complemented (e.g., \bar{x}), there are 2^n minterms for n variables
- Example: Two variables (X and Y) produce $2^2 = 4$ combinations:
 - XY (both normal)
 - $X\bar{Y}$ (X normal, Y complemented)
 - $\bar{X}Y$ (X complemented, Y normal)
 - $\bar{X}\bar{Y}$ (both complemented)
- Thus there are ***four minterms*** of two variables

Maxterms

- **Maxterms** are OR terms with *every variable* in true or complemented form
- Given that each binary variable may appear normal (e.g., x) or complemented (e.g., \bar{x}), there are 2^n maxterms for n variables
- Example: Two variables (X and Y) produce $2^2 = 4$ combinations:

$X + Y$ (both normal)

$X + \bar{Y}$ (X normal, Y complemented)

$\bar{X} + Y$ (X complemented, Y normal)

$\bar{X} + \bar{Y}$ (both complemented)

Maxterms and Minterms

- Examples: Three variable (X, Y, Z) minterms and maxterms

Index	Minterm (m)	Maxterm (M)
0	$\bar{X}\bar{Y}\bar{Z}$	$X + Y + Z$
1	$\bar{X}\bar{Y}Z$	$X + Y + \bar{Z}$
2	$\bar{X}Y\bar{Z}$	$X + \bar{Y} + Z$
3	$\bar{X}YZ$	$X + \bar{Y} + \bar{Z}$
4	$X\bar{Y}\bar{Z}$	$\bar{X} + Y + Z$
5	$X\bar{Y}Z$	$\bar{X} + Y + \bar{Z}$
6	$XY\bar{Z}$	$\bar{X} + \bar{Y} + Z$
7	XYZ	$\bar{X} + \bar{Y} + \bar{Z}$

- The *index* above is important for describing which variables in the terms are true and which are complemented

Standard Order

- Minterms and maxterms are designated with a subscript
- The subscript is a number, corresponding to a binary pattern
- The bits in the pattern represent the complemented or normal state of each variable listed in a standard order
- All variables will be present in a minterm or maxterm and will be listed in the *same order (usually alphabetically)*
- **Example: For variables a, b, c:**
 - **Maxterms:** $(a + b + \bar{c})$, $(a + b + c)$
 - **Terms:** $(b + a + c)$, $a\bar{c}b$, and $(c + b + a)$ are **NOT** in standard order.
 - **Minterms:** $a\bar{b}c$, abc , $\bar{a}b\bar{c}$
 - **Terms:** $(a + c)$, $\bar{b}c$, and $(\bar{a} + b)$ do not contain all variables

Purpose of the Index

- The *index* for the minterm or maxterm, expressed as a binary number, is used to determine whether the variable is shown in the true form or complemented form
- **For Minterms:**
 - “0” means the variable is “Complemented”
 - “1” means the variable is “Not Complemented”
- **For Maxterms:**
 - “0” means the variable is “Not Complemented”
 - “1” means the variable is “Complemented”

Index Example: Three Variables

Index (Decimal)	Index (Binary) n = 3 Variables	Minterm (m)	Maxterm (M)
0	000	$m_0 = \bar{X}\bar{Y}\bar{Z}$	$M_0 = X + Y + Z$
1	001	$m_1 = \bar{X}\bar{Y}Z$	$M_1 = X + Y + \bar{Z}$
2	010	$m_2 = \bar{X}Y\bar{Z}$	$M_2 = X + \bar{Y} + Z$
3	011	$m_3 = \bar{X}YZ$	$M_3 = X + \bar{Y} + \bar{Z}$
4	100	$m_4 = X\bar{Y}\bar{Z}$	$M_4 = \bar{X} + Y + Z$
5	101	$m_5 = X\bar{Y}Z$	$M_5 = \bar{X} + Y + \bar{Z}$
6	110	$m_6 = XY\bar{Z}$	$M_6 = \bar{X} + \bar{Y} + Z$
7	111	$m_7 = XYZ$	$M_7 = \bar{X} + \bar{Y} + \bar{Z}$

Index Example: Four Variables

i (Decimal)	i (Binary) n = 4 Variables	m_i	M_i
0	0000	$\bar{a}\bar{b}\bar{c}\bar{d}$	$a + b + c + d$
1	0001	$\bar{a}\bar{b}\bar{c}d$	$a + b + c + \bar{d}$
3	0011	$\bar{a}\bar{b}cd$	$a + b + \bar{c} + \bar{d}$
5	0101	$\bar{a}b\bar{c}d$	$a + \bar{b} + c + \bar{d}$
7	0111	$\bar{a}bcd$	$a + \bar{b} + \bar{c} + \bar{d}$
10	1010	$a\bar{b}\bar{c}\bar{d}$	$\bar{a} + b + \bar{c} + d$
13	1101	$ab\bar{c}d$	$\bar{a} + \bar{b} + c + \bar{d}$
15	1111	$abcd$	$\bar{a} + \bar{b} + \bar{c} + \bar{d}$

Minterm and Maxterm Relationship

- Review: DeMorgan's Theorem
 - $\overline{x \cdot y} = \bar{x} + \bar{y}$ and $\overline{\bar{x} + \bar{y}} = x \cdot y$
- Two-variable example:
 - $M_2 = \bar{x} + y$ and $m_2 = x \cdot \bar{y}$
 - Using DeMorgan's Theorem $\rightarrow \overline{\bar{x} + y} = \bar{\bar{x}} \cdot \bar{y} = x \cdot \bar{y}$
 - Using DeMorgan's Theorem $\rightarrow \overline{x \cdot \bar{y}} = \bar{x} + \bar{\bar{y}} = \bar{x} + y$
 - Thus, M_2 is the complement of m_2 and vice-versa
- Since DeMorgan's Theorem holds for n variables, the above holds for terms of n variables:

$$M_i = \overline{m_i} \text{ and } m_i = \overline{M_i}$$

- Thus, M_i is the complement of m_i and vice-versa

Function Tables for Both

- Minterms of 2 variables:

xy	m_0	m_1	m_2	m_3
00	1	0	0	0
01	0	1	0	0
10	0	0	1	0
11	0	0	0	1

- Maxterms of 2 variables:

xy	M_0	M_1	M_2	M_3
00	0	1	1	1
01	1	0	1	1
10	1	1	0	1
11	1	1	1	0

- Each column in the maxterm function table is the complement of the column in the minterm function table since M_i is the complement of m_i .

Observations

- In the function tables:
 - Each *minterm* has one and only one 1 present in the 2^n terms (a minimum of 1s). All other entries are 0.
 - Each *maxterm* has one and only one 0 present in the 2^n terms All other entries are 1 (a maximum of 1s).
- We can implement any function by
 - "ORing" the minterms corresponding to "1" entries in the function table. These are called the minterms of the function.
 - "ANDing" the maxterms corresponding to "0" entries in the function table. These are called the maxterms of the function.
- This gives us two canonical forms for stating any Boolean function:
 - *Sum of Minterms (SOM)*
 - *Product of Maxterms (POM)*

Minterm Function Example

- Example: Find $F_1 = m_1 + m_4 + m_7$
- $F_1 = x'y'z + xy'z' + xyz$

xyz	Index	$m_1 + m_4 + m_7 = F_1$
000	0	$0 + 0 + 0 = 0$
001	1	$1 + 0 + 0 = 1$
010	2	$0 + 0 + 0 = 0$
011	3	$0 + 0 + 0 = 0$
100	4	$0 + 1 + 0 = 1$
101	5	$0 + 0 + 0 = 0$
110	6	$0 + 0 + 0 = 0$
111	7	$0 + 0 + 1 = 1$

Minterm Function Example

- $F(A, B, C, D, E) = m_2 + m_9 + m_{17} + m_{23}$
- $F(A, B, C, D, E) = A'B'C'DE' + A'BC'D'E + AB'C'D'E + AB'CDE$

Maxterm Function Example

- **Example: Implement F1 in maxterms:**
- $F_1 = M_0 \cdot M_2 \cdot M_3 \cdot M_5 \cdot M_6$
- $F_1 = (x + y + z) \cdot (x + y' + z) \cdot (x + y' + z') \cdot (x' + y + z') \cdot (x' + y' + z)$

xyz	Index	$M_0 \cdot M_2 \cdot M_3 \cdot M_5 \cdot M_6 = F_1$
000	0	0 . 1 . 1 . 1 . 1 = 0
001	1	1 . 1 . 1 . 1 . 1 = 1
010	2	1 . 0 . 1 . 1 . 1 = 0
011	3	1 . 1 . 0 . 1 . 1 = 0
100	4	1 . 1 . 1 . 1 . 1 = 1
101	5	1 . 1 . 1 . 0 . 1 = 0
110	6	1 . 1 . 1 . 1 . 0 = 0
111	7	1 . 1 . 1 . 1 . 1 = 1

Maxterm Function Example

- $F(A, B, C, D) = M_3 \cdot M_8 \cdot M_{11} \cdot M_{14}$
- $F(A, B, C, D)$
 $= (A + B + C' + D') \cdot (A' + B + C + D) \cdot$
 $(A' + B + C' + D') \cdot (A' + B' + C' + D)$

Canonical Sum of Minterms

- Any Boolean function can be expressed as a Sum of Minterms (SOM):
 - For the function table, the minterms used are the terms corresponding to the 1's
 - For expressions, expand all terms first to explicitly list all minterms. Do this by “ANDing” any term missing a variable v with a term $(v + \bar{v})$
- Example: Implement $f = x + \bar{x}\bar{y}$ as a SOM?
 1. Expand terms $\rightarrow f = x(y + \bar{y}) + \bar{x}\bar{y}$
 2. Distributive law $\rightarrow f = xy + x\bar{y} + \bar{x}\bar{y}$
 3. Express as SOM $\rightarrow f = m_3 + m_2 + m_0 = m_0 + m_2 + m_3$

Another SOM Example

- Example: $F = A + \bar{B}C$
- There are three variables: A, B, and C which we take to be the standard order
- Expanding the terms with missing variables:
 - $F = A(B + \bar{B})(C + \bar{C}) + (A + \bar{A})\bar{B}C$
- Distributive law:
 - $F = ABC + A\bar{B}C + AB\bar{C} + A\bar{B}\bar{C} + \bar{A}BC + \bar{A}\bar{B}C$
- Collect terms (removing all but one of duplicate terms):
 - $F = ABC + AB\bar{C} + A\bar{B}C + A\bar{B}\bar{C} + \bar{A}\bar{B}C$
- Express as SOM:
 - $F = m_7 + m_6 + m_5 + m_4 + m_1$
 - $F = m_1 + m_4 + m_5 + m_6 + m_7$

Shorthand SOM Form

- From the previous example, we started with:
 - $F = A + \bar{B}C$
- We ended up with:
 - $F = m_1 + m_4 + m_5 + m_6 + m_7$
- This can be denoted in the *formal shorthand*:
 - $F(A, B, C) = \sum_m(1,4,5,6,7)$
- Note that we explicitly show the standard variables in order and drop the “m” designators.

Canonical Product of Maxterms

- Any Boolean Function can be expressed as a Product of Maxterms (POM):
 - For the function table, the maxterms used are the terms corresponding to the 0's
 - For an expression, expand all terms first to explicitly list all maxterms. Do this by first applying the second distributive law, “ORing” terms missing variable v with $(v \cdot \bar{v})$ and then applying the distributive law again
- Example: Convert $f(x, y, z) = x + \bar{x}\bar{y}$ to POM?
 - Distributive law $\rightarrow f = (x + \bar{x}) \cdot (x + \bar{y}) = x + \bar{y}$
 - ORing with missing variable (z) $\rightarrow f = x + \bar{y} + z \cdot \bar{z}$
 - Distributive law $\rightarrow f = (x + \bar{y} + z) \cdot (x + \bar{y} + \bar{z})$
 - Express as POS $\rightarrow f = M_2 \cdot M_3$

Another POM Example

- Convert $f(A, B, C) = AC' + BC + A'B'$ to POM?
- Use $x + yz = (x + y) \cdot (x + z)$, assuming $x = AC' + BC$ and $y = A'$ and $z = B'$
 - $f(A, B, C) = (AC' + BC + A') \cdot (AC' + BC + B')$
- Use Simplification theorem to get:
 - $f(A, B, C) = (BC + A' + C') \cdot (AC' + B' + C)$
- Use Simplification theorem again to get:
 - $f(A, B, C) = (A' + B + C') \cdot (A + B' + C) = M_5 \cdot M_2$
 - $f(A, B, C) = M_2 \cdot M_5 = \prod_M(2,5) \rightarrow$ ***Shorthand POM form***

Function Complements

- The complement of a function expressed as a sum of minterms is constructed by selecting the minterms missing in the sum-of-minterms canonical forms.
- Alternatively, the complement of a function expressed by a sum of minterms form is simply the Product of Maxterms with the same indices.
- Example: Given $F(x, y, z) = \sum_m(1,3,5,7)$, find complement F as SOM and POM?
 - $\bar{F}(x, y, z) = \sum_m(0,2,4,6)$
 - $\bar{F}(x, y, z) = \prod_M(1,3,5,7)$

Conversion Between Forms

- To convert between sum-of-minterms and product-of-maxterms form (or vice-versa) we follow these steps:
 - Find the function complement by swapping terms in the list with terms not in the list.
 - Change from products to sums, or vice versa.
- **Example:** Given F as before: $F(x, y, z) = \sum_m(1,3,5,7)$
 - Form the Complement:
$$\bar{F}(x, y, z) = \sum_m(0,2,4,6)$$
 - Then use the other form with the same indices – this forms the complement again, giving the other form of the original function:
$$F(x, y, z) = \prod_M(0,2,4,6)$$

Important Properties of Minterms

- Maxterms are seldom used directly to express Boolean functions
- Minterms properties:
 - For n Boolean variables, there are 2^n minterms (0 to $2^n - 1$)
 - Any Boolean function can be represented as a logical sum of minterms (SOM)
 - The complement of a function contains those minterms not included in the original function
 - A function that include all the 2^n minterms is equal to 1

Standard Forms

- Standard Sum-of-Products (SOP) form: equations are written as an OR of AND terms
- Standard Product-of-Sums (POS) form: equations are written as an AND of OR terms
- **Examples:**
 - SOP: $ABC + \bar{A}\bar{B}C + B$
 - POS: $(A + B) \cdot (A + \bar{B} + \bar{C}) \cdot C$
- These “mixed” forms are neither SOP nor POS
 - $(AB + C)(A + C)$
 - $AB\bar{C} + AC(A + B)$

Standard Sum-of-Products (SOP)

- A sum of minterms form for n variables can be written down directly from a truth table
- Implementation of this form is a two-level network of gates such that:
 - The first level consists of n -input AND gates, and
 - The second level is a single OR gate (with fewer than 2^n inputs)
- This form often can be simplified so that the corresponding circuit is simpler

Standard Sum-of-Products (SOP)

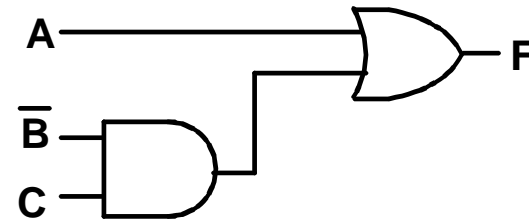
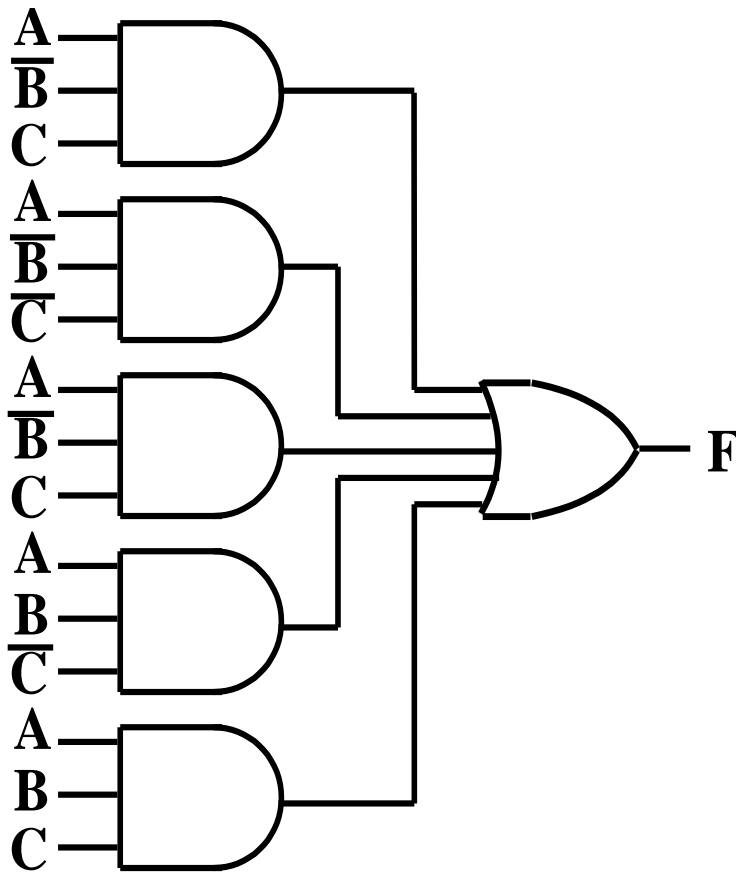
- A Simplification Example: $F(A, B, C) = \sum_m(1,4,5,6,7)$
- Writing the minterm expression:
 - $F(A, B, C) = A'B'C + AB'C' + AB'C + ABC' + ABC$
- Simplifying using boolean Algebra:

Simplification Steps	(identity or theorem)
$A'B'C + AB'C' + AB'C + ABC' + ABC$	
$= A'B'C + AB'(C' + C) + AB(C' + C)$	<i>Distributive law</i>
$= A'B'C + AB' + AB$	$X + X' = 1$
$= A'B'C + A(B' + B)$	<i>Distributive law</i>
$= A'B'C + A$	<i>Simplification Theorem</i>
$= A + B'C$	

- Simplified F contains 3 literals compared to 15 in minterm F

AND/OR Two-level Implementation of SOP Expression

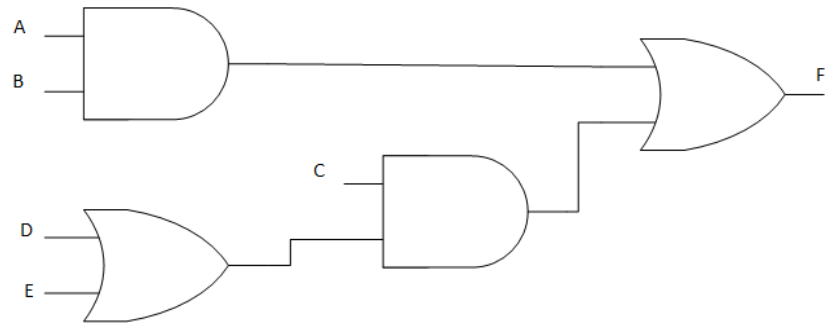
- The two implementations for F are shown below – it is quite apparent which is simpler!



Two-level Implementation

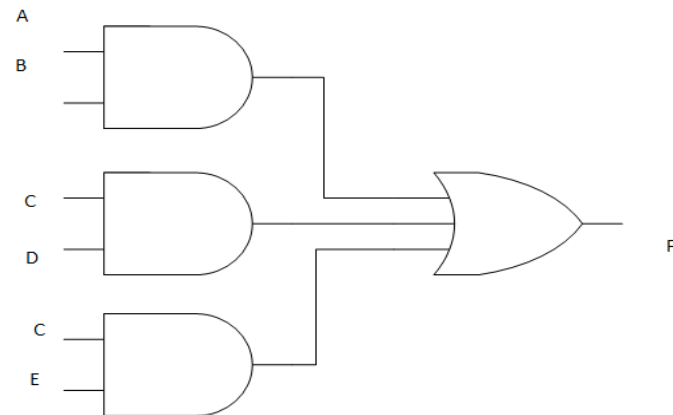
- Draw the logic diagram of the following boolean function:

- $f = AB + C(D + E)$



- Represent the function using two-level implementation:

- $f = AB + CD + CE \rightarrow \text{SOP}$



SOP and POS Observations

- **The previous examples show that:**
 - **Canonical Forms (Sum-of-minterms, Product-of-Maxterms), or other standard forms (SOP, POS) differ in complexity**
 - **Boolean algebra can be used to manipulate equations into simpler forms.**
 - **Simpler equations lead to simpler two-level implementations**
- **Questions:**
 - **How can we attain a “simplest” expression?**
 - **Is there only one minimum cost circuit?**
 - **The next part will deal with these issues.**

Logic and Computer Design Fundamentals

Chapter 2 – Combinational Logic Circuits

Part 2 – Circuit Optimization

Charles Kime & Thomas Kaminski

© 2008 Pearson Education, Inc.

(Hyperlinks are active in View Show mode)

Updated by Dr. Waleed Dweik

Overview

- **Part 1 – Gate Circuits and Boolean Equations**
 - **Binary Logic and Gates**
 - **Boolean Algebra**
 - **Standard Forms**
- **Part 2 – Circuit Optimization**
 - **Two-Level Optimization**
 - **Map Manipulation**
- **Part 3 – Additional Gates and Circuits**
 - **Other Gate Types**
 - **Exclusive-OR Operator and Gates**
 - **High-Impedance Outputs**

Circuit Optimization

- Goal: To obtain the simplest implementation for a given function
- Optimization is a more formal approach to simplification that is performed using a specific procedure or algorithm
- Optimization requires a cost criterion to measure the simplicity of a circuit
- Distinct cost criteria we will use:
 - **Literal cost (L)**
 - **Gate input cost (G)**
 - **Gate input cost with NOTs (GN)**

Literal Cost

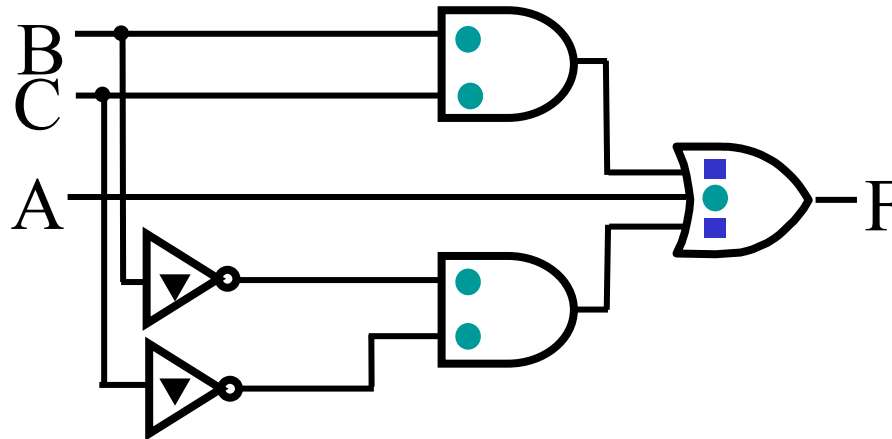
- **Literal:** a variable or its complement
- **Literal cost (L):** the number of literal appearances in a Boolean expression corresponding to the logic circuit diagram
- Examples:
 - $F = BD + AB'C + AC'D'$
 - $L = 8$ (Minimum cost \rightarrow Best solution)
 - $F = BD + AB'C + AB'D' + ABC'$
 - $L = 11$
 - $F = (A + B)(A + D)(B + C + D')(B' + C' + D)$
 - $L = 10$

Gate Input Cost

- **Gate input cost (G):** the number of inputs to the gates in the implementation corresponding exactly to the given equation or equations. (**G: inverters not counted, GN: inverters counted**)
- For SOP and POS equations, it can be found from the equation(s) by finding the sum of:
 - All literal appearances
 - The number of terms excluding single literal terms,(G) and
 - optionally, the number of distinct complemented single literals (GN).
- Examples:
 - $F = BD + AB'C + AC'D'$
 - $G = 11, GN = 14$ (Minimum cost → Best solution)
 - $F = BD + AB'C + AB'D' + ABC'$
 - $G = 15, GN = 18$
 - $F = (A + B)(A + D)(B + C + D')(B' + C' + D)$
 - $G = 14, GN = 17$

Cost Criteria (continued)

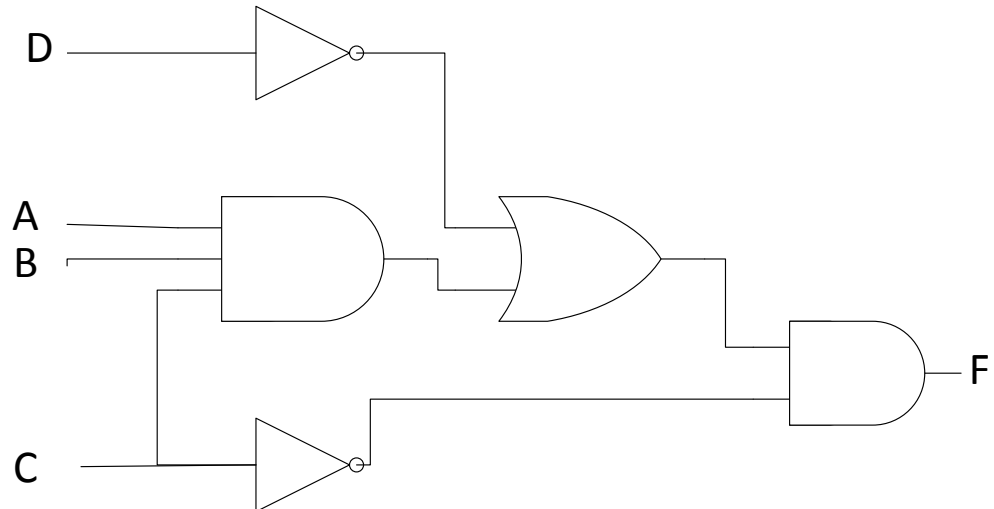
- Example 1: $\nabla \nabla$ $GN = G + 2 = 9$
- $F = \overset{\bullet}{A} + \overset{\bullet}{B} \overset{\bullet}{C} + \overset{\bullet}{\bar{B}} \overset{\bullet}{\bar{C}}$ $L = 5$
- $G = L + 2 = 7$



- L (literal count) counts the AND inputs and the single literal OR input.
- G (gate input count) adds the remaining OR gate inputs
- GN (gate input count with NOTs) adds the inverter inputs

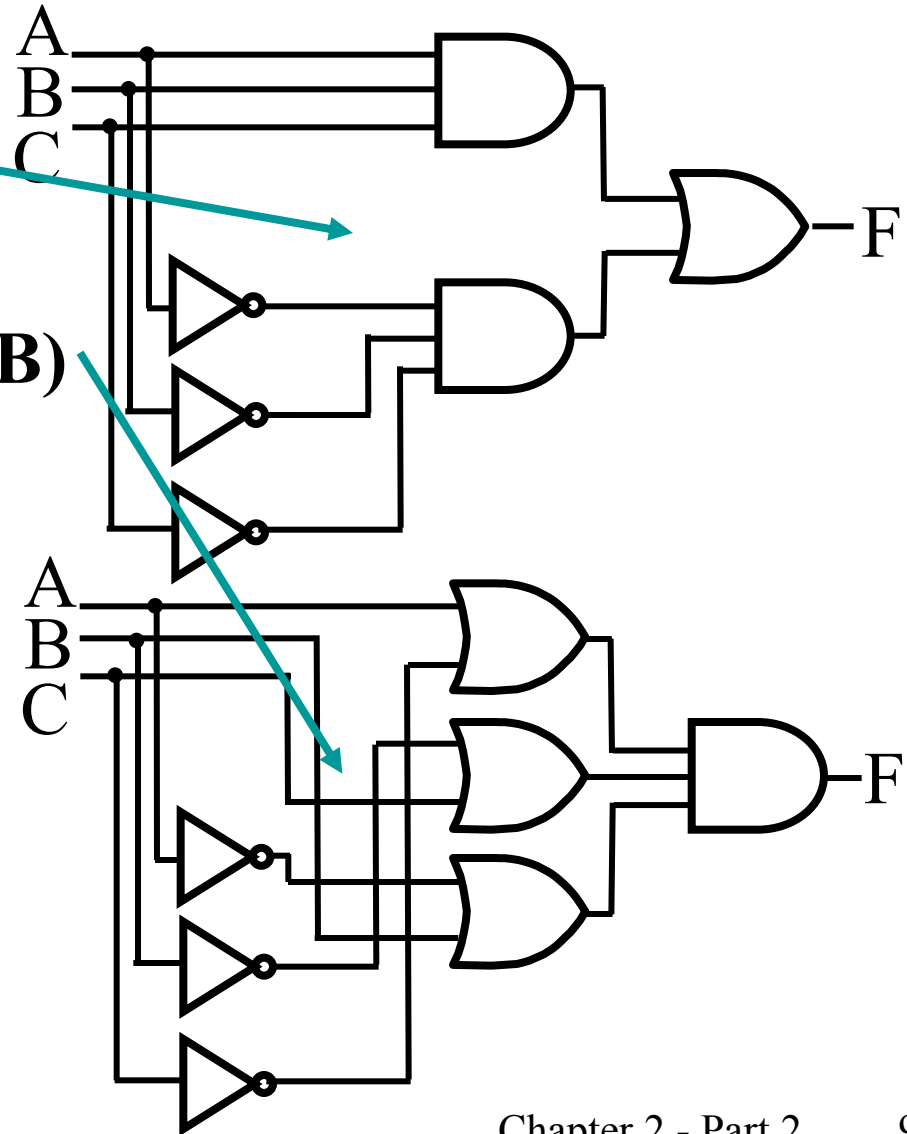
Cost Criteria (continued)

- **Example 2:**
- $F = (A, B, C, D) = (ABC + D') \cdot C'$
 - $L = 5$
 - $G = 5 + 2 = 7$
 - $GN = 7 + 2 = 9$



Cost Criteria (continued)

- **Example 3:**
- $F = A B C + \bar{A}\bar{B}\bar{C}$
- $L = 6, G = 8, GN = 11$
- $F = (A + \bar{C})(\bar{B} + C)(\bar{A} + B)$
- $L = 6, G = 9, GN = 12$
- Same function and same literal cost
- But first circuit has better gate input count and better gate input count with NOTs
- **Select it!**



Boolean Function Optimization

- Minimizing the gate input (or literal) cost of a (a set of) Boolean equation(s) reduces circuit cost
- We choose gate input cost
- Boolean Algebra and graphical techniques are tools to minimize cost criteria values
- Some important questions:
 - When do we stop trying to reduce the cost?
 - Do we know when we have a minimum cost?
- Treat optimum or near-optimum cost functions for two-level (SOP and POS) circuits
- Introduce a graphical technique using Karnaugh maps (K-maps, for short)

Karnaugh Maps (K-map)

- A K-map is a collection of squares
 - Graphical representation of the truth table
 - Each square represents a minterm, or a maxterm, or a row in the truth table
 - For n-variable, there are 2^n squares
 - The collection of squares is a graphical representation of a Boolean function
 - Adjacent squares differ in the value of one variable
 - Alternative algebraic expressions for the same function are derived by recognizing patterns of squares

Some Uses of K-Maps

- Finding optimum or near optimum
 - SOP and POS standard forms, and
 - two-level AND/OR and OR/AND circuit implementationsfor functions with small numbers of variables
- Visualizing concepts related to manipulating Boolean expressions, and
- Demonstrating concepts used by computer-aided design programs to simplify large circuits

Two Variable Maps

- **A 2-variable Karnaugh Map:**

- Note that minterm m_0 and minterm m_1 are “adjacent” and differ in the value of the variable y

	$y = 0$	$y = 1$
$x = 0$	$m_0 = \bar{x}\bar{y}$	$m_1 = \bar{x}y$
$x = 1$	$m_2 = x\bar{y}$	$m_3 = xy$

- Similarly, minterm m_0 and minterm m_2 differ in the x variable
- Also, m_1 and m_3 differ in the x variable as well
- Finally, m_2 and m_3 differ in the value of the variable y

K-Map and Truth Tables

- The K-Map is just a different form of the truth table
- Example: Two variable function
 - We choose a, b, c and d from the set $\{0, 1\}$ to implement a particular function, $F(x, y)$

Input Values (x, y)	$F(x, y)$
0 0	a
0 1	b
1 0	c
1 1	d

Truth Table

	$y = 0$	$y = 1$
$x = 0$	a	b
$x = 1$	c	d

K-Map

K-Map Function Representation

- Example: $F(x, y) = x$

$F(x, y) = x$	$y = 0$	$y = 1$
$x = 0$	0	0
$x = 1$	1	1

- For function $F(x, y)$, the two adjacent cells containing 1's can be combined using the Minimization Theorem:

$$F(x, y) = x\bar{y} + xy = x$$

K-Map Function Representation

- Example: $G(x, y) = x + y$

$G(x, y) = x + y$	$y = 0$	$y = 1$
$x = 0$	0	1
$x = 1$	1	1

- For $G(x, y)$, two pairs of adjacent cells containing 1's can be combined using the Minimization Theorem:

$$G(x, y) = (x\bar{y} + xy) + (\bar{x}y + xy)$$

$$G(x, y) = x + y$$

Three Variable Maps

- A three-variable K-map:

	yz = 00	yz = 01	yz = 11	yz = 10
x = 0	m_0	m_1	m_3	m_2
x = 1	m_4	m_5	m_7	m_6

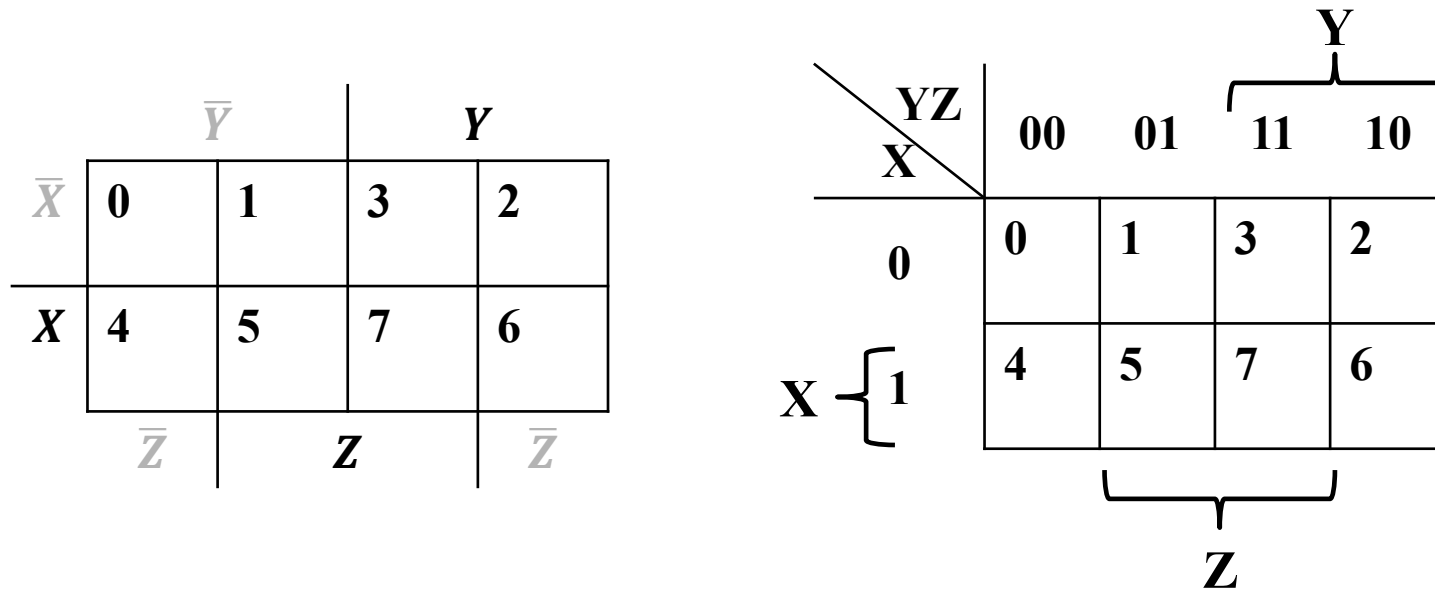
- Where each minterm corresponds to the product terms:

	yz = 00	yz = 01	yz = 11	yz = 10
x = 0	$\bar{x}\bar{y}\bar{z}$	$\bar{x}\bar{y}z$	$\bar{x}yz$	$\bar{x}y\bar{z}$
x = 1	$x\bar{y}\bar{z}$	$x\bar{y}z$	xyz	$xy\bar{z}$

- *Note that if the binary value for an index differs in one bit position, the minterms are adjacent on the K-Map*

Alternative Map Labeling

- Map use largely involves:
 - Entering values into the map, and
 - Reading off product terms from the map
- Alternate labelings are useful:



Example Functions

- By convention, we represent the minterms of F by a "1" in the map and leave the minterms of \bar{F} blank

- Example:

- $F(x, y, z) = \sum_m(2,3,4,5)$

		Y	
	0	1	3
			1
			1
X	4	5	7
	1	1	
			6
		Z	

- Example:

- $G(a, b, c) = \sum_m(3,4,6,7)$

		b	
	0	1	3
			1
a	4	5	7
	1		1
			6
		c	

- Learn the locations of the 8 indices based on the variable order shown (X, most significant and Z, least significant) on the map boundaries

Steps for using K-Maps to Simplify Boolean Functions

- Enter the function on the K-Map

- Function can be given in truth table, shorthand notation, SOP,...etc

- Example:

- $F(x, y) = \bar{x} + xy$

- $F(x, y) = \sum_m(0,1,3)$

x	y	F(x, y)
0	0	1
0	1	1
1	0	0
1	1	1

	y	
	0	1
x	1	1
	2	3
		1

- Combining squares for simplification

- Rectangles that include power of 2 squares {1, 2, 4, 8, ...}
- Goal: Fewest rectangles that cover all 1's → as large as possible

- Determine if any rectangle is not needed

- Read-off the SOP terms

Combining Squares

- By combining squares, we reduce number of literals in a product term, reducing the literal cost, thereby reducing the other two cost criteria
- On a 2-variable K-Map:
 - One square represents a minterm with two variables
 - Two adjacent squares represent a product term with one variable
 - Four “adjacent” terms is the function of all ones (no variables) = 1.
- On a 3-variable K-Map:
 - One square represents a minterm with three variables
 - Two adjacent squares represent a product term with two variables
 - Four “adjacent” terms represent a product term with one variable
 - Eight “adjacent” terms is the function of all ones (no variables) = 1.

Example: Combining Squares

- Example: $F(A, B) = \sum_m(0,1,2)$

$$F(A, B) = \bar{A}\bar{B} + \bar{A}B + A\bar{B}$$

	<i>B</i>	
	0	1
<i>A</i>	1	1
	2	3
	1	0

- Using Distributive law
 - $F(A, B) = \bar{A} + A\bar{B}$
- Using simplification theorem
 - $F(A, B) = \bar{A} + \bar{B}$
- **Thus, every two adjacent terms that form a 2×1 rectangle correspond to a product term with one variable**

Example: Combining Squares

- Example: $F(x, y, z) = \sum_m(2,3,6,7)$
- $F(x, y, z) = \bar{x}y\bar{z} + \bar{x}yz + xy\bar{z} + xyz$
- Using Distributive law
 - $F(x, y, z) = \bar{x}y + xy$
- Using Distributive law again
 - $F(x, y, z) = y$
- **Thus, the four adjacent terms that form a 2×2 square correspond to the term "y"**

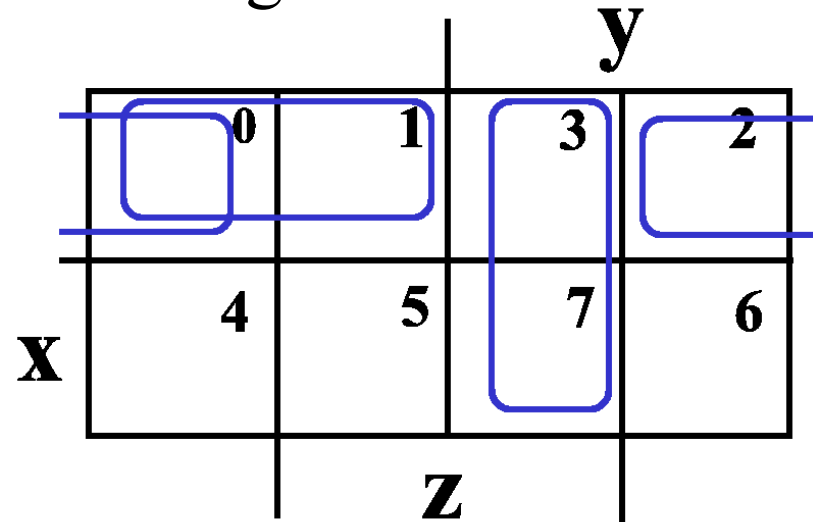
			<i>y</i>	
	0	1	3	2
<i>x</i>	4	5	7	6
			z	
			1	1
			1	1

Three-Variable Maps

- Reduced literal product terms for SOP standard forms correspond to rectangles on K-maps containing cell counts that are powers of 2
- Rectangles of 2 cells represent 2 adjacent minterms
- Rectangles of 4 cells represent 4 minterms that form a “pairwise adjacent” ring
- Rectangles can contain non-adjacent cells as illustrated by the “pairwise adjacent” ring above

Three-Variable Maps

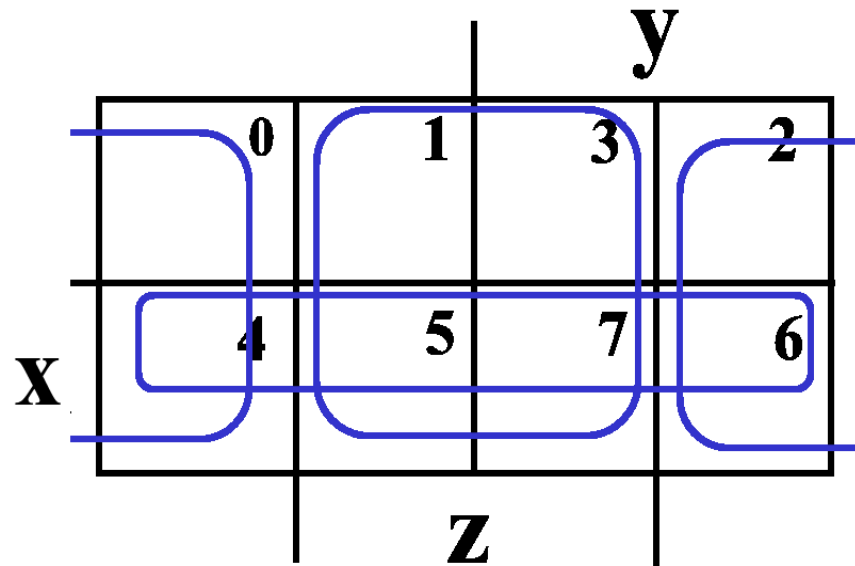
- Example shapes of 2-cell rectangles:



- Read-off the product terms for the rectangles shown:
 - $Rect(0,1) = \bar{X}\bar{Y}$
 - $Rect(0,2) = \bar{X}\bar{Z}$
 - $Rect(3,7) = YZ$

Three-Variable Maps

- Example shapes of 4-cell Rectangles:



- Read off the product terms for the rectangles shown:
 - $Rect(1,3,5,7) = Z$
 - $Rect(0,2,4,6) = \bar{Z}$
 - $Rect(4,5,6,7) = X$

Three Variable Maps

- K-maps can be used to simplify Boolean functions by systematic methods. Terms are selected to cover the “1s” in the map.
- Example: Simplify $F(x, y, z) = \sum_m(1,2,3,5,7)$

			<i>y</i>	
	0	1	3	2
		1	1	1
<i>x</i>	4	5	7	6
		1	1	
			<i>z</i>	

$$F(x, y, z) = z + \bar{x}y$$

Three-Variable Map Simplification

- Use a K-map to find an optimum SOP equation for $F(X, Y, Z) = \sum_m(0,1,2,4,6,7)$

A 2x4 Karnaugh map for variables X, Y, and Z. The vertical axis is labeled X with values 0 and 4. The horizontal axis is labeled Y with values 0, 1, 3, 2 and Z with values 0, 1, 3, 2. The cells contain 1s at (0,0), (0,1), (0,2), (4,0), (4,3), and (4,2). Red dashed ovals group the 1s at (0,0) and (0,1), and the 1s at (4,3) and (4,2). Red solid lines group the 1s at (0,0) and (4,0), and the 1s at (0,2) and (4,2).

					Y
	0	1	3	2	
	1	1		1	
X	4	5	7	6	
	1		1	1	
					Z

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

Four Variable Maps

- Map and location of minterms

$F(W, X, Y, Z)$:

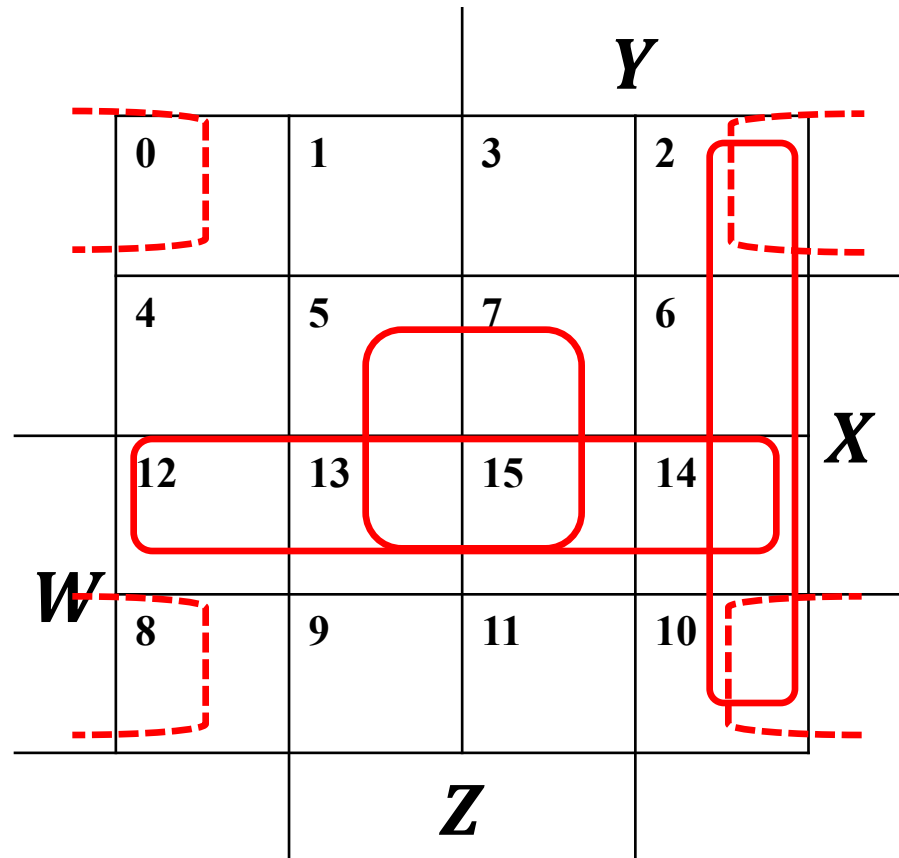
					Y
	0	1	3	2	
	4	5	7	6	
	12	13	15	14	X
W	8	9	11	10	
					Z

Four Variable Terms

- Four variable maps can have rectangles corresponding to:
 - A single 1: 4 variables (i.e. Minterm)
 - Two 1's: 3 variables
 - Four 1's: 2 variables
 - Eight 1's: 1 variable
 - Sixteen 1's: zero variables (function of all ones)

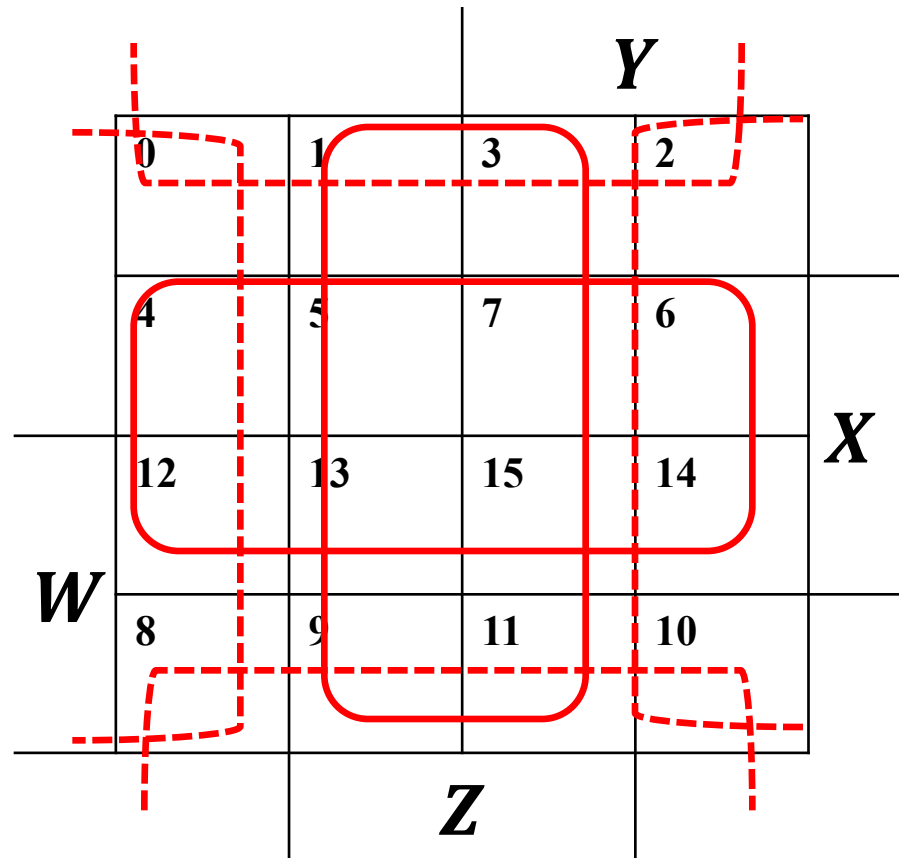
Four-Variable Maps

- Example shapes of 4-cell rectangles:



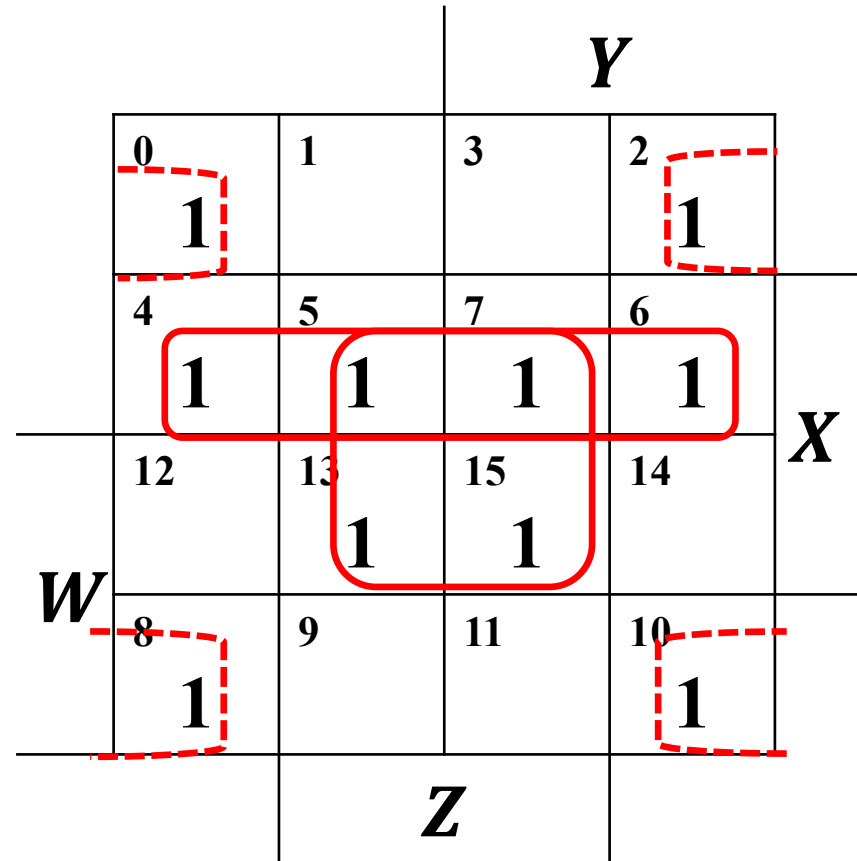
Four-Variable Maps

- Example shapes of 8-cell rectangles:



Four-Variable Map Simplification

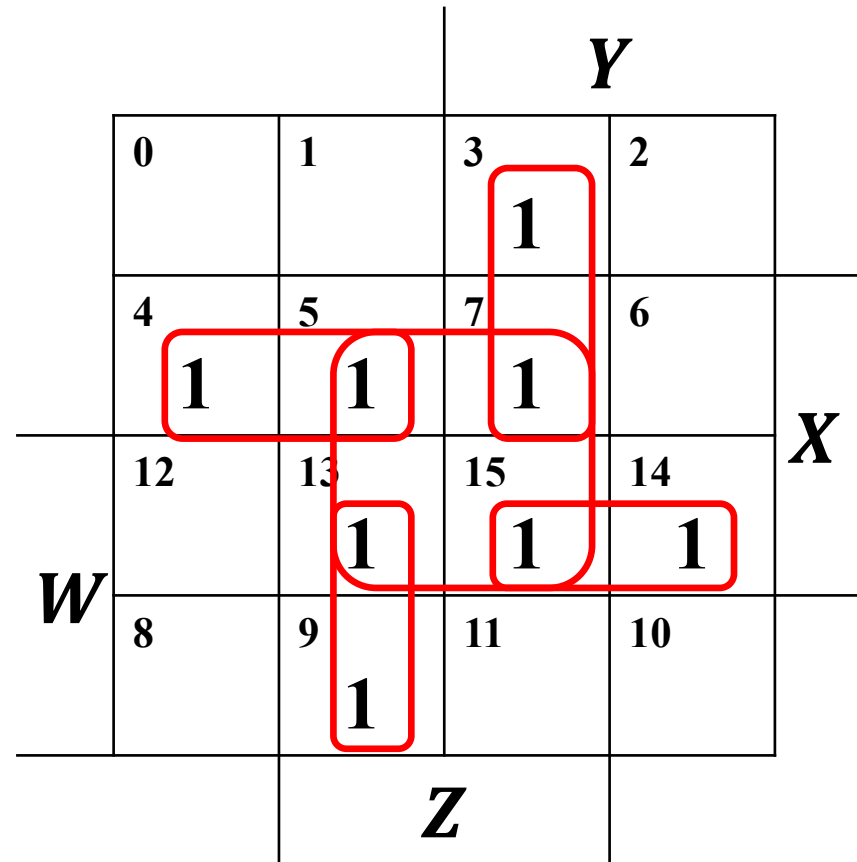
- $F(W, X, Y, Z) = \sum_m(0, 2, 4, 5, 6, 7, 8, 10, 13, 15)$



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

Four-Variable Map Simplification

- $F(W, X, Y, Z) = \sum_m(3, 4, 5, 7, 9, 13, 14, 15)$



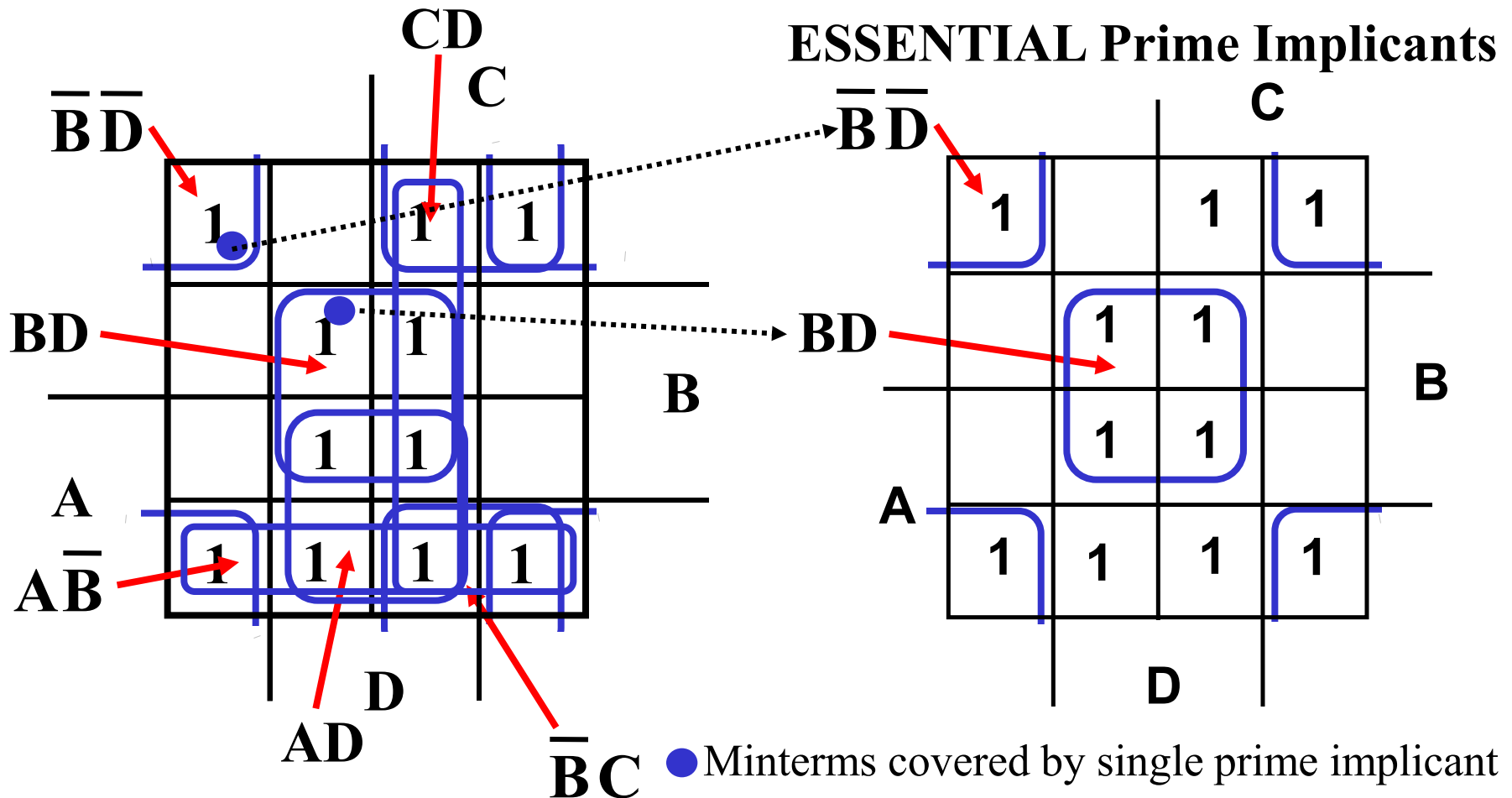
$$F(W, X, Y, Z) = \bar{W}YZ + \bar{W}X\bar{Y} + WXY + W\bar{Y}Z$$

Systematic Simplification

- ***Prime Implicant:*** is a product term obtained by combining the maximum possible number of adjacent squares in the map into a rectangle with the number of squares a power of 2
- A prime implicant is called an ***Essential Prime Implicant*** if it is the only prime implicant that covers (includes) one or more minterms
- Prime Implicants and Essential Prime Implicants can be determined by inspection of a K-Map
- A set of prime implicants "*covers all minterms*" if, for each minterm of the function, at least one prime implicant in the set of prime implicants includes the minterm

Example of Prime Implicants

- Find ALL Prime Implicants



Prime Implicant Practice

- Find all prime implicants for:

$$F(A, B, C, D) = \sum_m (0, 2, 3, 8, 9, 10, 11, 12, 13, 14, 15)$$

- Prime Implicants:

- A
- $\bar{B}C$
- $\bar{B}\bar{D}$

		C			
	0 1	1	3 1	2 1	
	4	5	7	6	
	12 1	13 1	15 1	14 1	B
A	8 1	9 1	11 1	10 1	
		D			

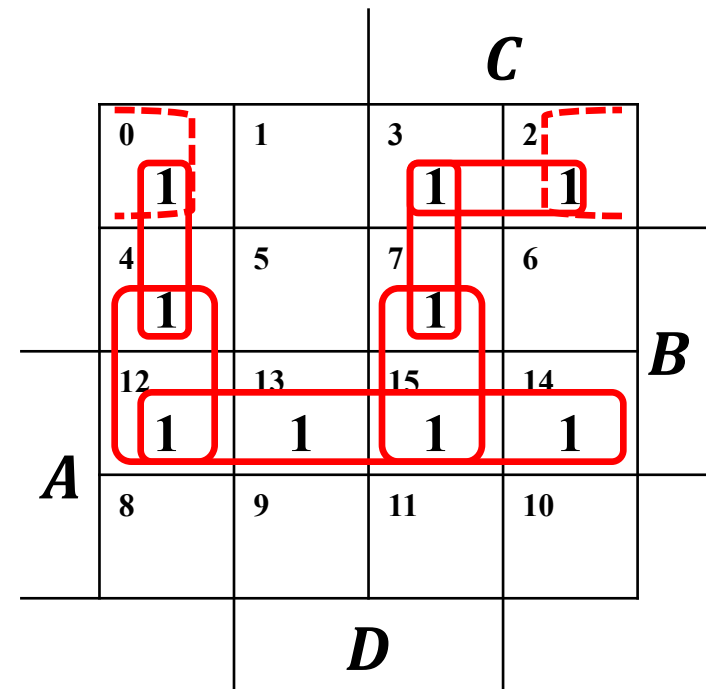
Another Example

- Find all prime implicants for:

$$G(A, B, C, D) = \sum_m (0, 2, 3, 4, 7, 12, 13, 14, 15)$$

- Hint: There are seven prime implicants!
- Prime Implicants:

- AB
- BCD
- $B\bar{C}\bar{D}$
- $\bar{A}CD$
- $\bar{A}\bar{C}\bar{D}$
- $\bar{A}\bar{B}C$
- $\bar{A}\bar{B}\bar{D}$

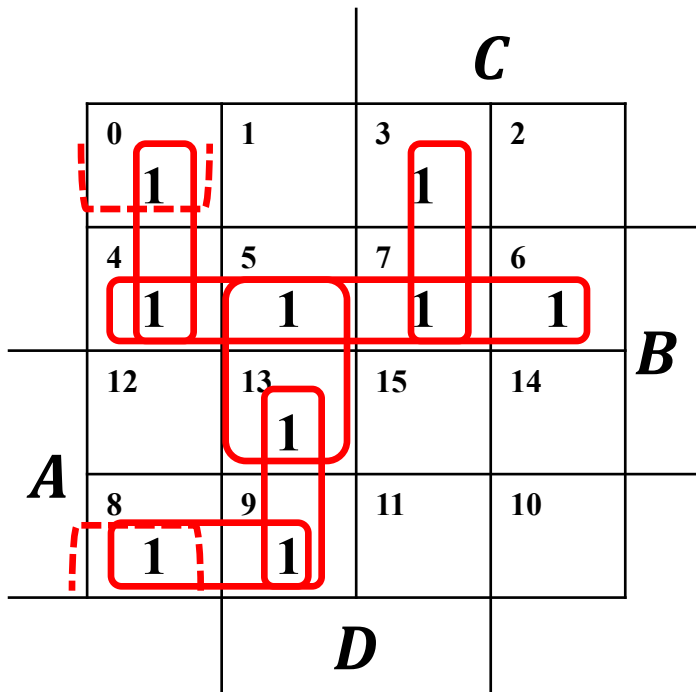


Optimization Algorithm

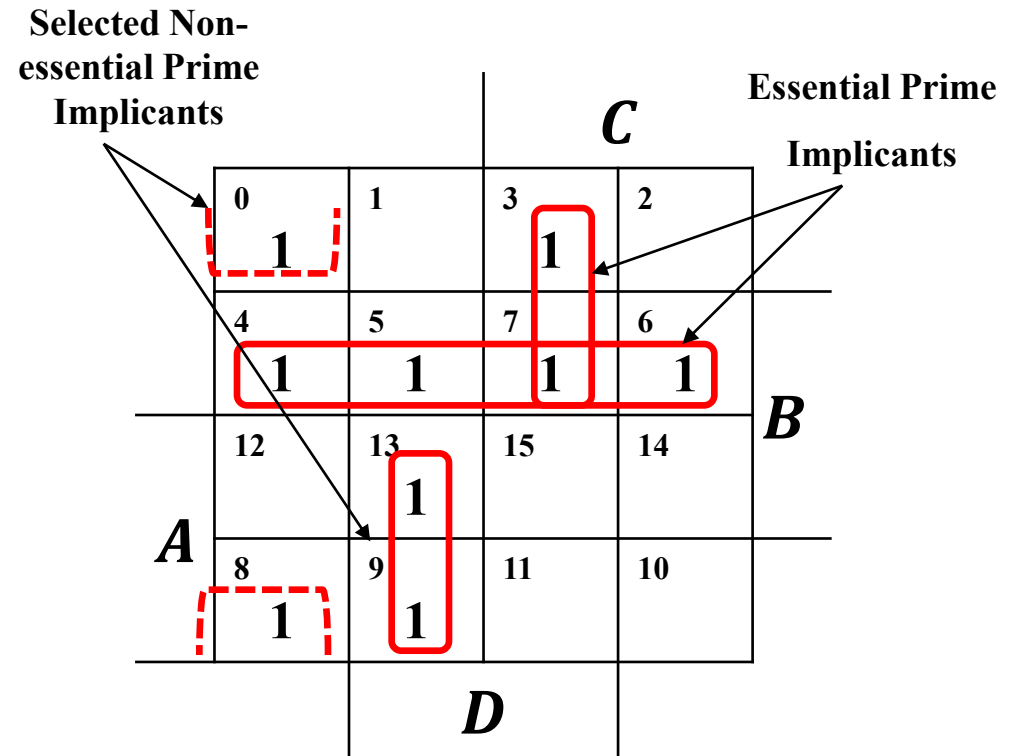
1. Find all prime implicants
2. Include all essential prime implicants in the solution
3. Select a *minimum cost* set of non-essential prime implicants to cover all minterms not yet covered
 - Selection Rule: Minimize the overlap among prime implicants as much as possible. In particular, in the final solution, make sure that each prime implicant selected includes at least one minterm not included in any other prime implicant selected

Selection Rule Example

- Simplify $F(A, B, C, D)$ given on the K-map



Prime Implicants



Essential and Selected Non-essential Prime Implicants

Product of Sums Example

- Find the optimum POS solution for:

$$F(A, B, C, D) = \sum_m (1, 3, 9, 11, 12, 13, 14, 15)$$

- Solution:

- Find optimized SOP for \bar{F} by combining 0's in K-Map of F
- Complement \bar{F} to obtain optimized POS for F

- $\bar{F}(A, B, C, D) = \bar{A}B + \bar{B}\bar{D}$

- Using Demorgan's Law:

$$F(A, B, C, D) = (A + \bar{B})(B + D)$$

	C				
	0	1	3	2	
	0	1	1	0	
	4	5	7	6	
	0	0	0	0	
	12	13	15	14	B
	1	1	1	1	
A	8	9	11	10	
	0	1	1	0	
		D			

Example

- Find the optimum POS and SOP solution for:

$$F(A, B, C, D) = \prod_M (0, 2, 4, 5, 6, 7)$$

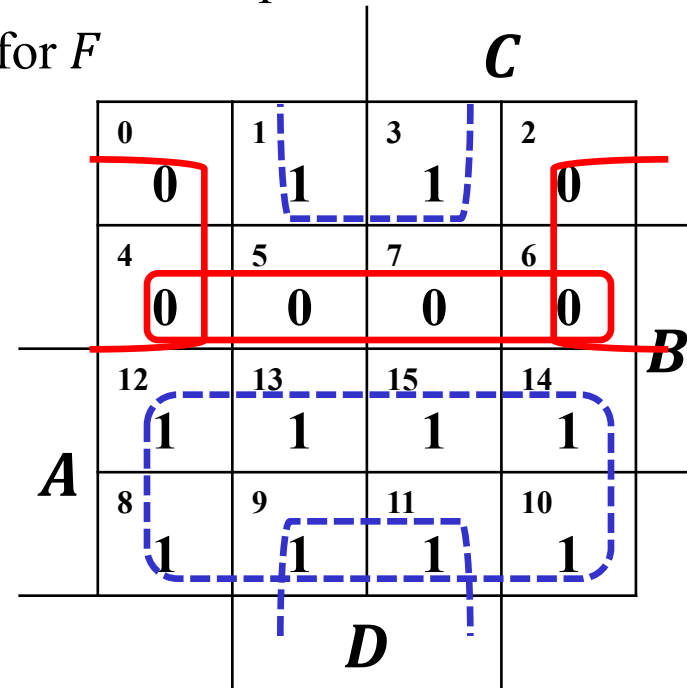
- POS solution (Red):
 - Find optimized SOP for \bar{F} by combining 0's in K-Map of F
 - Complement \bar{F} to obtain optimized POS for F

$$\bar{F}(A, B, C, D) = \bar{A}B + \bar{A}\bar{D}$$

$$F(A, B, C, D) = (A + \bar{B})(A + D)$$

- SOP solution (Blue):
 - Combining 1's in K-Map of F

$$F(A, B, C, D) = A + \bar{B}D$$



Don't Cares in K-Maps

- Incompletely specified functions: Sometimes a function table or map contains entries for which it is known:
 - the input values for the minterm will never occur, or
 - The output value for the minterm is not used
- In these cases, the output value is defined as a “don't care”
- By placing “don't cares” (an “x” entry) in the function table or map, the cost of the logic circuit may be lowered
- **Example:** A logic function having the binary codes for the BCD digits as its inputs. Only the codes for 0 through 9 are used. The six codes, 1010 through 1111 never occur, so the output values for these codes are “x” to represent “don't cares”
- ***“Don't care” minterms cannot be replaced with 1's or 0's because that would require the function to be always 1 or 0 for the associated input combination***

Example: BCD “5 or More”

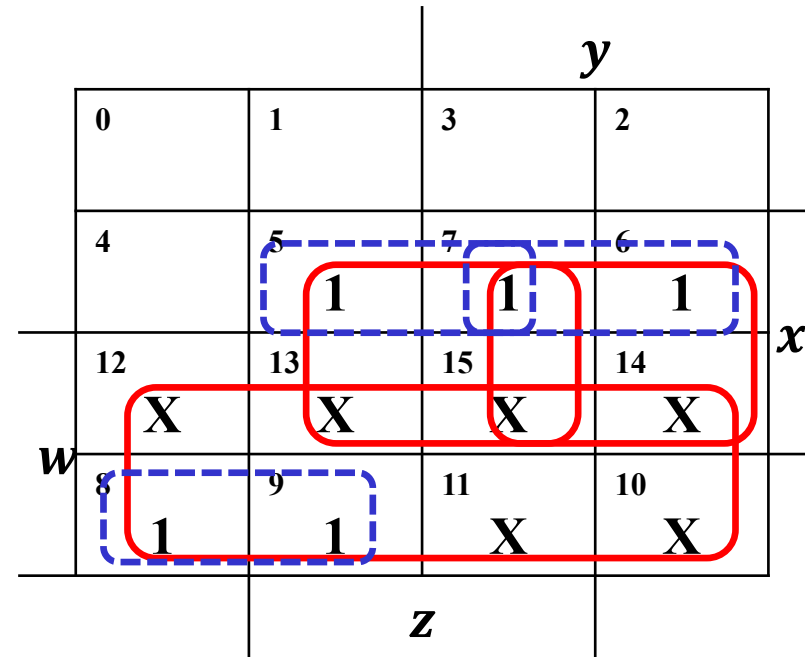
- The map below gives a function $F(w, x, y, z)$ which is defined as "5 or more" over BCD inputs. With the don't cares used for the 6 non-BCD combinations:

- If don't cares are treated as 1's (Red):

- $$F_1(w, x, y, z) = w + xy + xz$$
 - $G = 7$

- If don't cares are treated as 0's (Blue):

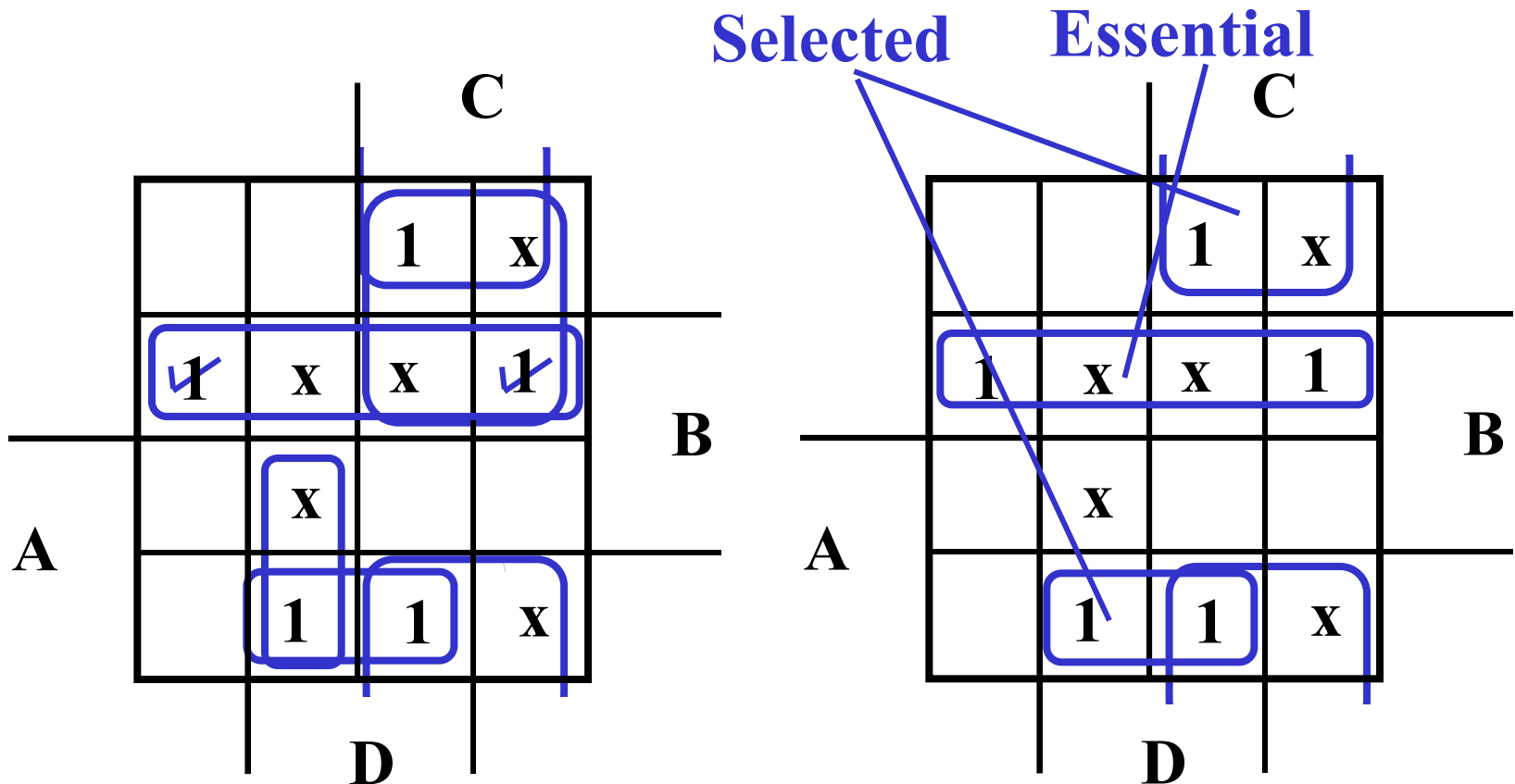
- $$F_2(w, x, y, z) = \bar{w}xz + \bar{w}xy + w\bar{x}\bar{y}$$
 - $G = 12$



- For this particular function, cost G for the POS solution for $F(w, x, y, z)$ is not changed by using the don't cares***
 - Choose the one less inverters (i.e. less GN)***

Selection Rule Example with Don't Cares

- Simplify $F(A, B, C, D)$ given on the K-map.



✓ Minterms covered by essential prime implicants

Product of Sums with Don't Care Example

- Find the optimum **POS** solution for:

$$F(A, B, C, D) = \sum_m (3, 9, 11, 12, 13, 14, 15) + \sum_d (1, 4, 6)$$

	<i>C</i>				
	0	1	3	2	
	0	X	1	0	
	4	5	7	6	
	X	0	0	X	<i>B</i>
	12	13	15	14	
<i>A</i>	1	1	1	1	
	8	9	11	10	
	0	1	1	0	<i>D</i>

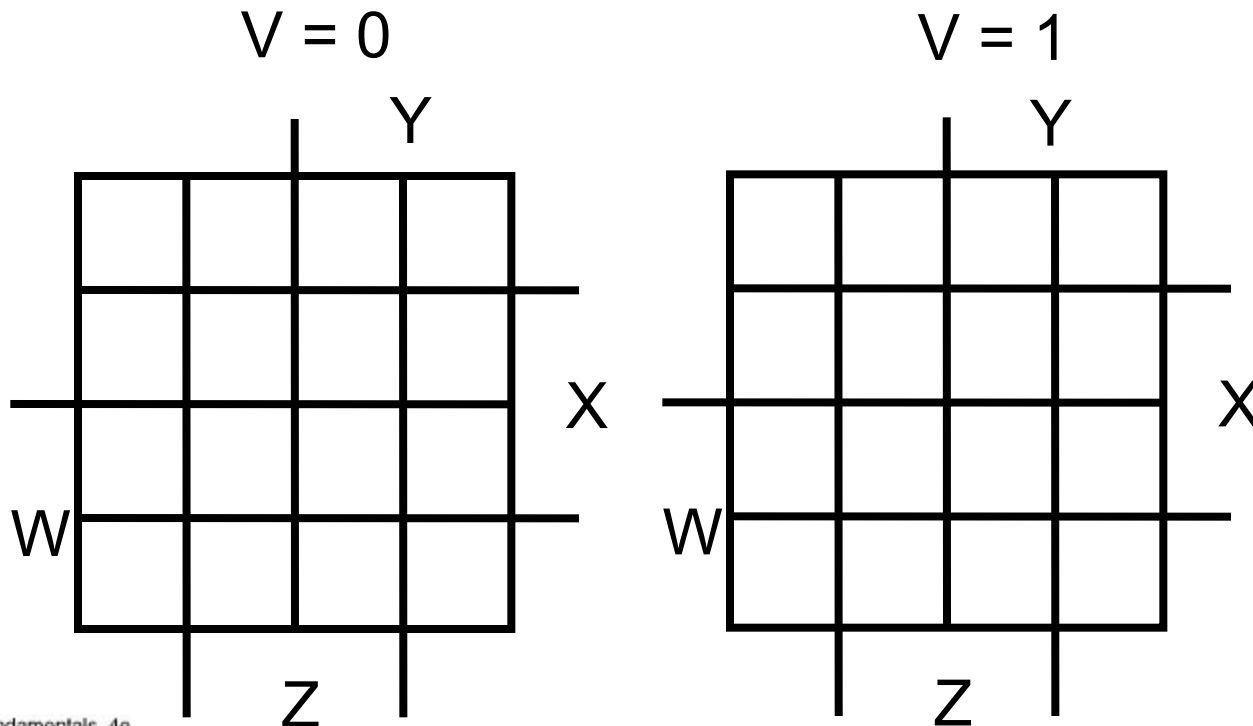
	<i>C</i>				
	0	1	3	2	
	0	X	1	0	
	4	5	7	6	
	X	0	0	X	<i>B</i>
	12	13	15	14	
<i>A</i>	1	1	1	1	
	8	9	11	10	
	0	1	1	0	<i>D</i>

$$\bar{F}(A, B, C, D) = \bar{A}B + \bar{B}\bar{D}$$

$$F(A, B, C, D) = (A + \bar{B})(B + D)$$

Five Variable or More K-Maps

- For five variable problems, we use *two adjacent K-maps*. It becomes harder to visualize adjacent minterms for selecting PIs. You can extend the problem to six variables by using four K-Maps.



Terms of Use

- **All (or portions) of this material © 2008 by Pearson Education, Inc.**
- **Permission is given to incorporate this material or adaptations thereof into classroom presentations and handouts to instructors in courses adopting the latest edition of Logic and Computer Design Fundamentals as the course textbook.**
- **These materials or adaptations thereof are not to be sold or otherwise offered for consideration.**
- **This Terms of Use slide or page is to be included within the original materials or any adaptations thereof.**

Logic and Computer Design Fundamentals

Chapter 2 – Combinational Logic Circuits

Part 3 – Additional Gates and Circuits

Charles Kime & Thomas Kaminski

© 2008 Pearson Education, Inc.

(Hyperlinks are active in View Show mode)

Updated by Dr. Waleed Dweik

Overview

- **Part 1 – Gate Circuits and Boolean Equations**
 - **Binary Logic and Gates**
 - **Boolean Algebra**
 - **Standard Forms**
- **Part 2 – Circuit Optimization**
 - **Two-Level Optimization**
 - **Map Manipulation**
- **Part 3 – Additional Gates and Circuits**
 - **Other Gate Types**
 - **Exclusive-OR Operator and Gates**
 - **High-Impedance Outputs**

Other Gate Types

- **Why?**

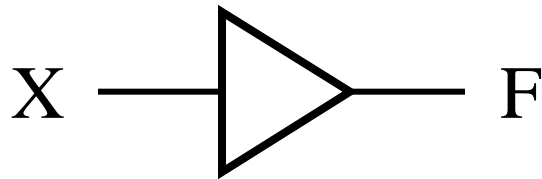
- Implementation feasibility and low cost
- Power in implementing Boolean functions
- Convenient conceptual representation

- **Gate classifications:**

- **Primitive gate:** a gate that can be described using a single primitive operation type (AND or OR) plus an optional inversion(s).
- **Complex gate:** a gate that requires more than one primitive operation type for its description

Buffer

- A *buffer* is a gate with the function $F = X$:

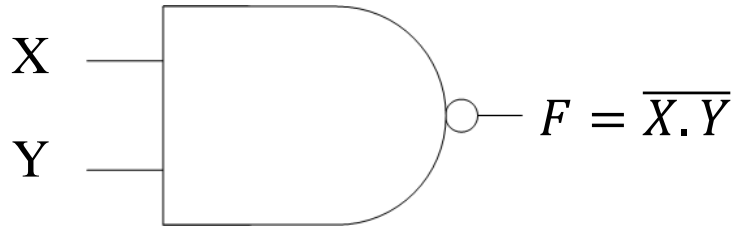


X	F
0	0
1	1

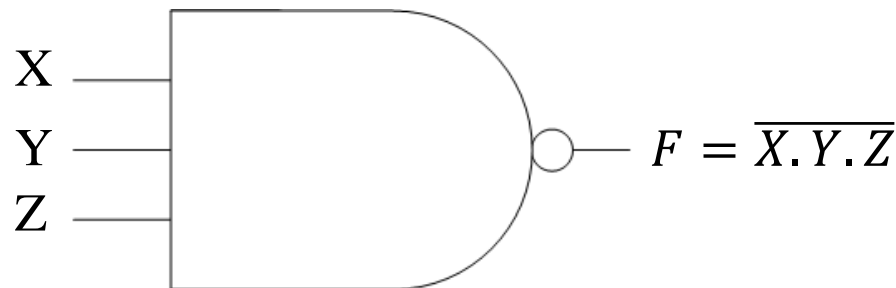
- In terms of Boolean function, a buffer is the same as a connection!
- **So why use it?**
 - A buffer is an electronic amplifier used to improve circuit voltage levels and increase the speed of circuit operation
 - Protection and isolation between circuits

NAND Gate

- The NAND gate has the following symbol and truth table:



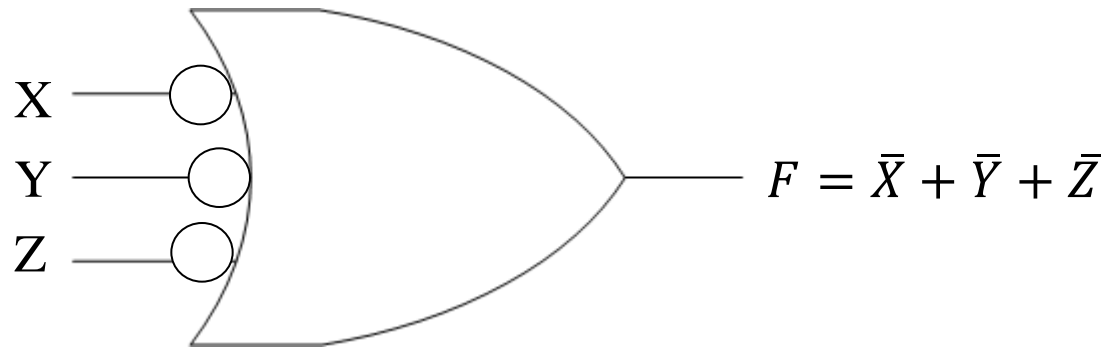
<i>X</i>	<i>Y</i>	<i>F</i>
<i>0</i>	<i>0</i>	<i>1</i>
<i>0</i>	<i>1</i>	<i>1</i>
<i>1</i>	<i>0</i>	<i>1</i>
<i>1</i>	<i>1</i>	<i>0</i>



- NAND** represents **NOT-AND**, i.e., the AND function with a NOT applied. The symbol shown is an **AND-Invert**. The small circle (“bubble”) represents the invert function

NAND Gates (continued)

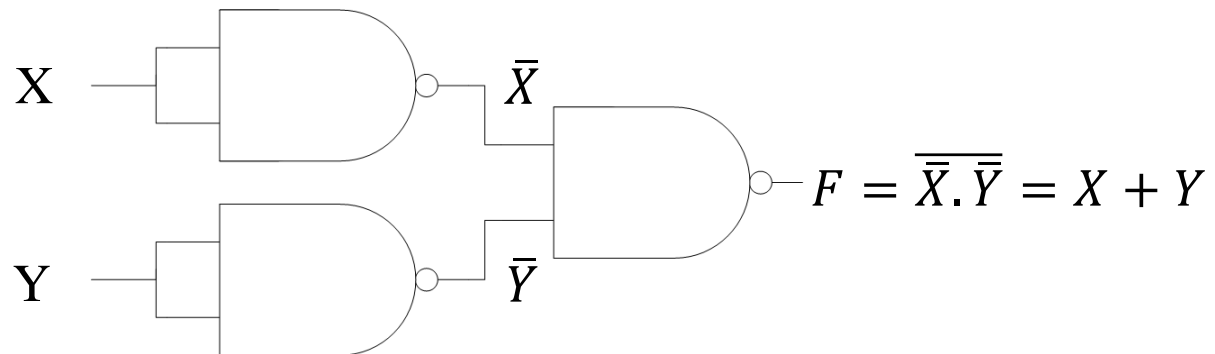
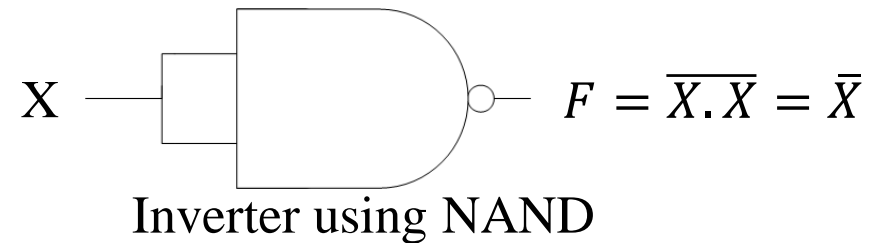
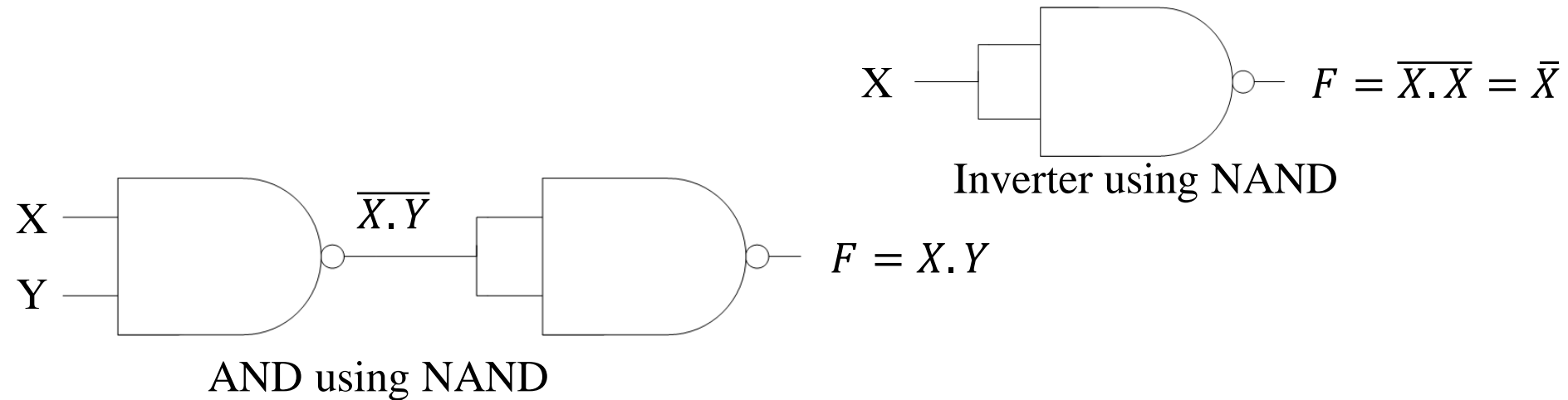
- Applying DeMorgan's Law gives **Invert-OR** (NAND)



- This NAND symbol is called **Invert-OR**, since inputs are inverted and then ORed together
- **AND-Invert** and **Invert-OR** both represent the NAND gate. Having both makes visualization of circuit function easier

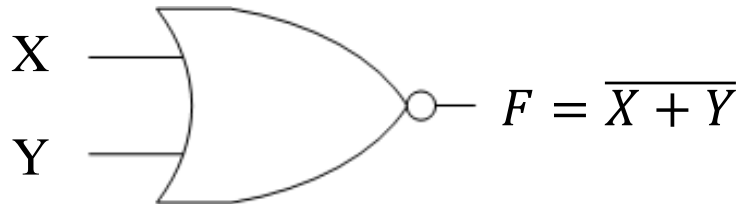
NAND Gates (continued)

- **Universal gate:** a gate type that can implement any Boolean function. **The NAND gate is a universal gate:**

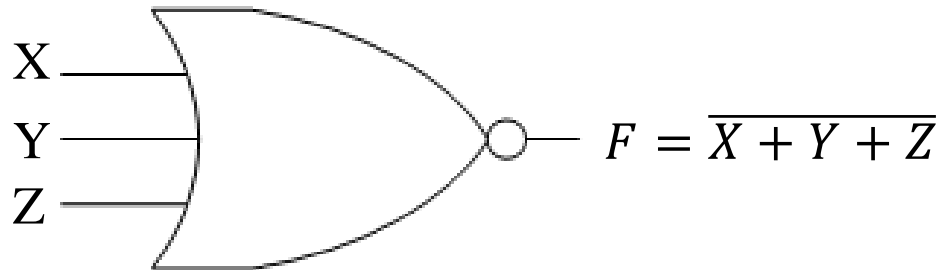


NOR Gate

- The NOR gate has the following symbol and truth table:



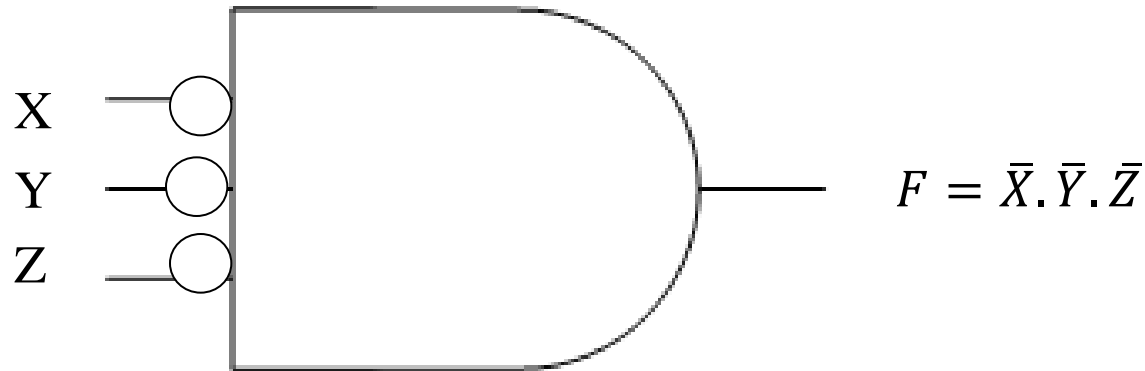
<i>X</i>	<i>Y</i>	<i>F</i>
<i>0</i>	<i>0</i>	<i>1</i>
<i>0</i>	<i>1</i>	<i>0</i>
<i>1</i>	<i>0</i>	<i>0</i>
<i>1</i>	<i>1</i>	<i>0</i>



- NOR** represents **NOT-OR**, i.e., the OR function with a NOT applied. The symbol shown is an **OR-Invert**. The small circle (“bubble”) represents the invert function

NOR Gates (continued)

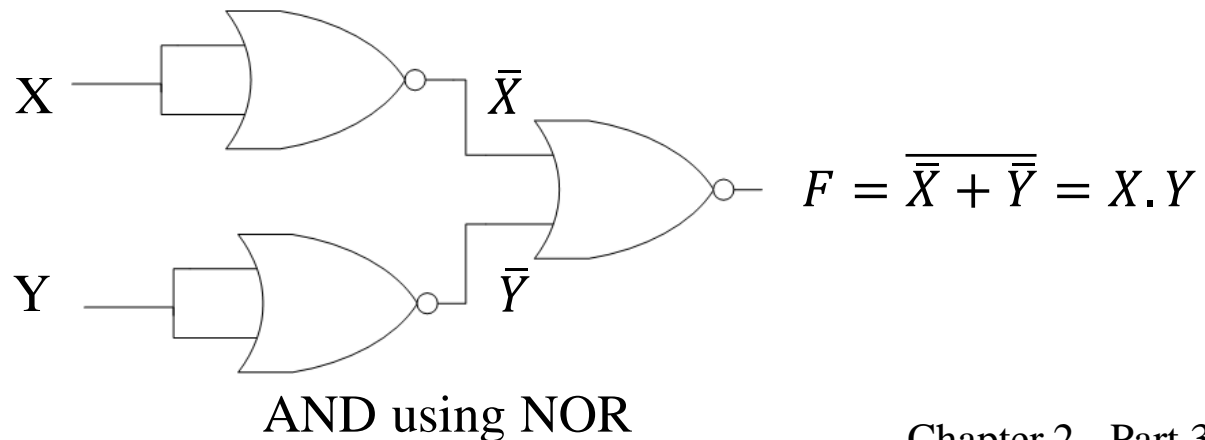
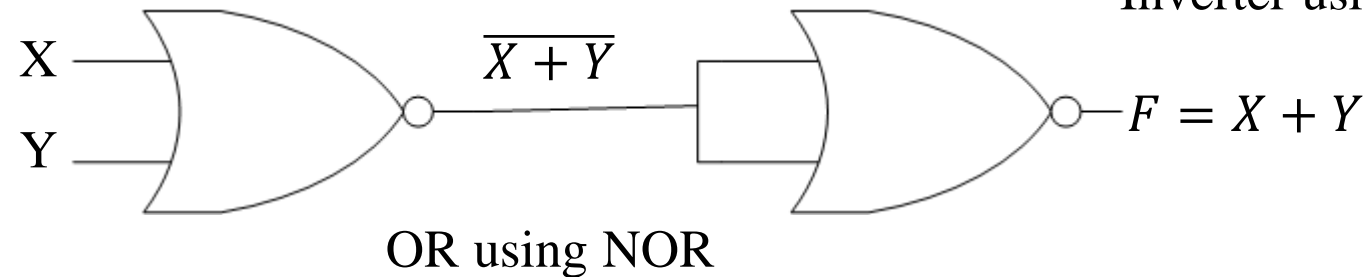
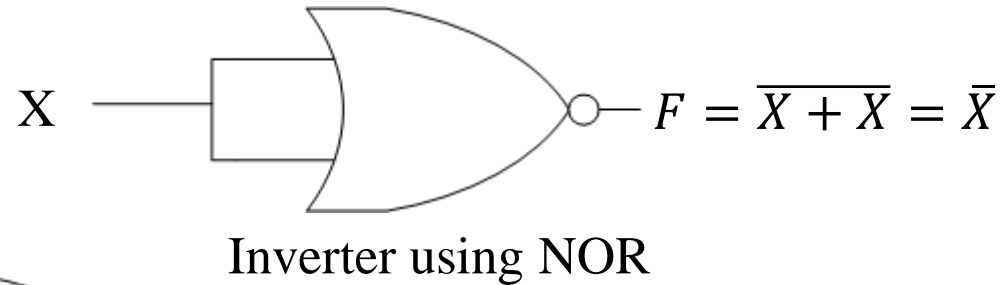
- Applying DeMorgan's Law gives **Invert-AND** (NOR)



- This NOR symbol is called **Invert-AND**, since inputs are inverted and then ANDed together
- OR-Invert** and **Invert-AND** both represent the NOR gate. Having both makes visualization of circuit function easier

NOR Gates (continued)

- The NOR gate is a universal gate:



Hi-Impedance Outputs

- Logic gates introduced thus far
 - have 1 and 0 output values,
 - cannot have their outputs connected together, and
 - transmit signals on connections in only one direction
- Three-state logic adds a third logic value, **Hi-Impedance (Hi-Z)**, giving three states: 0, 1, and Hi-Z on the outputs.
- *Hi-Z can be also denoted as Z or z*
- The presence of a Hi-Z state makes a gate output as described above behave quite differently:
 - “1 and 0” become “1, 0, and Hi-Z”
 - “cannot” becomes “can,” and
 - “only one” becomes “two”

Hi-Impedance Outputs (continued)

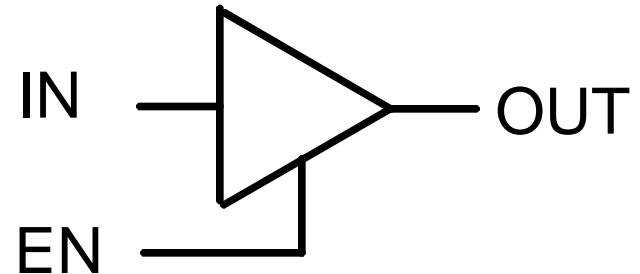
- **What is a Hi-Z value?**
 - The Hi-Z value behaves as an open circuit
 - This means that, looking back into the circuit, the output appears to be disconnected
 - It is as if a switch between the internal circuitry and the output has been opened

- Hi-Z may appear on the output of any gate, but we restrict gates to **3-state buffer**

Tri-State Buffer (3-State Buffer)

- For the symbol and truth table, **IN** is the data input, and **EN** is the control input
- For **EN = 0**, regardless of the value on IN (denoted by X), the output value is Hi-Z
- For **EN = 1**, the output value follows the input value

Symbol

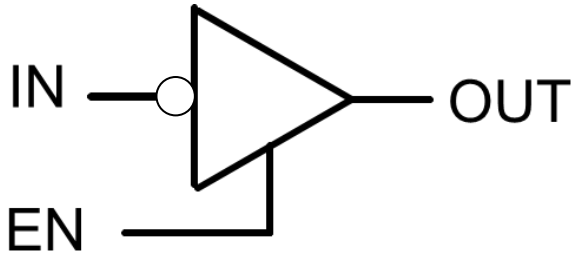


Truth Table

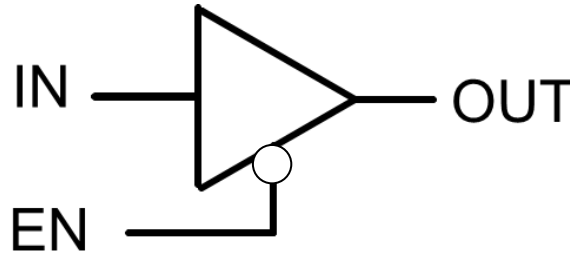
<i>EN</i>	<i>IN</i>	<i>OUT</i>
<i>0</i>	<i>X</i>	<i>Hi-Z</i>
<i>1</i>	<i>0</i>	<i>0</i>
<i>1</i>	<i>1</i>	<i>1</i>

Tri-State Buffer Variations

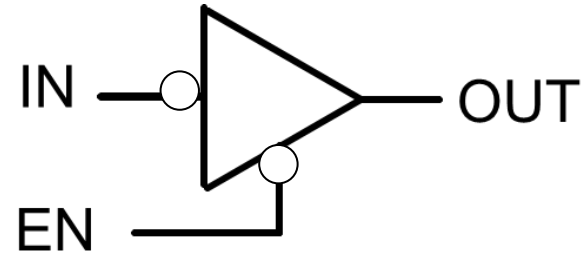
- By adding “bubbles” to signals:
 - Data input, IN, can be inverted
 - Control input, EN, can be inverted



<i>EN</i>	<i>IN</i>	<i>OUT</i>
<i>0</i>	<i>X</i>	<i>Hi-Z</i>
<i>1</i>	<i>0</i>	<i>1</i>
<i>1</i>	<i>1</i>	<i>0</i>



<i>EN</i>	<i>IN</i>	<i>OUT</i>
<i>0</i>	<i>0</i>	<i>0</i>
<i>0</i>	<i>1</i>	<i>1</i>
<i>1</i>	<i>X</i>	<i>Hi-Z</i>

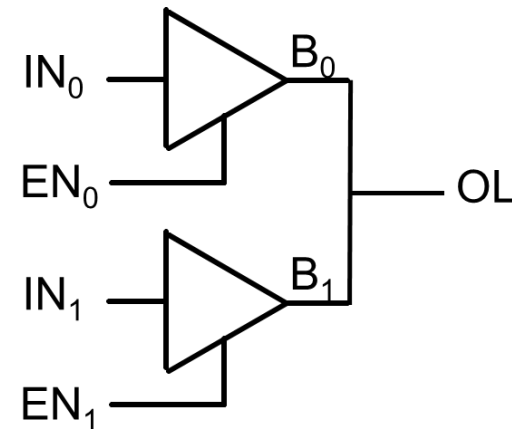


<i>EN</i>	<i>IN</i>	<i>OUT</i>
<i>0</i>	<i>0</i>	<i>1</i>
<i>0</i>	<i>1</i>	<i>0</i>
<i>1</i>	<i>X</i>	<i>Hi-Z</i>

Resolving 3-State Values on a Connection

- Connection of two tri-state buffer outputs, B_1 and B_0 , to a wire, OL (Output Line) \rightarrow Multiplexed Output

EN_1	EN_0	IN_1	IN_0	B_1	B_0	OL
0	0	X	X	Hi-Z	Hi-Z	Hi-Z
0	1	X	0	Hi-Z	0	0
0	1	X	1	Hi-Z	1	1
1	0	0	X	0	Hi-Z	0
1	0	1	X	1	Hi-Z	1
1	1	0	0	0	0	0
1	1	1	1	1	1	1
1	1	0	1	0	1	Fire
1	1	1	0	1	0	Fire



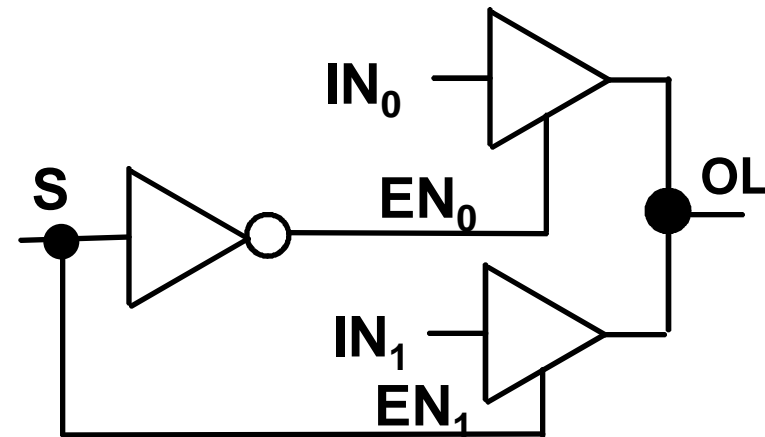
Resolving 3-State Values on a Connection

- **Resulting Rule: At least one buffer output value must be Hi-Z. Why?**
 - Because any data combinations including (0,1) and (1,0) can occur. If one of these combinations occurs, and no buffers are Hi-Z, then high currents can occur, destroying or damaging the circuit
- **How many valid buffer output combinations exist?**
 - 5 valid output combination
- **What is the rule for “ n ” tri-state buffers connected to wire, OL?**
 - At least “ $n-1$ ” buffer outputs must be Hi-Z
 - **How many valid buffer output combinations exist ?**
 - Each of the n -buffers can have a 0 or 1 output with all others at Hi-Z. Also all buffers can be Hi-Z. So there are $2n + 1$ valid combinations.

Tri-State Logic Circuit

- **Data Selection Function:** If $s = 0$, $OL = IN_0$, else $OL = IN_1$
- Performing data selection with tri-state buffers:

EN_1	EN_0	IN_1	IN_0	OL
0	1	X	0	0
0	1	X	1	1
1	0	0	X	0
1	0	1	X	1



- Since $EN_0 = \bar{s}$ and $EN_1 = s$, one of the two buffer outputs is always Hi-Z.

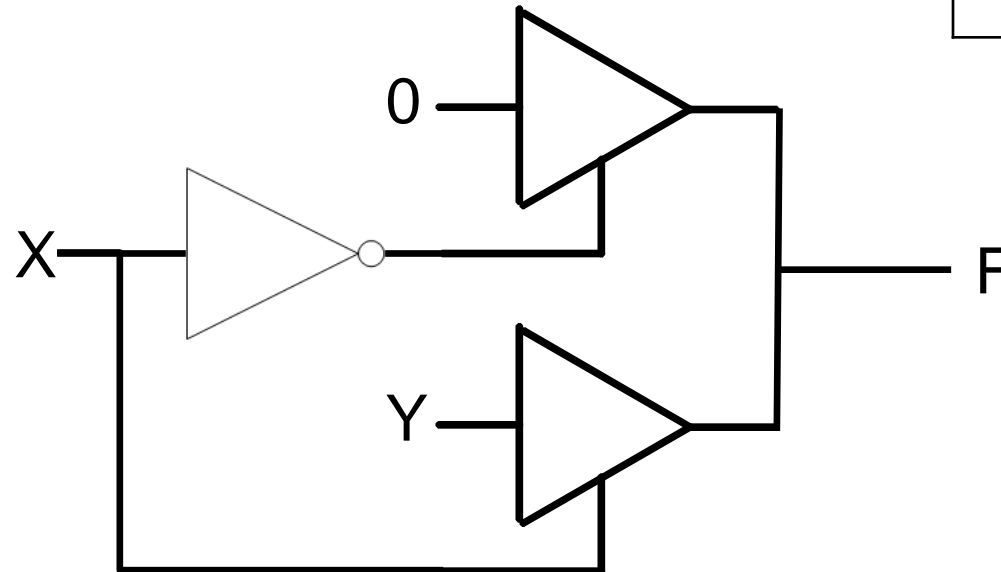
Logic Functions using Tri-State Buffers

- Implement AND gate using 3-State buffers and inverters

$$F(X, Y) = X \cdot Y$$

- Use X as control input:
 - When $X = 0$, $F = 0$ regardless of the value of Y
 - When $X = 1$, $F = Y$

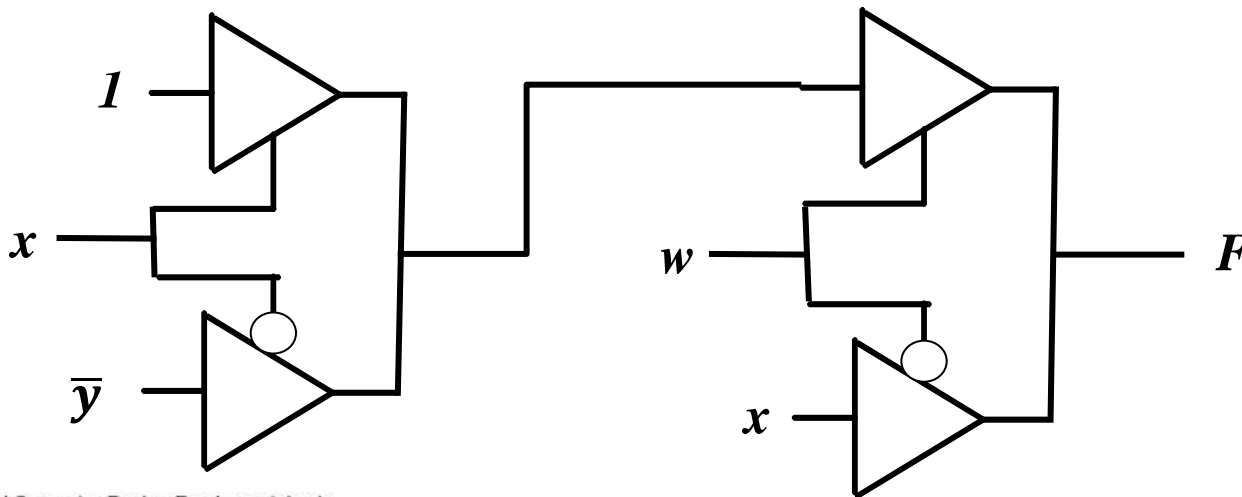
X	Y	F
0	0	0
0	1	0
1	0	0
1	1	1



Logic Functions using Tri-State Buffers

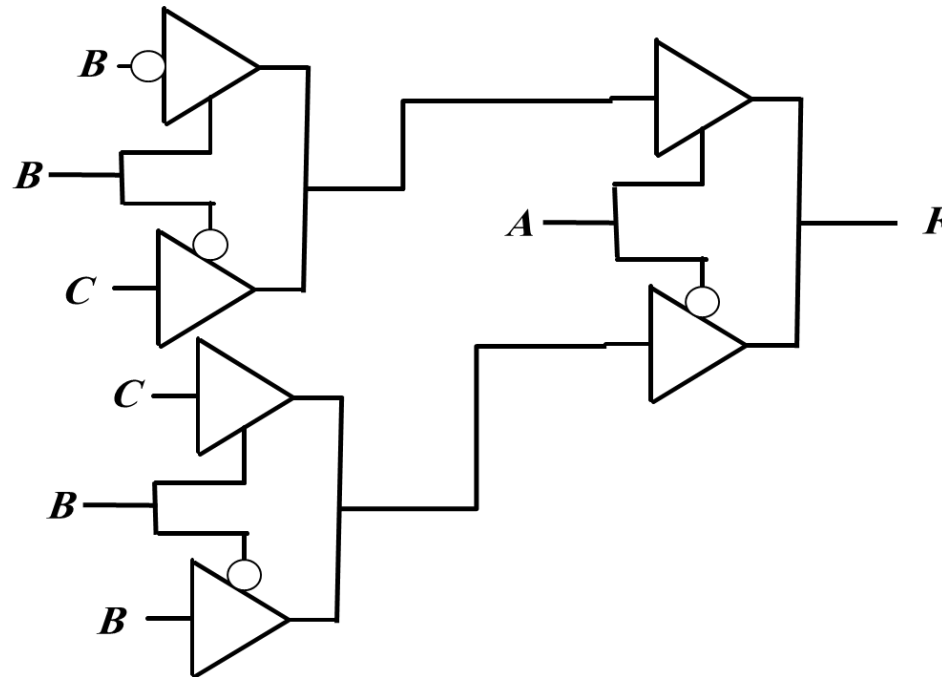
- Implement the following function using 3-State buffers and inverters: $F(w, x, y) = \bar{w}x + w\bar{y} + xy$
- Use w as control input:
 - When $w = 0$, $F = x$ regardless of the value of Y
 - When $w = 1$
 - If $x = 0$, $F = \bar{y}$
 - If $x = 1$, $F = 1$

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



Logic Functions using Tri-State Buffers

- Write the Boolean expression of $F(A, B, C)$ given the diagram below:



$$F(A, B, C) = A\bar{B}C + \bar{A}BC$$

Exclusive OR/ Exclusive NOR

- The *eXclusive OR (XOR)* function is an important Boolean function used extensively in logic circuits
- The XOR function may be:
 - implemented directly as an electronic circuit (truly a gate) or
 - implemented by interconnecting other gate types (used as a convenient representation)
- The *eXclusive NOR (XNOR)* function is the complement of the XOR function
- By our definition, XOR and XNOR gates are *complex gates*

Exclusive OR/ Exclusive NOR

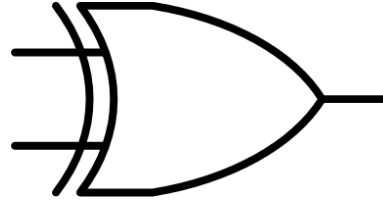
- Uses for the XOR and XNORs gate include:
 - Adders/subtractors/multipliers
 - Counters/incrementers/decrementers
 - Parity generators/checkers
- Definitions
 - The XOR function is: $X \oplus Y = \bar{X}Y + X\bar{Y}$
 - The XNOR function is: $X \odot Y = \overline{X \oplus Y} = XY + \bar{X}\bar{Y}$
- Strictly speaking, XOR and XNOR gates ***do not exist for more than two inputs***. Instead, they are replaced by odd and even functions

Proof: XNOR is the complement of XOR

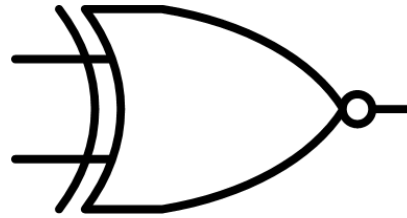
- $\overline{X \oplus Y} = \overline{\bar{X}Y + X\bar{Y}}$
- $\overline{X \oplus Y} = \overline{\bar{X}Y} \cdot \overline{X\bar{Y}}$
- $\overline{X \oplus Y} = (X + \bar{Y})(\bar{X} + Y)$
- $\overline{X \oplus Y} = X\bar{X} + X\bar{Y} + \bar{X}Y + Y\bar{Y}$
- $X \odot Y = \overline{X \oplus Y} = X\bar{Y} + \bar{X}Y$

Symbols For XOR and XNOR

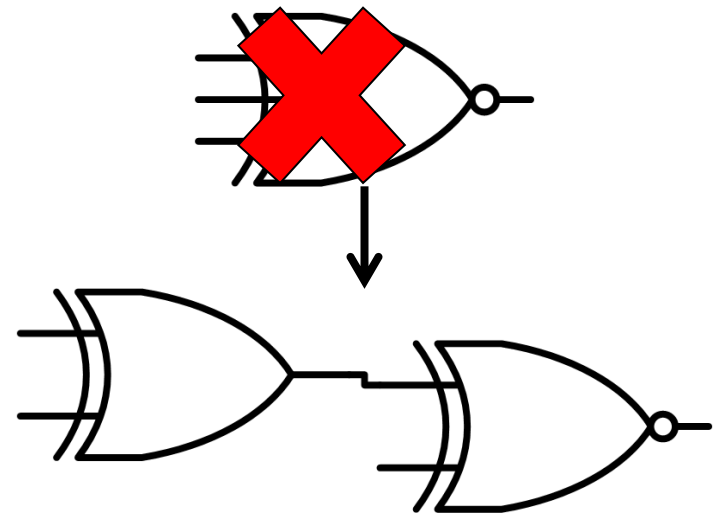
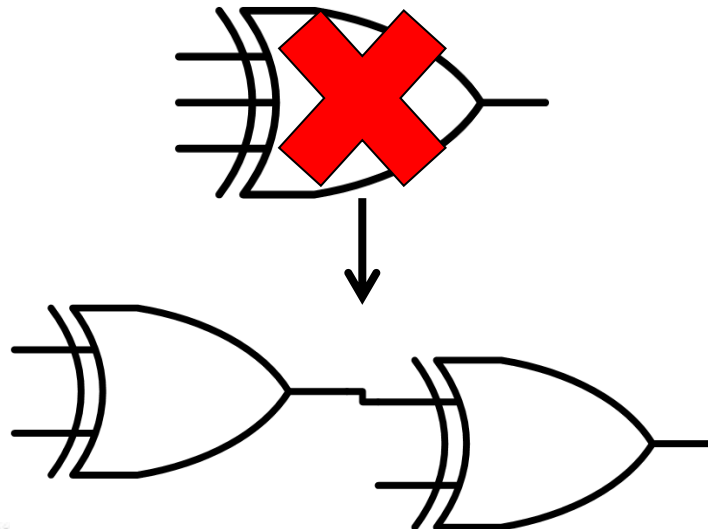
- XOR symbol:



- XNOR symbol:



- *Shaped symbols exist only for two inputs*



Truth Tables for XOR/XNOR

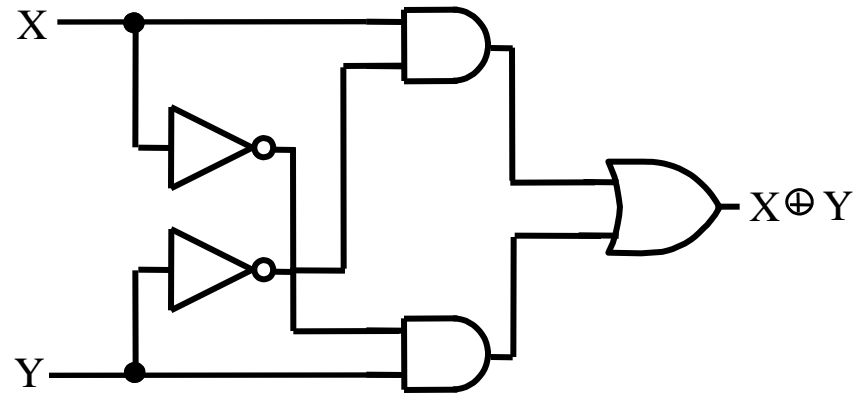
X	Y	$X \oplus Y$
0	0	0
0	1	1
1	0	1
1	1	0

X	Y	$X \odot Y (X \equiv Y)$
0	0	1
0	1	0
1	0	0
1	1	1

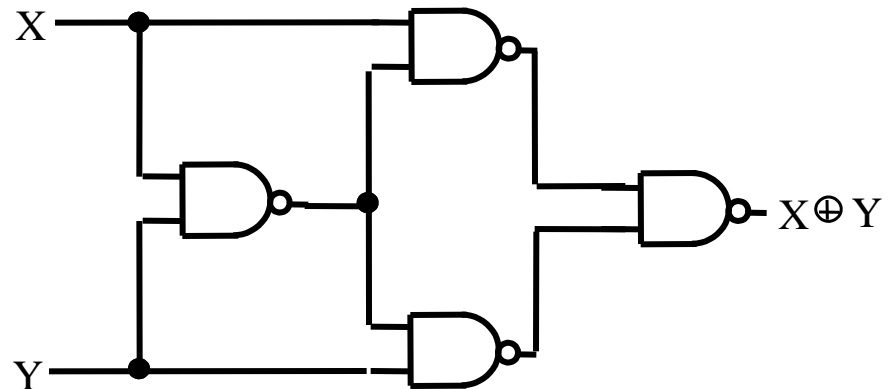
- The XOR function means: ***X OR Y, but NOT BOTH***
- Why is the XNOR function also known as the ***equivalence*** function, denoted by the operator \equiv ?
 - Because the function equals 1 if and only if **$X = Y$**

XOR Implementations

- The simple SOP implementation uses the following structure:



- A NAND only implementation is:



XOR

- The XOR identities:

$X \oplus 0 = X$	$X \oplus 1 = \bar{X}$
$X \oplus X = 0$	$X \oplus \bar{X} = 1$
$X \oplus \bar{Y} = \bar{X} \oplus Y$	$\bar{X} \oplus Y = \bar{X} \oplus Y$
$X \oplus Y = Y \oplus X$	
$(X \oplus Y) \oplus Z = X \oplus (Y \oplus Z) = X \oplus Y \oplus Z$	

- The XOR function can be extended to 3 or more variables. For more than 2 variables, it is called an *odd function* or *modulo 2 sum* (*Mod 2 sum*), not an XOR:

$$X \oplus Y \oplus Z = \bar{X}\bar{Y}Z + \bar{X}Y\bar{Z} + X\bar{Y}\bar{Z} + XYZ \text{ (Odd \# of 1's)}$$

XNOR

- The XNOR identities:

$X \odot 0 = \bar{X}$	$X \odot 1 = X$
$X \odot X = 1$	$X \odot \bar{X} = 0$
$X \odot Y = Y \odot X$	
$X \odot Y \odot Z = (X \oplus Y) \odot Z = X \odot (Y \oplus Z)$	

- The XNOR function can be extended to 3 or more variables. For more than 2 variables, it is called an **even function**, not an XNOR:

$$X \odot Y \odot Z = \bar{X}YZ + X\bar{Y}Z + XY\bar{Z} + \bar{X}\bar{Y}\bar{Z} \text{ (Even \# of 1's)}$$

- ***The even function is the complement of the odd function***

Odd and Even Functions

- The 1s of an *odd function* correspond to minterms having an index with an odd number of 1s.

		<i>y</i>		
	0	1	3	2
		1		1
<i>x</i>	4	5	7	6
	1		1	
		<i>z</i>		

		<i>C</i>		
	0	1	3	2
		1		1
	4	5	7	6
	1		1	
<i>B</i>	12	13	15	14
		1		1
<i>A</i>	8	9	11	10
	1		1	
		<i>D</i>		

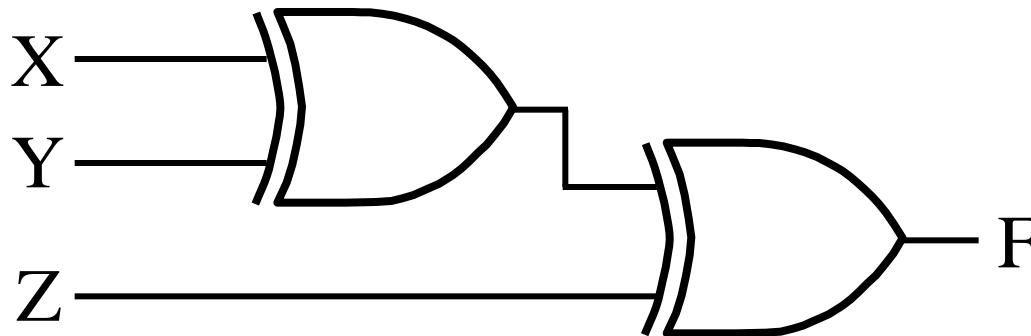
- The 1s of an *even function* correspond to minterms having an index with an even number of 1s.

		<i>y</i>		
	0	1	3	2
	1		1	
<i>x</i>	4	5	7	6
		1		1
		<i>z</i>		

		<i>C</i>		
	0	1	3	2
	1		1	
	4	5	7	6
		1		1
<i>B</i>	12	13	15	14
	1		1	
<i>A</i>	8	9	11	10
		1		1
		<i>D</i>		

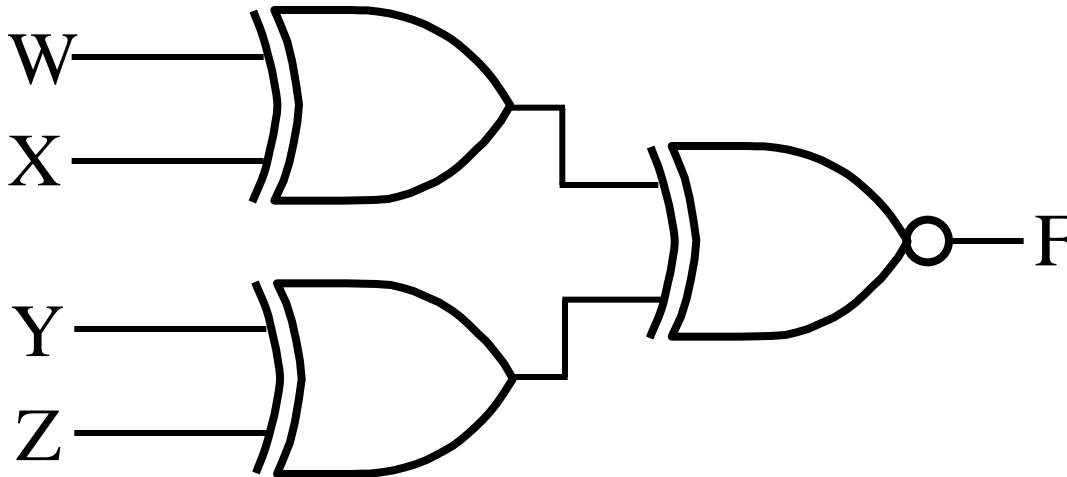
Example: Odd Function Implementation

- Design a 3-input odd function $F = X \oplus Y \oplus Z$ with 2-input XOR gates
- Factoring, $F = (X \oplus Y) \oplus Z$
- The circuit:



Example: Even Function Implementation

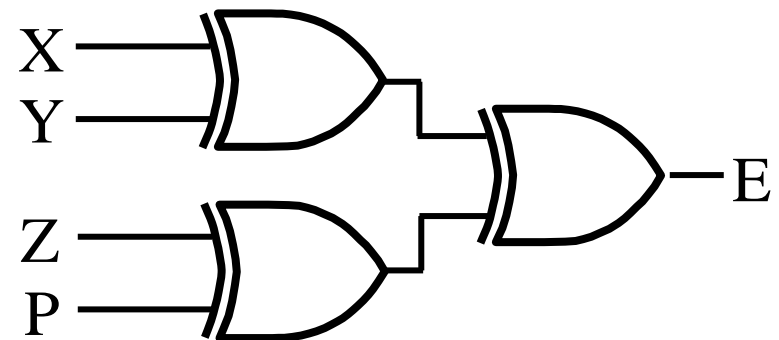
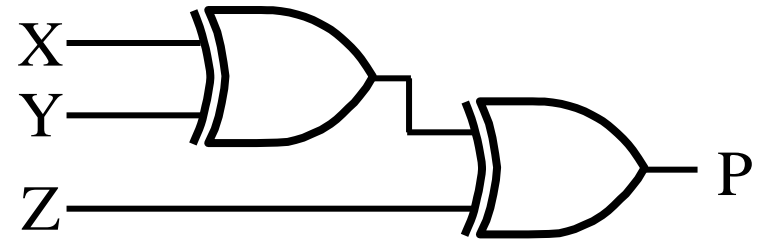
- Design 4-input even function $F = \overline{W \oplus X \oplus Y \oplus Z}$ with 2-input XOR and XNOR gates
- Factoring, $F = \overline{(W \oplus X) \oplus (Y \oplus Z)}$
- The circuit:



Parity Generators and Checkers

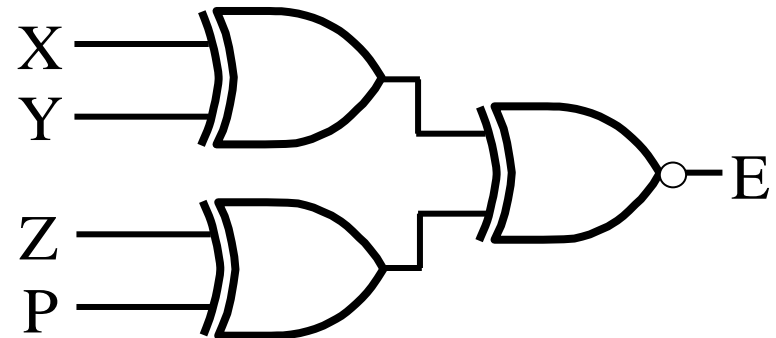
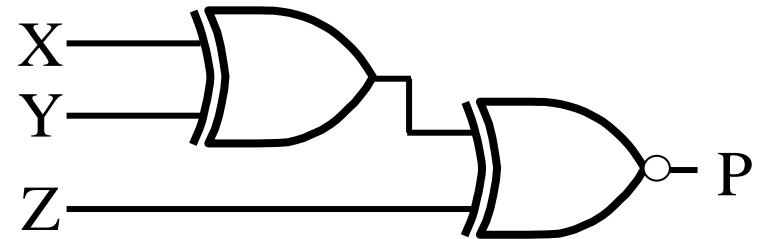
- In Chapter 1, a parity bit added to n-bit code to produce an n+1 bit code:

- Example: $n = 3$. **Generate even parity** code words of length four with **odd function (XOR)**:
- Check even parity** code words of length four with **odd function (XOR)**:
- Operation: $(X, Y, Z) = (0, 0, 1)$ gives $(X, Y, Z, P) = (0, 0, 1, 1)$ and $E = 0$.
If Y changes from 0 to 1 between generator and checker, then $E = 1$ indicates an error



Parity Generator and Checker

- Example: $n = 3$. **Generate odd parity** code words of length four with **even function (XNOR)**:
- **Check odd parity** code words of length four with **even function (XNOR)**:
- Operation: $(X, Y, Z) = (0, 0, 1)$ gives $(X, Y, Z, P) = (0, 0, 1, 0)$ and $E = 0$.
If Y changes from 0 to 1 between generator and checker, then $E = 1$ indicates an error



Terms of Use

- **All (or portions) of this material © 2008 by Pearson Education, Inc.**
- **Permission is given to incorporate this material or adaptations thereof into classroom presentations and handouts to instructors in courses adopting the latest edition of Logic and Computer Design Fundamentals as the course textbook.**
- **These materials or adaptations thereof are not to be sold or otherwise offered for consideration.**
- **This Terms of Use slide or page is to be included within the original materials or any adaptations thereof.**

Logic and Computer Design Fundamentals

Chapter 3 – Combinational Logic Design

Part 1 – Implementation Technology and Logic Design

Charles Kime & Thomas Kaminski

© 2008 Pearson Education, Inc.

(Hyperlinks are active in View Show mode)

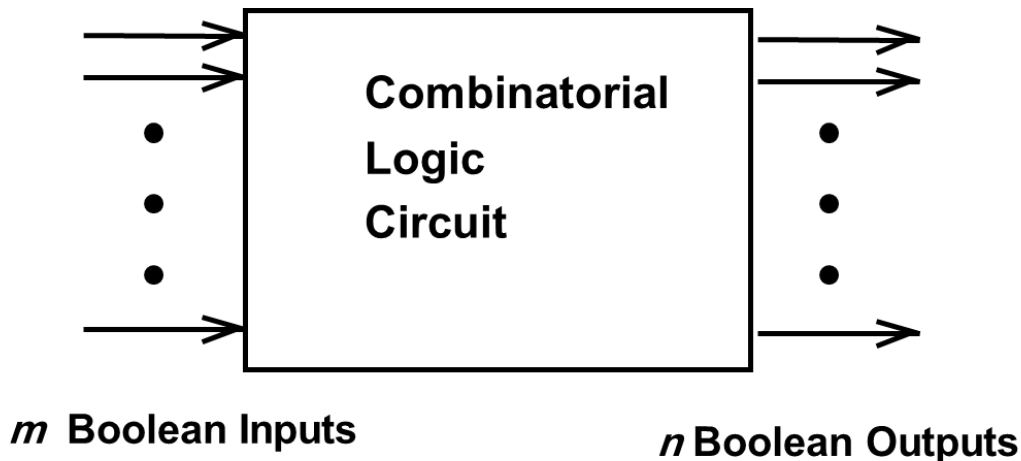
Updated by thoroughly by Dr. Waleed Dweik

Overview

- **Part 1 – Design Procedure**
 - **Steps**
 - **Specification**
 - **Formulation**
 - **Optimization**
 - **Technology Mapping**
 - **Verification**
 - **Technology Mapping - AND, OR, and NOT to NAND or NOR**

Combinational Circuits

- A combinational logic circuit has:
 - A set of m Boolean inputs,
 - A set of n Boolean outputs, and
 - n switching functions, each mapping the 2^m input combinations to an output such that the current output depends only on the current input values
- A block diagram:



Design Procedure

1. Specification

- Write a specification for the circuit if one is not already available. *What does the circuit do? Including names or symbols for inputs and outputs*

2. Formulation

- Derive a *truth table* or *initial Boolean equations* that define the required relationships between the inputs and outputs, if not in the specification

3. Optimization

- Apply 2-level optimization using K-maps
- Draw a logic diagram for the resulting circuit using ANDs, ORs, and inverters

Design Procedure

4. Technology Mapping

- Map the logic diagram to the implementation technology selected

5. Verification

- Verify the correctness of the final design *manually* or using *simulation*

Design Example1

- **Specification:** Design a combinational circuit that has **3 inputs (X, Y, Z)** and **one output F** , such that $F = 1$ when the number of 1's in the input is greater than the number of 0's (i.e. number of 1's ≥ 2)
 - This is called **majority function** (i.e. majority of inputs must be 1 for the function to be 1)

- **Formulation:**

X	Y	Z	F
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Design Example1 Cont.

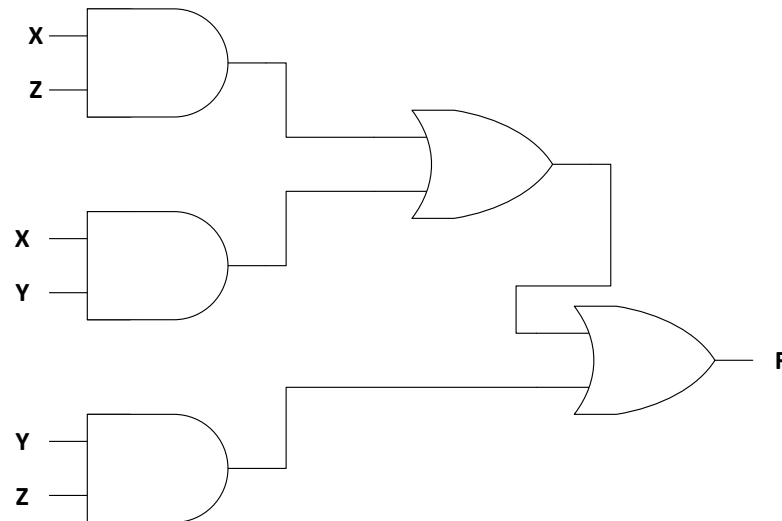
- **Optimization:**

$$F(X, Y, Z) = XY + XZ + YZ$$

		Y			
		0	1	3	2
X	4			1	
	5		1	1	1
		Z			

- **Technology Mapping:**

- Mapping with a library containing inverters, 2-input AND, 2-input OR



Design Example2

- Specification:** Design a combinational circuit that compares *2-bit* Binary number (A, B) and produce two outputs (O_1, O_0), such that:

$O_1O_0 = 00$	When $A = B$ and Both are even
$O_1O_0 = 01$	When $A < B$
$O_1O_0 = 10$	When $A > B$
$O_1O_0 = 11$	When $A = B$ and Both are odd

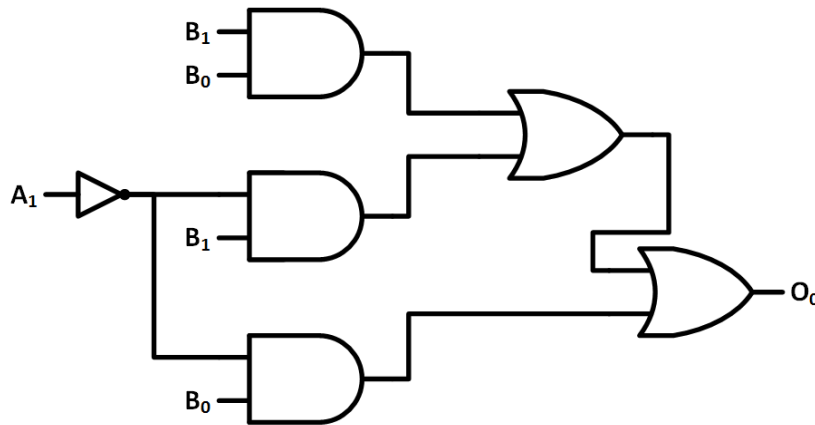
- Formulation:**

$A(A_1A_0)$	$B(B_1B_0)$	$O(O_1O_0)$
00	00	00
00	01	01
00	10	01
00	11	01
01	00	10
01	01	11
01	10	01
01	11	01
10	00	10
10	01	10
10	10	00
10	11	01
11	00	10
11	01	10
11	10	10
11	11	11

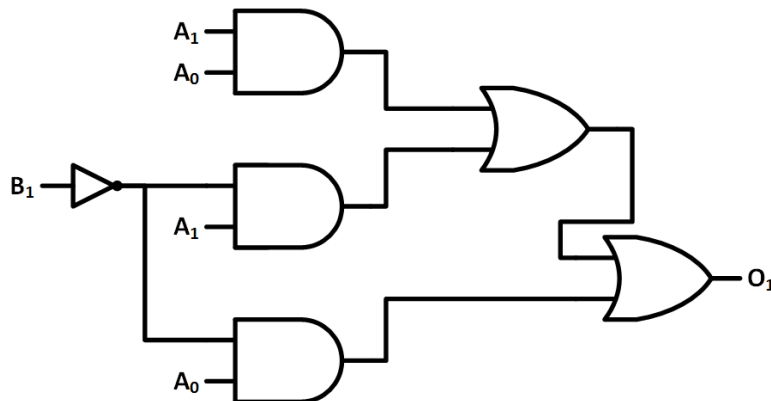
Design Example2 Cont.

- Optimization and Technology Mapping:

$$O_0 = B_1B_0 + \overline{A_1}B_1 + \overline{A_1}B_0$$



$$O_1 = A_1A_0 + A_0\overline{B_1} + A_1\overline{B_1}$$



O_0		B_1			
		0	1	3	2
		4	5	7	6
		12	13	15	14
A_1		8	9	11	10
		B_0			

O_1		B_1			
		0	1	3	2
		4	5	7	6
		12	13	15	14
A_1		8	9	11	10
		B_0			

Design Example3

1. Specification

- *BCD to Excess-3 code converter*
- Transforms BCD code for the decimal digits to Excess-3 code for the decimal digits
- BCD code words for digits 0 through 9: 4-bit patterns 0000 to 1001, respectively
- Excess-3 code words for digits 0 through 9: 4-bit patterns consisting of 3 (binary 0011) added to each BCD code word
- *BCD input is labeled A, B, C, D*
- *Excess-3 output is labeled W, X, Y, Z*

Design Example3 Cont.

2. Formulation

<i>ABCD</i>	<i>WXYZ</i>
0000	0011
0001	0100
0010	0101
0011	0110
0100	0111
0101	1000
0110	1001
0111	1010
1000	1011
1001	1100
1010	XXXX
1011	XXXX
1100	XXXX
1101	XXXX
1110	XXXX
1111	XXXX

Design Example3 Cont.

3. Optimization

$$W = A + BC + BD$$

$$X = \bar{B}D + \bar{B}C + B\bar{C}\bar{D}$$

$$Y = \bar{C}\bar{D} + CD$$

$$Z = \bar{D}$$

	<i>C</i>				
<i>W</i>	0	1	3	2	
	4	5	7	6	
	12	13	15	14	<i>B</i>
<i>A</i>	8	9	11	10	
	1	1	X	X	
					<i>D</i>

	<i>C</i>				
<i>X</i>	0	1	3	2	
	4	5	7	6	
	12	13	15	14	<i>B</i>
<i>A</i>	8	9	11	10	
	1	1	X	X	
					<i>D</i>

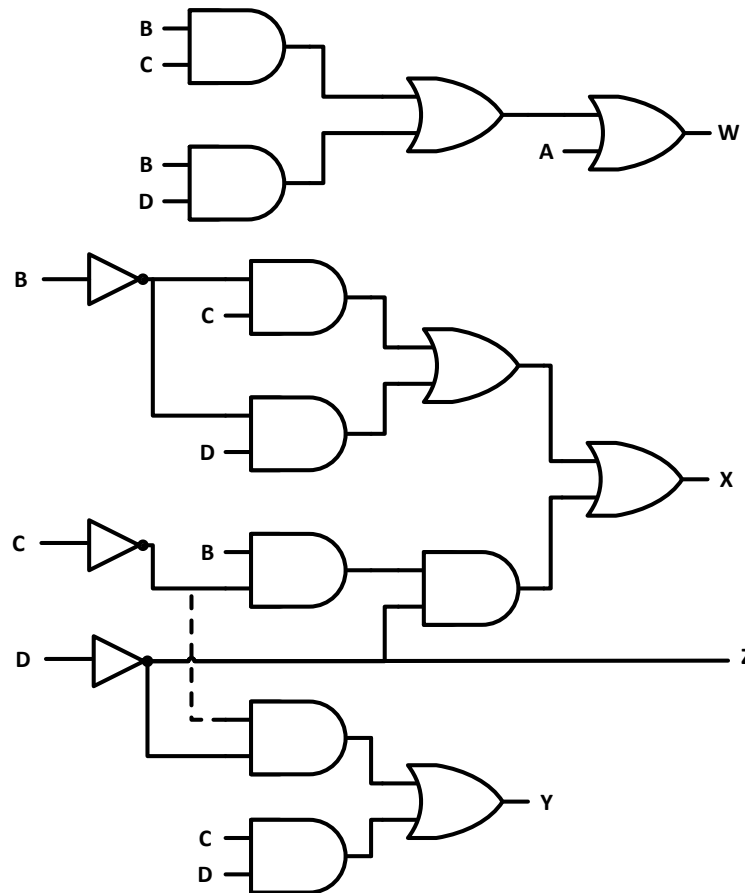
	<i>C</i>				
<i>Y</i>	0	1	3	2	
	4	5	7	6	
	12	13	15	14	<i>B</i>
<i>A</i>	8	9	11	10	
	1		X	X	
					<i>D</i>

	<i>C</i>				
<i>Z</i>	0	1	3	2	
	4	5	7	6	
	12	13	15	14	<i>B</i>
<i>A</i>	8	9	11	10	
	1		X	X	
					<i>D</i>

Design Example3 Cont.

4. Technology Mapping

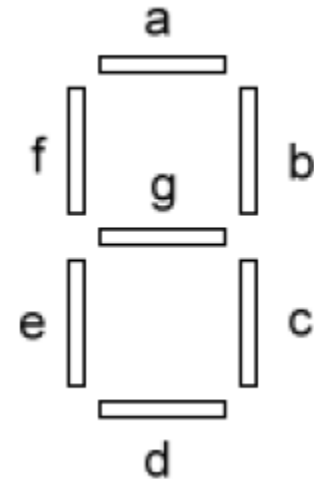
- Mapping with a library containing inverters, 2-input AND, 2-input OR



Homework: BCD to 7-Segment

- **Specification:**

- **Inputs:** (*A, B, C, D*) BCD code from 0000-to-1001
- **Outputs:** (*g, f, e, d, c, b, a*)



- **Formulation:**

<i>ABCD</i>	<i>gfedcba</i>
<i>0000</i>	<i>0111111</i>
<i>0001</i>	<i>0000110</i>
	/
	/
	/
<i>1001</i>	<i>1100111</i>
<i>1010</i>	<i>0000000</i>
	/
	/
	/
<i>1111</i>	<i>0000000</i>

- **Optimization:**

- How many K-maps?

Technology Mapping

- **Mapping Procedures**
 - **To NAND gates**
 - **To NOR gates**

Mapping to NAND gates

■ Assumptions:

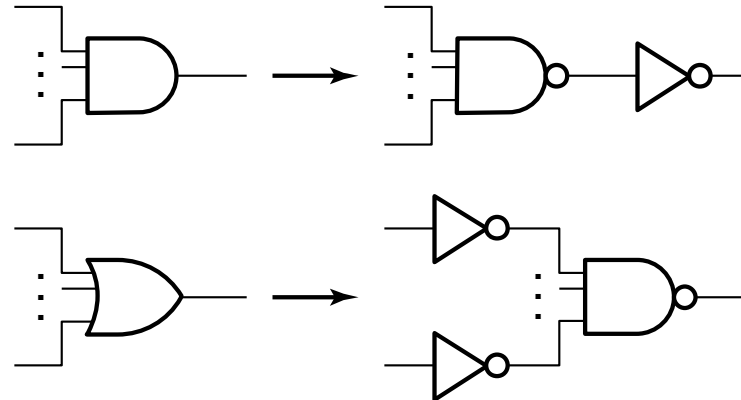
- Gate loading and delay are ignored
- Cell library contains an inverter and n -input NAND gates, $n = 2, 3, \dots$
- An AND, OR, inverter schematic for the circuit is available

■ The mapping is accomplished by:

- Replacing AND and OR symbols,
- Pushing inverters through circuit fan-out points, and
- Canceling inverter pairs

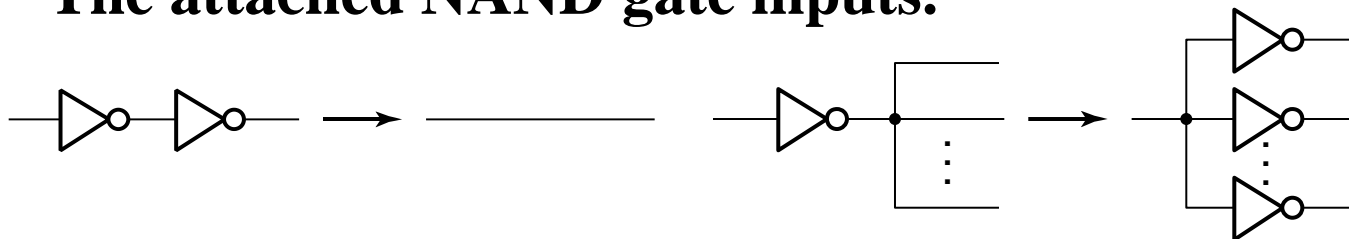
NAND Mapping Algorithm

1. Replace ANDs and ORs:

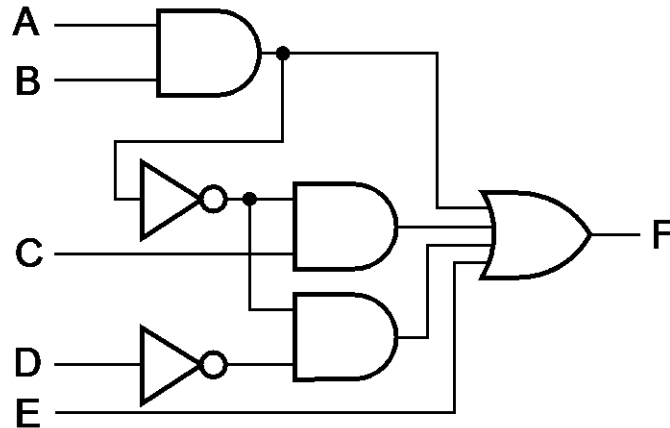


2. Repeat the following pair of actions until there is at most one inverter between :

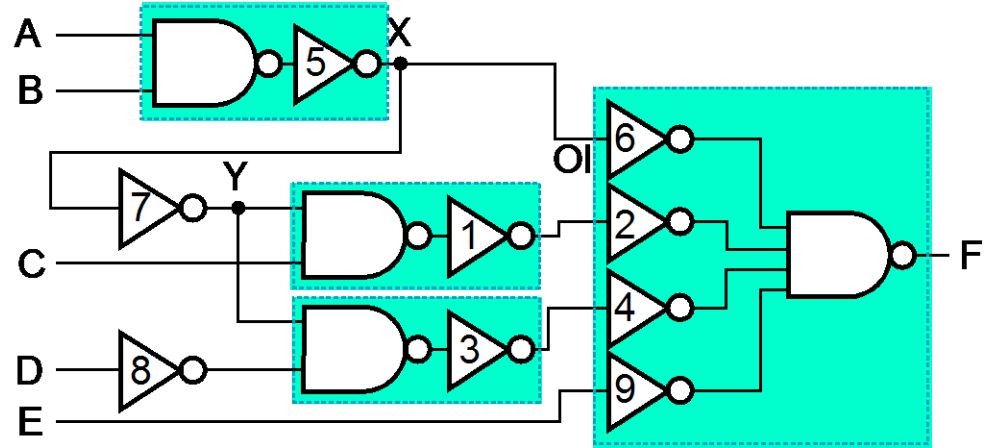
- A circuit input or driving NAND gate output, and
- The attached NAND gate inputs.



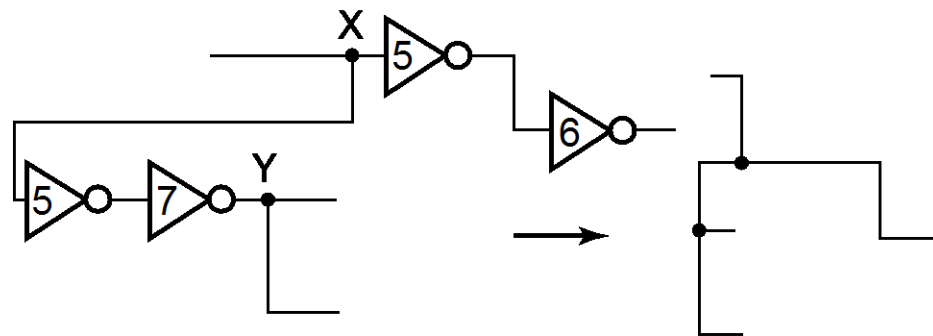
NAND Mapping Example



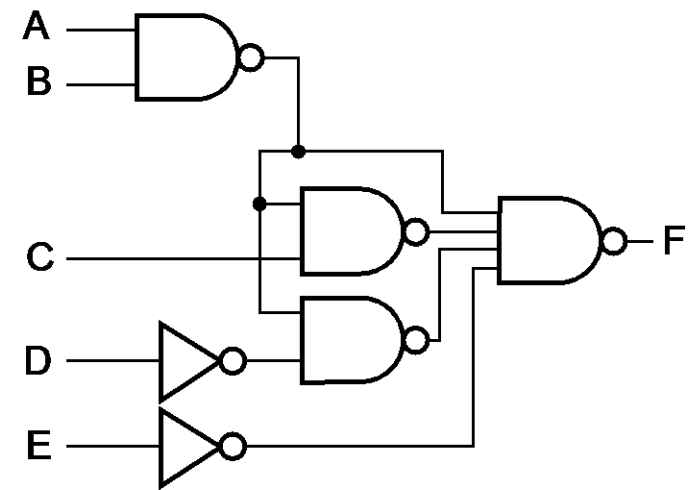
(a)



(b)



(c)



(d)

Mapping to NOR gates

■ Assumptions:

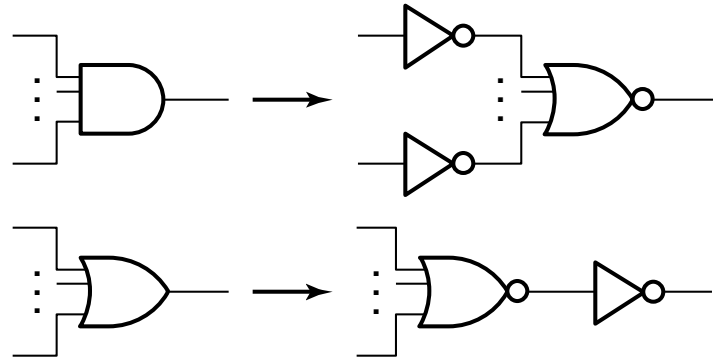
- Gate loading and delay are ignored
- Cell library contains an inverter and n -input NOR gates, $n = 2, 3, \dots$
- An AND, OR, inverter schematic for the circuit is available

■ The mapping is accomplished by:

- Replacing AND and OR symbols,
- Pushing inverters through circuit fan-out points, and
- Canceling inverter pairs

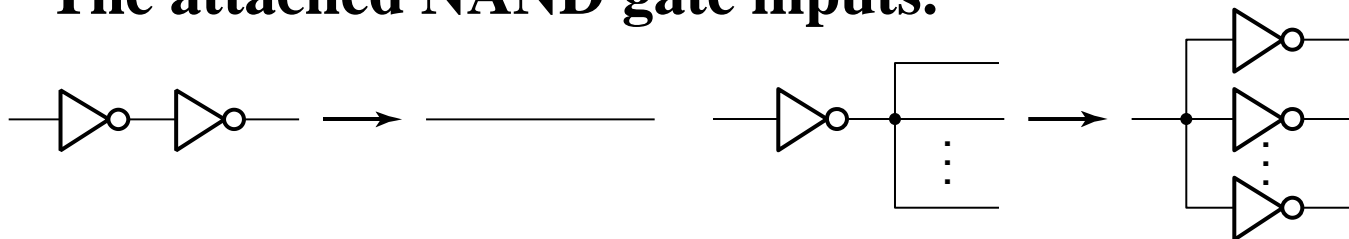
NOR Mapping Algorithm

1. Replace ANDs and ORs:

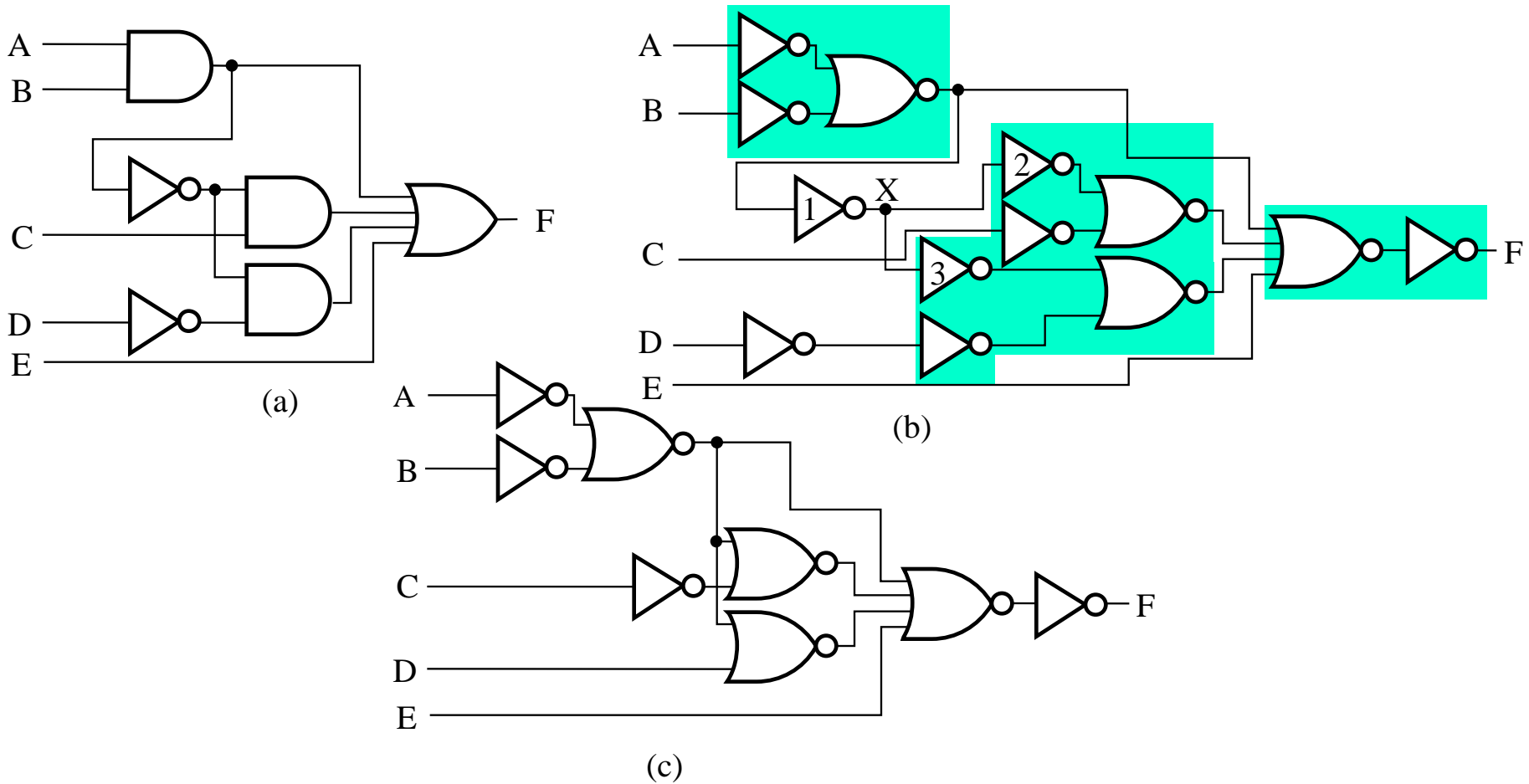


2. Repeat the following pair of actions until there is at most one inverter between :

- A circuit input or driving NAND gate output, and
- The attached NAND gate inputs.



NOR Mapping Example



Terms of Use

- **All (or portions) of this material © 2008 by Pearson Education, Inc.**
- **Permission is given to incorporate this material or adaptations thereof into classroom presentations and handouts to instructors in courses adopting the latest edition of Logic and Computer Design Fundamentals as the course textbook.**
- **These materials or adaptations thereof are not to be sold or otherwise offered for consideration.**
- **This Terms of Use slide or page is to be included within the original materials or any adaptations thereof.**

Logic and Computer Design Fundamentals

Chapter 3 – Combinational Logic Design

Part 2 – Combinational Logic

Charles Kime & Thomas Kaminski

© 2008 Pearson Education, Inc.
(Hyperlinks are active in View Show mode)

Updated Thoroughly by Dr. Waleed Dweik

Overview

■ Part 2 – Combinational Logic

- Functions and functional blocks
- Rudimentary logic functions
- Decoding using Decoders
 - Implementing Combinational Functions with Decoders
- Encoding using Encoders
- Selecting using Multiplexers
 - Implementing Combinational Functions with Multiplexers

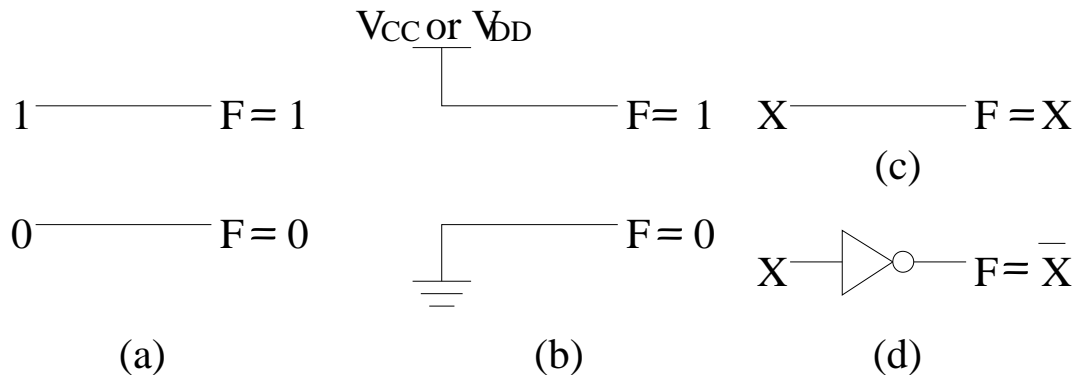
Functions and Functional Blocks

- The functions considered are those found to be very useful in design
- Corresponding to each of the functions is a combinational circuit implementation called a *functional block*
- In the past, functional blocks were packaged as small-scale-integrated (SSI), medium-scale integrated (MSI), and large-scale-integrated (LSI) circuits
- Today, they are often simply implemented within a very-large-scale-integrated (VLSI) circuit

Rudimentary Logic Functions

- Functions of a single variable X
- Can be used on the inputs to functional blocks to implement other than the block's intended function
- *Value fixing : a, b*
- *Transferring : c*
- *Inverting : d*
- *Enabling : next slide*

Functions of One Variable				
X	$F = 0$	$F = 1$	$F = X$	$F = \bar{X}$
0	0	1	0	1
1	0	1	1	0

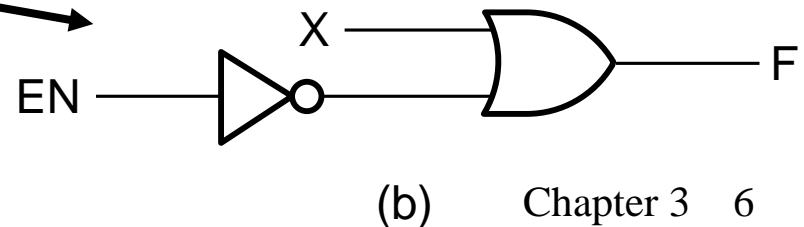
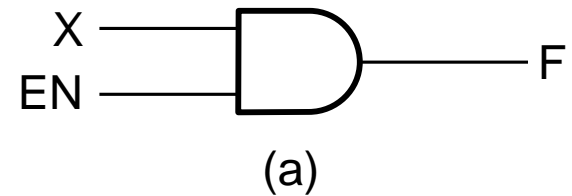


Enabling Function

- **Enabling** permits an input signal to pass through to an output
- **Disabling** blocks an input signal from passing through to an output, replacing it with a fixed value
- The value on the output when it is disabled can be **Hi-Z** (as for three-state buffers and transmission gates), 0, or 1

- When disabled, **0 output**

- When disabled, **1 output**



Decoding

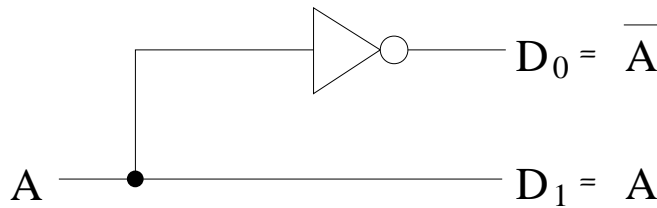
- **Decoding:** the conversion of an ***n -bit*** input code to an ***m -bit*** output code with $n \leq m \leq 2^n$ such that each valid code word produces a unique output code
- Circuits that perform decoding are called ***decoders***
- Functional blocks for decoding are
 - called ***n -to- m line decoders***, where $m \leq 2^n$, and
 - generate 2^n (or fewer) minterms for the n input variables

1-to-2 Line Decoder

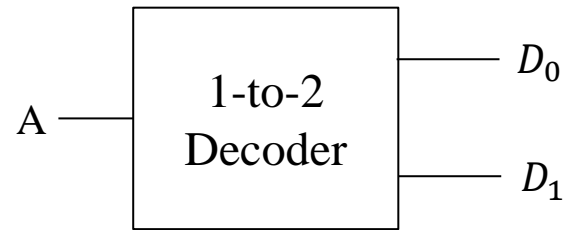
- When the decimal value of A equals the subscript of D_i , that D_i will be 1 and all others will be 0's
- Only one output is active at a time

A	D_0	D_1
0	1	0
1	0	1

(a)



(b)



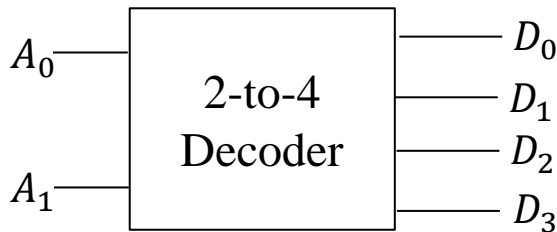
(c)

- Decoders are used to control multiple circuits by enabling only one of them at a time

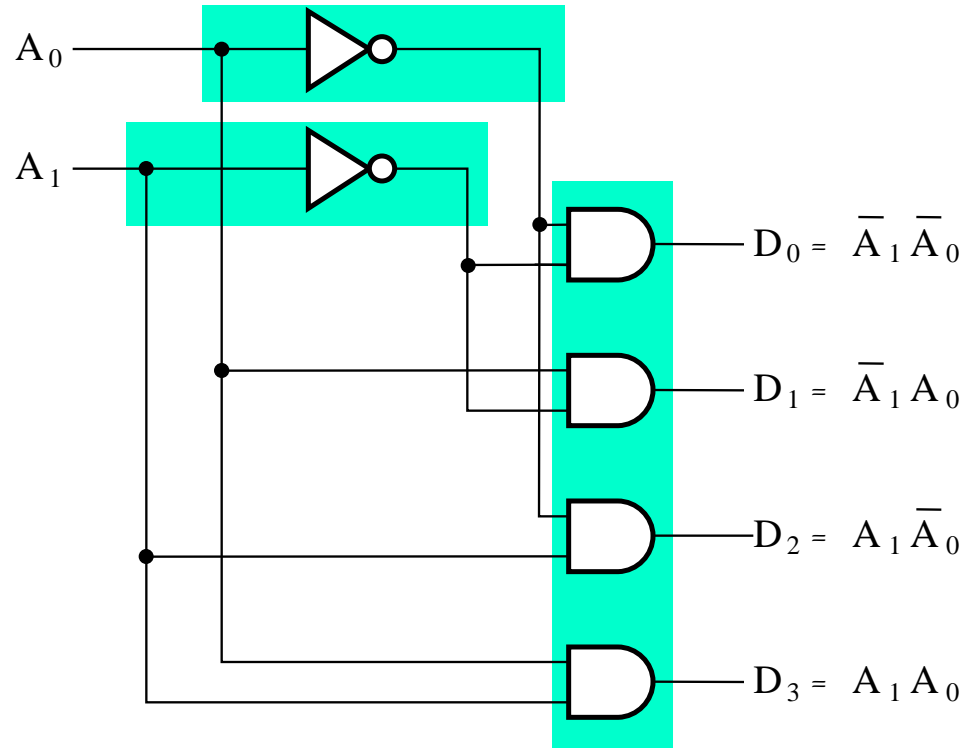
2-to-4 Line Decoder

A_1	A_0	D_0	D_1	D_2	D_3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

(a)



(c)



(b)

- No more optimization is possible
- ***Note that the 2-to-4 line decoder is made up of two 1-to-2-line decoders and 4 AND gates***

Decoder Expansion

- General procedure given in book for any decoder with ***n*** ***inputs and 2ⁿ outputs***
- This procedure builds a decoder backward from the outputs using
 1. Let $k = n$
 2. We need 2^k 2-input AND gates driven as follows:
 - **If k is even,** drive the gates using two $k/2$ -to- $2^{k/2}$ decoders
 - **If k is odd,** drive the gates using one $(k+1)/2$ -to- $2^{(k+1)/2}$ decoder and one $(k-1)/2$ -to- $2^{(k-1)/2}$ decoder
 3. For each decoder resulting from step2, repeat step2 until $k = 1$. For $k = 1$, use 1-to-2 decoder

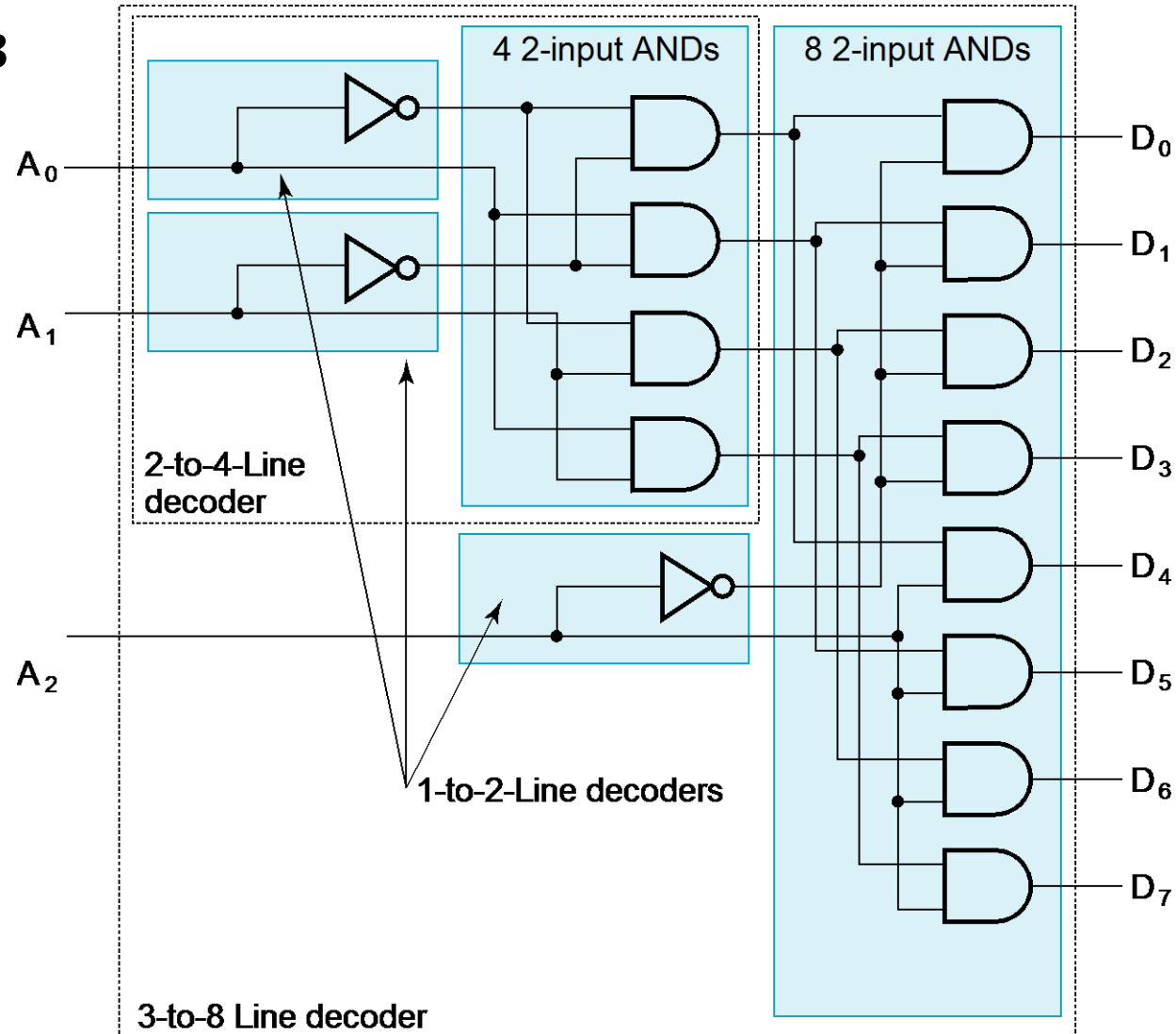
Decoder Expansion - Example 1

- 3-to-8-line decoder
 - $k = n = 3$
 - We need 2^3 (8) 2-input AND gates driven as follows:
 - k is odd, so split to:
 - 2-to-4-line decoder
 - 1-to-2-line decoder
 - 2-to-4-line decoder $\rightarrow k = n = 2$
 - We need 2^2 (4) 2-input AND gates driven as follows:
 - k is even, so split to:
 - Two 1-to-2-line decoder

- See next slide for result

Decoder Expansion - Example 1

- $GN = 8 \times 2 + 4 \times 2 + 3$
- $GN = 27$
- Straight forward design has the same GN cost

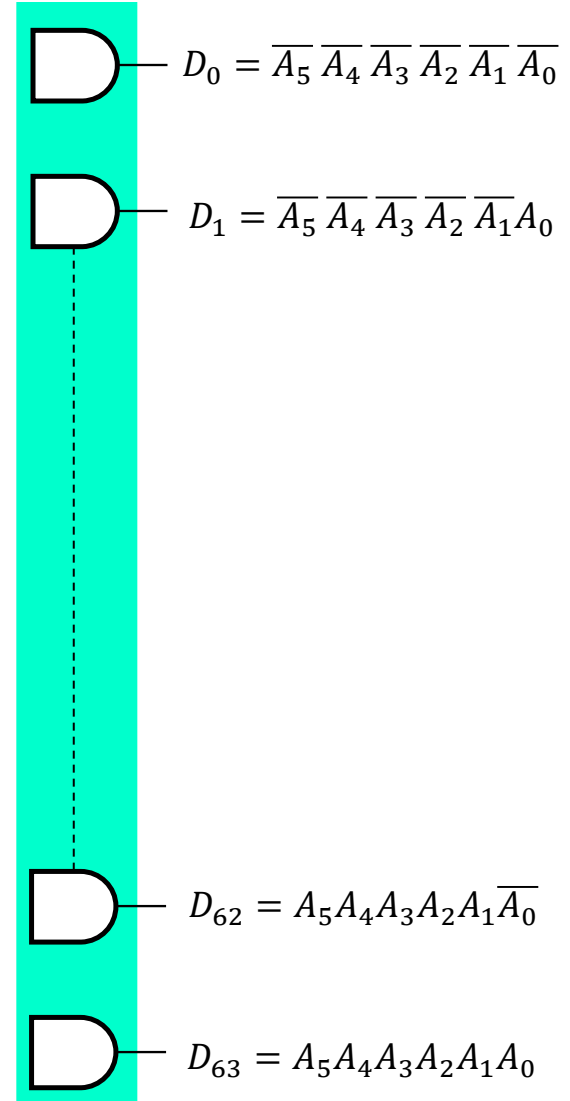
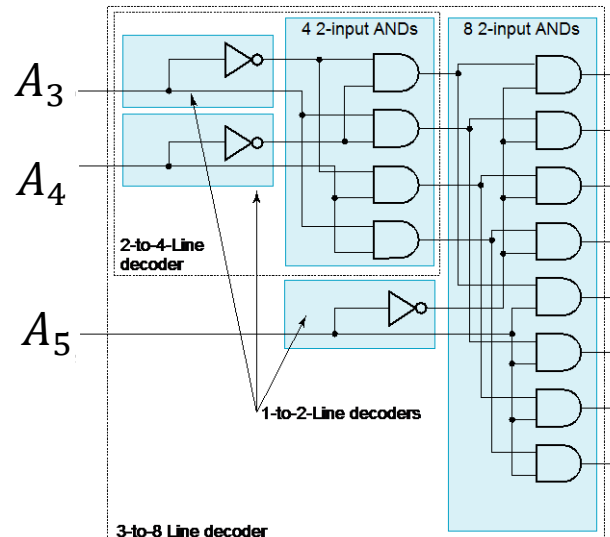
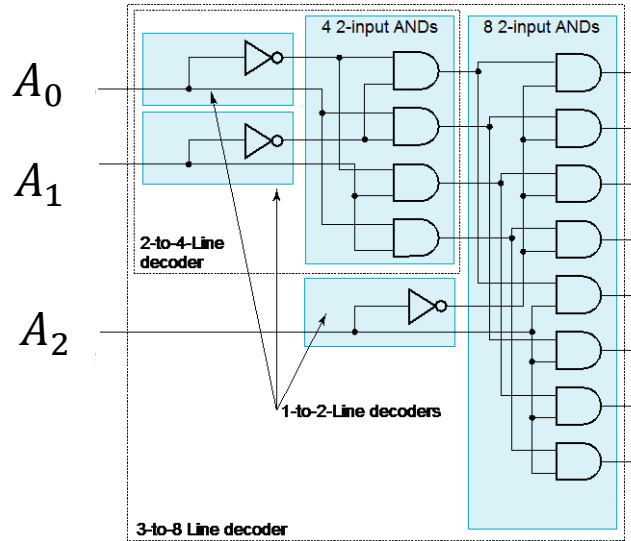


Decoder Expansion - Example 2

- 6-to-64-line decoder
 - $k = n = 6$
 - We need 2^6 (64) 2-input AND gates driven as follows:
 - k is even, so split to:
 - Two 3-to-8-line decoders
 - Each 3-to-8-line decoder is designed as shown in Example 1

Decoder Expansion - Example 2

- GN = 64 × 2 + 16 × 2 + 8 × 2 + 6
- GN = 182
- Straight forward design has GN cost of 390



Decoder Expansion - Example 3

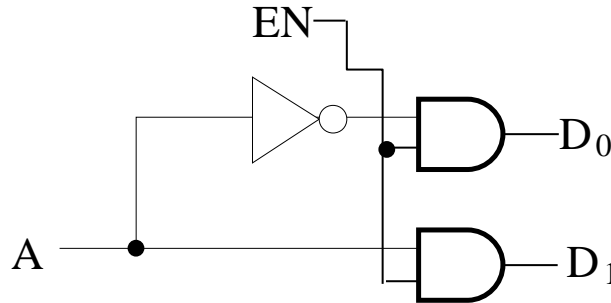
- **7-to-128-line decoder**
 - $k = n = 7$
 - We need 2^7 (128) 2-input AND gates driven as follows:
 - k is odd, so split to:
 - 4-to-16-line decoder
 - 3-to-8-line decoder
 - 4-to-16-line decoder
 - $k = n = 4$
 - We need 2^4 (16) 2-input AND gates driven as follows:
 - k is even, so split to:
 - Two 2-to-4-line decoders
 - Complete using known 3-8 and 2-to-4 line decoders
- $GN = 128 \times 2 + 16 \times 2 + 8 \times 2 + 12 \times 2 + 7 = 335$
- Compare to straight forward design with GN cost of 903

Building Larger Decoders

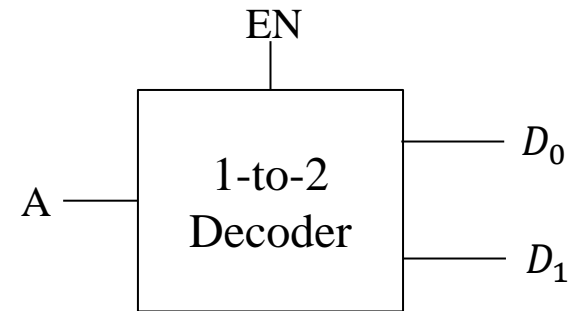
- **Method_1:** Decoder Expansion
- **Method_2:** Using Small *Decoders with Enable input*
- Example: 1-to-2 line decoder with enable
 - In general, attach *m-enabling* circuits to the outputs
 - See truth table below for function
 - Note use of X's to denote both 0 and 1
 - Combination containing two X's represent two binary combinations
- Alternatively, can be viewed as distributing value of signal EN to 1 of 2 outputs
 - In this case, it is called a *Demultiplexer*

EN	A	D ₀	D ₁
0	X	0	0
1	0	1	0
1	1	0	1

(a)



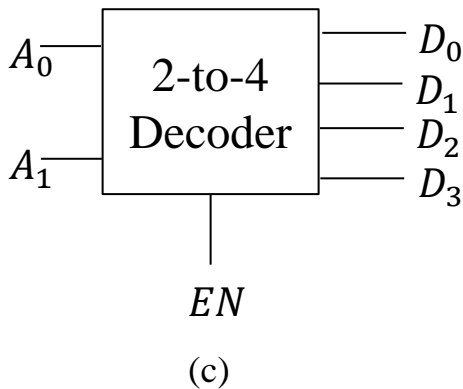
(b)



(c)

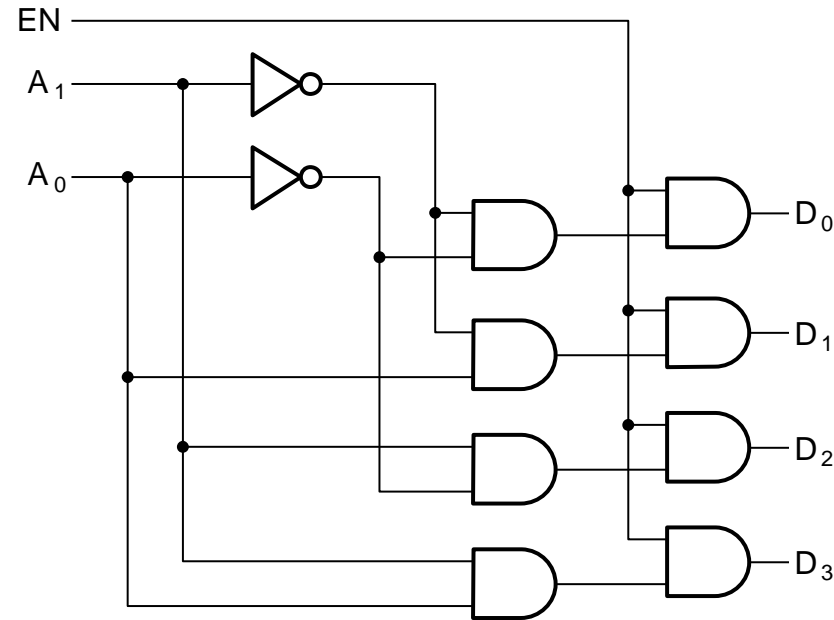
2-to-4 Line Decoder with Enable

- Attach **4-enabling** circuits to the outputs
- See truth table below for function
 - Combination containing two X's represent four binary combinations
- Alternatively, can be viewed as distributing value of signal EN to 1 of 4 outputs
 - In this case, it is called a *Demultiplexer*



EN	A ₁	A ₀	D ₀	D ₁	D ₂	D ₃
0	X	X	0	0	0	0
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1

(a)



2-to-4 Decoder using 1-to-2 Decoders and Inverters

A_1
0
0
1
1

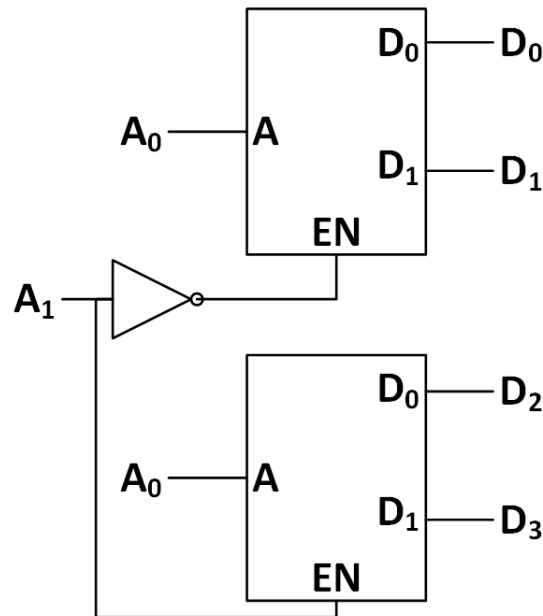
A_0
0
1
0
1

D_0	D_1
1	0
0	1
0	0
0	0

1st 1-to-2 Decoder

D_2	D_3
0	0
0	0
1	0
0	1

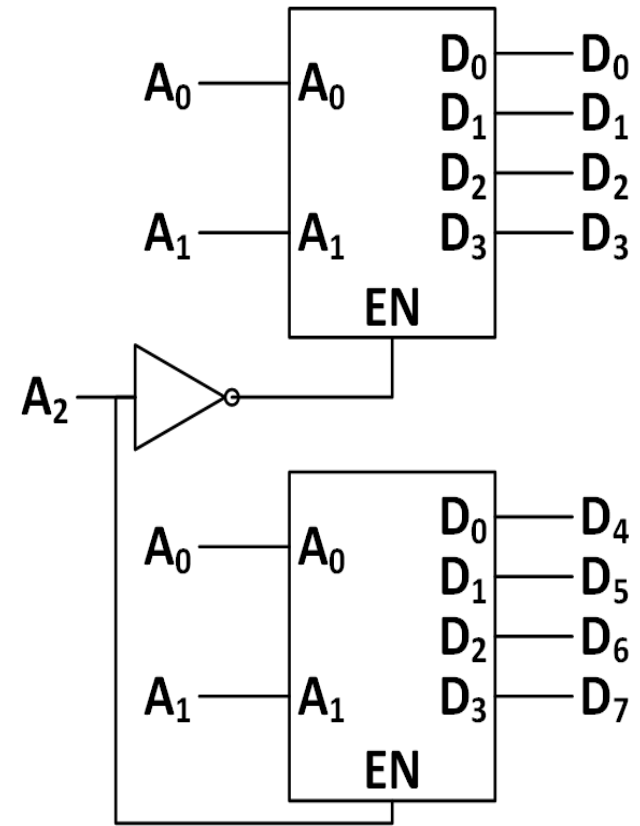
2nd 1-to-2 Decoder



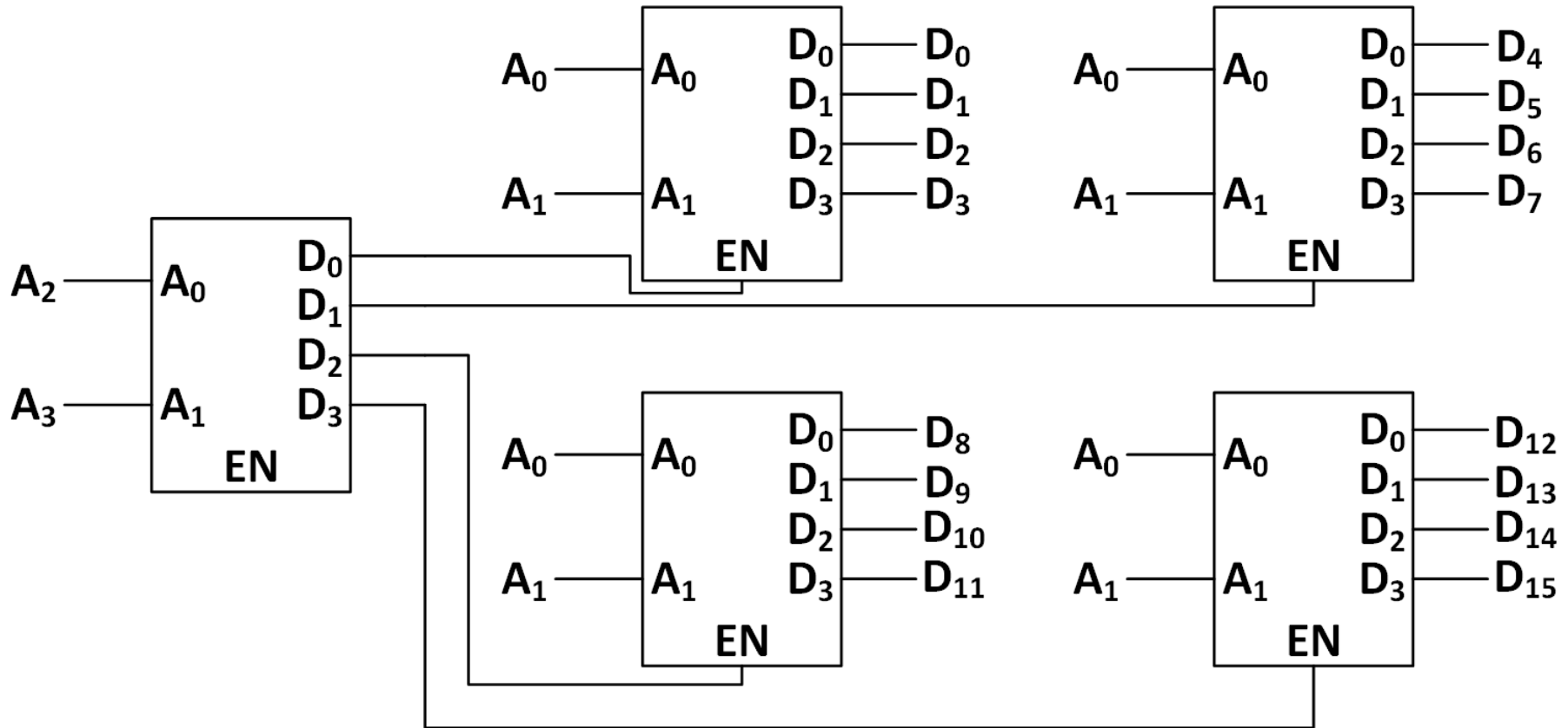
3-to-8 Decoder using 2-to-4 Decoders and Inverters

A_2	A_1	A_0	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

1st 2-to4 Decoder
 2nd 2-to4 Decoder



4-to-16 Decoder using Only 2-to-4 Decoders



Combinational Logic Implementation

- Decoder and OR Gates

- Implement m functions of n variables with:
 - Sum-of-minterms expressions
 - One n -to- 2^n -line decoder
 - m OR gates, one for each function
 - For each function, the OR gate has k inputs, where k is the number of minterms in the function
- **Approach 1:**
 - Find the truth table for the functions
 - Make a connection to the corresponding OR from the corresponding decoder output wherever a 1 appears in the truth table
- **Approach 2**
 - Find the minterms for each output function
 - OR the minterms together

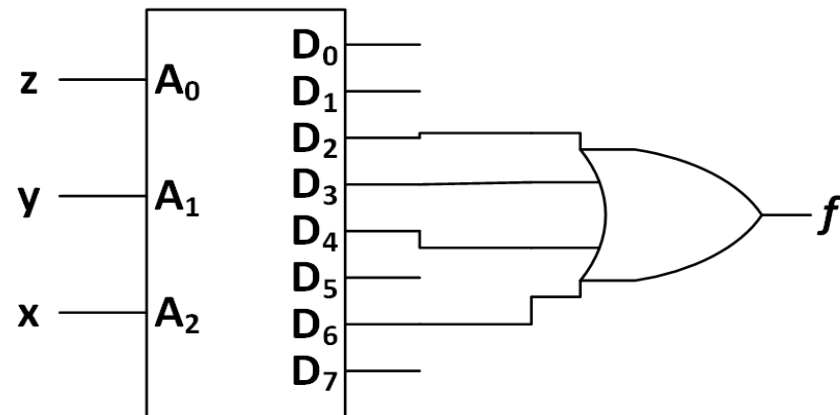
Example 1

- Implement function f using decoder and OR gate:

$$f(x, y, z) = x\bar{z} + \bar{x}y$$

- $n = 3$ variables \rightarrow **3-to-8 decoder**
- One function \rightarrow **One OR gate**
- Solution: Convert f to SOM format
 - $f = x\bar{z}(y + \bar{y}) + \bar{x}y(z + \bar{z}) = xy\bar{z} + x\bar{y}\bar{z} + \bar{x}yz + \bar{x}y\bar{z}$
 - $f(x, y, z) = \sum_m(2,3,4,6) \rightarrow$ **4-input OR gate**

- Decoder is a Minterm Generator**



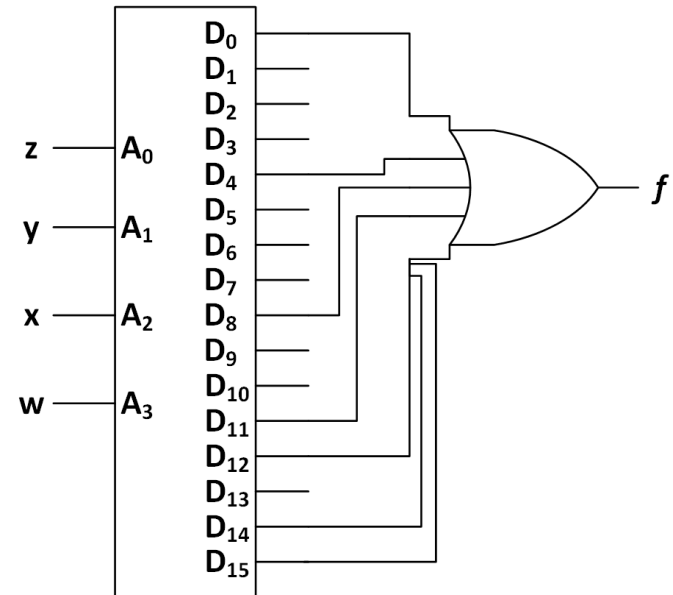
Example2

- Implement function f using decoder and OR gate:

$$f(w, x, y, z) = \sum_m (0, 4, 8, 11, 12, 14, 15)$$

- $n = 4$ variables \rightarrow **4-to-16 decoder**
- One function with 7 minterms \rightarrow **One 7-input OR gate**

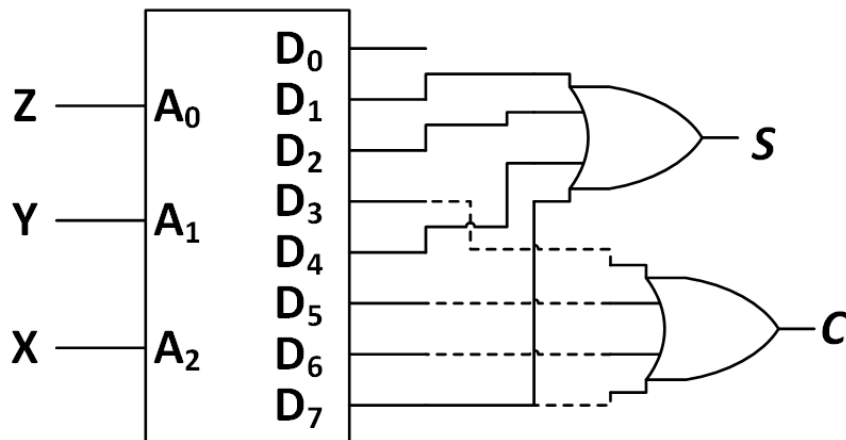
- If number of minterms is greater than $\frac{2^n}{2}$, then design for complement F (\bar{F}) and use NOR gate instead of OR to generate F**



Example 3

- Implement functions C and S using decoder and OR gates:
- $n = 3$ variables \rightarrow **3-to-8 decoder**
- Two function \rightarrow **Two OR gates**
- Solution:
 - $C = \sum_m(3,5,6,7) \rightarrow$ **4-input OR gate**
 - $S = \sum_m(1,2,4,7) \rightarrow$ **4-input OR gate**

X	Y	Z	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1



Example4

- Implement the following set of odd parity functions of

(A_7, A_6, A_5, A_4)

$$P_1 = A_7 \oplus A_5 \oplus A_4$$

$$P_2 = A_7 \oplus A_6 \oplus A_4$$

$$P_3 = A_7 \oplus A_6 \oplus A_5$$

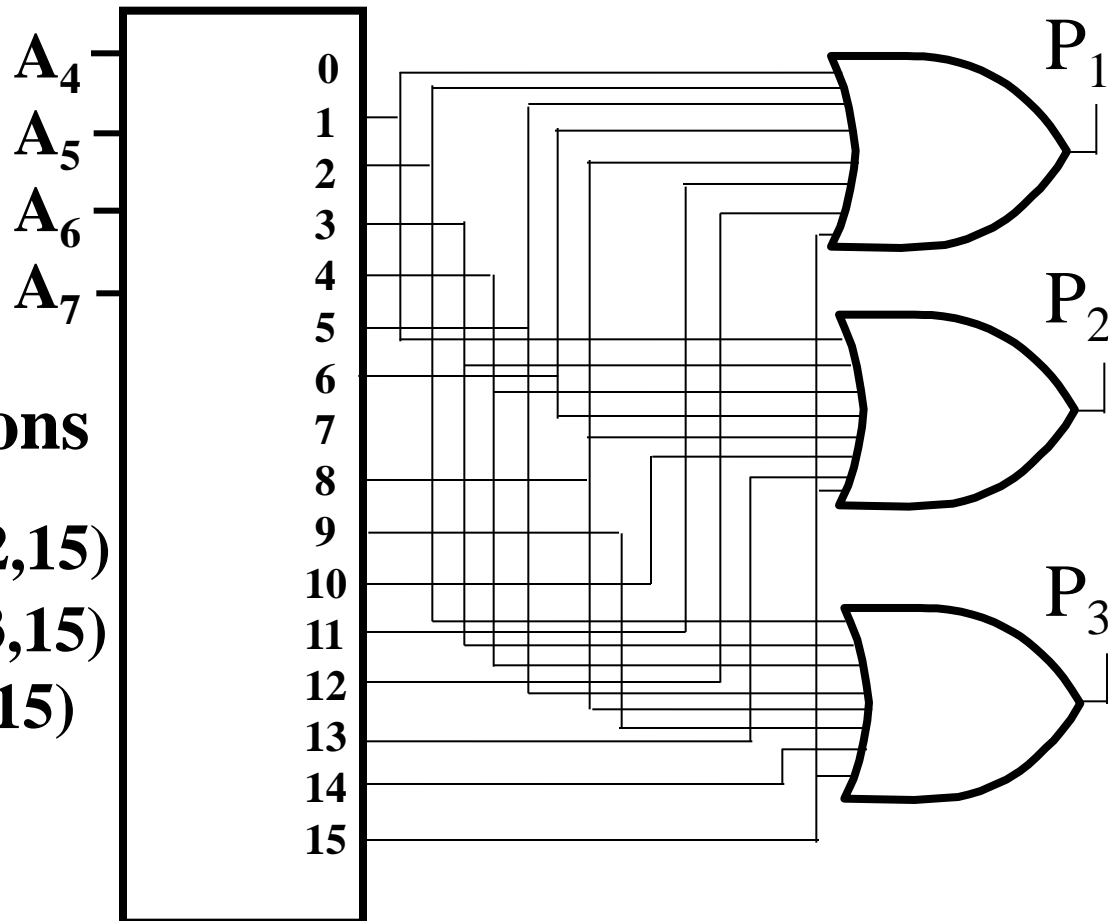
- Finding sum of minterms expressions

$$P_1 = \sum_m(1,2,5,6,8,11,12,15)$$

$$P_2 = \sum_m(1,3,4,6,8,10,13,15)$$

$$P_3 = \sum_m(2,3,4,5,8,9,14,15)$$

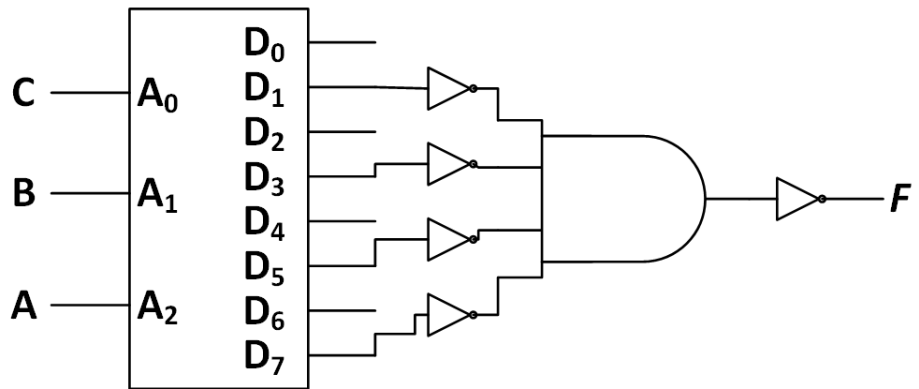
- Find circuit
- Is this a good idea?



Example 5

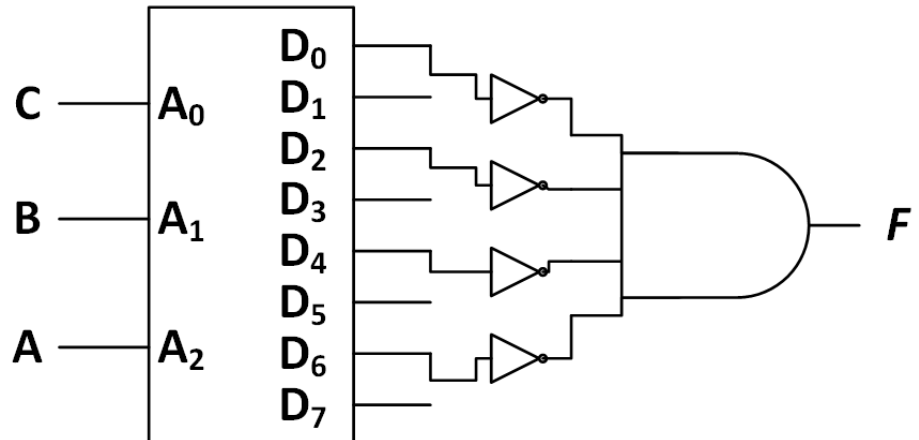
- Implement function F using 3-to-8 decoder, AND gate and inverters: $F(A, B, C) = \sum_m(1,3,5,7)$

- Solution with 5 inverters:



- Solution with 4 inverters:

- $F(A, B, C) = \prod_M(0,2,4,6)$



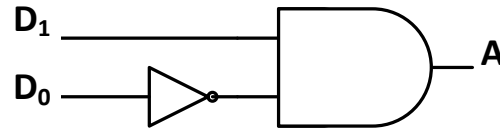
Encoding

- ***Encoding***: the opposite of decoding - the conversion of an m -bit input code to a n -bit output code with $n \leq m \leq 2^n$ such that each valid code word produces a unique output code
- Circuits that perform encoding are called ***encoders***
- An encoder has 2^n (or fewer) input lines and n output lines which ***generate the binary code corresponding to the input values***
- Typically, an encoder converts a code containing exactly one bit that is 1 to a binary code corresponding to the position in which the 1 appears

2-to-1 Encoder & 4-to-2 Encoder

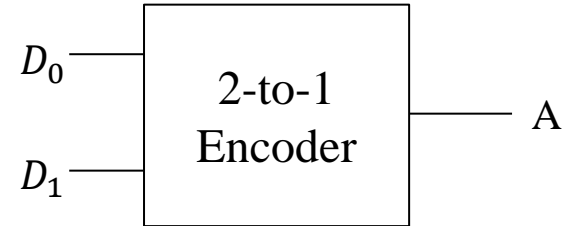
D_1	D_0	A
0	0	Invalid Input
0	1	0
1	0	1
1	1	Invalid Input

(a)



$$A = D_1 \cdot \overline{D_0}$$

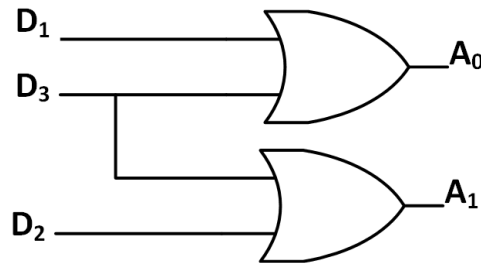
(b)



(c)

D_3	D_2	D_1	D_0	A_1	A_0
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1

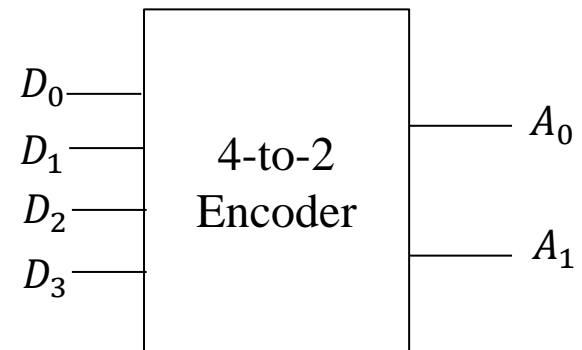
(a)



$$A_0 = D_1 + D_3$$

$$A_1 = D_2 + D_3$$

(b)



(c)

8-to-3 Encoder (Octal-to-Binary Encoder)

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

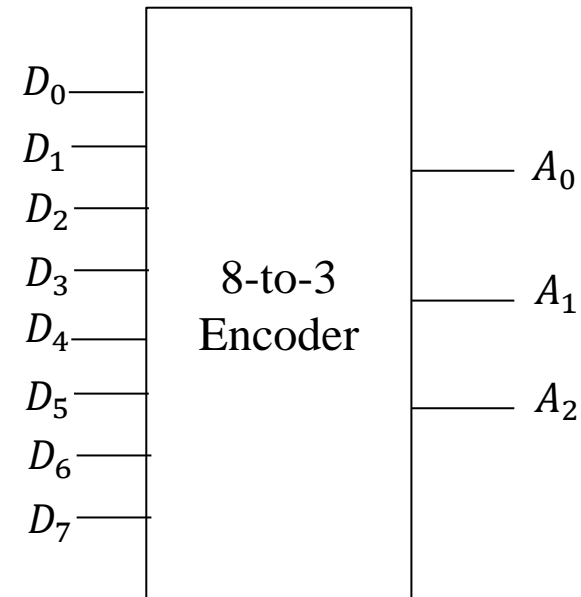
(a)

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

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

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

(b)



(c)

Decimal-to-BCD Encoder

- **Inputs:** 10 bits corresponding to decimal digits 0 through 9, (D_0, \dots, D_9)
- **Outputs:** 4 bits with BCD codes (A_3, A_2, A_1, A_0)
- **Function:** If input bit D_i is a 1, then the output is the BCD code for i
- The truth table could be formed, but alternatively, the equations for each of the four outputs can be obtained directly

Decimal-to-BCD Encoder Cont.

- Input D_i is a term in equation A_j if bit A_j is 1 in the binary value for i
- Equations:
$$A_3 = D_8 + D_9$$
$$A_2 = D_4 + D_5 + D_6 + D_7$$
$$A_1 = D_2 + D_3 + D_6 + D_7$$
$$A_0 = D_1 + D_3 + D_5 + D_7 + D_9$$
- What happens if two inputs are high simultaneously?
 - For example if D_3 and D_6 are high, then the output is 0111 which indicates that only D_7 is high ???
 - ***Solution: Establish input priority***

Priority Encoder

- If more than one input value is 1, then the encoder just designed does not work
- One encoder that can accept all possible combinations of input values and produce a meaningful result is a *priority encoder*
- Among the 1s that appear, it selects the most significant input position (or the least significant input position) containing a 1 and responds with the corresponding binary code for that position
 - *High priority encoder*: gives priority for the input whose value is 1 and has the highest subscript
 - *low priority encoder*: gives priority for the input whose value is 1 and has the lowest subscript
- If all inputs are 0's, what happens?
 - Define an output (V) to encode whether the input is valid or not
 - When all inputs are 0's, V is set to 0 indicating that the input is invalid, otherwise V is set to 1

4-to-2 Low Priority Encoder

#_of_Minterms/ Rows	D_3	D_2	D_1	D_0	A_1	A_0	V
1	0	0	0	0	X	X	0
8	X	X	X	1	0	0	1
4	X	X	1	0	0	1	1
2	X	1	0	0	1	0	1
1	1	0	0	0	1	1	1

(a)

$$A_0 = D_1 \overline{D_0} + D_3 \overline{D_2} \overline{D_1} \overline{D_0}$$

$$A_0 = \overline{D_0} (D_1 + D_3 \overline{D_2} \overline{D_1})$$

$$A_0 = \overline{D_0} (D_1 + D_3 \overline{D_2})$$

$$A_0 = D_1 \overline{D_0} + D_3 \overline{D_2} \overline{D_0}$$

$$A_1 = D_2 \overline{D_1} \overline{D_0} + D_3 \overline{D_2} \overline{D_1} \overline{D_0}$$

$$A_1 = \overline{D_1} \overline{D_0} (D_2 + D_3 \overline{D_2})$$

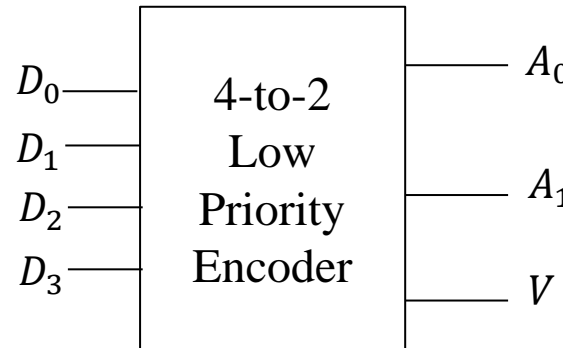
$$A_1 = \overline{D_1} \overline{D_0} (D_2 + D_3)$$

$$A_1 = D_2 \overline{D_1} \overline{D_0} + D_3 \overline{D_1} \overline{D_0}$$

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

(b)

Number of Minterms per Row = $2^{\# \text{ of don't cares}}$



(c)

4-to-2 High Priority Encoder

#_of_Minterms/ Rows	D_3	D_2	D_1	D_0	A_1	A_0	V
1	0	0	0	0	X	X	0
1	0	0	0	1	0	0	1
2	0	0	1	X	0	1	1
4	0	1	X	X	1	0	1
8	1	X	X	X	1	1	1

(a)

$$A_0 = D_3 + \overline{D_3} \overline{D_2} D_1$$

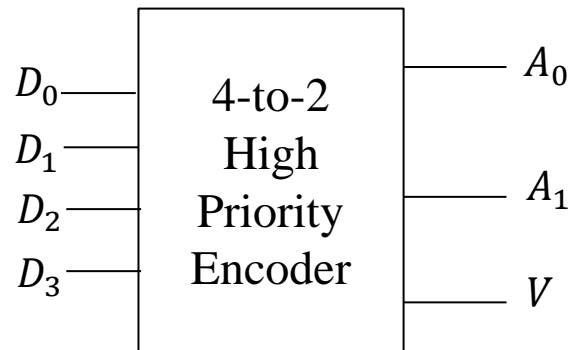
$$A_0 = D_3 + \overline{D_2} D_1$$

$$A_1 = D_3 + \overline{D_3} D_2$$

$$A_1 = D_3 + D_2$$

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

(b)



(c)

5-input Priority Encoder

- Priority encoder with 5 inputs (D_4, D_3, D_2, D_1, D_0) - highest priority to most significant 1 present - Code outputs A_2, A_1, A_0 and V where V indicates at least one 1 present

No. of Min-terms/Row	Inputs					Outputs			
	D_4	D_3	D_2	D_1	D_0	A_2	A_1	A_0	V
1	0	0	0	0	0	X	X	X	0
1	0	0	0	0	1	0	0	0	1
2	0	0	0	1	X	0	0	1	1
4	0	0	1	X	X	0	1	0	1
8	0	1	X	X	X	0	1	1	1
16	1	X	X	X	X	1	0	0	1

- X's in input part of table represent 0 or 1; thus table entries correspond to product terms instead of minterms. The column on the left shows that all 32 minterms are present in the product terms in the table

5-input Priority Encoder Cont.

- Could use a K-map to get equations, but can be read directly from table and manually optimized if careful:

$$\mathbf{A}_2 = \mathbf{D}_4$$

$$\mathbf{A}_1 = \overline{\mathbf{D}}_4 \mathbf{D}_3 + \overline{\mathbf{D}}_4 \overline{\mathbf{D}}_3 \mathbf{D}_2 = \overline{\mathbf{D}}_4 (\mathbf{D}_3 + \mathbf{D}_2)$$

$$\mathbf{A}_1 = \overline{\mathbf{D}}_4 \mathbf{D}_3 + \overline{\mathbf{D}}_4 \mathbf{D}_2$$

$$\mathbf{A}_0 = \overline{\mathbf{D}}_4 \mathbf{D}_3 + \overline{\mathbf{D}}_4 \overline{\mathbf{D}}_3 \overline{\mathbf{D}}_2 \mathbf{D}_1 = \overline{\mathbf{D}}_4 (\mathbf{D}_3 + \overline{\mathbf{D}}_2 \mathbf{D}_1)$$

$$\mathbf{A}_0 = \overline{\mathbf{D}}_4 \mathbf{D}_3 + \overline{\mathbf{D}}_4 \overline{\mathbf{D}}_2 \mathbf{D}_1$$

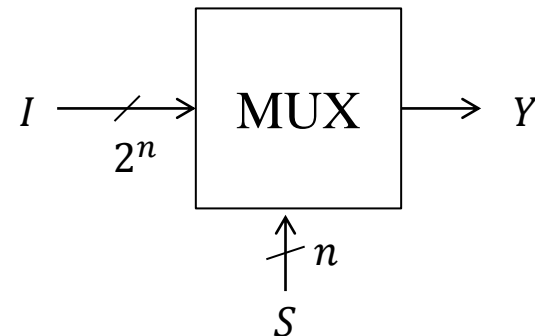
$$\mathbf{V} = \mathbf{D}_4 + \mathbf{D}_3 + \mathbf{D}_2 + \mathbf{D}_1 + \mathbf{D}_0$$

Selecting

- Selecting of data or information is a critical function in digital systems and computers
- Circuits that perform selecting have:
 - A set of information inputs from which the selection is made
 - A single output
 - A set of control lines for making the selection
- Logic circuits that perform selecting are called ***multiplexers***
- Selecting can also be done by three-state logic

Multiplexers (MUX) (Data Selectors)

- A multiplexer selects information from an input line and directs the information to an output line
- A typical multiplexer has **n control inputs** (S_{n-1}, \dots, S_0) called *selection inputs*, **2^n information inputs** (I_{2^n-1}, \dots, I_0), and **one output Y**
- A multiplexer can be designed to have m information inputs with **$m < 2^n$** as well as n selection inputs
- Multiplexers allow sharing of resources and reduce the cost by reducing the number of wires



2-to-1-Line MUX

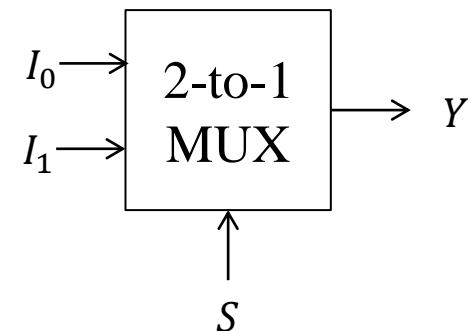
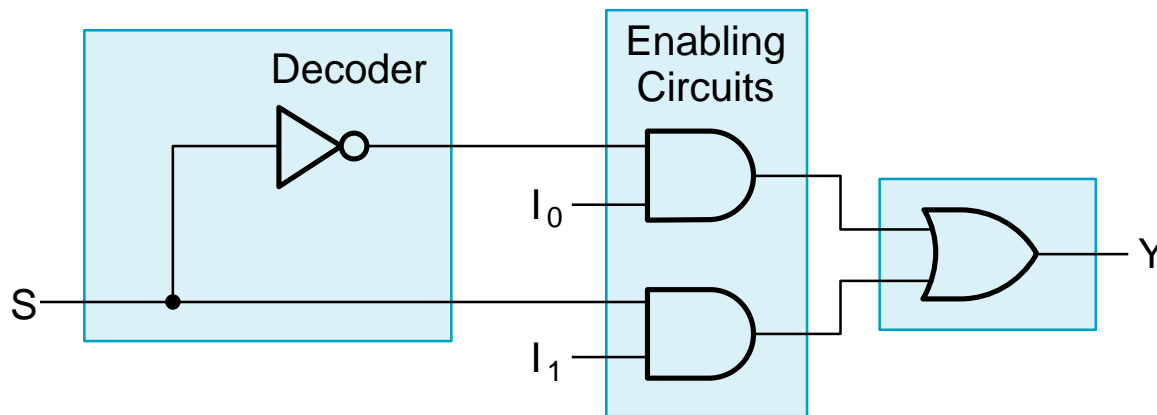
- Since $2 = 2^1$, $n = 1$
- The single selection variable S has two values:
 - $S = 0$ selects input I_0
 - $S = 1$ selects input I_1
- The equation:

$$Y = \bar{S}I_0 + SI_1$$
- The circuit:

S	I_1	I_0	Y	
0	0	0	0	$Y = I_0$
0	0	1	1	
0	1	0	0	
0	1	1	1	
1	0	0	0	$Y = I_1$
1	0	1	0	
1	1	0	1	
1	1	1	1	

S	Y
0	I_0
1	I_1

		I_1	
		0	1
S	0	0	1
	1	1	0
		I_0	
		0	1
0	0	1	1
1	1	0	0



2-to-1-Line MUX Cont.

- **Note the regions of the multiplexer circuit shown:**
 - **1-to-2-line Decoder**
 - **2 Enabling circuits**
 - **2-input OR gate**

- ***In general, for an 2^n -to-1-line multiplexer:***
 - ***n-to- 2^n -line decoder***
 - ***2^n 2-input AND gate***
 - ***One 2^n -input OR gate***

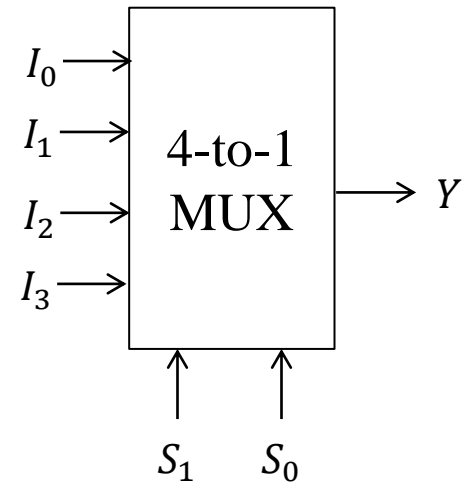
4-to-1-Line MUX

- Since $4 = 2^2$, $n = 2$
- There are two selection variables (S_1S_0) and they have four values:
 - $S_1S_0 = 00$ selects input I_0
 - $S_1S_0 = 01$ selects input I_1
 - $S_1S_0 = 10$ selects input I_2
 - $S_1S_0 = 11$ selects input I_3

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

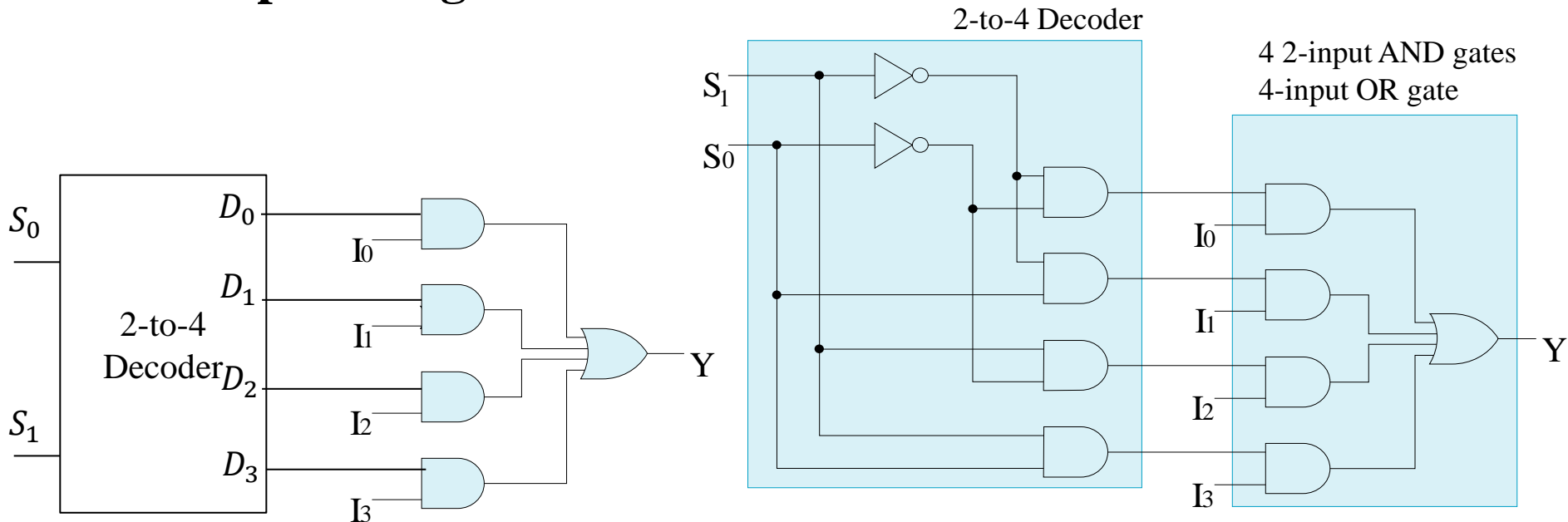
- The equation:

$$Y = \overline{S_1} \overline{S_0} I_0 + \overline{S_1} S_0 I_1 + S_1 \overline{S_0} I_2 + S_1 S_0 I_3$$



4-to-1-line MUX Cont.

- 2-to-4-line decoder
- 4 2-input AND gates
- 4-input OR gate

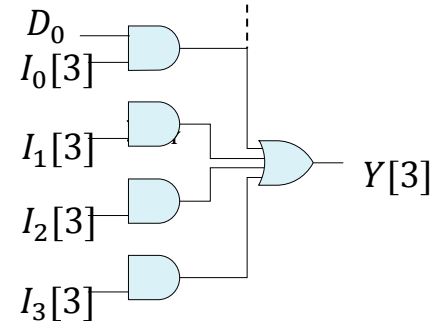
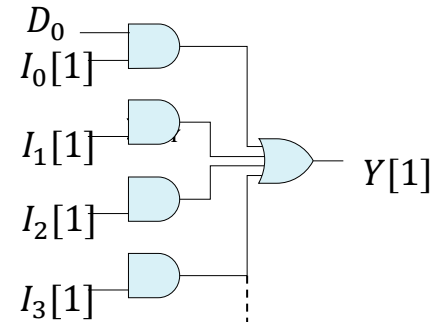
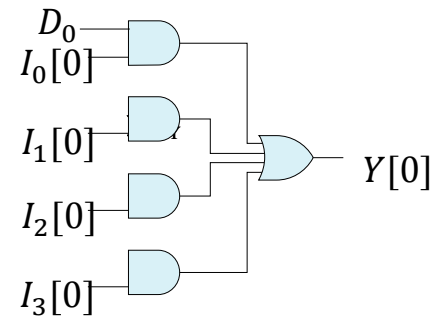
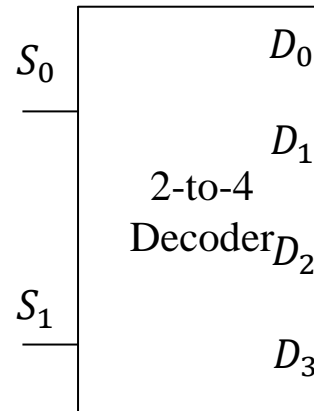
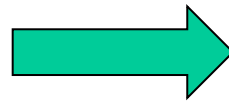
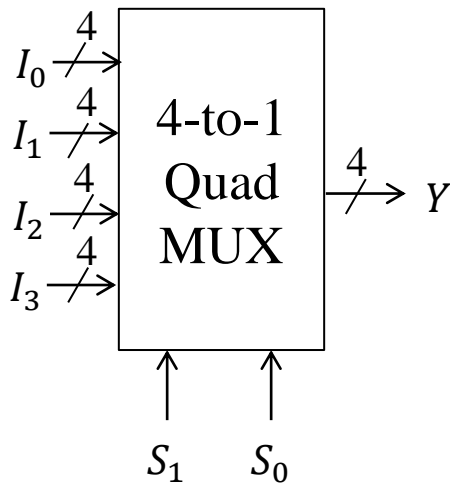


Homework

- **Implement 8-to-1-Line MUX and 64-to-1 MUX:**
 - **How many select lines are needed?**
 - **Decoder size?**
 - **How many 2-input AND gates are needed?**
 - **What is the size of the OR gate?**

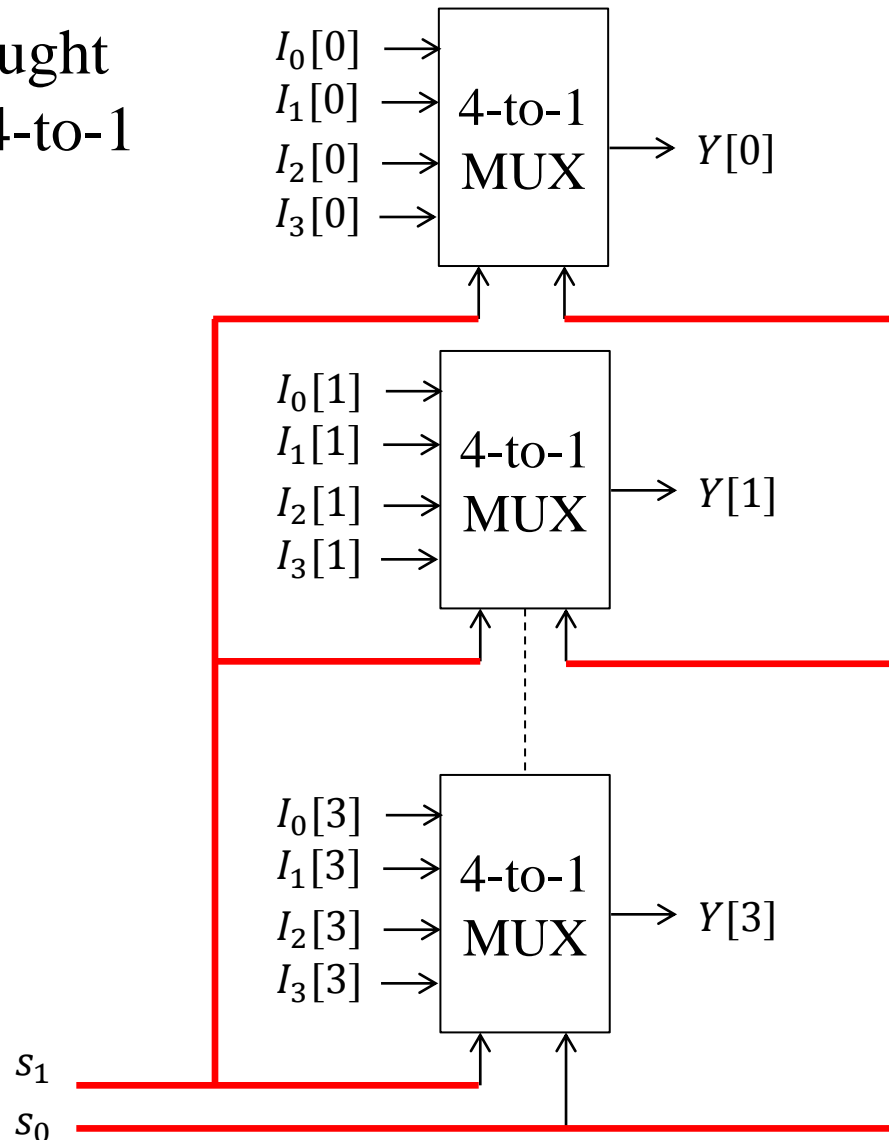
Multiplexer Width Expansion

- Select “vectors of bits” instead of “bits”
- Example: *4-to-1-line quad multiplexer*



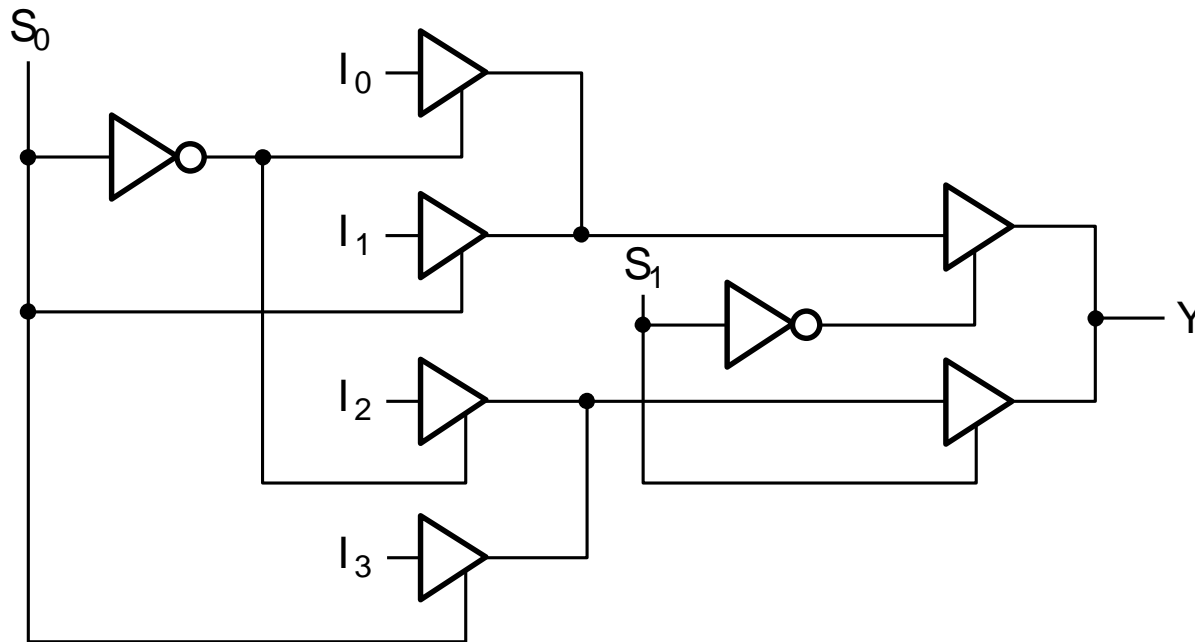
Multiplexer Width Expansion Cont.

- Can be thought of as four 4-to-1 MUXes:



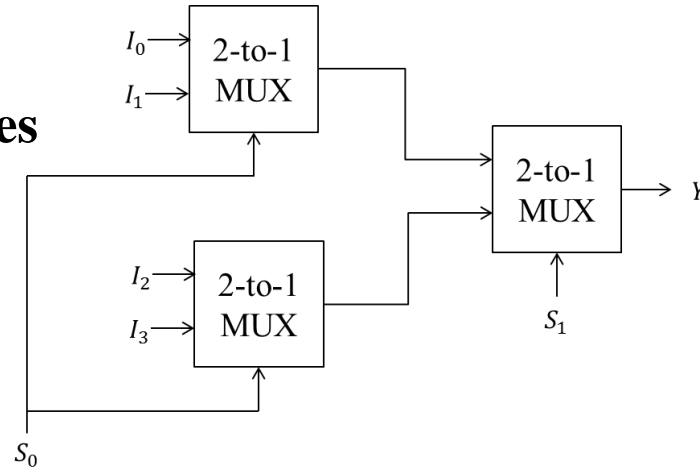
Other Selection Implementations

- Three-state logic



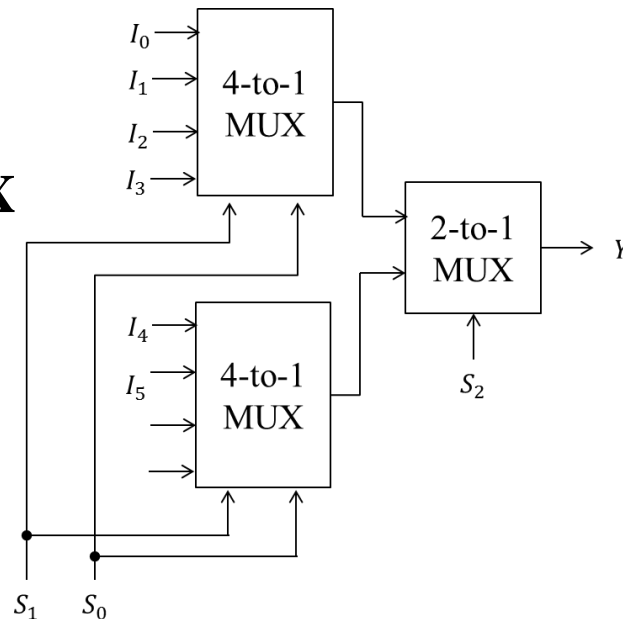
Building Large MUXEs from Smaller Ones

- 4-to-1 MUX using three 2-to-1 MUXEs



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

- 6-to-1 MUX using two 4-to-1 MUXEs and one 2-to-1 MUX



S_2	S_1	S_0	Y
0	0	0	I_0
0	0	1	I_1
0	1	0	I_2
0	1	1	I_3
1	0	0	I_4
1	0	1	I_5
1	1	0	X
1	1	1	X

Homework

- **Build an 8-to-1 MUX using:**
 - **Two 4-to-1 MUX and one 2-to-1 MUX**
 - **One 4-to-1 MUX and multiple 2-to-1 MUXes**
 - **Only 2-to-1 MUXes (How many MUXes are need?)**

Combinational Logic Implementation

- Multiplexer Approach 1

- Implement m functions of n variables with:
 - Sum-of-minterms expressions
 - An m -wide 2^n -to-1-line multiplexer
- Design:
 - Find the truth table for the functions
 - In the order they appear in the truth table:
 - Apply the function input variables to the multiplexer select inputs S_{n-1}, \dots, S_0
 - Label the outputs of the multiplexer with the output variables
 - Value-fix the information inputs to the multiplexer using the values from the truth table (for don't cares, apply either 0 or 1)

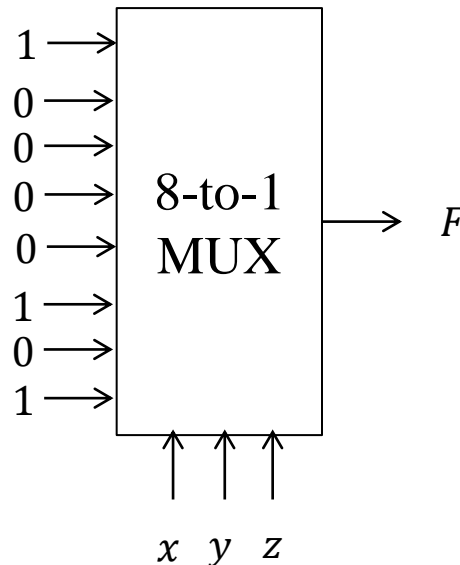
Example 1

- Implement the following function using a single MUX based on Approach 1 : $F(x, y, z) = \sum_m(0, 5, 7)$

- Solution:

- Single function $\rightarrow m = 1$
- 3 variables $\rightarrow n = 3 \rightarrow 8\text{-to-1 MUX}$
- Fill the truth table of F

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



Example2: Gray to Binary Code

- Design a circuit to convert a 3-bit Gray code to a binary code
- The formulation gives the truth table on the right

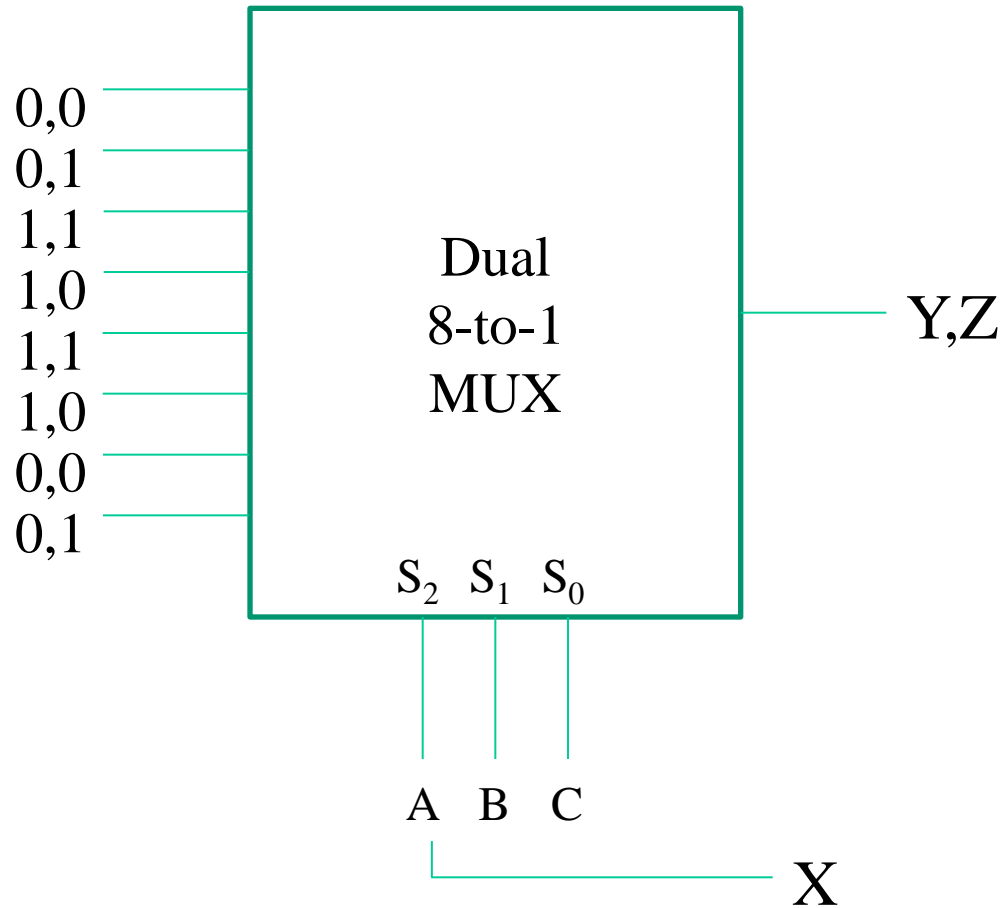
Gray Code ABC	Binary Code XYZ
000	000
001	001
011	010
010	011
110	100
111	101
101	110
100	111

Gray to Binary Code Cont.

- Rearrange the table so that the input combinations are in counting order
- It is obvious from this table that $X = A$. However, Y and Z are more complex
- Two functions (Y and Z) $\rightarrow m = 2$
- 3 variables (A , B , and C) $\rightarrow n = 3$
- Functions Y and Z can be implemented using a **dual** 8-to-1-line multiplexer by:
 - connecting A , B , and C to the multiplexer select inputs
 - placing Y and Z on the two multiplexer outputs
 - connecting their respective truth table values to the inputs

Gray Code ABC	Binary Code XYZ
000	000
001	001
010	011
011	010
100	111
101	110
110	100
111	101

Gray to Binary Code Cont.



Combinational Logic Implementation

- Multiplexer Approach 2

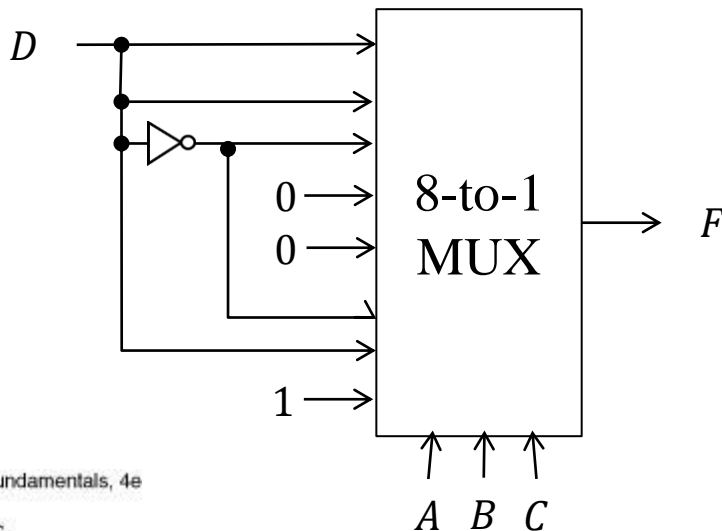
- Implement any m functions of n variables by using:
 - An m -wide $2^{(n-1)}$ -to-1-line multiplexer
 - A single inverter if needed
- Design:
 - Find the truth table for the functions
 - Based on the values of the most significant $(n-1)$ variables, separate the truth table rows into pairs
 - For each pair and output, define a rudimentary function of the least significant variable ($0, 1, X, \bar{X}$)
 - Connect the most significant $(n-1)$ variables to the select lines of the MUX, value-fix the information inputs to the multiplexer with the corresponding rudimentary functions
 - Use the inverter to generate the rudimentary function \bar{X}

Example 1

- Implement the following function using a single MUX and an inverter (if needed) based on Approach 2 :

$$F(A, B, C, D) = \sum_m (1, 3, 4, 10, 13, 14, 15)$$

- Solution:
 - Single function $\rightarrow m = 1$
 - 4 variables $\rightarrow n = 4 \rightarrow 8\text{-to-1 MUX}$
 - Fill the truth table of F



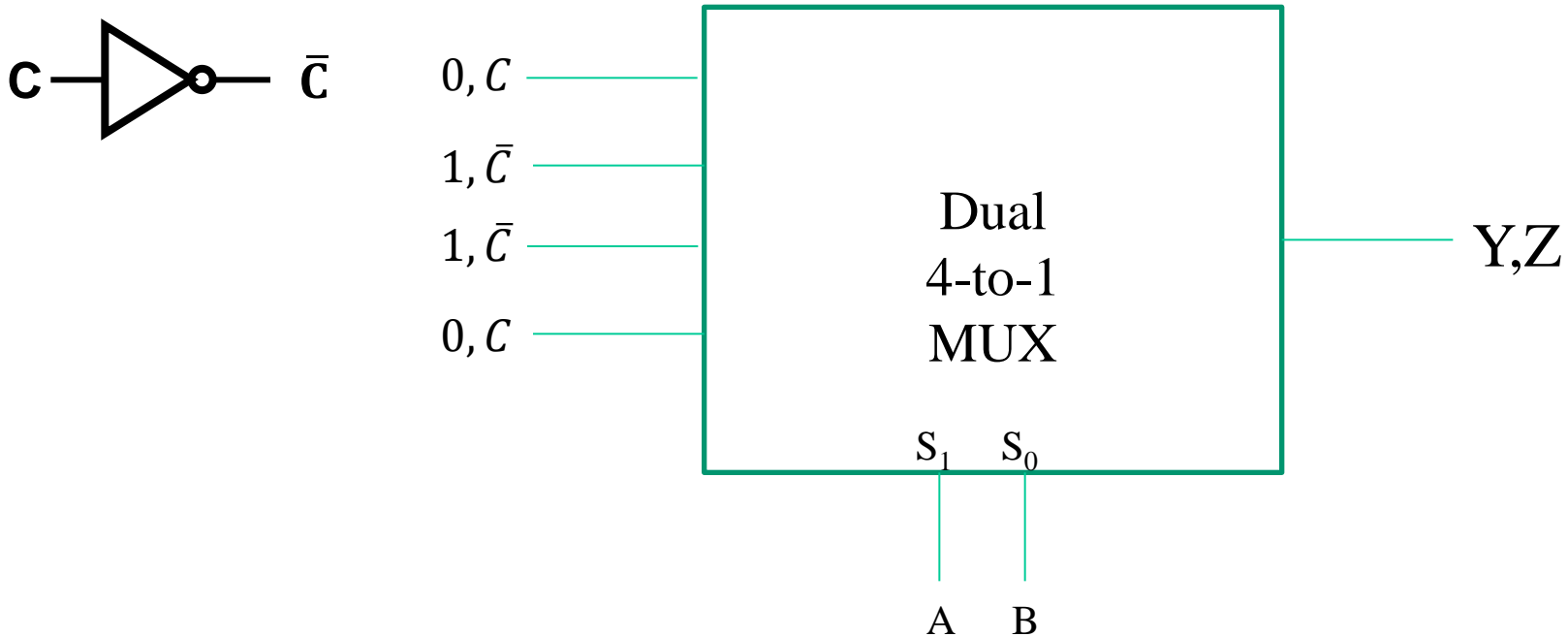
A	B	C	D	F	
0	0	0	0	0	$F = D$
0	0	0	1	1	
0	0	1	0	0	$F = D$
0	0	1	1	1	
0	1	0	0	1	$F = \overline{D}$
0	1	0	1	0	
0	1	1	0	0	$F = 0$
0	1	1	1	0	
1	0	0	0	0	$F = 0$
1	0	0	1	0	
1	0	1	0	1	$F = \overline{D}$
1	0	1	1	0	
1	1	0	0	0	$F = D$
1	1	0	1	1	
1	1	1	0	1	$F = 1$
1	1	1	1	1	

Example2: Gray to Binary Code

Gray Code ABC	Binary Code XYZ	Rudimentary Functions of C for Y	Rudimentary Functions of C for Z
000	000	$Y = 0$	$Z = C$
001	001		
010	011	$Y = 1$	$Z = \bar{C}$
011	010		
100	111	$Y = 1$	$Z = \bar{C}$
101	110		
110	100	$Y = 0$	$Z = C$
111	101		

Gray to Binary Code Cont.

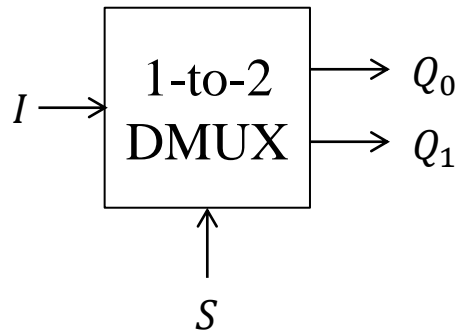
- Assign the variables and functions to the multiplexer inputs:



- Note that Approach2 reduces the cost by almost half compared to Approach1*

Demultiplexer (DMUX)

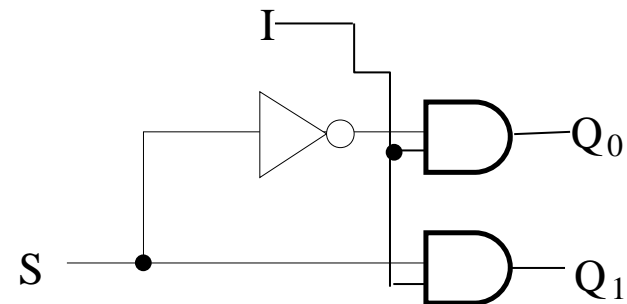
- Opposite of multiplexer
- Receives **one** input and directs it to one from **2^n** outputs based on **n-select** lines
- Example: 1-to-2 DMUX



$$Q_0 = \bar{S}I$$
$$Q_1 = SI$$

S	I	Q_1	Q_0
0	0	0	0
0	1	0	1
1	0	0	0
1	1	1	0

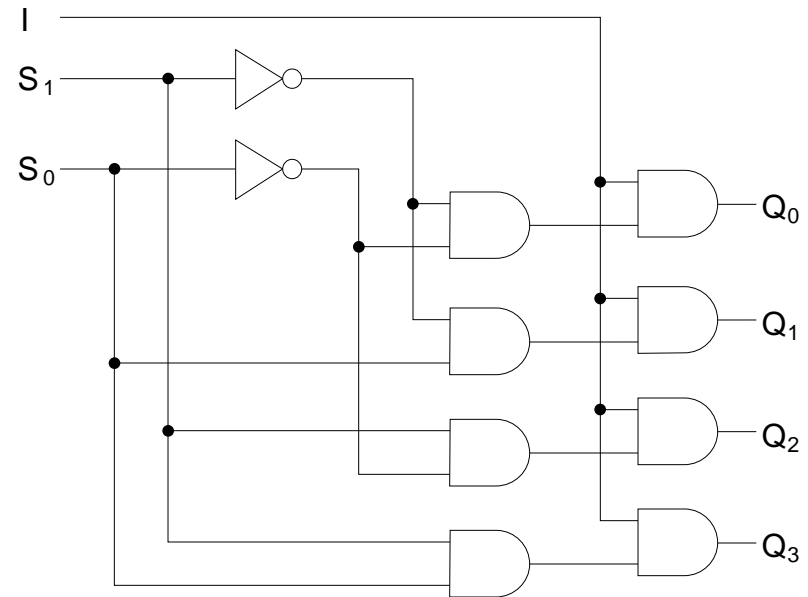
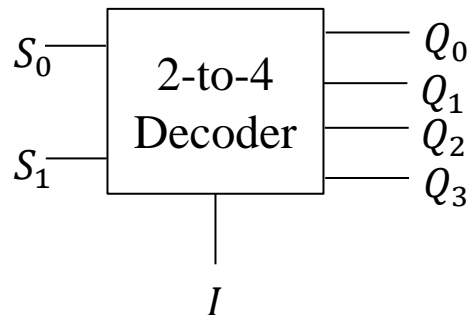
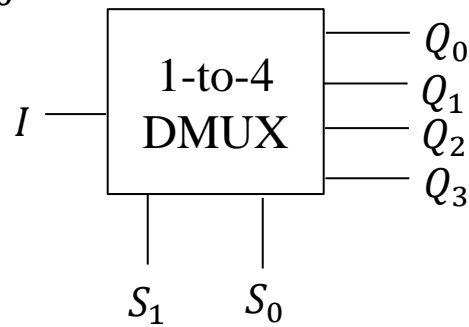
- ***DMUX \equiv Decoder with Enable***



1-to-4 DMUX

- $Q_0 = \bar{S}_1 \bar{S}_0 I$
- $Q_1 = \bar{S}_1 S_0 I$
- $Q_2 = S_1 \bar{S}_0 I$
- $Q_3 = S_1 S_0 I$

S_1	S_0	Q_3	Q_2	Q_1	Q_0
0	0	0	0	0	I
0	1	0	0	I	0
1	0	0	I	0	0
1	1	I	0	0	0



Terms of Use

- **All (or portions) of this material © 2008 by Pearson Education, Inc.**
- **Permission is given to incorporate this material or adaptations thereof into classroom presentations and handouts to instructors in courses adopting the latest edition of Logic and Computer Design Fundamentals as the course textbook.**
- **These materials or adaptations thereof are not to be sold or otherwise offered for consideration.**
- **This Terms of Use slide or page is to be included within the original materials or any adaptations thereof.**

Logic and Computer Design Fundamentals

Chapter 4 – Arithmetic Functions

Charles Kime & Thomas Kaminski

© 2008 Pearson Education, Inc.

(Hyperlinks are active in View Show mode)

Updated Thoroughly by Dr. Waleed Dweik

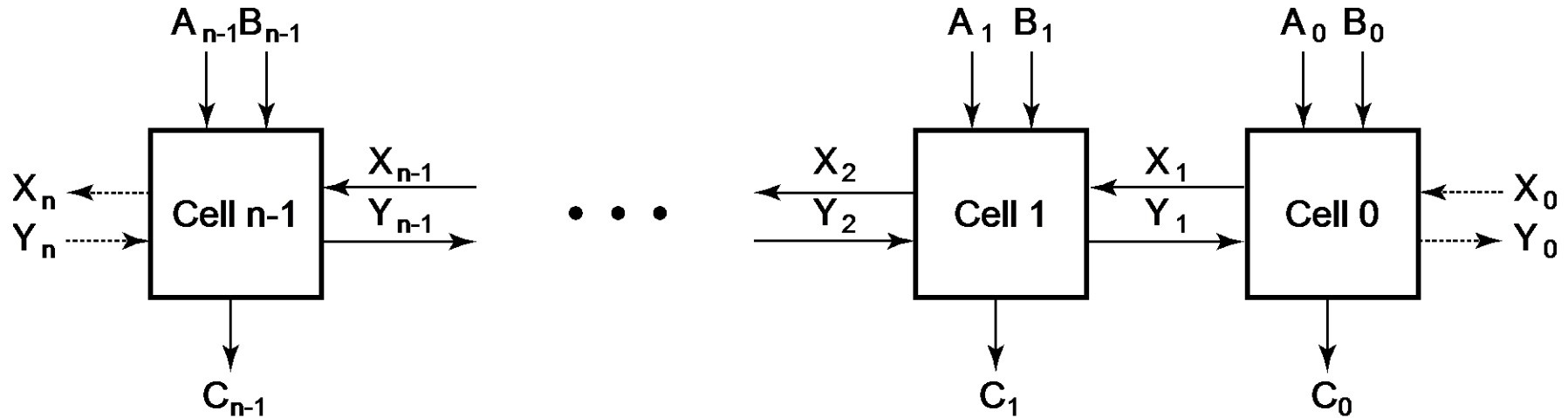
Overview

- Iterative combinational circuits
- Binary adders
 - Half and full adders
 - Ripple carry adders
- Binary subtraction
- Binary adder-subtractors
 - Signed binary numbers
 - Signed binary addition and subtraction
 - Overflow
- Binary multiplication
- Other arithmetic functions
 - Design by contraction

Iterative Combinational Circuits

- Arithmetic functions
 - Operate on binary vectors
 - Use the same sub-function in each bit position
- Can design functional block for the sub-function and repeat to obtain functional block for overall function
- ***Cell:*** sub-function block
- ***Iterative array:*** array of interconnected cells

Block Diagram of an Iterative Array



- Example: $n = 32$
 - Number of inputs = $32 \cdot 2 + 1 + 1 = 66$
 - Truth table rows = 2^{66}
 - Equations with up to 66 input variables
 - Equations with huge number of terms
 - Design impractical!
- Iterative array takes advantage of the regularity to make design feasible

Functional Blocks: Addition

- Binary addition used frequently
- Addition Development:
 - ***Half-Adder (HA)***: a 2-input bit-wise addition functional block
 - ***Full-Adder (FA)***: a 3-input bit-wise addition functional block
 - ***Ripple Carry Adder***: an iterative array to perform vector binary addition

Functional Block: Half-Adder

- A 2-input, 1-bit width binary adder that performs the following computations:

X	0	0	1	1
+ Y	+ 0	+ 1	+ 0	+ 1
C S	0 0	0 1	0 1	1 0

- A half adder adds two bits to produce a two-bit sum

- The sum is expressed as a *sum bit (S)* and a *carry bit (C)*

- The half adder can be specified as a truth table for S and C \Rightarrow

X	Y	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Logic Simplification and Implementation: Half-Adder

- The K-Map for S, C is:

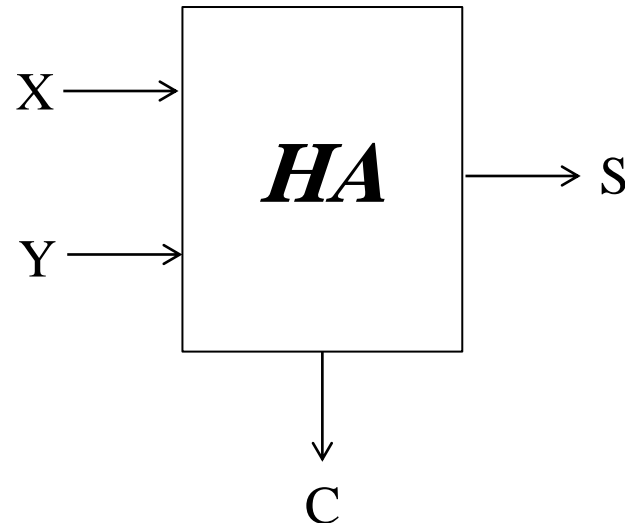
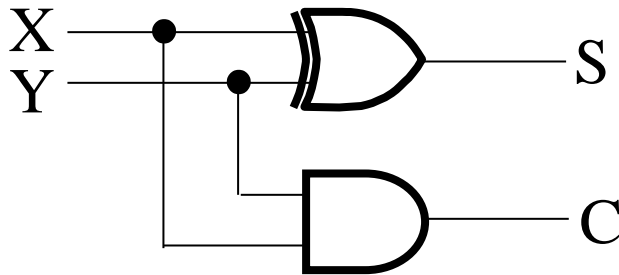
$$S = X \cdot \bar{Y} + \bar{X} \cdot Y = X \oplus Y$$

$$C = X \cdot Y$$

S	Y	
	0	1
X	1 ₂	3

C	Y	
	0	1
X	2	1 ₃

- The most common half adder implementation is:



Functional Block: Full-Adder

- A full adder is similar to a half adder, but includes a carry-in bit from lower stages. Like the half-adder, it computes a *sum bit (S)* and a *carry bit (C)*

- For a carry-in (Z) of 0, it is the same as the half-adder:

Z	0	0	0	0
X	0	0	1	1
+ Y	+ 0	+ 1	+ 0	+ 1
C S	0 0	0 1	0 1	1 0

- For a carry-in (Z) of 1:

Z	1	1	1	1
X	0	0	1	1
+ Y	+ 0	+ 1	+ 0	+ 1
C S	0 1	1 0	1 0	1 1

Logic Optimization: Full-Adder

- Full-Adder Truth Table:

X	Y	Z	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

- Full-Adder K-Map:

S

	Y		
	0	1	
X	1	0	1
	4	5	6
	Z		
	0	1	

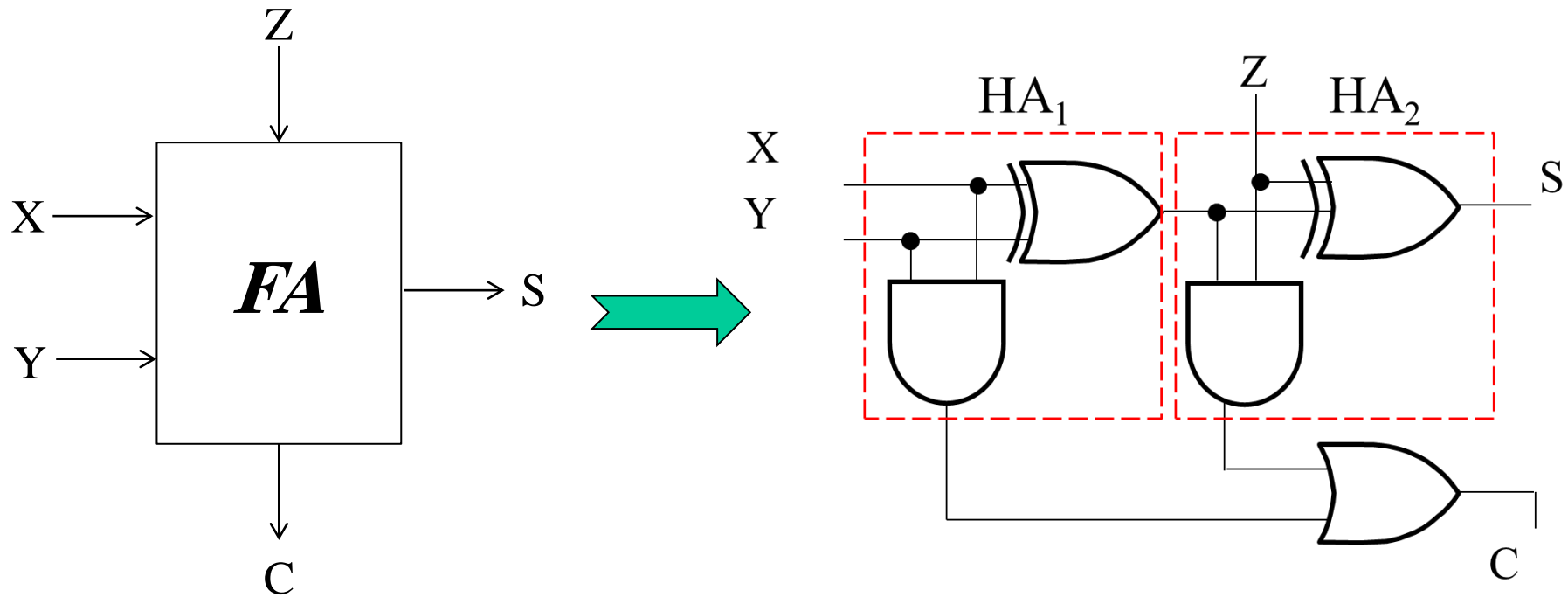
C

	Y		
	0	1	
X	0	1	1
	4	5	6
	Z		
	0	1	

$$S = \bar{X}\bar{Y}Z + \bar{X}Y\bar{Z} + X\bar{Y}\bar{Z} + XYZ \quad C = XZ + XY + YZ$$

- The S function is the three-bit XOR function (Odd Function):
 - $S = X \oplus Y \oplus Z$
- The Carry bit C is 1 if both X and Y are 1 (the sum is 2), or if the sum is 1 and a carry-in (Z) occurs. Thus C can be re-written as:
 - $C = XY + (X \oplus Y)Z$

Implementation: Full Adder



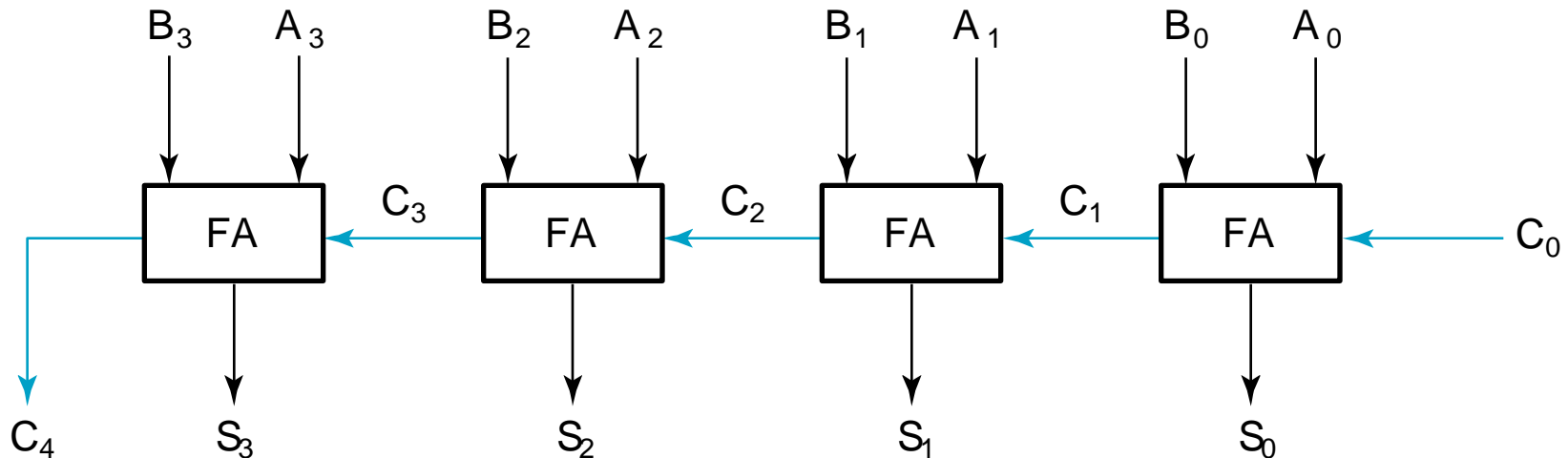
Binary Adders

- To add multiple operands, we “bundle” logical signals together into vectors and use functional blocks that operate on the vectors
- Example: *4-bit ripple carry adder* adds input vectors $A(3:0)$ and $B(3:0)$ to get a sum vector $S(3:0)$
- Note: carry-out of *cell i* becomes carry-in of *cell $i + 1$*

Description	Subscript	Name
	3 2 1 0	
Carry In	0 1 1 0	C_i
Augend	1 0 1 1	A_i
Addend	<u>0 0 1 1</u>	B_i
Sum	1 1 1 0	S_i
Carry out	0 0 1 1	C_{i+1}

4-bit Ripple-Carry Binary Adder

- A four-bit Ripple Carry Adder made from four 1-bit Full Adders:



Homework

- **Design a 4-bit ripple-carry adder using HA's only?**

Unsigned Subtraction

- When we subtract one bit from another, two bits are produced: *difference bit (D)* and *borrow bit (B)*

X	0	1	0	0
0	0	0	1	1
– Y	– 0	– 1	– 0	– 1
—	—	—	—	—
B D	0 0	1 1	0 1	0 0

- Algorithm:**
 - Subtract the *subtrahend (N)* from the *minuend (M)*
 - If no end borrow occurs, then $M \geq N$ and the result is a non-negative number and correct
 - If an end borrow occurs, then $N > M$ and the difference $(M - N + 2^n)$ is subtracted from 2^n , and a minus sign is appended to the result

Unsigned Subtraction

- Examples:

$$\begin{array}{r}
 0 \\
 1001 \\
 - 0111 \\
 \hline
 0010
 \end{array}$$

$$\begin{array}{r}
 1 \\
 0100 \\
 - 0111 \\
 \hline
 1101
 \end{array}$$

$$\begin{array}{r}
 1 \\
 10011 \\
 - 11110 \\
 \hline
 10101
 \end{array}$$

$$\begin{array}{r}
 0 \\
 10010110 \\
 - 01100100 \\
 \hline
 00110010
 \end{array}$$

$$\begin{array}{r}
 1 \\
 01100100 \\
 - 10010110 \\
 \hline
 11001110
 \end{array}$$

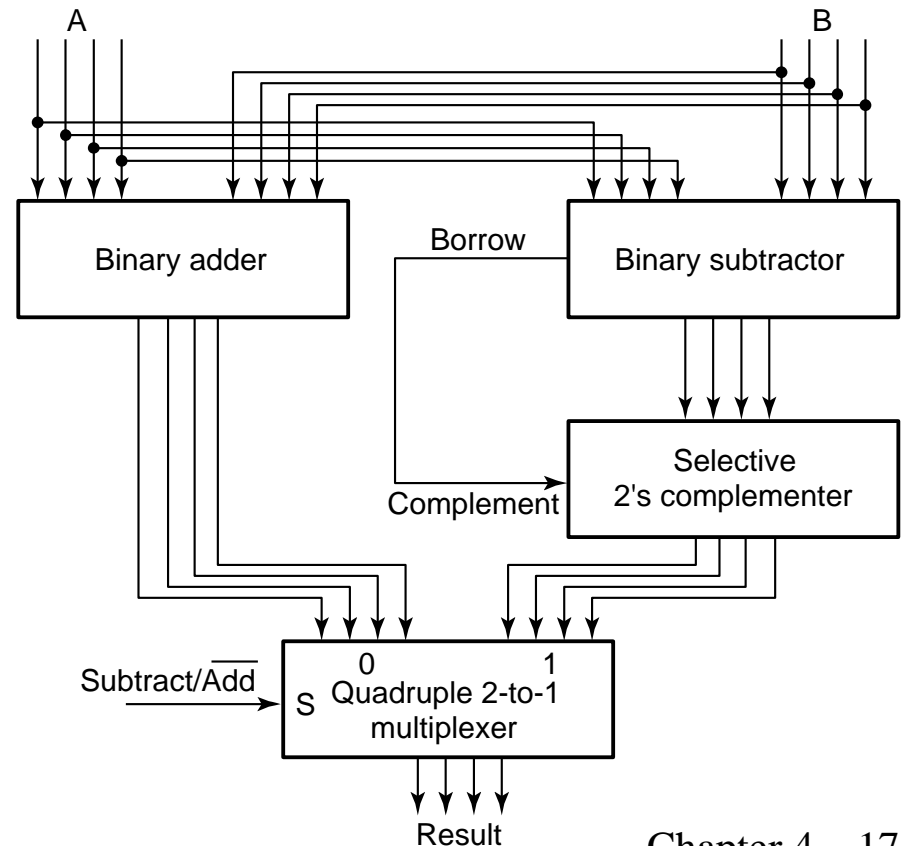
$$\begin{array}{r}
 10000 \\
 - 1101 \\
 \hline
 (-) 0011
 \end{array}$$

$$\begin{array}{r}
 100000 \\
 - 10101 \\
 \hline
 (-) 01011
 \end{array}$$

$$\begin{array}{r}
 100000000 \\
 - 11001110 \\
 \hline
 (-) 00110010
 \end{array}$$

Unsigned Subtraction (continued)

- The subtraction, $2^n - D$, is taking the **2's complement of D**
- To do both unsigned addition and unsigned subtraction requires:
 - Addition and Subtraction are performed in parallel and $\overline{\text{Subtract/Add}}$ chooses between them
- Quite complex!
- **Goal:** Shared simpler logic for both addition and subtraction
- Introduce complements as an approach



Complements

- For a number system with radix (r), there are two complements:
 - ***Diminished Radix Complement***
 - Famously known as *$(r - 1)$'s complement*
 - Examples:
 - 1's complement for radix 2
 - 9's complement for radix 10
 - For a number (N) with n -digits, the diminished radix complement is defined as:
 - $(r^n - 1) - N$
 - ***Radix Complement***
 - Famously known as *r 's complement* for radix r
 - Examples:
 - 2's complement in binary
 - 10's complement in decimal
 - For a number (N) with n -digits, r 's complement is defined as:
 - $r^n - N$, when $N \neq 0$
 - 0 , when $N = 0$

Diminished Radix Complement

- If N is a number of n -digits with radix (r) , then
 - $N + (r - 1)$'s complement of $N = \underbrace{(r - 1)(r - 1)(r - 1) \dots (r - 1)}_{n\text{-digits}}$
 - *The $(r - 1)$'s complement can be computed by subtracting each digit from $(r - 1)$*
- Example: Find 1's complement of $(1011)_2$
 - $r = 2, n = 4$
 - Answer is $(2^4 - 1) - (1011)_2 = (0100)_2$
 - Notice that $(1011)_2 + (0100)_2 = (1111)_2$ which is $\underbrace{(2 - 1)(2 - 1)(2 - 1)(2 - 1)}_{4\text{-digits}}$
- Example: Find 9's complement of $(45)_{10}$
 - $r = 10, n = 2$
 - Answer is $(10^2 - 1) - (45)_{10} = (54)_{10}$
 - Notice that $(45)_{10} + (54)_{10} = (99)_{10}$ which is $\underbrace{(10 - 1)(10 - 1)}_{2\text{-digits}}$
- Example: Find 7's complement of $(671)_8$
 - $r = 8, n = 3$
 - Answer is $(8^3 - 1) - (671)_8 = (106)_8$
 - Notice that $(671)_8 + (106)_8 = (777)_8$ which is $\underbrace{(8 - 1)(8 - 1)(8 - 1)}_{3\text{-digits}}$

Binary 1's Complement

- For $r = 2$, $N = 01110011_2$, $n = 8$ (8 digits):

$$(r^n - 1) = 256 - 1 = 255_{10} \text{ or } 11111111_2$$

- The 1's complement of 01110011_2 is then:

$$\begin{array}{r} 11111111 \\ - \underline{01110011} \\ 10001100 \end{array}$$

- Since the $2^n - 1$ factor consists of all 1's and since $1 - 0 = 1$ and $1 - 1 = 0$, the one's complement is obtained ***by complementing each individual bit (bitwise NOT)***

Radix Complement

- For number N with n -digit and radix (r):
 - If $N \neq 0$, r 's complement of $N = r^n - N$
 - *r 's complement = $(r-1)$'s complement + 1*
 - If $N = 0$, r 's complement of $N = 0$
- Example: Find 10's complement of $(92)_{10}$
 - $r = 10, n = 2$
 - Answer is $10^2 - (92)_{10} = (8)_{10}$
 - Notice that 9's complement of $(92)_{10}$ is $(7)_{10}$
 - *10's complement = 9's complement + 1*
- Example: Find 16's complement of $(3AE7)_{16}$
 - $r = 16, n = 4$
 - Answer is $16^4 - (3AE7)_{16} = (10000)_{16} - (3AE7)_{16} = (C519)_{16}$
 - 15's complement = $(C518)_{16} \rightarrow 16$'s complement = $(C518)_{16} + 1 = (C519)_{16}$

Binary 2's Complement

- For $r = 2$, $N = 01110011_2$, $n = 8$ (8 digits), we have:
 - $(r^n) = 256_{10}$ or 100000000_2
- The 2's complement of 01110011 is then:

$$\begin{array}{r} 100000000 \\ - \underline{01110011} \\ 10001101 \end{array}$$

- *Note the result is the 1's complement plus 1, a fact that can be used in designing hardware*
- *Remember the 2's complement of $(000..00)_2$ is $(000..00)_2$*
- *Complement of a complement restores the number to its original value:*
 - *The Complement of complement $N = 2^n - (2^n - N) = N$*

Alternate 2's Complement Method

- Given: an n -bit binary number, beginning at the least significant bit and proceeding upward:
 - Copy all least significant 0's
 - Copy the first 1
 - Complement all bits thereafter

- 2's Complement Example:

10010100

- Copy underlined bits:

100

- and complement bits to the left:

01101100

Subtraction with 2's Complement

- For n-digit, unsigned numbers M and N, find $M - N$ in base 2:
 - Add the 2's complement of the subtrahend N to the minuend M:
 - $M - N \longrightarrow M + (2^n - N) = M - N + 2^n$
- If $M \geq N$, the *sum* produces end carry 2^n which is discarded; and from above, $M - N$ remains
- If $M < N$, the *sum* does not produce end carry, and from above, is equal to $2^n - (N - M)$ which is the 2's complement of $(N - M)$
- To obtain the result $-(N - M)$, take the 2's complement of the sum and place a “-” to its left

Unsigned 2's Complement Subtraction Example: (M > N)

- Find $01010100_2 - 01000011_2$

$$\begin{array}{r} 01010100 \\ - 01000011 \\ \hline \end{array} \xrightarrow{\text{2's comp}} \begin{array}{r} 101010100 \\ + 10111101 \\ \hline 00010001 \end{array}$$

- The carry of 1 indicates that no correction of the result is required

Unsigned 2's Complement Subtraction Example: (M < N)

- Find $01000011_2 - 01010100_2$

$$\begin{array}{r}
 \mathbf{01000011} \\
 - \mathbf{01010100} \\
 \hline
 \end{array}
 \xrightarrow{\text{2's comp}}
 \begin{array}{r}
 \mathbf{0} \mathbf{01000011} \\
 + \mathbf{10101100} \\
 \hline
 \mathbf{11101111} \\
 \hline
 \mathbf{00010001}
 \end{array}
 \xrightarrow{\text{2's comp}}$$

- The carry of 0 indicates that a correction of the result is required
- Result = $-(00010001)$

Signed Integers

- *Positive numbers and zero* can be represented by unsigned *n-digit, radix r* numbers. ***We need a representation for negative numbers***
- To represent a sign (+ or –) we need exactly one more bit of information (1 binary digit gives $2^1 = 2$ elements which is exactly what is needed).
- Since computers use binary numbers, by convention, ***the most significant bit is interpreted as a sign bit.***

$$\mathbf{s \ a_{n-2} \ \dots \ a_2 a_1 a_0}$$

where:

s = 0 for Positive numbers

s = 1 for Negative numbers

and $a_i = 0$ or 1 represent the magnitude in some form

Signed Integer Representations

- ***Signed-Magnitude:*** here the $(n - 1)$ digits are interpreted as a positive magnitude
 - Max = $+(2^{n-1} - 1)$
 - Min = $-(2^{n-1} - 1)$
 - Two representation for zero (i.e. ± 0)
- ***Signed-Complement:*** here the digits are interpreted as the rest of the complement of the number. There are two possibilities here:
 - ***Signed 1's Complement:*** Uses 1's Complement Arithmetic
 - Max = $+(2^{n-1} - 1)$
 - Min = $-(2^{n-1} - 1)$
 - Two representation for zero (i.e. ± 0)
 - ***Signed 2's Complement:*** Uses 2's Complement Arithmetic
 - Max = $+(2^{n-1} - 1)$
 - Min = -2^{n-1}
 - Single representation for zero

Signed Integer Representation Example

- $r = 2, n = 3$

Number	Signed-Magnitude	1's Complement	2's Complement
+3	011	011	011
+2	010	010	010
+1	001	001	001
+0	000	000	000
-0	100	111	----
-1	101	110	111
-2	110	101	110
-3	111	100	101
-4	----	----	100

- **Represent the number -9 using 8-bits**
 - Sign-Magnitude = 10001001
 - 1's complement = 11110110
 - 2's complement = 11110111

2's Complement Signed Numbers

- Signed 2's complement is the most common representation for signed numbers
 - Focus of the course
- For any n-bit 2's complement signed number ($b_{n-1}b_{n-2}b_{n-3} \dots b_2b_1b_0$), the decimal value is given by

$$Value = (-2^{n-1} \times b_{n-1}) + \sum_{i=0}^{n-2} 2^i \times b_i$$

- Example: What is value of the 2's complement number $(100111)_2$?

$$Value = -2^5 \times 1 + 7 = -25$$

Signed-2's Complement Arithmetic

■ Addition:

- Add the numbers including the sign bits
- Discard the carry out of the sign bits

■ Subtraction:

- Form the complement of the number you are subtracting
- Follow the same rules for addition
- $A - B = A + (-B) = A + (\bar{B} + 1)$

Signed 2's Complement Addition

$$\begin{array}{r} (+6) \quad 00000110 \\ + \quad + \\ (+13) \quad \underline{00001101} \\ \hline 00010011 \quad (+19) \end{array}$$

$$\begin{array}{r} (-6) \quad 11111010 \\ + \quad + \\ (+13) \quad \underline{00001101} \\ \hline \boxed{1}00000111 \quad (+7) \end{array}$$

Carry-out is ignored

$$\begin{array}{r} (+6) \quad 00000110 \\ + \quad + \\ (-13) \quad \underline{11110011} \\ \hline 11111001 \quad (-7) \end{array}$$

$$\begin{array}{r} (-6) \quad 11111010 \\ + \quad + \\ (-13) \quad \underline{11110011} \\ \hline \boxed{1}11101101 \quad (-19) \end{array}$$

Carry-out is ignored

Signed 2's Complement Subtraction

$$\begin{array}{r} (+6) \quad 00000110 \\ - \quad + \\ (+13) \quad \underline{11110011} \\ \hline 11111001 \quad (-7) \end{array}$$

$$\begin{array}{r} (-6) \quad 11111010 \\ - \quad + \\ (+13) \quad \underline{11110011} \\ \hline \boxed{1}11101101 \quad (-19) \end{array}$$

Carry-out is ignored

$$\begin{array}{r} (+6) \quad 00000110 \\ - \quad + \\ (-13) \quad \underline{00001101} \\ \hline 00010011 \quad (+19) \end{array}$$

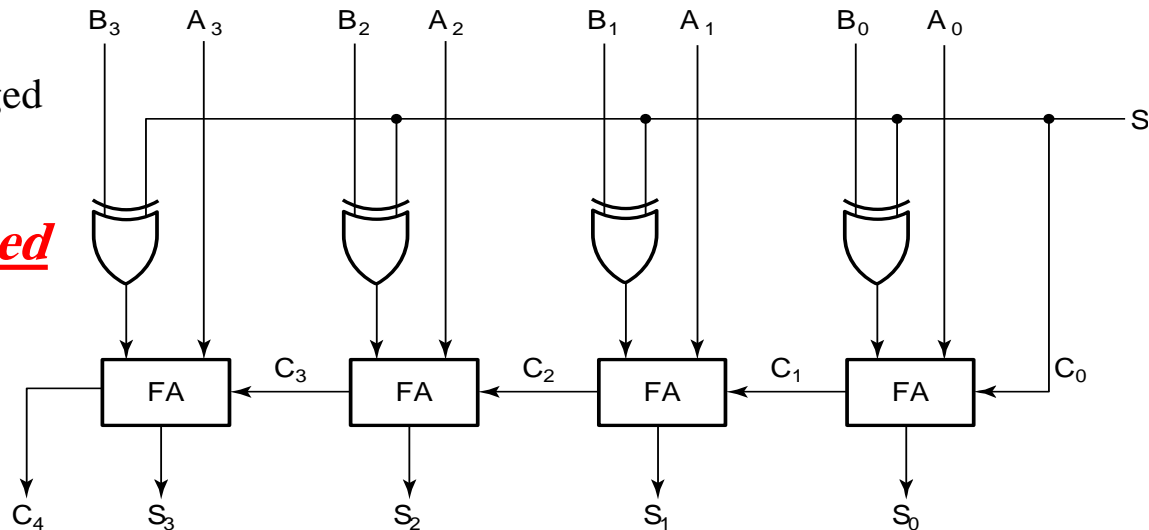
$$\begin{array}{r} (-6) \quad 11111010 \\ - \quad + \\ (-13) \quad \underline{00001101} \\ \hline \boxed{1}00000111 \quad (+7) \end{array}$$

Carry-out is ignored

2's Complement Adder/Subtractor for Signed Numbers

- ***Subtraction can be done by addition of the 2's Complement***
 1. Complement each bit (1's Complement)
 2. Add 1 to the result
- The circuit shown computes $A + B$ and $A - B$:
- Subtract ($S = 1$): $A - B = A + (2^n - B) = A + \bar{B} + 1$
 - The 2's complement of B is formed by using XORs to form the 1's complement and adding the 1 applied to C_0
- Add ($S = 0$): $A + B$
 - B is passed through unchanged

- ***Same Hardware for Signed and Unsigned numbers***



Overflow Detection

- In computers, the number of bits is fixed
- **Overflow** occurs if $n + 1$ bits are required to contain the result from an n -bit addition or subtraction
- **Unsigned number overflow** is detected from the **end carry-out** when **adding** two unsigned numbers
 - Overflow is **impossible** for unsigned subtraction

$$\begin{array}{r} (8) \quad 1000 \\ + \quad + \\ (12) \quad \underline{1100} \\ \hline 10100 \quad (4) \end{array}$$

Carry-out = 1 → Overflow

- **Signed number overflow** can occur for:
 - Addition of two operands with the same sign
 - Subtraction of operands with different signs

Signed-number Overflow Detection

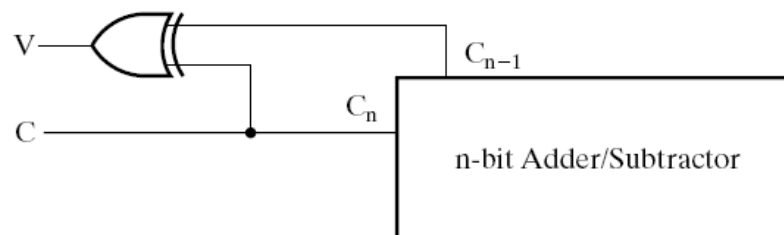
- Signed number cases with carries C_n and C_{n-1} shown for correct result signs:

0	0	0	0	1	1	1	1
0	0	0	0	1	1	1	1
+	<u>0</u>	-	<u>1</u>	-	<u>0</u>	+	<u>1</u>
0	0	0	0	1	1	1	1

- Signed number cases with carries shown for erroneous result signs (indicating overflow):

0	1	0	1	1	0	1	0
0	0	0	0	1	1	1	1
+	<u>0</u>	-	<u>1</u>	-	<u>0</u>	+	<u>1</u>
1	1	1	1	0	0	0	0

- Simplest way to implement signed overflow is: $V = C_n \oplus C_{n-1}$*



Signed-number Overflow Examples

- 8-bit signed number range between: -128 to +127

$$\begin{array}{r} (+70) \ 01000110 \\ + \quad + \\ (+80) \ \underline{01010000} \\ \quad \quad 10010110 \ (-106) \\ V = C_7 \oplus C_8 = 1 \oplus 0 = 1 \end{array}$$

$$\begin{array}{r} (-70) \ 10111010 \\ + \quad + \\ (-80) \ \underline{10110000} \\ \quad \quad 101101010 \ (+106) \\ V = C_7 \oplus C_8 = 0 \oplus 1 = 1 \end{array}$$

$$\begin{array}{r} (+70) \ 01000110 \\ - \quad + \\ (-80) \ \underline{01010000} \\ \quad \quad 10010110 \ (-106) \\ V = C_7 \oplus C_8 = 1 \oplus 0 = 1 \end{array}$$

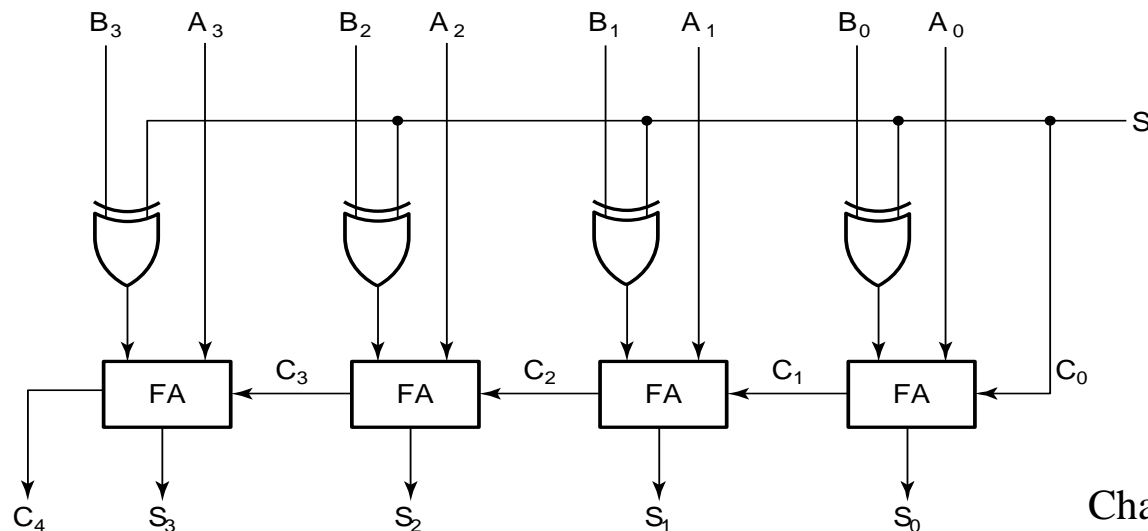
$$\begin{array}{r} (-70) \ 10111010 \\ - \quad + \\ (+80) \ \underline{10110000} \\ \quad \quad 101101010 \ (+106) \\ V = C_7 \oplus C_8 = 0 \oplus 1 = 1 \end{array}$$

Other Arithmetic Functions

- Incrementing
- Decrementing
- Multiplication by Constant
- Division by Constant
- Zero Fill and Extension

Incrementer and Decrementer

- Start with Adder/Subtractor
- Set $B_3B_2B_1B_0 = 0001$
- For Incrementer \rightarrow Set $S = 0$
- For Decrementer \rightarrow Set $S = 1$
- For Incrementer/Decrementer \rightarrow S remains variable
 - In this case, the full adder complexity stays the same in the typical bit positions (i.e. Cell₁ and Cell₂)



Binary Multiplication

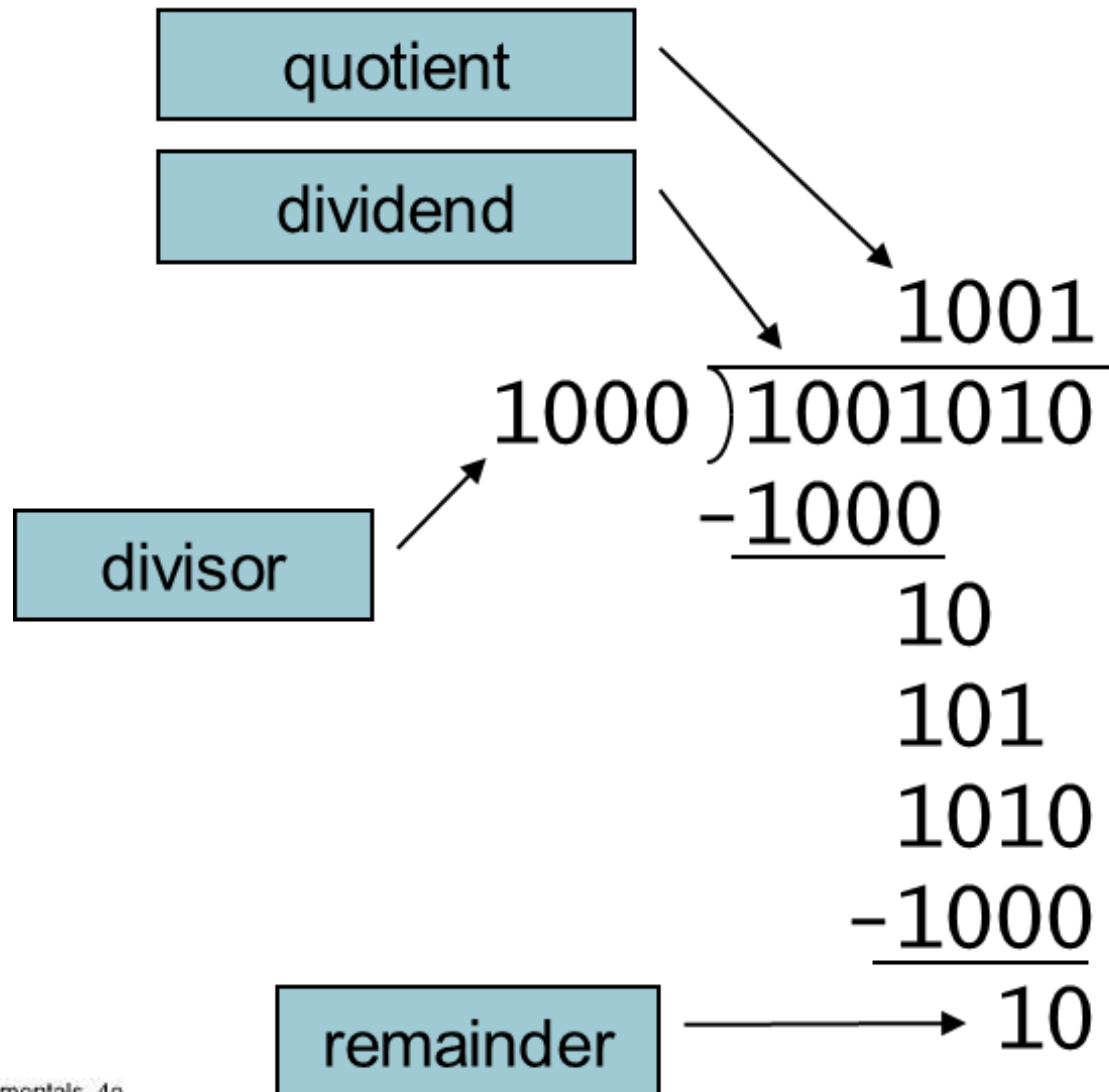
The binary multiplication table is simple:

$$0 * 0 = 0 \quad | \quad 1 * 0 = 0 \quad | \quad 0 * 1 = 0 \quad | \quad 1 * 1 = 1$$

Extending multiplication to multiple digits:

Multiplicand	1011
Multiplier	<u>x 101</u>
Partial Products	1011
	0000 -
	<u>1011 - -</u>
Product	110111

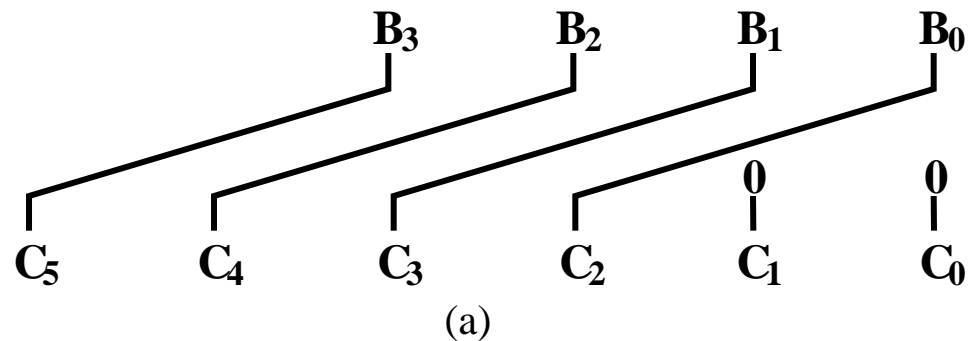
Binary Division



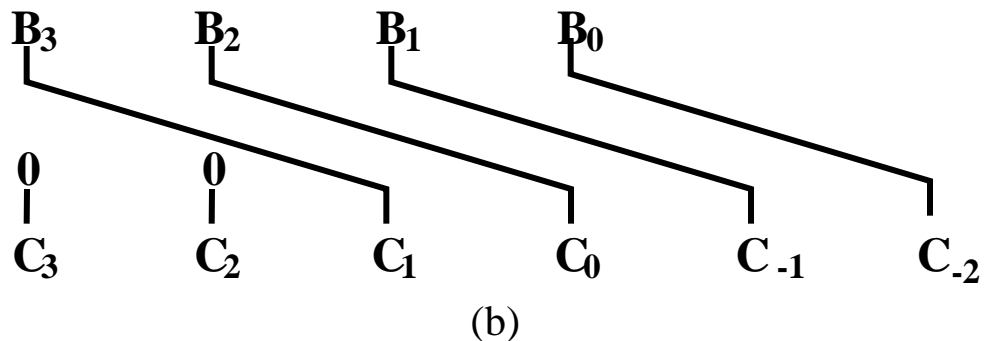
Multiplication/Division by 2^n

- Multiplication by 2^n : Shift left by n
 - Add n-zeros on the left
- Division by 2^n : Shift right by n
 - Add n-zeros on the right

- **Multiplication by $(100)_2$**
 - **Shift left by 2**

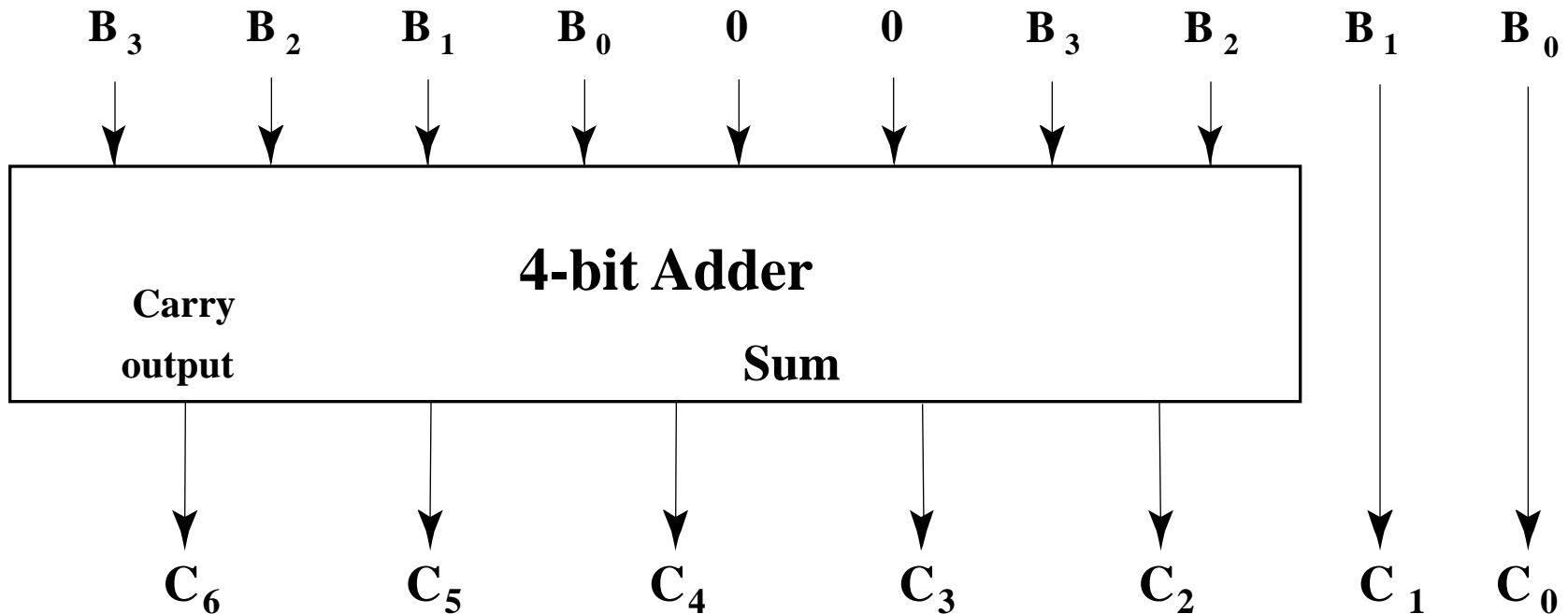


- **Division by $(100)_2$**
 - **Shift right by 2**
 - **Quotient = $C_3C_2C_1C_0$**
 - **Remainder = $C_{-1}C_{-2}$**



Multiplication by a Constant

- Multiplication of $B(3:0)$ by 101
- See text Figure 4-10 in page 171 for contraction



Zero Fill

- ***Zero fill:*** filling an m -bit operand with 0s to become an n -bit operand with $n > m$
- Filling usually is applied to the MSB end of the operand, but can also be done on the LSB end
- **Example: 11110101 filled to 16 bits**
 - MSB end: 0000000011110101 (**Zero Extension**)
 - LSB end: 1111010100000000

Extension

- ***Extension:*** increase in the number of bits at the MSB end of an operand by using a complement representation

- **Copies the MSB of the operand into the new positions**
- **Positive operand example - 01110101 extended to 16 bits:**

000000001110101

- **Negative operand example - 11110101 extended to 16 bits:**

1111111111110101

Hexadecimal, Octal, BCD Addition

Hexadecimal and Octal Addition:

- Add each digit then take modulus (r)

$$\begin{array}{r}
 (59F)_{16} \\
 + \\
 (E46)_{16} \\
 \hline
 (13E5)_{16}
 \end{array}$$

$$\begin{array}{r}
 (762)_8 \\
 + \\
 (345)_8 \\
 \hline
 (1327)_8
 \end{array}$$

BCD Addition:

- Add each 4-bit together
 - If the binary sum is greater than 1001
 - Add 0110 to the result

$$\begin{array}{r}
 (448)_{10} \\
 + \\
 (489)_{10} \\
 \hline
 (937)_{10}
 \end{array}
 \qquad
 \begin{array}{r}
 (0100\ 0100\ 1000)_{\text{BCD}} \\
 + \\
 (0100\ 1000\ 1001)_{\text{BCD}} \\
 \hline
 1001\ 1101^1\ 0001 \\
 + \quad \quad \quad 0110\ 0110 \\
 \hline
 1001^1\ 0011^1\ 0111
 \end{array}$$

Terms of Use

- **All (or portions) of this material © 2008 by Pearson Education, Inc.**
- **Permission is given to incorporate this material or adaptations thereof into classroom presentations and handouts to instructors in courses adopting the latest edition of Logic and Computer Design Fundamentals as the course textbook.**
- **These materials or adaptations thereof are not to be sold or otherwise offered for consideration.**
- **This Terms of Use slide or page is to be included within the original materials or any adaptations thereof.**

Logic and Computer Design Fundamentals

Chapter 5 – Sequential Circuits

Charles Kime & Thomas Kaminski

© 2008 Pearson Education, Inc.
(Hyperlinks are active in View Show mode)

Updated based on Dr. Fahed Jubair slides

Overview

- **Part 1 - Storage Elements and Analysis**
 - **Introduction to sequential circuits**
 - **Types of sequential circuits**
 - **Storage elements**
 - **Latches**
 - **Flip-flops**
 - **Sequential circuit analysis**
 - **State tables**
 - **State diagrams**
 - **Equivalent states**
 - **Moore and Mealy Models**
- **Part 2 - Sequential Circuit Design**

Introduction to Sequential Circuits

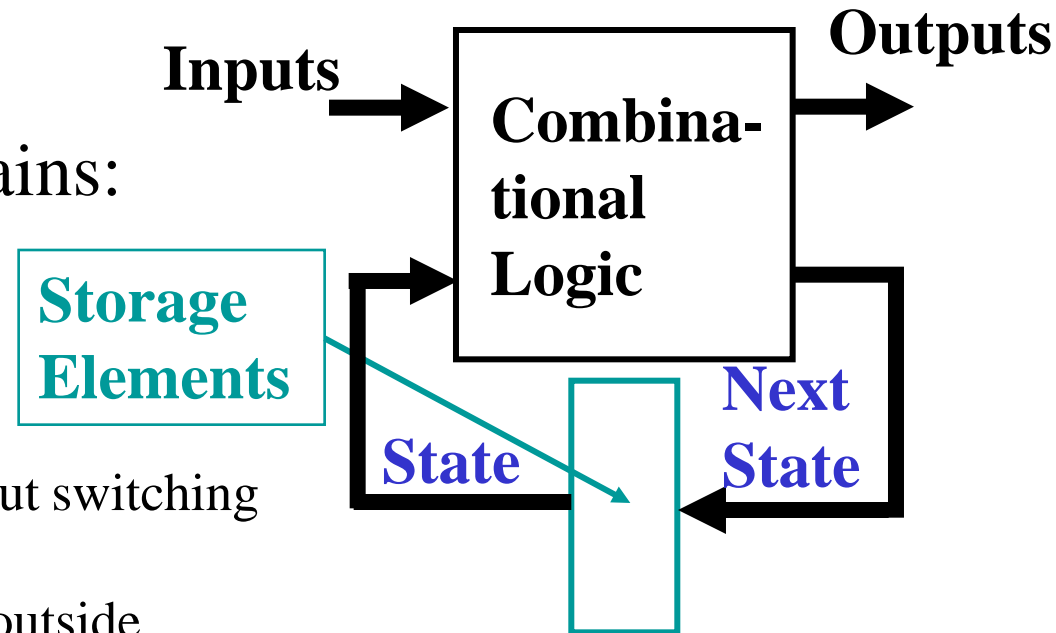
- A Sequential circuit contains:

- ***Storage elements:***

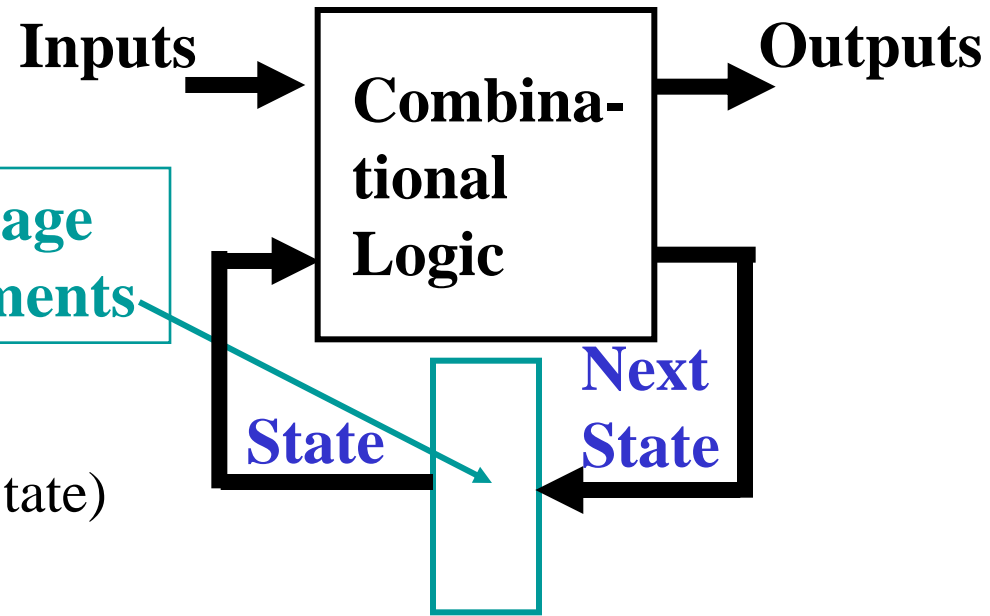
- Latches or Flip-Flops

- ***Combinational Logic:***

- Implements a multiple-output switching function
- Inputs are signals from the outside
- Outputs are signals to the outside
- Other inputs, State or Present State, are signals from storage elements
- The remaining outputs, Next State are inputs to storage elements



Introduction to Sequential Circuits



- **Combinatorial Logic**

- ***Next state function***

$$\text{Next State} = f(\text{Inputs}, \text{State})$$

OR $\text{Next State} = f(\text{State})$

- ***Output function (Mealy)***

$$\text{Outputs} = g(\text{Inputs}, \text{State})$$

- ***Output function (Moore)***

$$\text{Outputs} = g(\text{State})$$

- **Output function type depends on specification and affects the design significantly**

Types of Sequential Circuits

- **Depends on the times at which:**
 - storage elements observe their inputs, and
 - storage elements change their state
- **Synchronous**
 - Behavior defined from knowledge of its signals at *discrete* instances of time
 - **Storage elements observe inputs and can change state only in relation to a timing signal (*clock pulses* from a *clock*)**
 - *Simple to design but slow*
- **Asynchronous**
 - Behavior defined from knowledge of inputs at any instant of time and the order in *continuous* time in which inputs change
 - *Complex to design but fast*

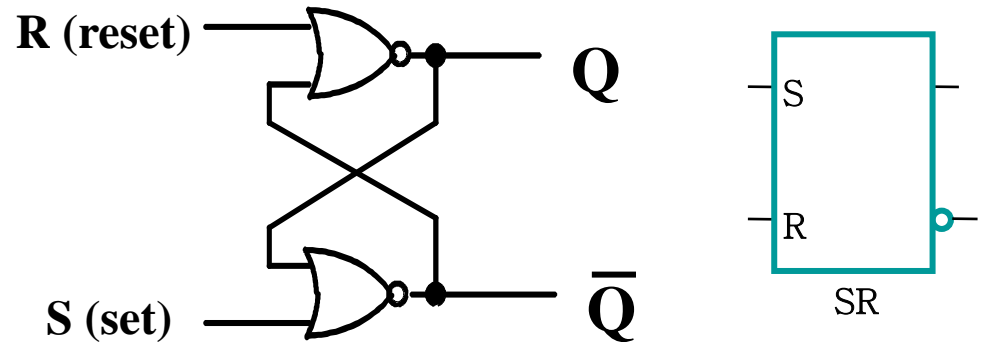
Storage Elements

- Any storage element can maintain a binary state indefinitely (as long as the power is on) until directed by the input signals to switch
- Storage elements: *Latches* and *Flip-flops (FFs)*
- Latches and FFs differ in:
 - Number of inputs
 - Manner in which the inputs affect the binary state
- Latch:
 - Asynchronous
 - Although difficult to design, we discuss latches first because they are the building blocks for flip-flops

Basic (NOR) SR Latch

- Cross-coupling two NOR gates

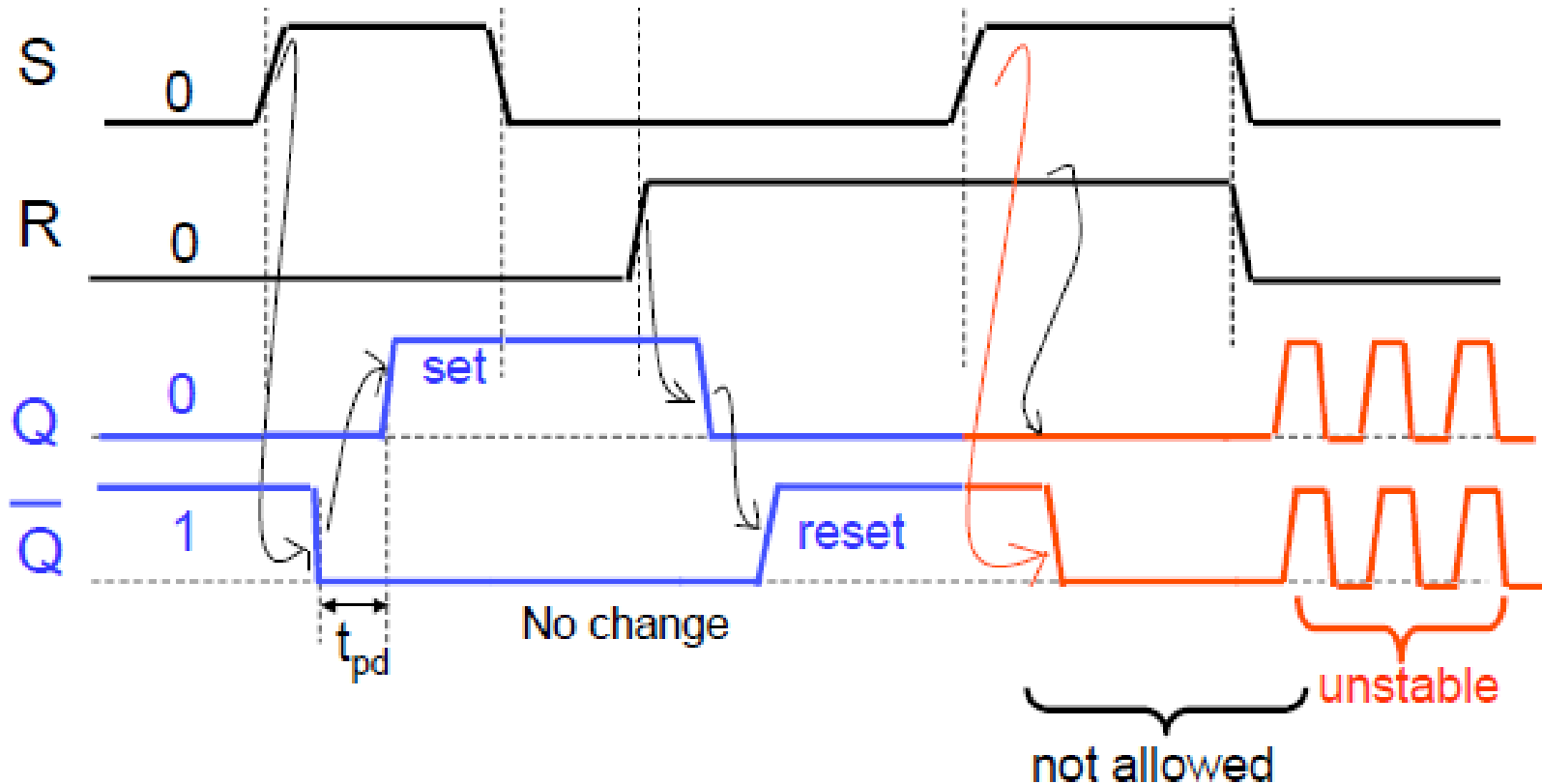
R	S	Q	\bar{Q}	Comment
0	0	Q	\bar{Q}	Hold, no change
0	1	1	0	Set
1	0	0	1	Reset
1	1	0	0	Not allowed



- Time sequence behavior:
- $S = 1, R = 1$ is forbidden as input pattern

Time	R	S	Q	\bar{Q}	Comment
	0	0	?	?	Stored state unknown
	0	1	1	0	“Set” Q to 1
	0	0	1	0	Now Q “remembers” 1
	1	0	0	1	“Reset” Q to 0
	0	0	0	1	Now Q “remembers” 0
	1	1	0	0	Both go low
	0	0	?	?	Unstable!

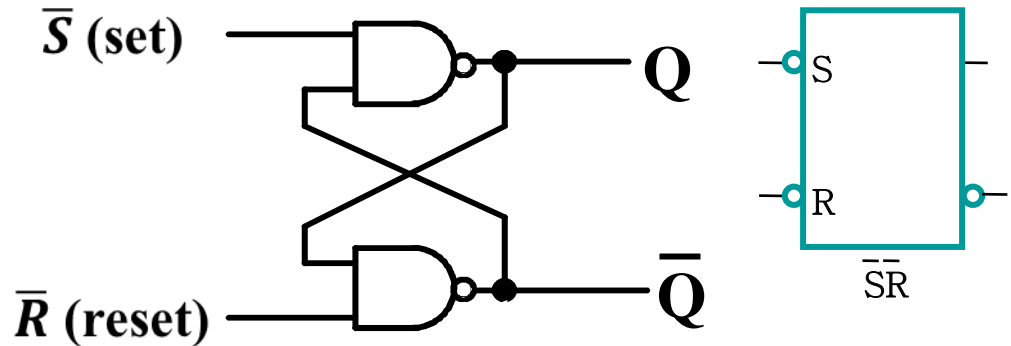
Timing Waveform of NOR SR Latch



Basic (NAND) $\bar{S}\bar{R}$ Latch

- **Cross-coupling** two NAND gates
- **Active low inputs**

\bar{R}	\bar{S}	Q	\bar{Q}	Comment
0	0	1	1	Not allowed
0	1	0	1	Reset
1	0	1	0	Set
1	1	Q	\bar{Q}	Hold, no change



- **Time sequence behavior:**

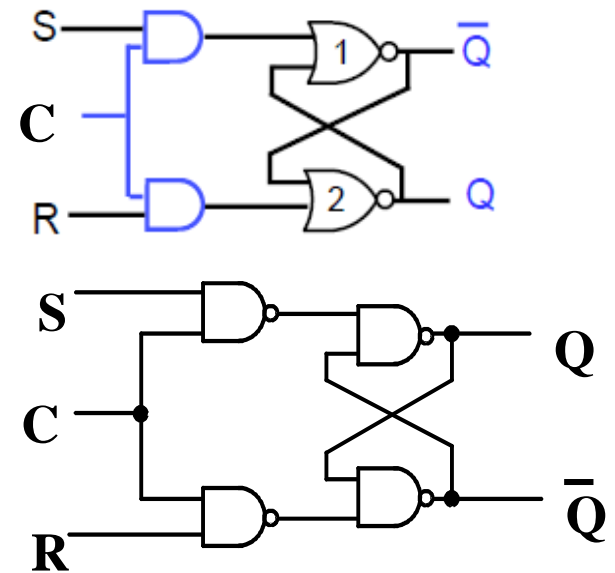
- $\bar{S} = 0, \bar{R} = 0$ is **forbidden** as

Time	\bar{R}	\bar{S}	Q	\bar{Q}	Comment
	1	1	?	?	Stored state unknown
	1	0	1	0	“Set” Q to 1
	1	1	1	0	Now Q “remembers” 1
	0	1	0	1	“Reset” Q to 0
	1	1	0	1	Now Q “remembers” 0
	0	0	1	1	Both go high
	1	1	?	?	Unstable!

Clocked SR Latch (Pulse-triggered Latch)

- The operation of the basic NOR and basic NAND latches can be modified by providing a control input (C) that determines when the state of the latch can be changed
- Adding two AND gates to basic SR latch **OR**
- Adding two NAND gates to $\overline{S}\overline{R}$ basic latch

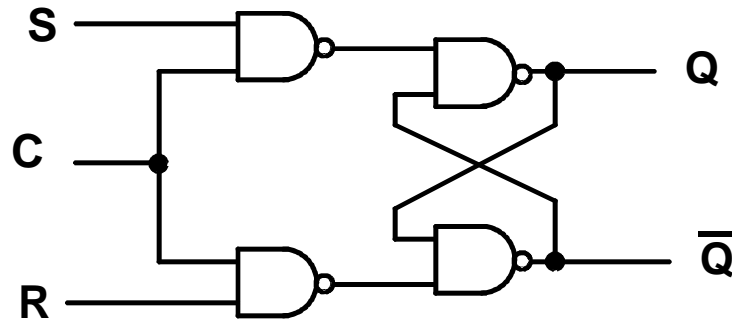
C	R	S	Q	\overline{Q}	Comment
0	x	x	Q	\overline{Q}	Hold, no change
1	0	0	Q	\overline{Q}	Hold, no change
1	0	1	1	0	Set
1	1	0	0	1	Reset
1	1	1	Not allowed		



- Has a time sequence behavior similar to the basic S-R latch except that the S and R inputs are only observed when the line C is high
- C means “control” or “clock”

Clocked SR Latch (continued)

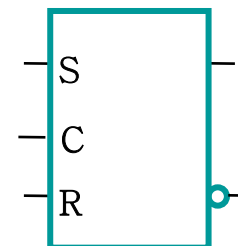
- The Clocked SR Latch can be described by a table:



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

- The table describes what happens after the clock [at time (t+1)] based on:

- current inputs (S,R) and
- current state Q(t)

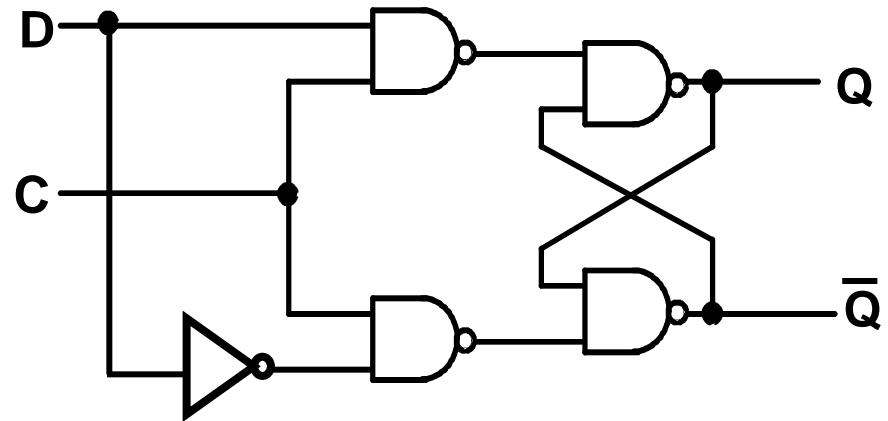


Clocked SR

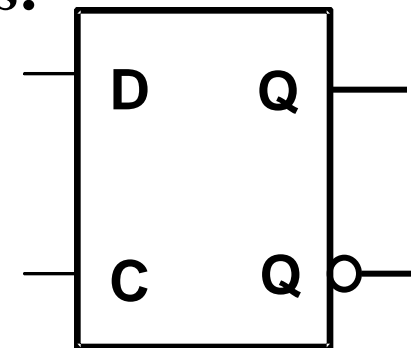
D Latch

- Adding an inverter to the S-R Latch, gives the D Latch:
- Note that there are no **“indeterminate”** states!

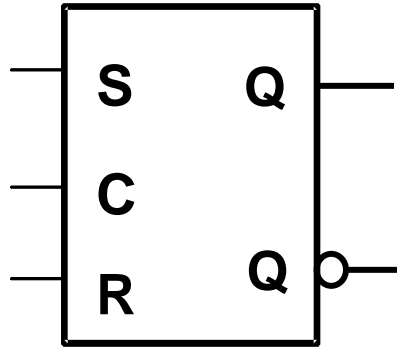
C	D	Q	\bar{Q}	Comment
0	x	Q	\bar{Q}	Hold, no change
1	0	0	1	Reset
1	1	1	0	Set



The graphic symbol for a D Latch is:

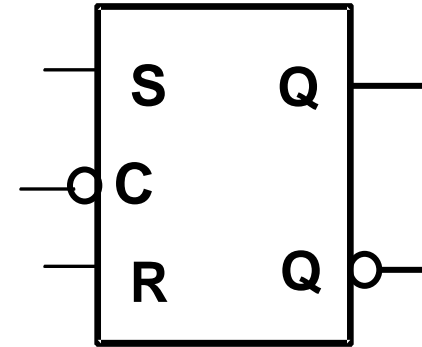


Variations of Clocked SR and D Latches

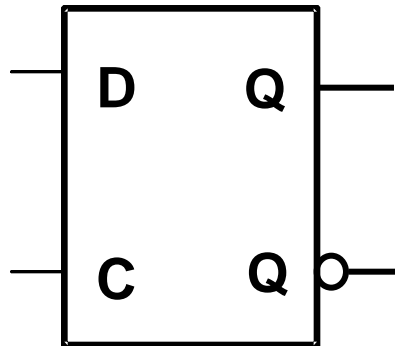


+ve pulse-triggered SR latch

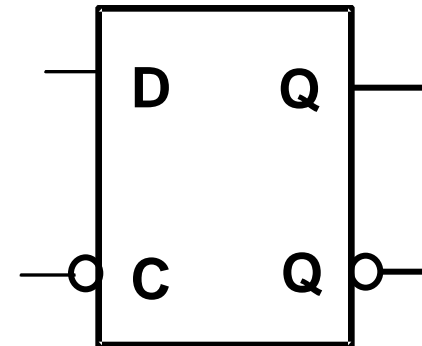
*$C = 0 \rightarrow \text{Hold}$
 $C = 1 \rightarrow \text{Change}$*



-ve pulse-triggered SR latch
 *$C = 0 \rightarrow \text{Change}$
 $C = 1 \rightarrow \text{Hold}$*



+ve pulse-triggered D latch



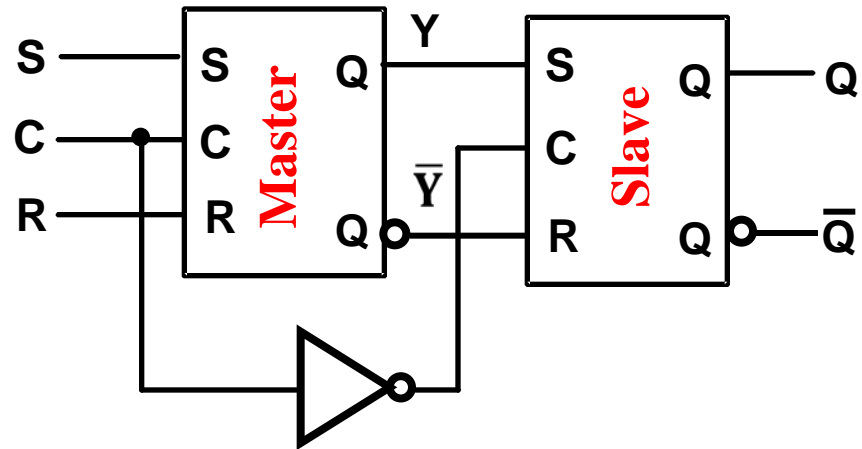
-ve pulse-triggered D latch

Flip-Flops

- **Master-slave flip-flop**
- **Edge-triggered flip-flop**
- **Standard symbols for storage elements**
- **Direct inputs to flip-flops**

SR Master-Slave Flip-Flop

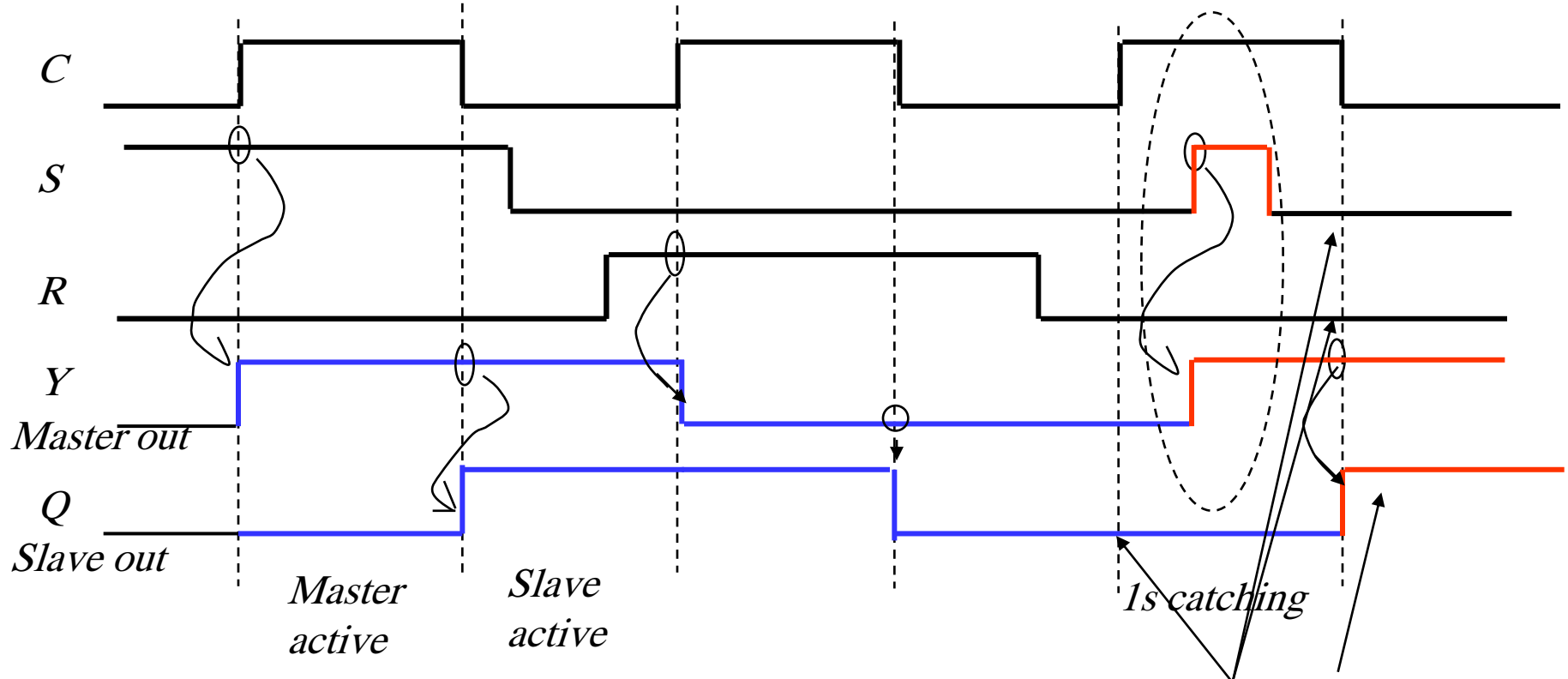
- Consists of two clocked SR latches in series with the clock on the second latch inverted



- The input is observed by the first latch with $C = 1$
- The output is changed by the second latch with $C = 0$
- The path from input to output is broken by the difference in clocking values ($C = 1$ and $C = 0$)

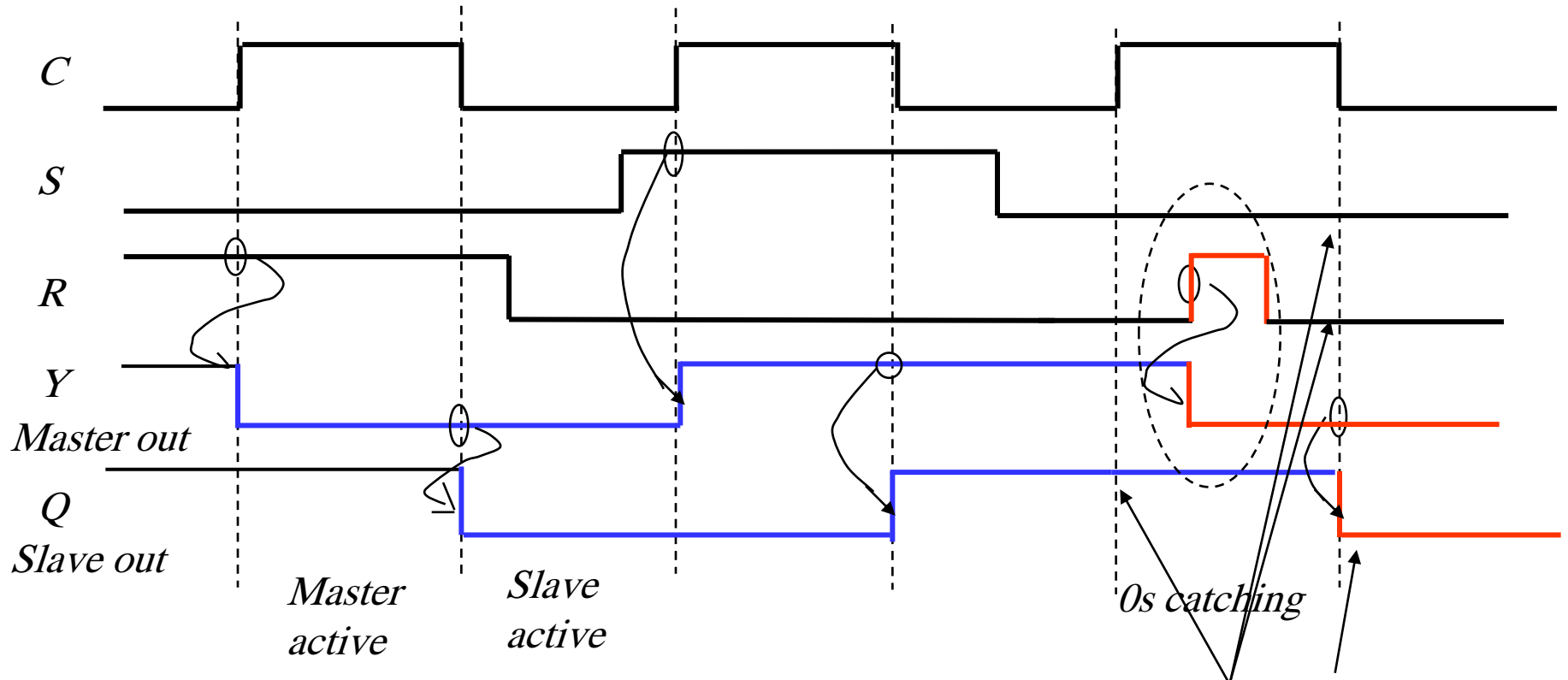
Master-Slave Flip-Flop Problem

- **S and/or R are permitted to change while C = 1**
 - Chances of **0s or 1s catching**



*wrong output
should have been 0*

0s Catching



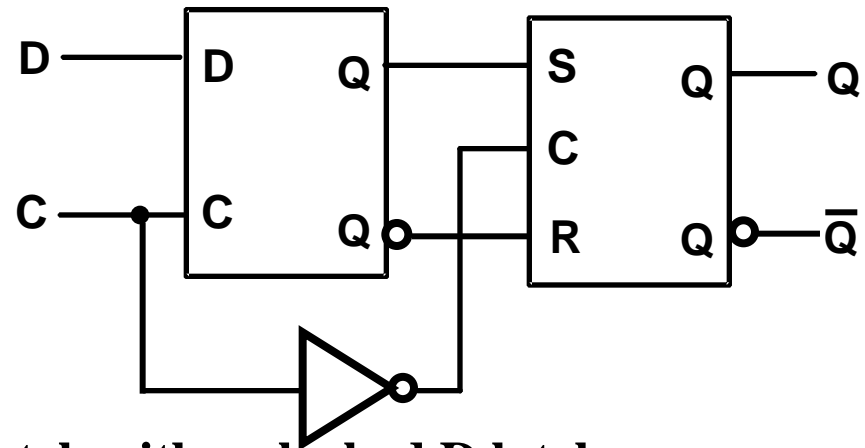
*wrong output
should have been 1*

Flip-Flop Solution

- Use *edge-triggering* instead of master-slave
- An *edge-triggered* flip-flop ignores the pulse while it is at a constant level and triggers only during a *transition* of the clock signal
- Edge-triggered flip-flops can be built directly at the electronic circuit level, or
- A *master-slave D flip-flop* which also exhibits *edge-triggered behavior* can be used

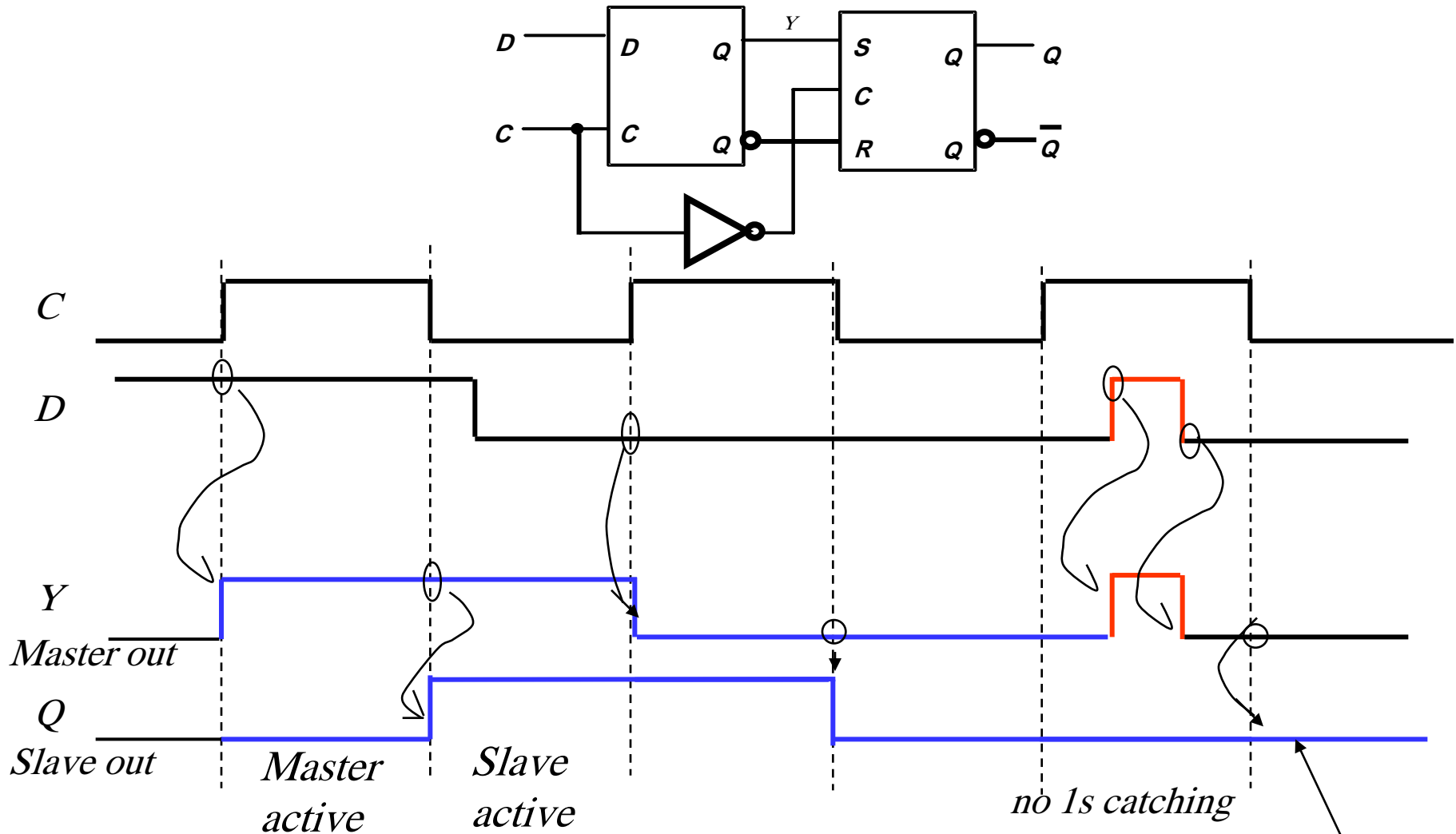
Edge-Triggered D Flip-Flop

- *The edge-triggered D flip-flop is the same as the master-slave D flip-flop*



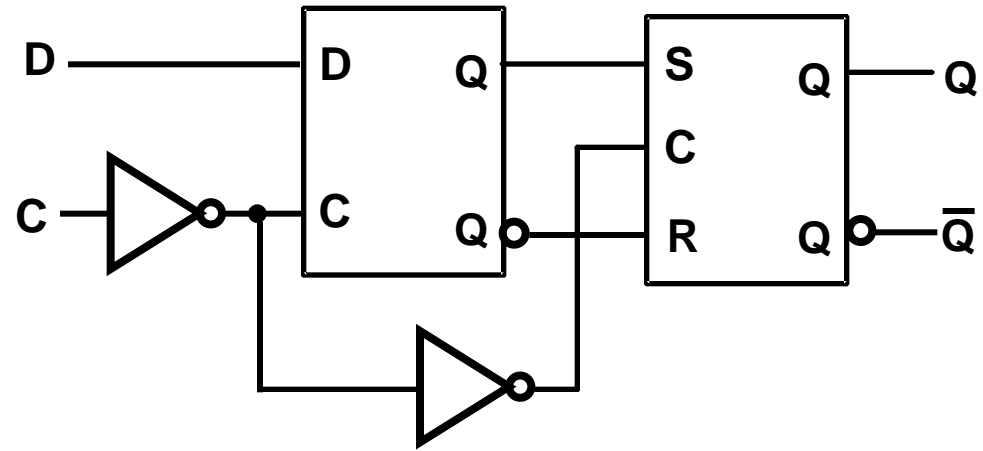
- It can be formed by:
 - Replacing the first clocked SR latch with a clocked D latch or
 - Adding a D input and inverter to a master-slave SR flip-flop
- The 1s and 0s catching behaviors are not present with D replacing S and R inputs
- The change of the D flip-flop output is associated with the negative edge at the end of the pulse
- *It is called a negative-edge triggered flip-flop*

No 1s catching in the edge-triggered D Flip-Flops



Positive-Edge Triggered D Flip-Flop

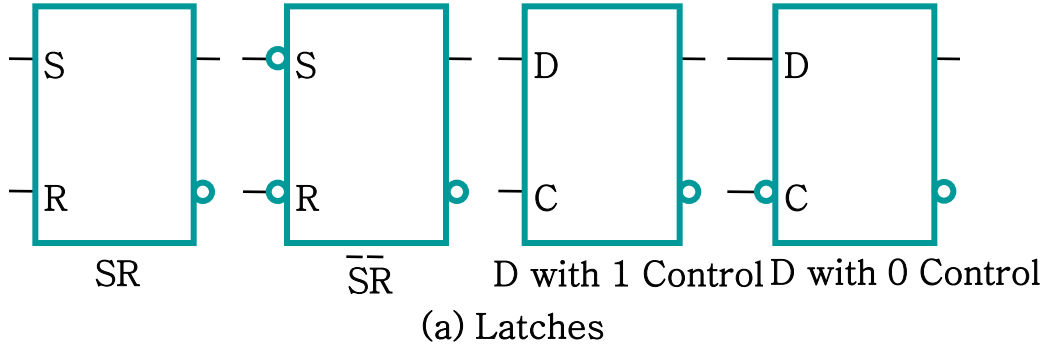
- Formed by adding inverter to clock input



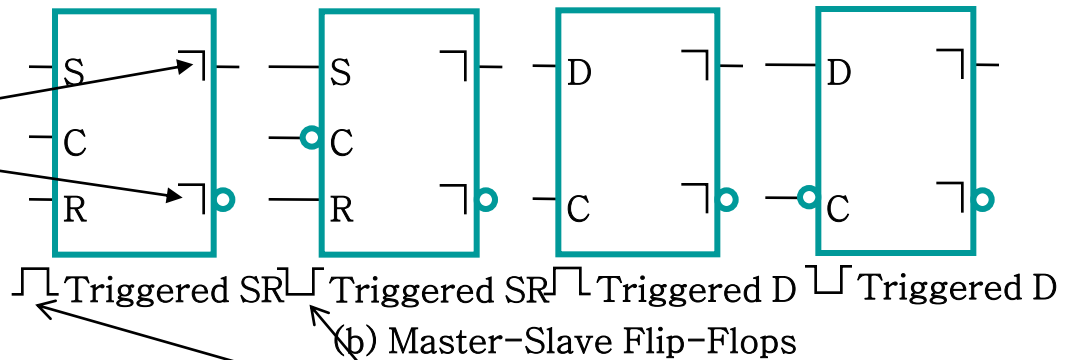
- Q changes to the value on D applied at the positive clock edge
- Our choice as the standard flip-flop for most sequential circuits

Standard Symbols for Storage Elements

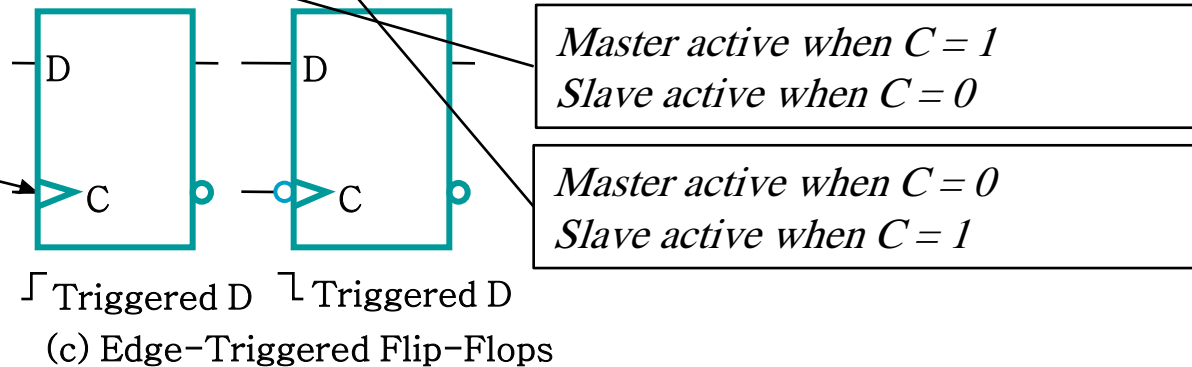
- Latches:**



- Master-Slave:**
Postponed output indicators

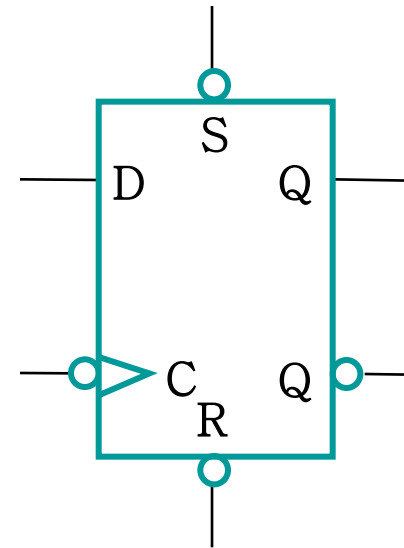


- Edge-Triggered:**
Dynamic indicator



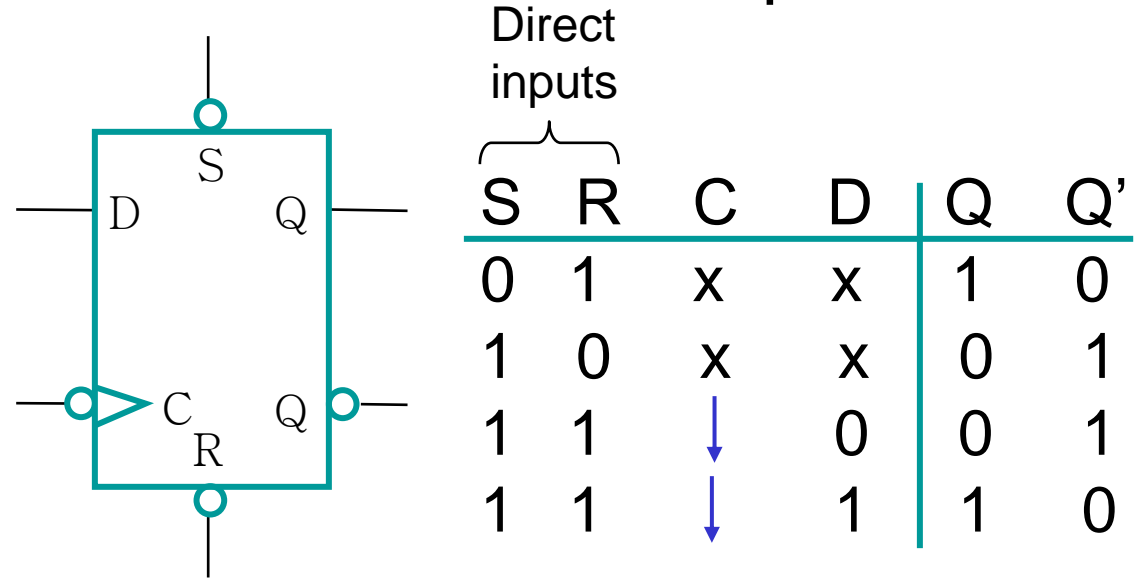
Direct Inputs

- At power up or at reset, all or part of a sequential circuit usually is initialized to a known state before it begins operation
- This initialization is often done outside of the clocked behavior of the circuit, i.e., asynchronously
- Direct R and/or S inputs that control the state of the latches within the flip-flops are used for this initialization
- For the example flip-flop shown
 - 0 applied to R resets the flip-flop to the 0 state
 - 0 applied to S sets the flip-flop to the 1 state

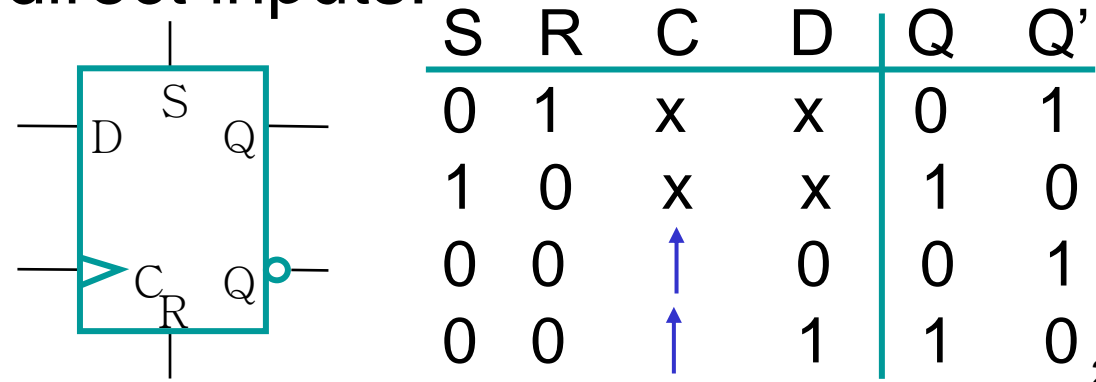


Direct inputs

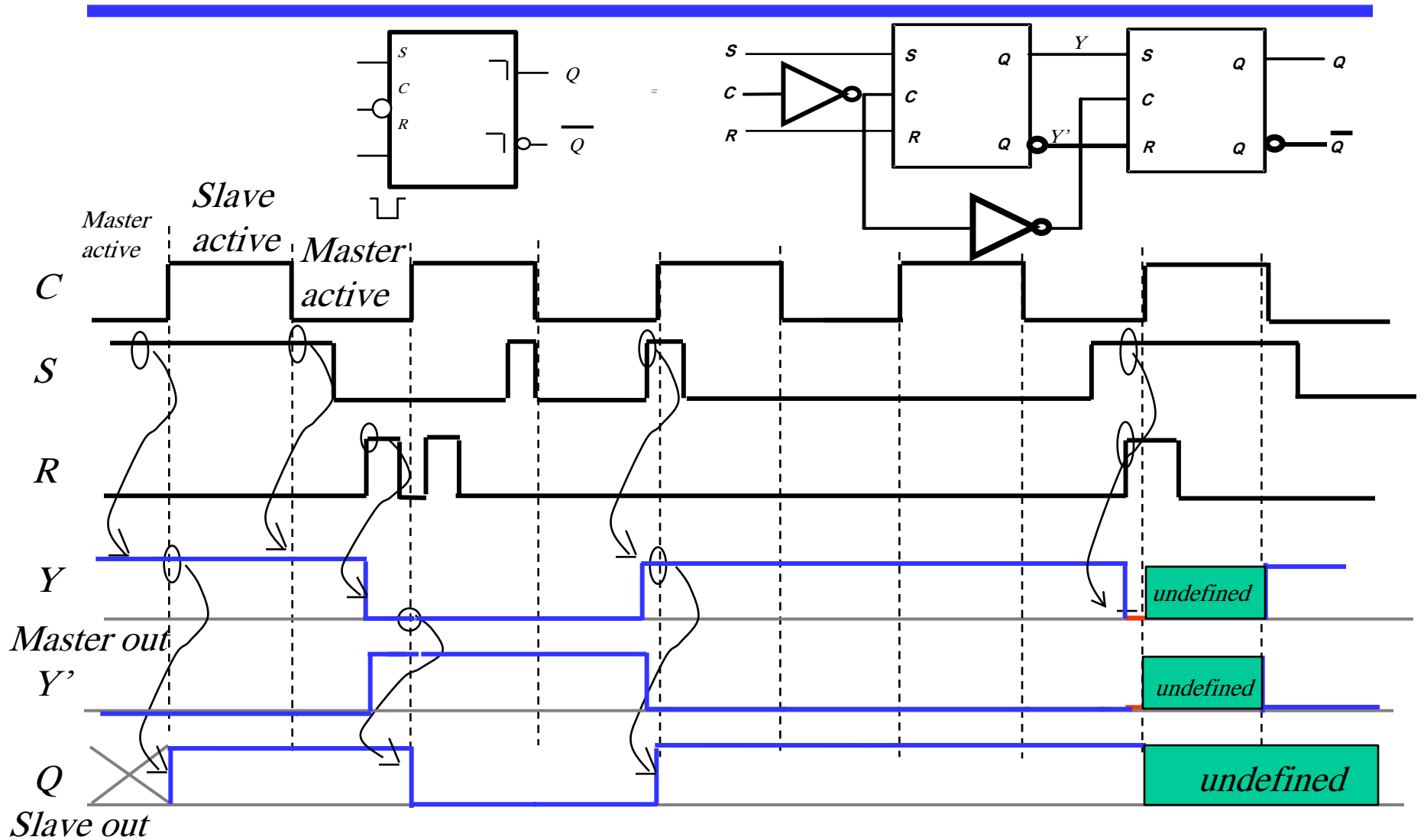
- D flip-flop with active-low direct inputs :



- Active high direct inputs:



Timing diagram of A SR Master-Slave Flip-Flop

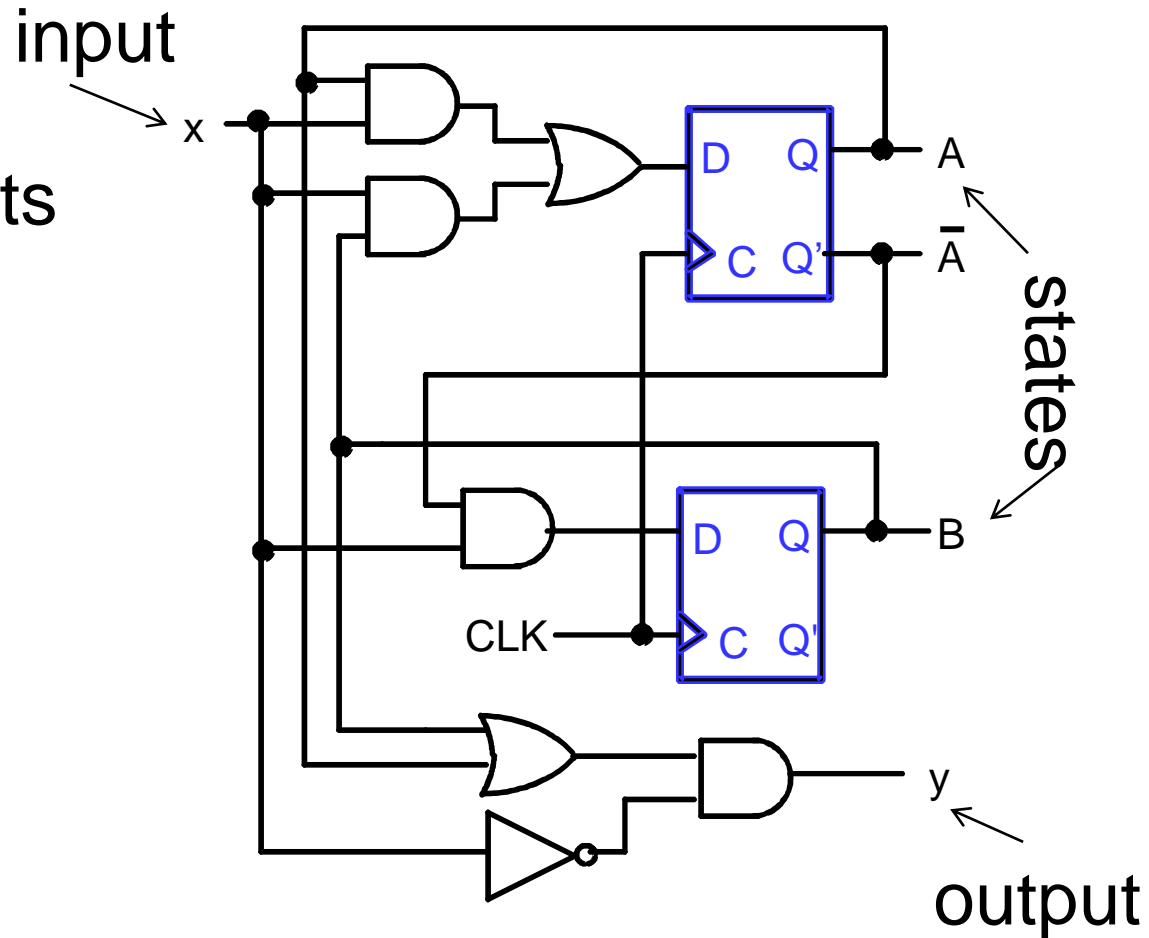


Sections 5.4, 5.5, and 5.6 courtesy Dr. Fahed Jubair

5-4 Sequential Circuit Analysis

- Consider the following circuit:

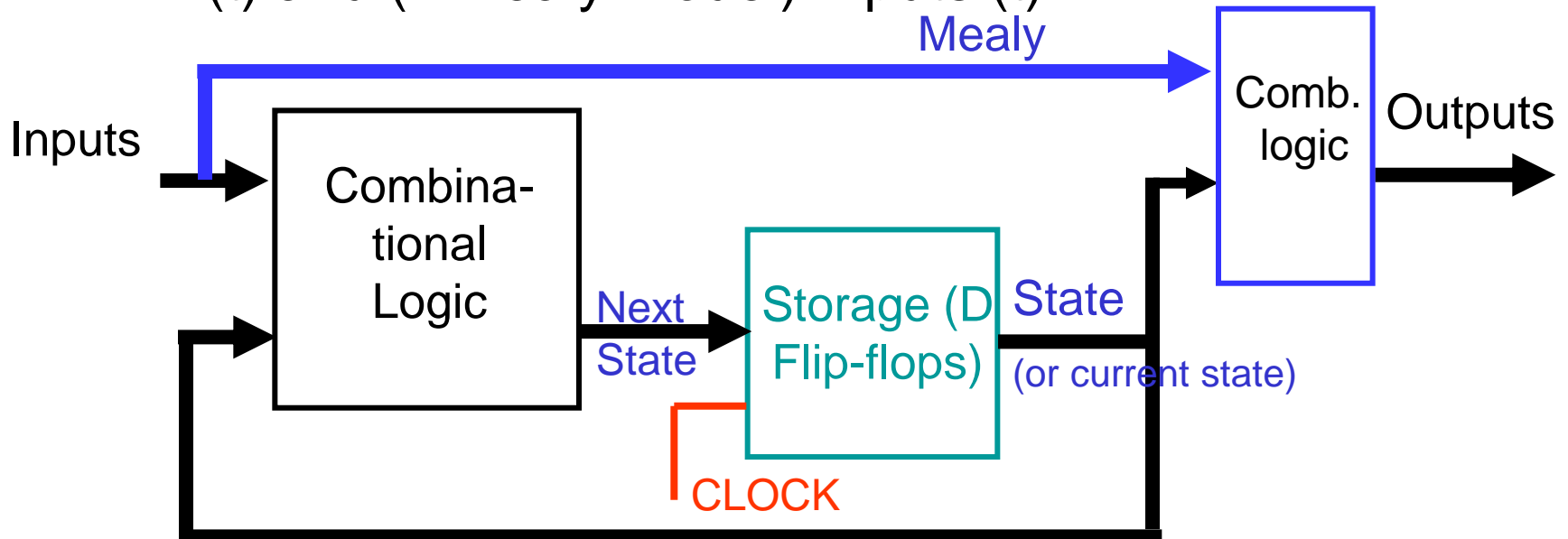
- What does it do?
- How do the outputs change when an input arrives?



Sequential Circuit Model

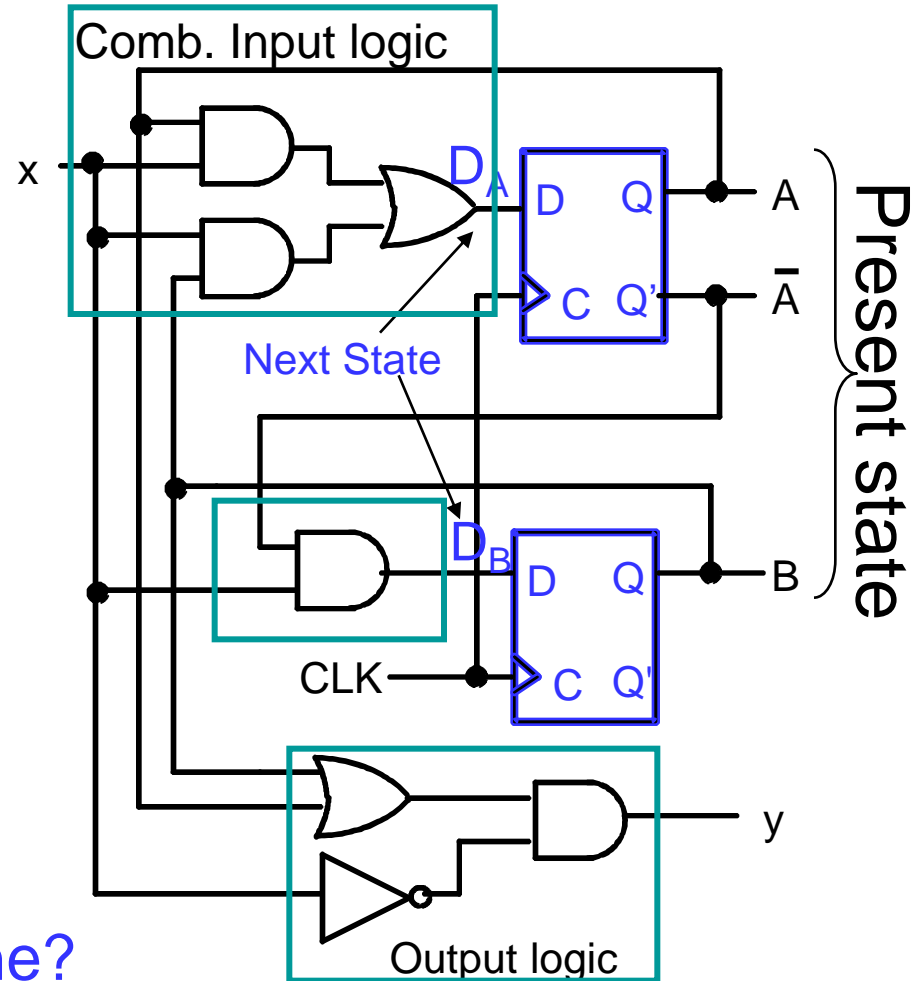
■ General Model

- Current or Present State at time (t) is stored in an array of flip-flops.
- Next State is a Boolean function of State and Inputs.
- Outputs at time (t) are a Boolean function of State (t) and (if Mealy model) Inputs (t).



Previous Example (from Fig. 5-15)

- Input: X
- Output: Y
- State: $(A(t), B(t))$
Example: $(AB) = (01), (10)$
- Next State:
 $(D_A(t), D_B(t))$
 $= (A(t+1), B(t+1))$



Is this a Moore or Mealy machine?

Steps for Analyzing a Sequential Circuit

1. Find the **input equations** (D_A, D_B) to the flip-flops (next state equations) and the output equation.
2. Derive the **State Table** (describes the behavior of a sequential circuit).
3. Draw the **State Diagram** (graphical description of the behavior of the sequential circuit).
4. Simulation

Step 1: Input and output equations

- Boolean equations for the inputs to the flip flops:

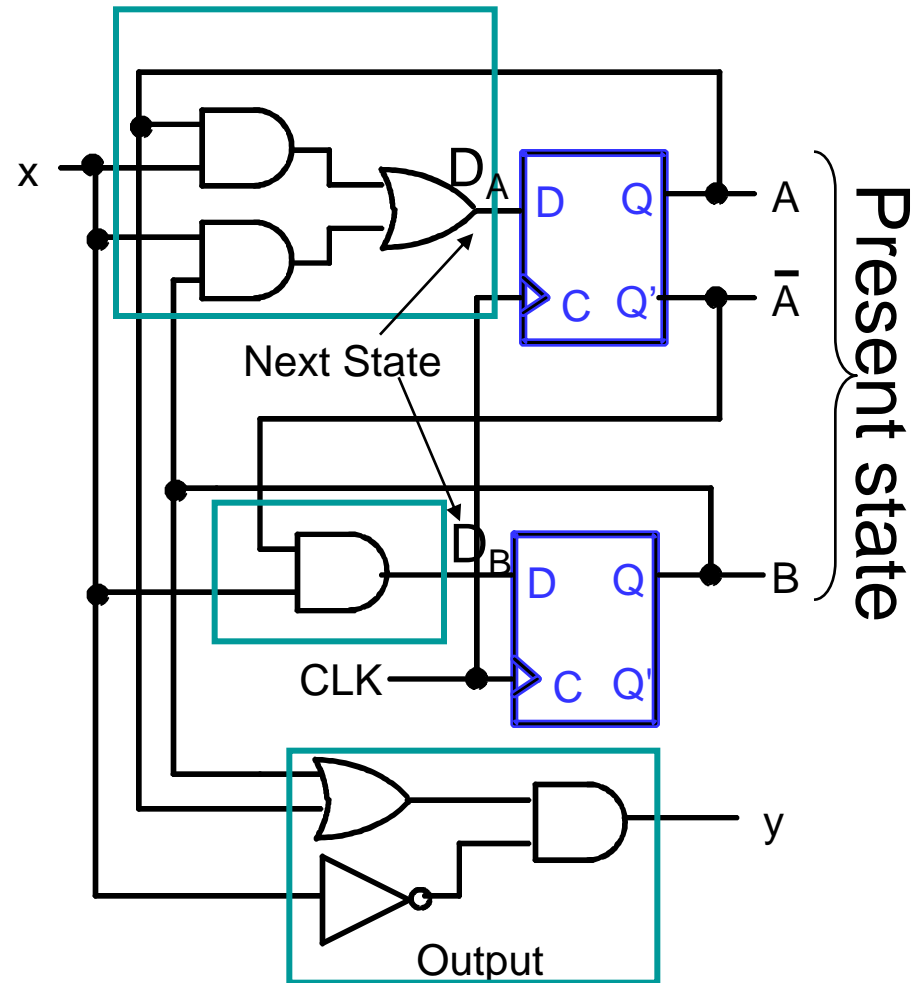
- $D_A = AX + BX$
- $D_B = \overline{A}X$

- Output Y

- $Y = \overline{X}(A + B)$

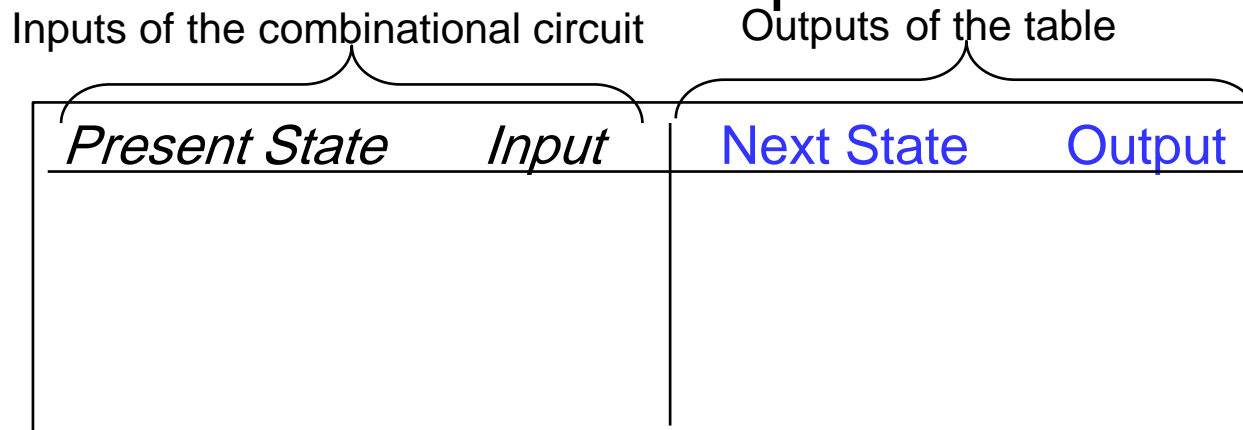
- Also can be written as

- $A(t+1) = D_A = A(t)X + B(t)X$
- $B(t+1) = D_B = \overline{A}(t)X$
- $Y = \overline{X}(A(t) + B(t))$



Step 2: State Table

- The state table: shows what the *next state* and the *output* will be as a function of the present state and the input:



- The State Table can be considered a truth table defining the combinational circuits:
 - the inputs are *Present State* and *Input*,
 - and the outputs are *Next State* and *Output*

State Table For The Example

- For the example: $A(t+1) = A(t) x + B(t) x$
 $B(t+1) = A'(t) x$
 $Y(t) = X' (B(t) + A(t))$

Inputs of the table Outputs of the table

2^3 ROWS
 (2^{m+n}) ROWS

m : no. of flip-flops
 n : no. of inputs

Present State		Input	Next State		Output
A(t)	B(t)	X	A(t+1)	B(t+1)	Y
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	0	1
0	1	1	1	1	0
1	0	0	0	0	1
1	0	1	1	0	0
1	1	0	0	0	1
1	1	1	1	0	0

Alternate State Table

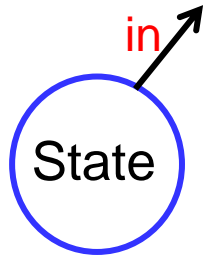
- The previous (1-dimensional table) can become quite lengthy with 2^{m+n} rows (m =no. of flip-flops; n =no. of inputs)
- Alternatively, a 2-dimensional table has the present state in the left column and inputs across the top row
 - $A(t+1) = A(t) X + B(t) X$
 - $B(t+1) = A'(t) X$
 - $Y = X' (B(t) + A(t))$

Present State A(t) B(t)	Next State				Output	
	X = 0		X = 1		X=0	X=1
	A(t+1)	B(t+1)	A(t+1)	B(t+1)	Y	Y
0 0	0	0	0	1	0	0
0 1	0	0	1	1	1	0
1 0	0	0	1	0	1	0
1 1	0	0	1	0	1	0

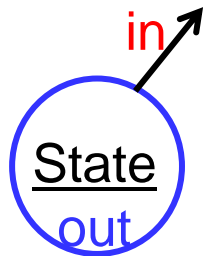
2^m {

Step 3: State Diagrams

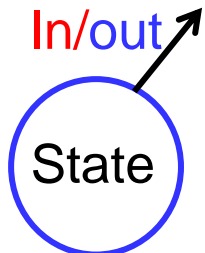
- The sequential circuit function can be represented in graphical form as a state diagram with the following components:



- A circle with the state name in it for each state
- A directed arc from the Present State to the Next State for each state transition
- A label on each directed arc with the Input values which causes the state transition, and

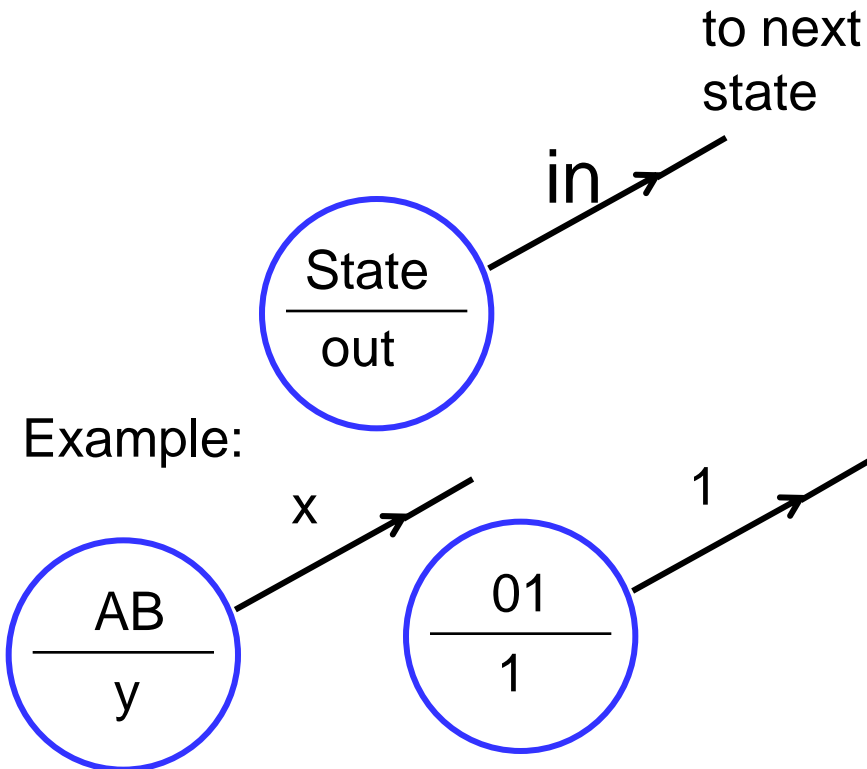


- A label:
 - In each circle with the output value produced, or
 - On each directed arc with the output value produced.



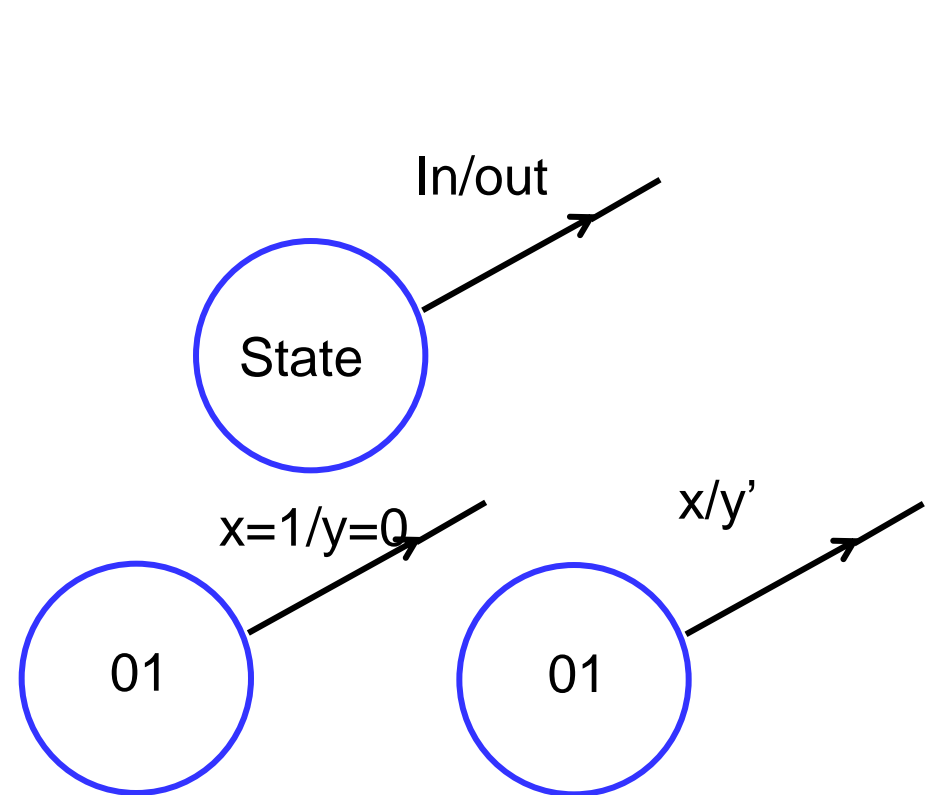
State Diagram Convention

Moore Machine:



Moore type output depends only on state

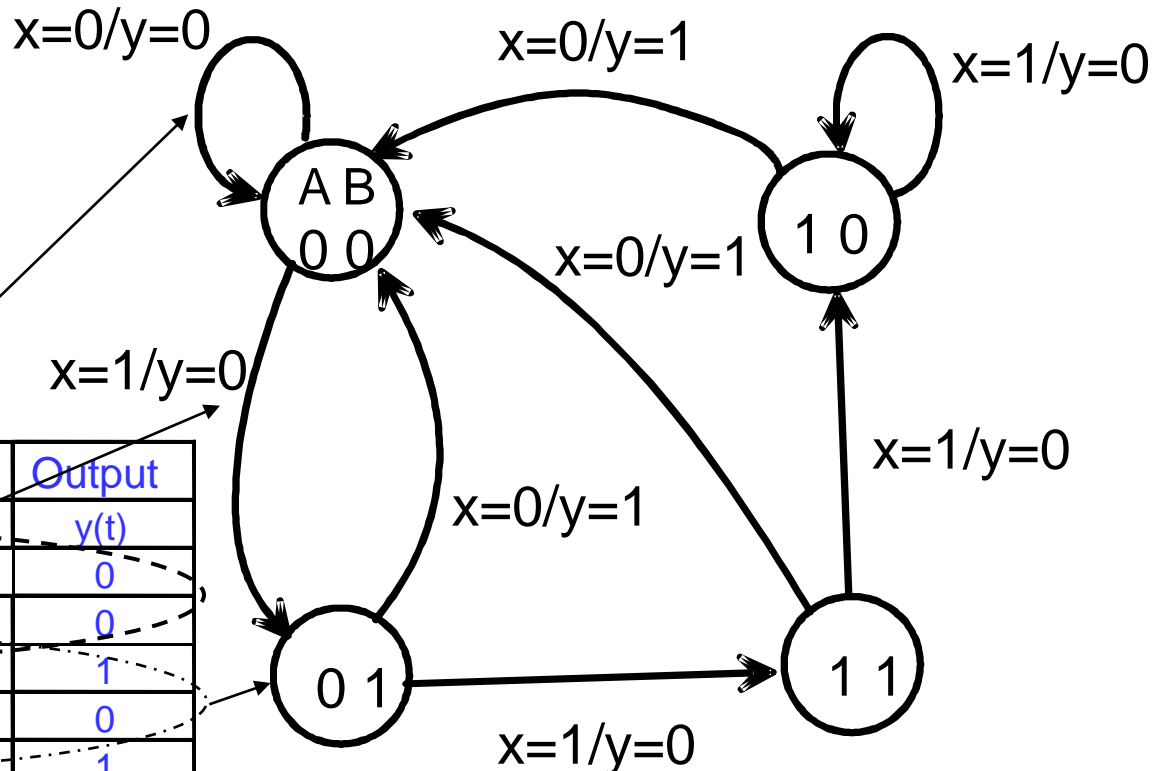
Mealy Machine:



Mealy type output depends on state and input

State Diagram For The Example

- Graphical representation of the state table:



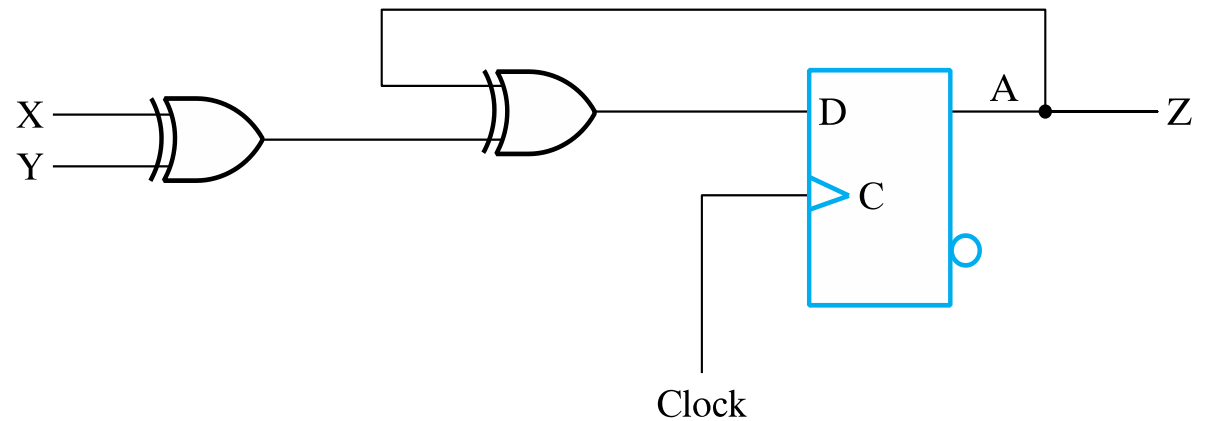
Present State	Input	Next State	Output
A(t) B(t)	x(t)	A(t+1) B(t+1)	y(t)
0 0	0	0 0	0
0 0	1	0 1	0
0 1	0	0 0	1
0 1	1	1 1	0
1 0	0	0 0	1
1 0	1	1 0	0
1 1	0	0 0	1
1 1	1	1 0	0

Step 4: Simulation

- Two types:
 - Functional simulation: objective is to verify the functionality of the circuit
 - Timing simulation: objective is to perform a more realistic testing (with gate delays counted)
- More about this step in the lab (CPE0907234)

Example2

- Derive the state table and state diagram for the sequential circuit:

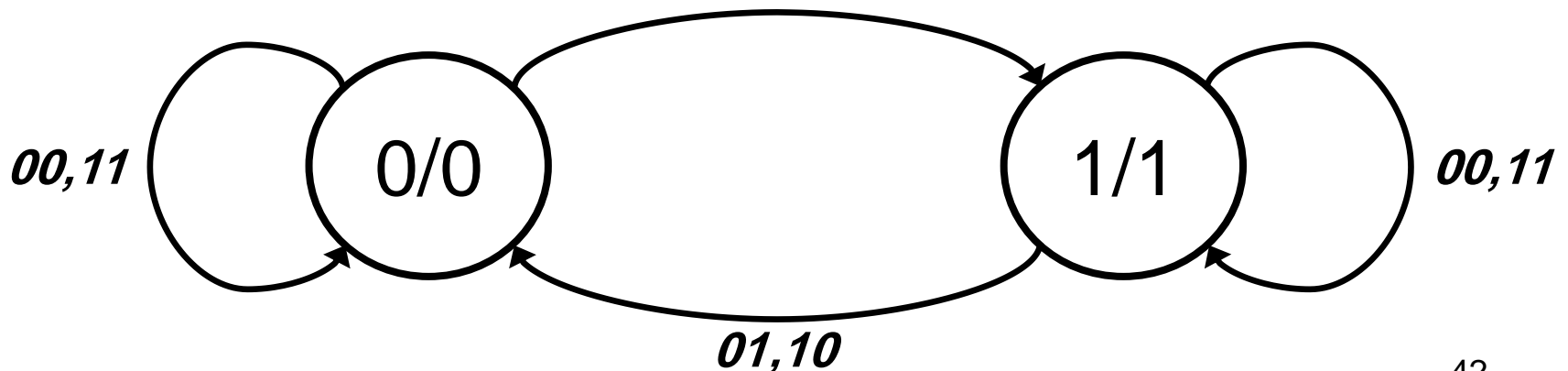


Example2 Cont.

- State Table:

Preset State A(t)	Next State				Z
	XY = 00	XY = 01	XY = 10	XY = 11	
0	0	1	1	0	0
1	1	0	0	1	1

- State Diagram:

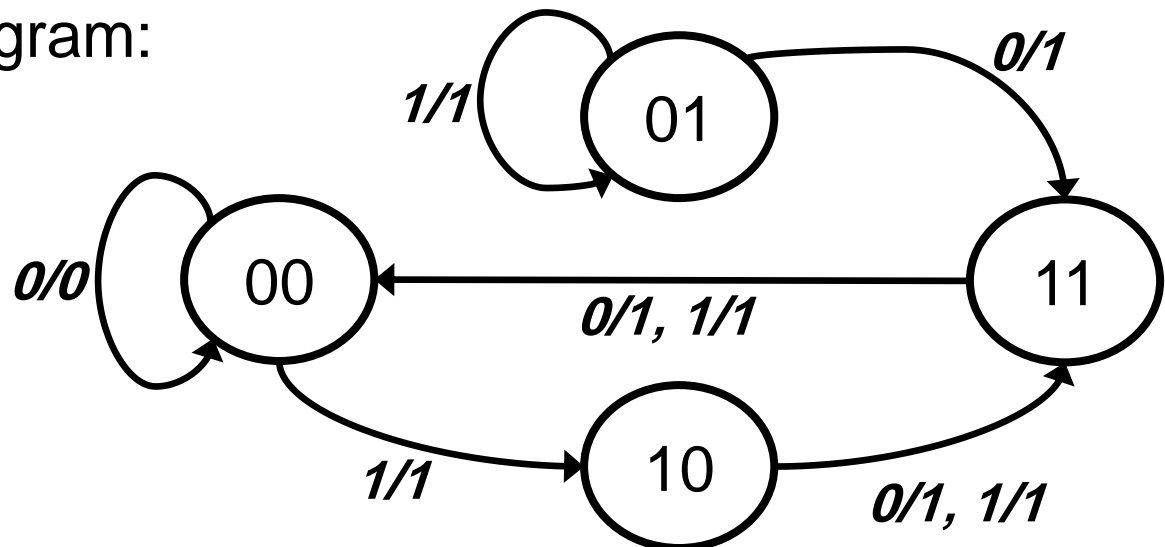


Example3 Cont.

- State Table:

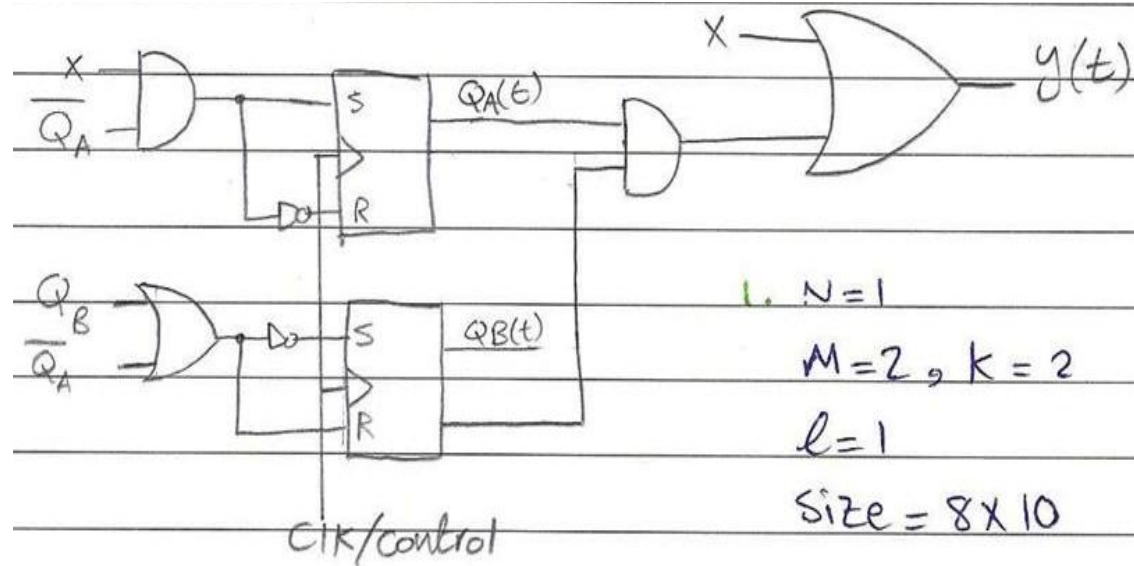
Preset State A(t) B(t)	Next State		Output	
	X = 0	X = 1	X = 0	X = 1
	A(t+1) B(t+1)	A(t+1) B(t+1)	Y	Y
0 0	0 0	1 0	0	1
0 1	1 1	0 1	1	1
1 0	1 1	1 1	1	1
1 1	0 0	0 0	1	1

- State Diagram:



Example 4

- Derive the state table and state diagram for the sequential circuit:

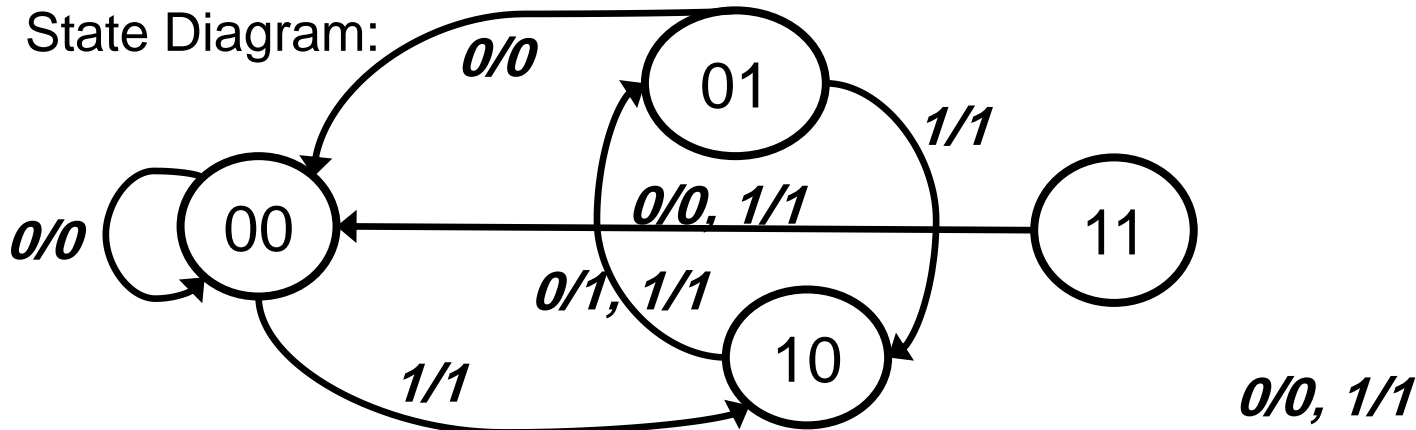


Example4 Cont.

- State Table

Present State $Q_A Q_B$	Input X	$S_A R_A$	$S_B R_B$	Next State $Q_A(t+1) Q_B(t+1)$	Output Y
0 0	0	0 1	0 1	0 0	0
0 0	1	1 0	0 1	1 0	1
0 1	0	0 1	0 1	0 0	0
0 1	1	1 0	0 1	1 0	1
1 0	0	0 1	1 0	0 1	1
1 0	1	0 1	1 0	0 1	1
1 1	0	0 1	0 1	0 0	0
1 1	1	0 1	0 1	0 0	1

- State Diagram:

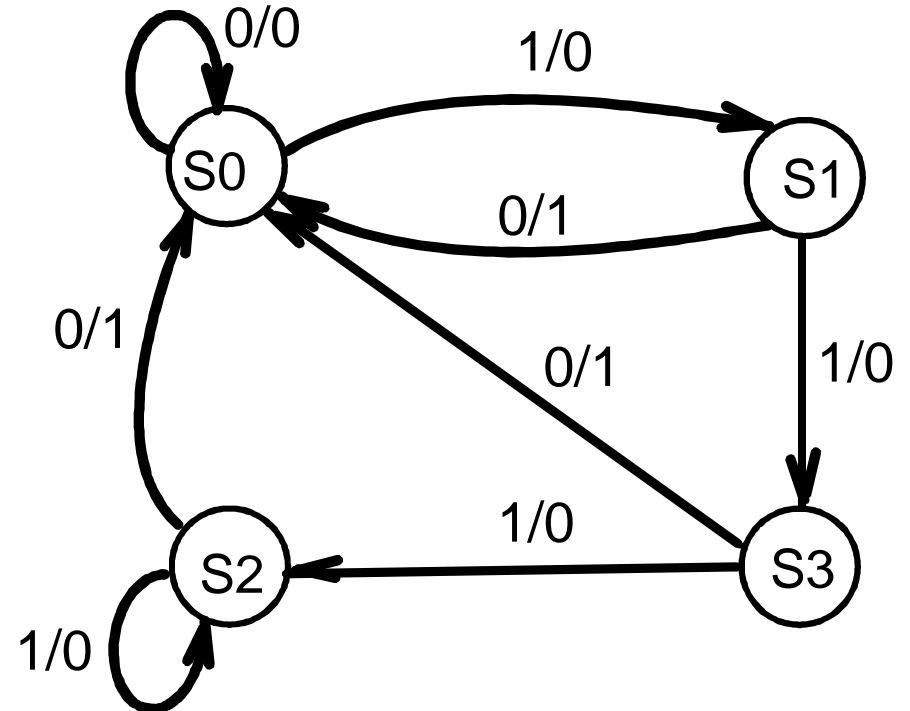


Equivalent State Definitions

- Two states are *equivalent* if their response for each possible input sequence is an identical output sequence.
- Alternatively, two states are *equivalent* if their **outputs produced for each input symbol is identical** and their **next states for each input symbol** are the same or equivalent.

Equivalent State Example

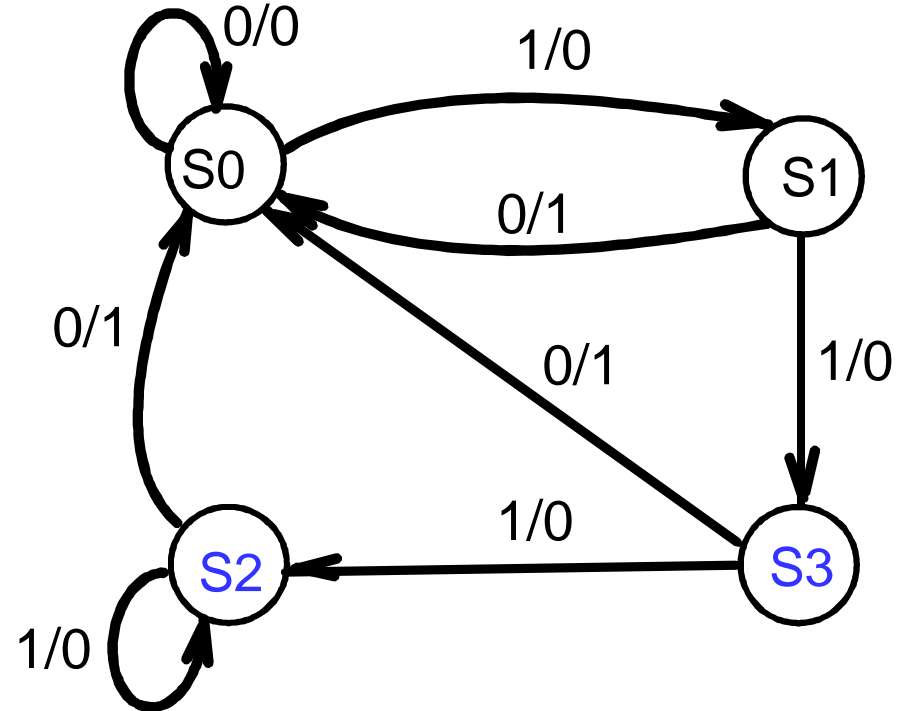
- Consider the following state diagram:



- Which states are equivalent?

Equivalent State Example

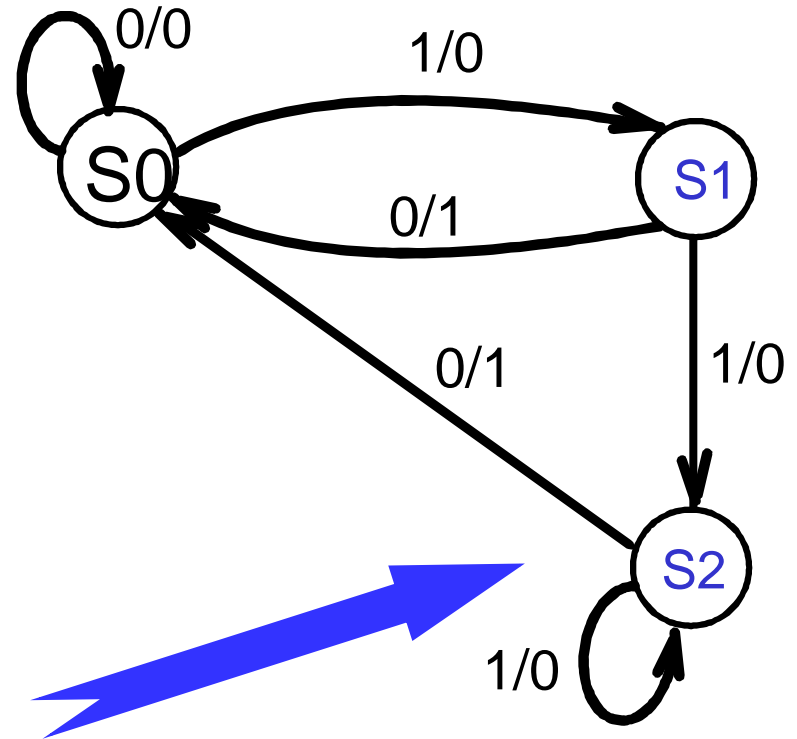
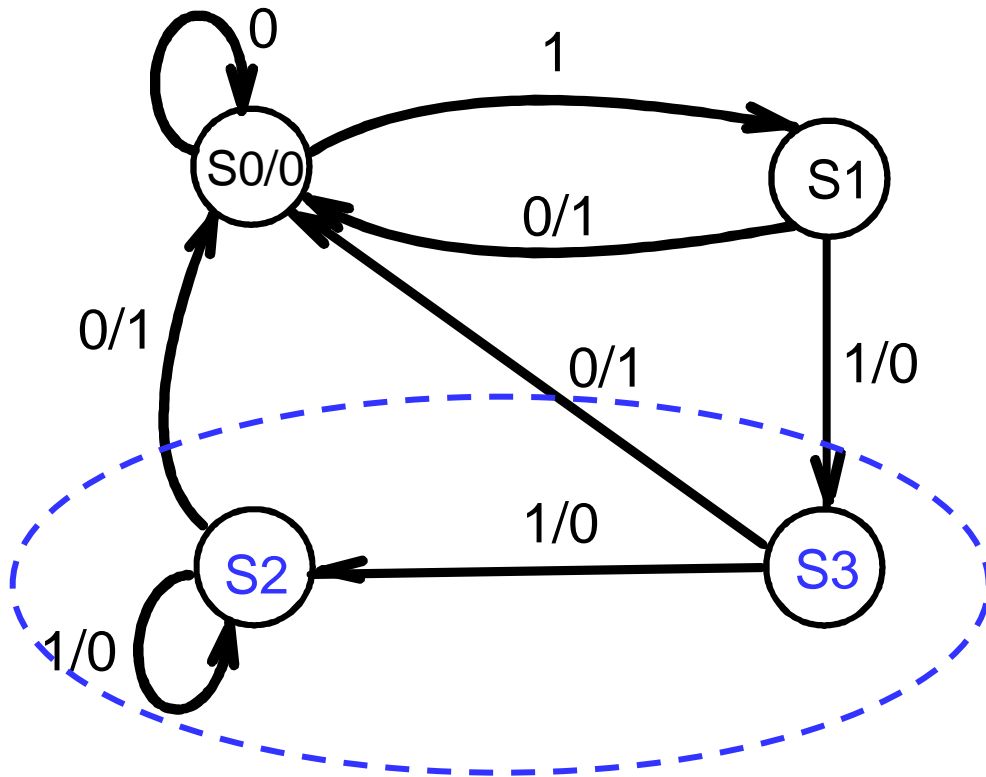
- Equivalent states in the state diagram:



- For states S2 and S3,
 - the **output** for input 0 is **1** and the for input 1, the output is **0**
 - the next state for input 0 is S0 and for input 1 is S2.
 - By the alternative definition, states S2 and S3 are equivalent.

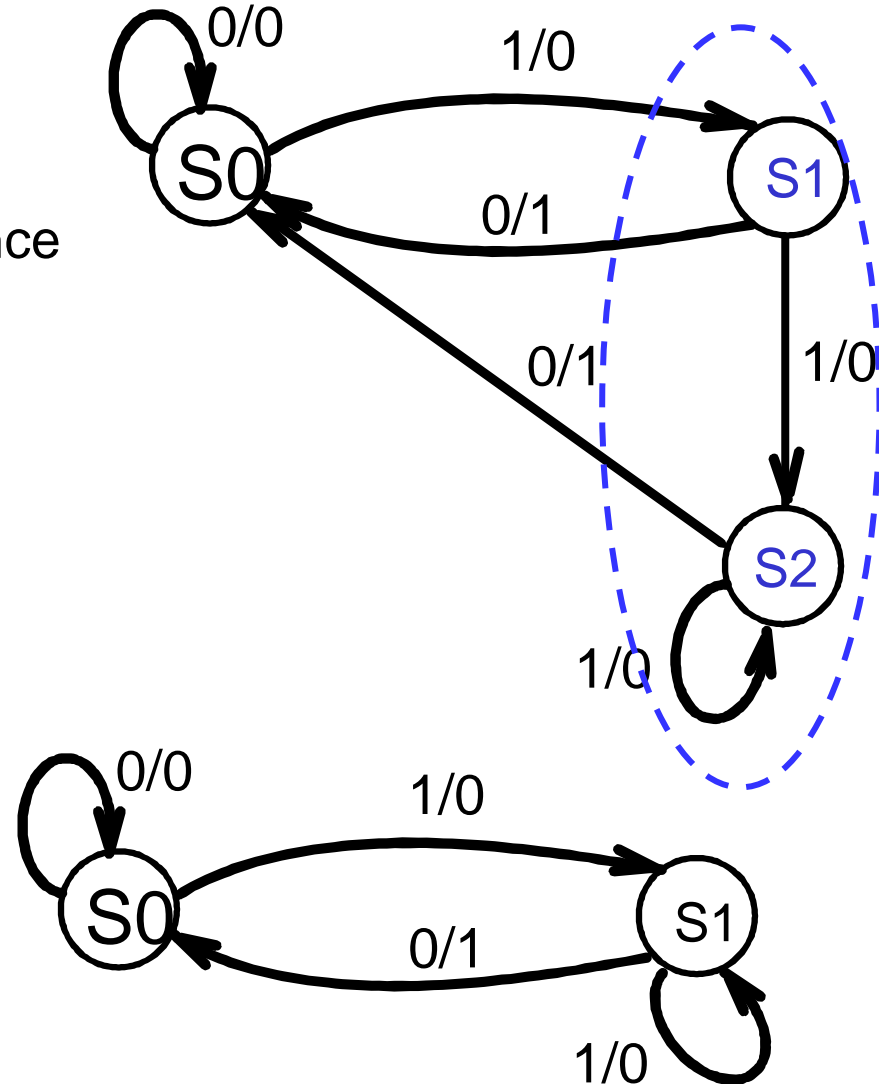
Equivalent State Example

- Replacing S2 and S3 by a single state gives state diagram:



Equivalent State Example

- Are there other equivalent states?
- Examining the new diagram, states **S1** and **S2** are equivalent since
 - their outputs for input 0 is 1 and input 1 is 0, and
 - their next state for input 0 is both S0 and for input 1 is both S2,
- Replacing S1 and S2 by a single state gives state diagram:

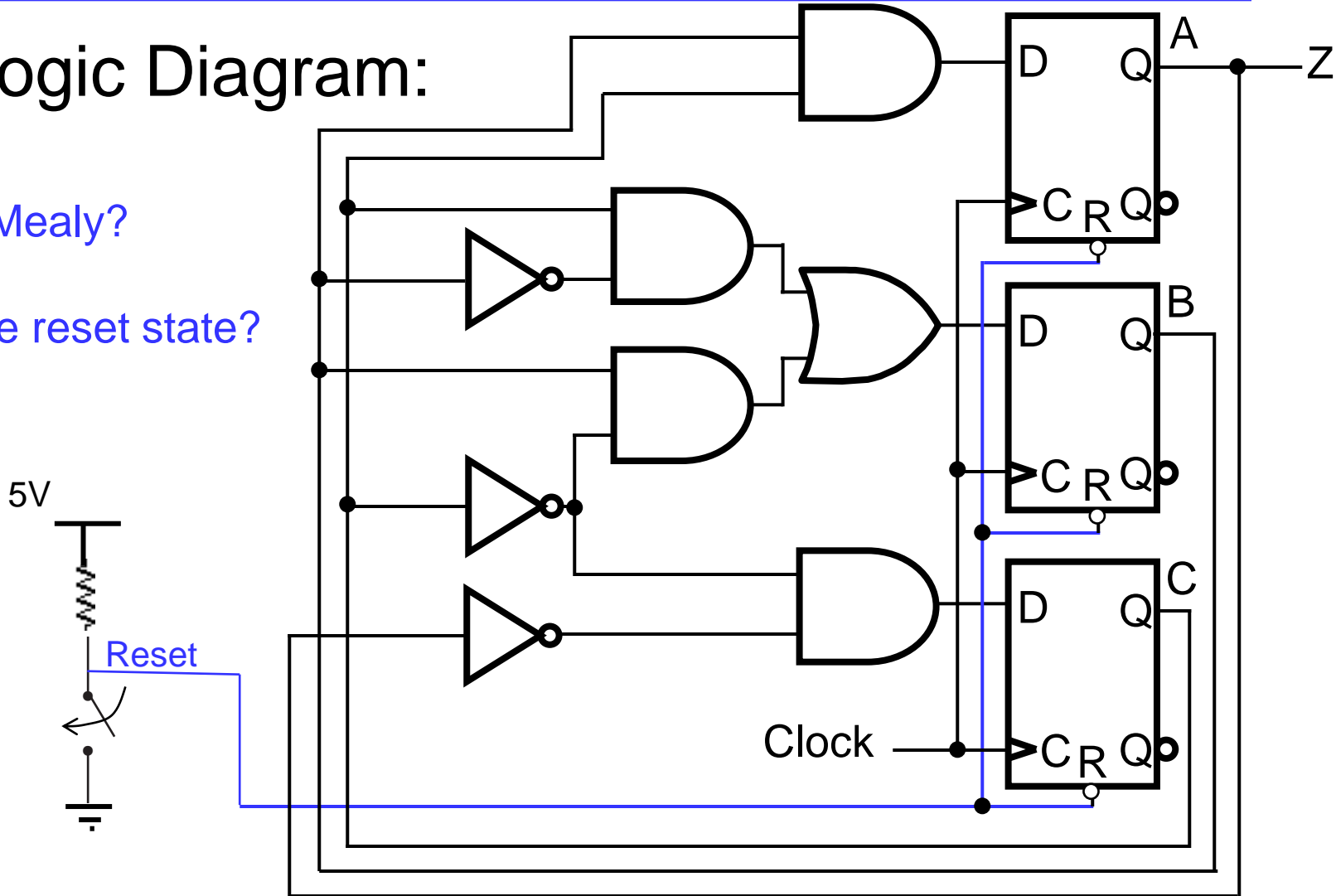


Exercise: Derive the state diagram of the following Circuit

- Logic Diagram:

Moore or Mealy?

What is the reset state?



Step1: Flip-Flop Input Equations

- Variables
 - Inputs: None
 - Outputs: Z
 - State Variables: A, B, C
- Initialization: Reset to (0,0,0)
- Equations
 - $A(t+1) = BC$ $Z = A$
 - $B(t+1) = B'C + BC' = B \oplus C$
 - $C(t+1) = A'C'$

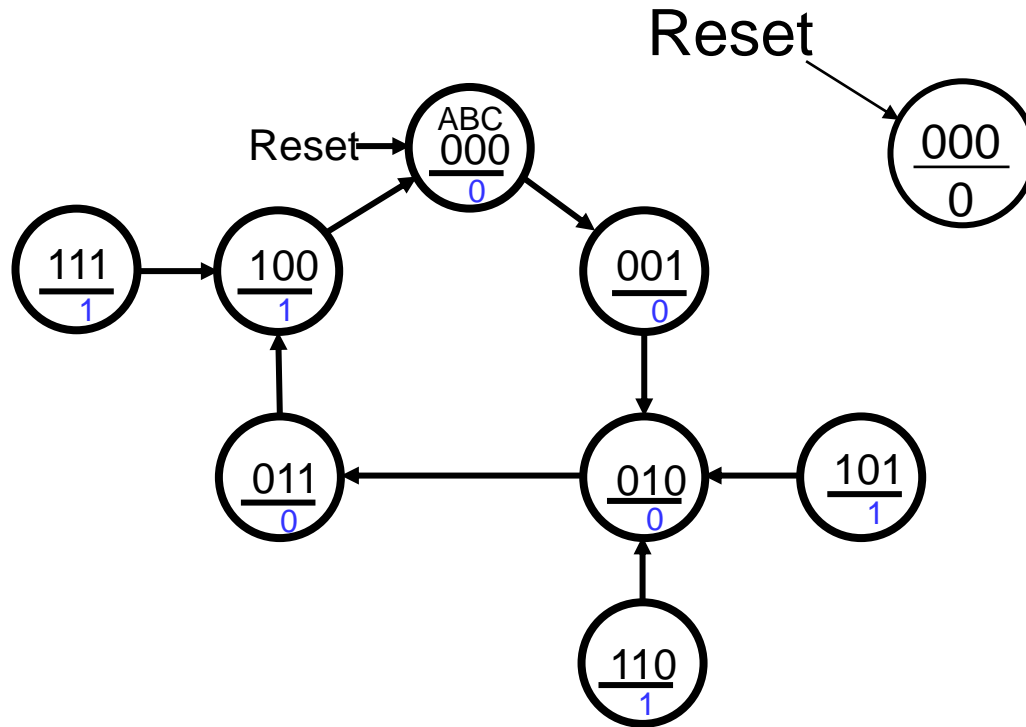
Step 2: State Table

$$\begin{aligned}A(t+1) &= BC & Z &= A \\B(t+1) &= B'C + BC' = \\ & B \oplus C \\C(t+1) &= A'C'\end{aligned}$$

A B C	A ⁺ B ⁺ C ⁺	Z
0 0 0	0 0 1	0
0 0 1	0 1 0	0
0 1 0	0 1 1	0
0 1 1	1 0 0	0
1 0 0	0 0 0	1
1 0 1	0 1 0	1
1 1 0	0 1 0	1
1 1 1	1 0 0	1

Step 3: State Diagram

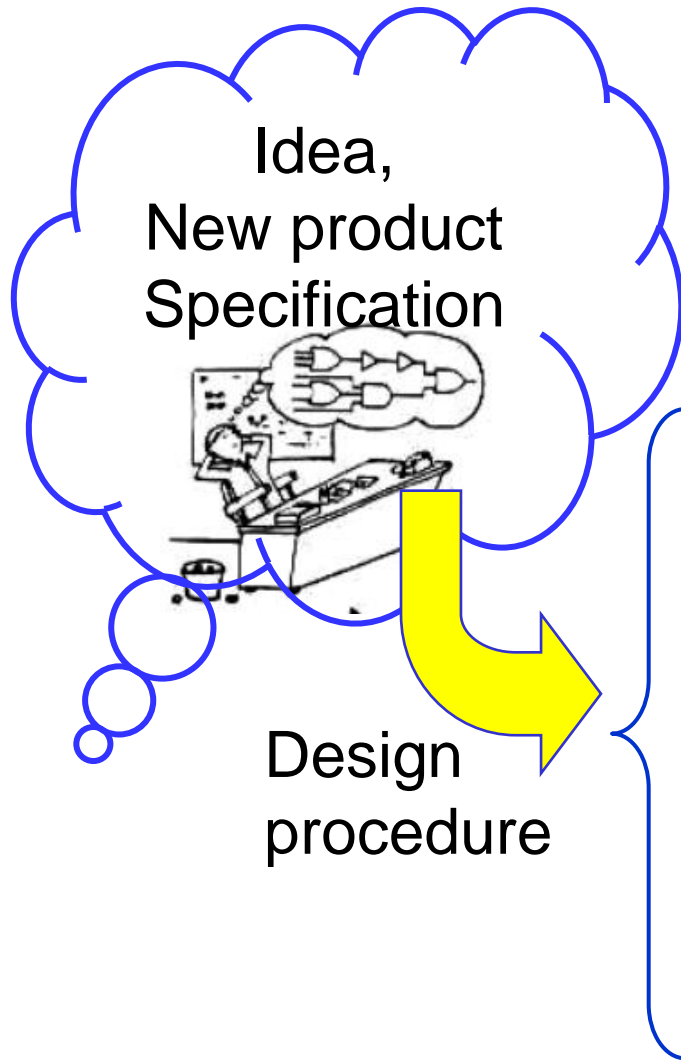
Start from the reset state



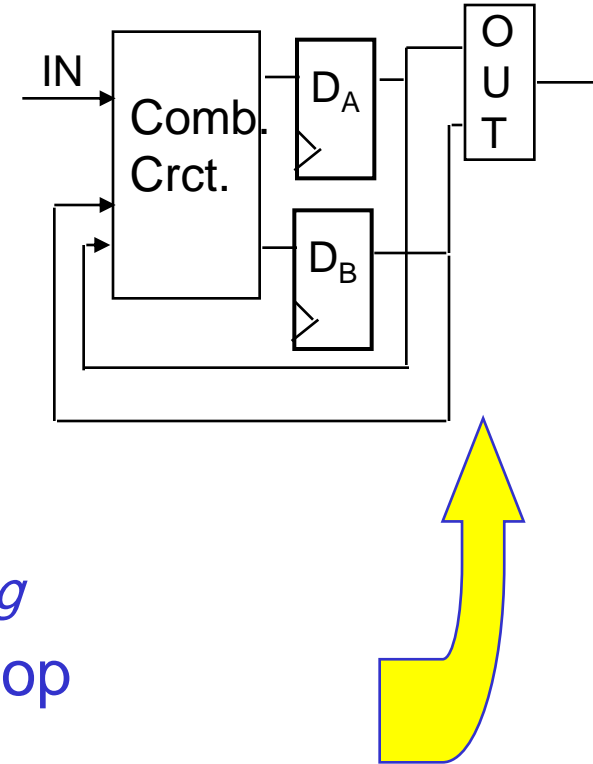
A B C	$A+B+C$	Z
0 0 0	0 0 1	0
0 0 1	0 1 0	0
0 1 0	0 1 1	0
0 1 1	1 0 0	0
1 0 0	0 0 0	1
1 0 1	0 1 0	1
1 1 0	0 1 0	1
1 1 1	1 0 0	1

- Are all states used? Which ones?

5-5 Sequential Circuit Design



- ?
- Word description
State Diagram
- ↓
- State Table
- ↓ *State encoding*
- Select type of Flip-flop
- ↓
- Input equations to FF, output eq.
- ↓
- Verification



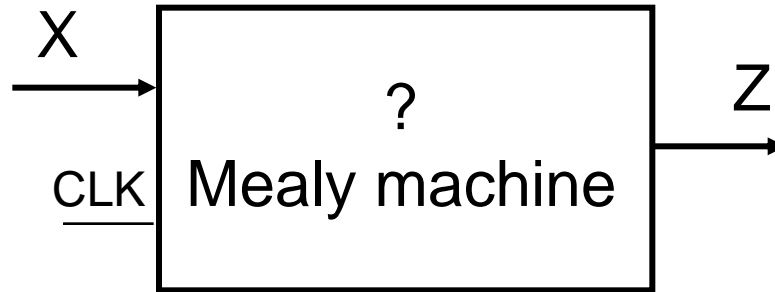
Specification

- Component Forms of Specification
 - Written description
 - Mathematical description
 - Hardware description language
 - Tabular description
 - Equation description
 - Diagram describing operation (not just structure)

Formulation: Finding a State Diagram

- In specifying a circuit, we use states to remember meaningful properties of past input sequences that are essential to predicting future output values.
- As an example, a sequence recognizer is a sequential circuit that produces a distinct output value whenever a prescribed pattern of input symbols occur in sequence, i.e, recognizes an input sequence occurrence.
- Next, the state diagram, will be converted to a state table from which the circuit will be designed.

Sequence Detector Example: 1101



Input X:		00111001101011011010011110111
Output Z:		000000000100001001000000100

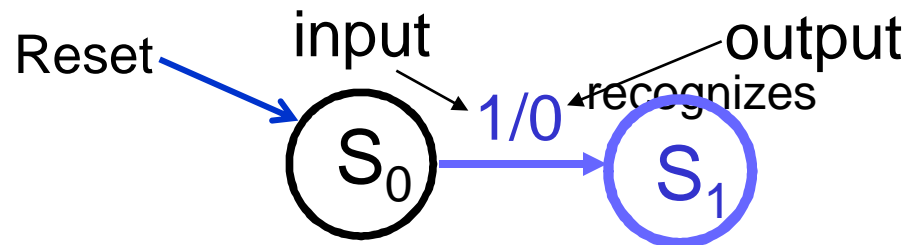
The diagram shows the input and output sequences for the Mealy machine. The input sequence is 00111001101011011010011110111 and the output sequence is 000000000100001001000000100. Blue brackets above the input sequence highlight the overlapping occurrences of the sequence 1101. A white bracket below the output sequence highlights the corresponding output 1001, which occurs once for each of the two overlapping 1101 sequences in the input.

Overlapping sequences are allowed

Step2: Finding A State Diagram

- Define states for the sequence to be recognized:
 - assuming it starts with first symbol $X=1$,
 - continues through the right sequence to be recognized, and
 - uses output 1 to mean the full sequence has occurred,
 - with output 0 otherwise.
- Starting in the initial state (named " S_0 "):

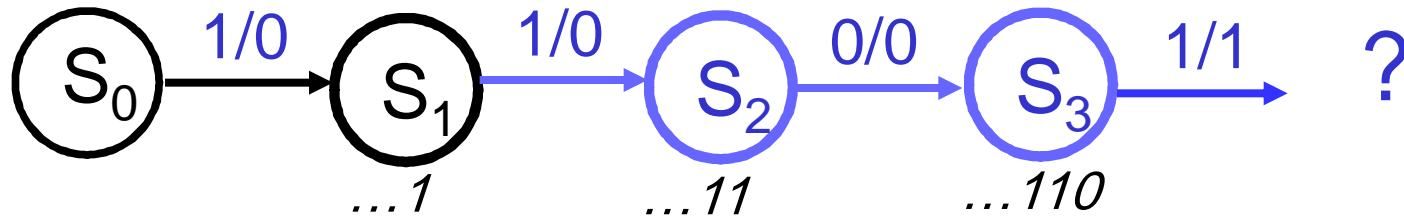
- Add a state that the first "1."



- State " S_0 " is the initial state, and state " S_1 " is the state which represents the fact that the "first" one in the input subsequence has occurred. The first "1" occurred while being in state S_0 during the clock edge.

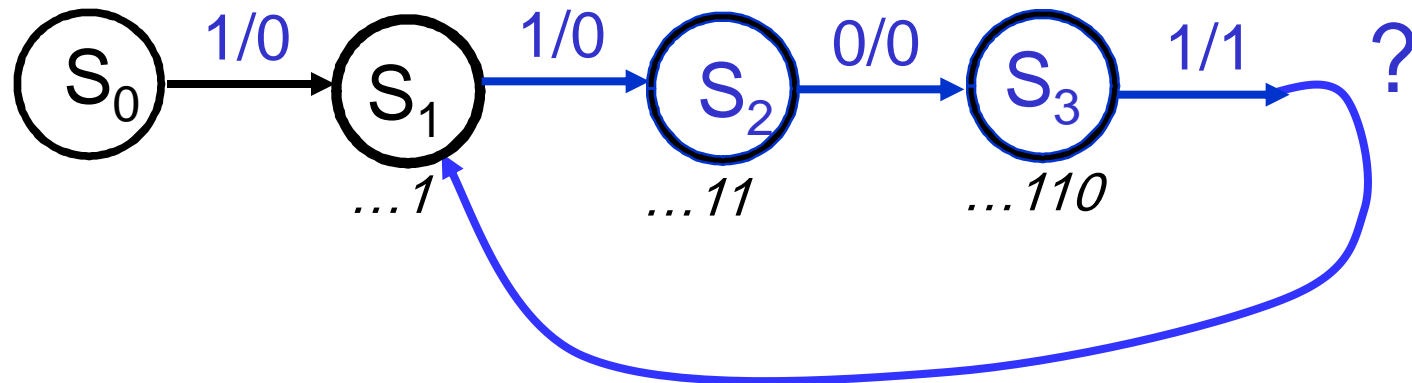
Finding a State Diagram(cont.)

- Assume that the 2nd 1 arrives of the sequence 1101: needs to be remembered: add a state S_2



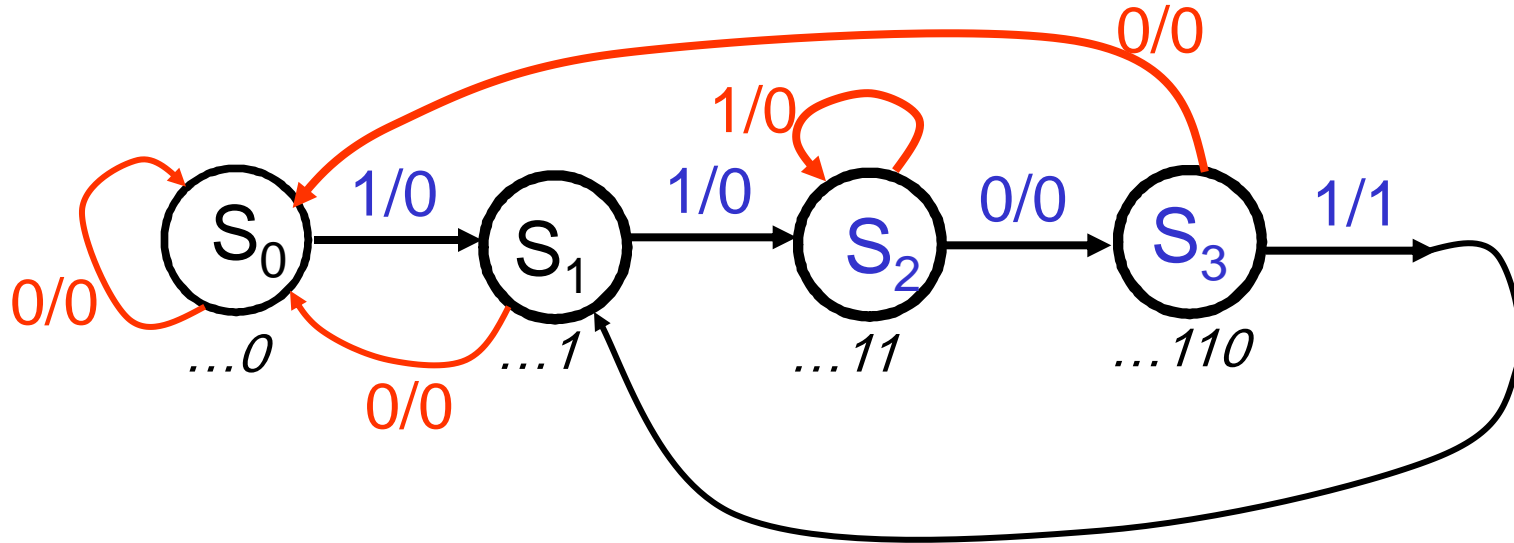
- Next, a “0” arrives: part of the sequence 1101 that needs to be remembered; add state S_3
- The next input is “1” which is part of the right sequence 1101; now output $Z=1$

Completing The State Diagram



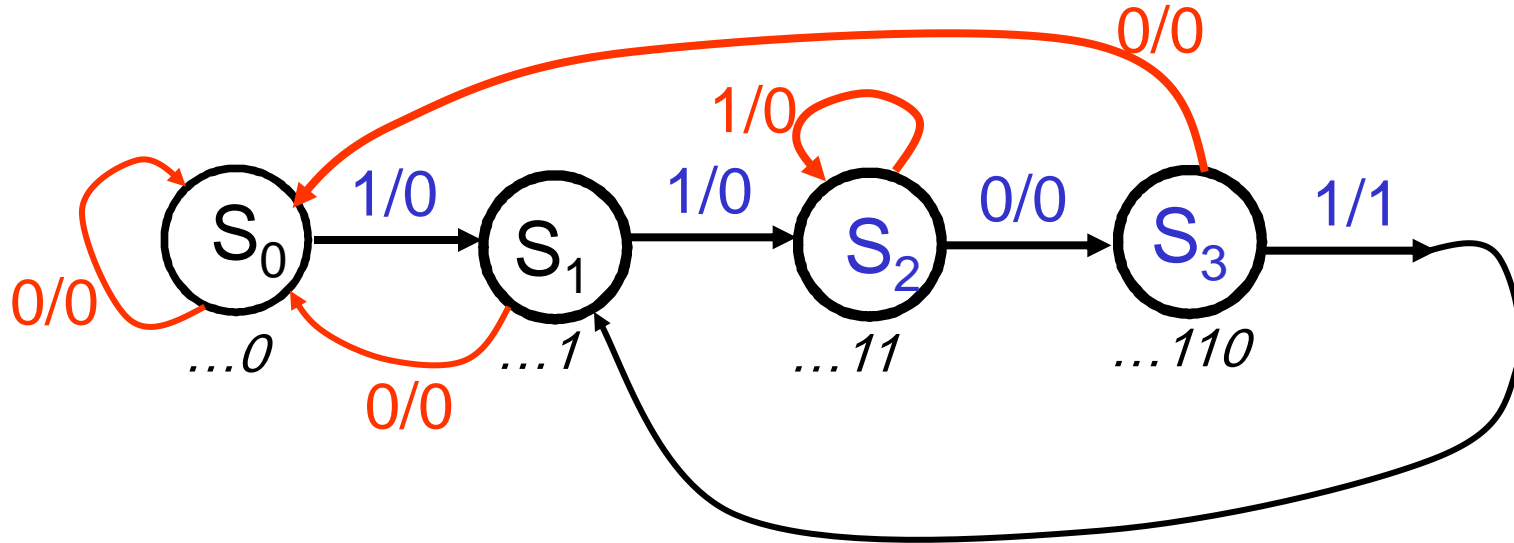
- Where does the final arrow go to:
 - The final **1** of the sequence **1101** can be the beginning of another sequence; thus the arrow should go to state S_1

Completing The State Diagram



- Start is state S_0 : assume an input $X=0$ arrives; what is the next state?
- Next, consider state S_1 : input $X=0$; next state?
- Next state S_2 and S_3 : completes the diagram
- Each state should have two arrows leaving

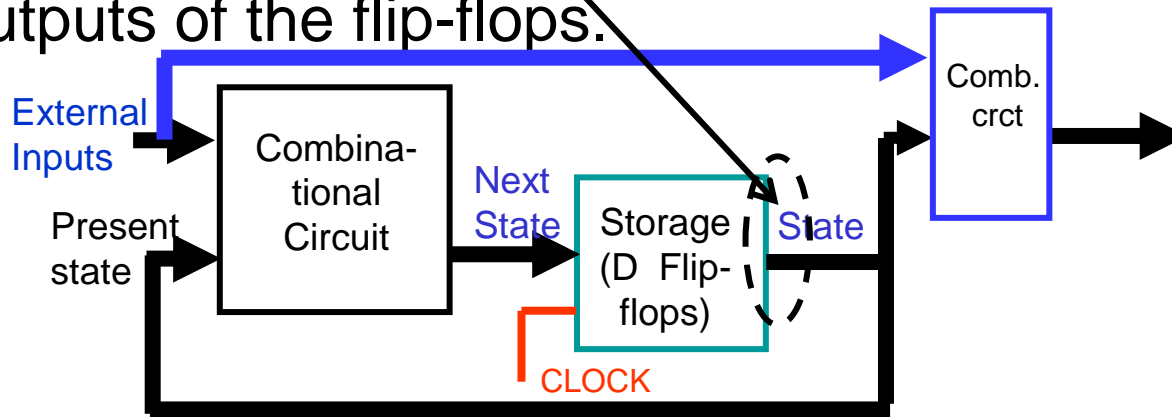
Deriving State Table



Present State	Next State		Output	
	x=0	x=1	x=0	x=1
S_0	S_0	S_1	0	0
S_1	S_0	S_2	0	0
S_2	S_3	S_2	0	0
S_3	S_0	S_1	0	1

Step 3: State Assignment

- Right now States have names such as S_0 , S_1 , S_2 and S_3
- In actuality these state need to be represented by the outputs of the flip-flops.



- We need to assign each state to a certain output combination AB of the flip-flops:
 - e.g. State $S_0=00$, $S_1=01$, $S_2=10$, $S_3=11$
 - Other combinations are possible: $S_0=00$, $S_1=10$, $S_2=11$, $S_3=01$

Popular State Assignments

- 1. Counting order assignment:
 - 00, 01, 10, 11
- 2. Gray code assignment:
 - 00, 01, 11, 10
- 3. One-hot state assignment
 - 0001, 0010, 0100, 1000
- Does state assignment make a difference in cost?

State Assignment: Counting order

“Counting Order” Assignment:

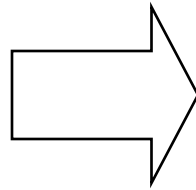
State Table:

$$S_0 = 00$$

$$S_1 = 01$$

$$S_2 = 10$$

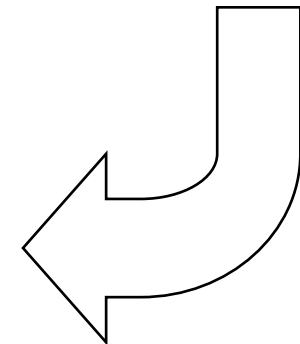
$$S_3 = 11$$



Present State	Next State		Output	
	x=0	x=1	x=0	x=1
S_0	S_0	S_1	0	0
S_1	S_0	S_2	0	0
S_2	S_3	S_2	0	0
S_3	S_0	S_1	0	1

Resulting coded state table:

Present State A B	Next State		Output	
	x = 0 $A^+ B^+$	x = 1 $A^+ B^+$	x = 0 Z	x = 1 Z
00	00	01	0	0
01	00	10	0	0
10	11	10	0	0
11	00	01	0	1



Step 4: Find Flip-Flop Input and Output Equations



• State Diagram

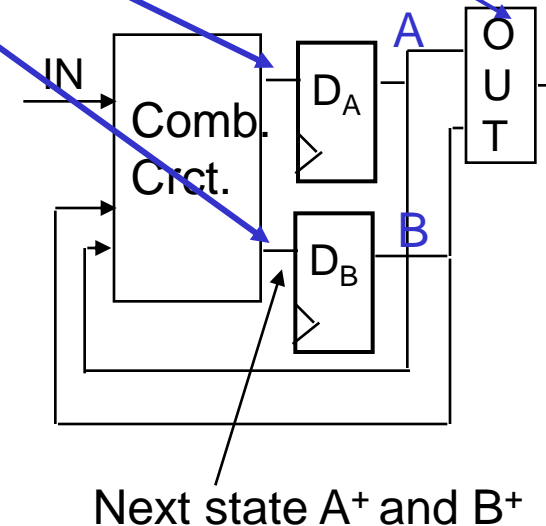
• State Table

State encoding

• Select type of Flip-flop

• Input equations to FF, output eq.

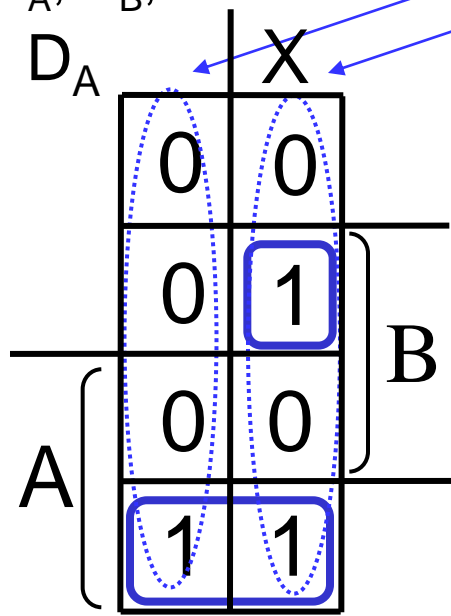
• Verification



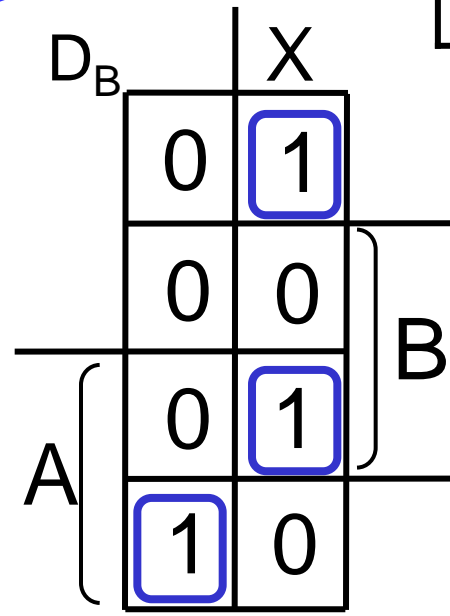
Find Flip-Flop Input and Output Equations: Example – Counting Order Assignment

- Using D flip-flops: thus $D_A = A^+$, $D_B = B^+$ (the state table is the truth table for D_A and D_B).
- Interchange the bottom two rows of the state table, to obtain **K-maps** for D_A , D_B , and Z :

Present State	Next State		Output	
	x = 0	x = 1	x = 0	x = 1
AB	$A^+ B^+$	$A^- B^+$	Z	Z
00	00	01	0	0
01	00	10	0	0
10	11	10	0	0
11	00	01	0	1



$$D_A = A\bar{B} + X\bar{A}B$$



$$D_B = \bar{X}\bar{A}\bar{B} + XAB + \bar{X}A\bar{B}$$

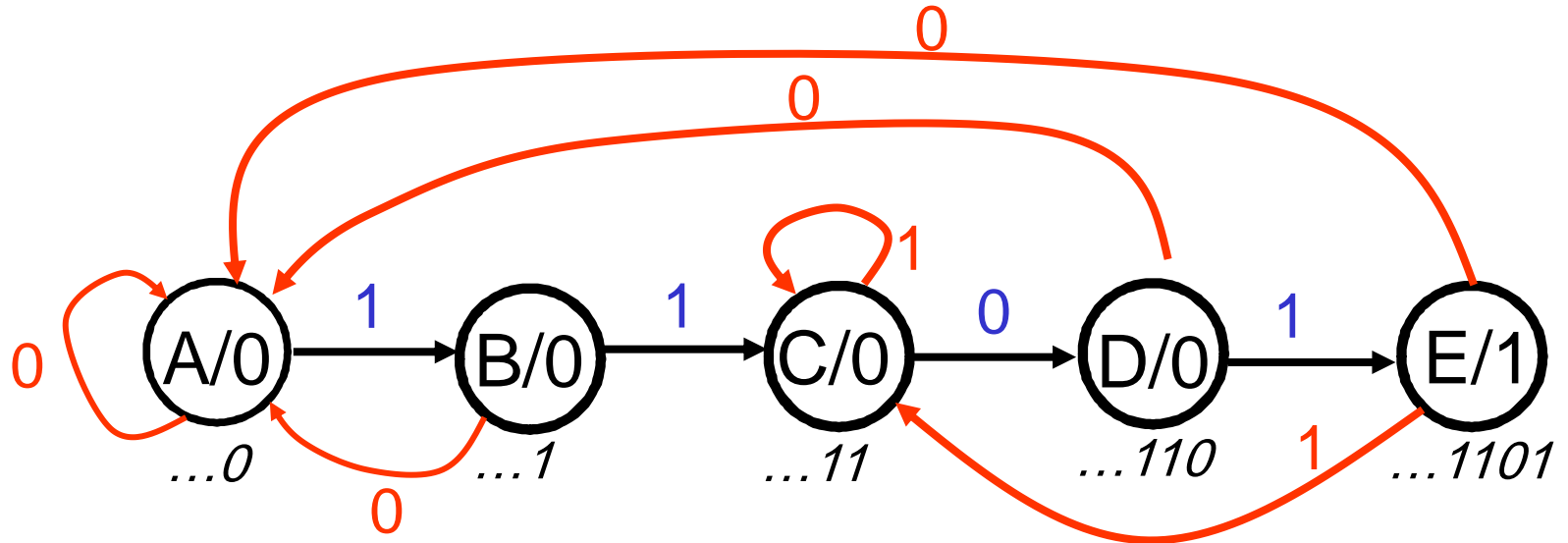
$$Z = XAB$$

Gate Input Cost = 22

Step 5: Verification

- We will learn software tools for verifying the functionality of sequential circuits in the lab (CPE0907234)

Moore model for Sequence Recognizer "1101"



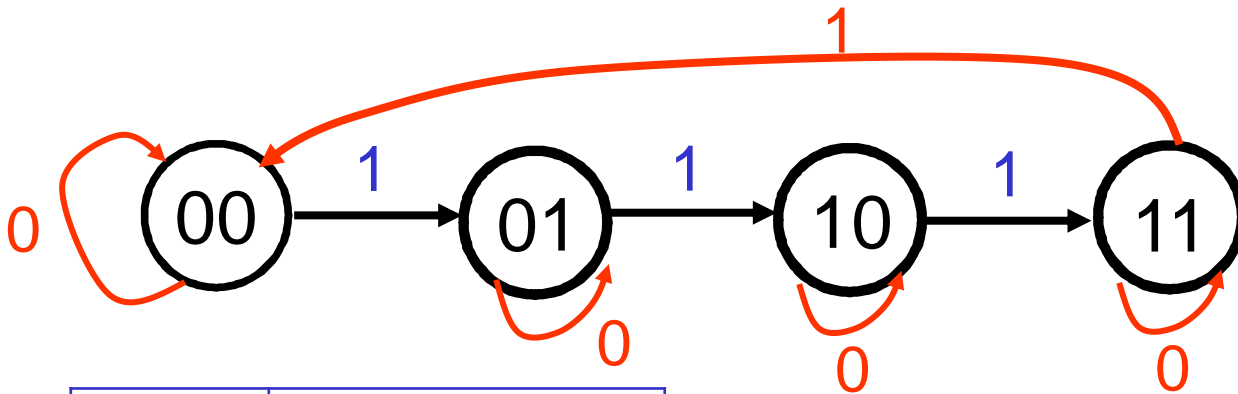
State Assignment:

- Counting order (3 Flip-flops):
 - A = 000, B = 001, C = 010, D = 011, E = 100
- Gray code (3 Flip-flops):
 - A = 000, B = 001, C = 011, D = 010, E = 110
- One hot (5 Flip-flops):
 - A = 00001, B = 00010, C = 00100, D = 01000, E = 10000

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

Exercise

- Use D Flip-Flops design a counter that counts 00,01,10,11,00,01,10,11, ..etc.
- The counter also has an input x such that the counter pauses if x=0 and proceeds to the next state if x=1.



Present State Q_1Q_0	Next State	
	X = 0 $Q_1^+Q_0^+$	X = 1 $Q_1^+Q_0^+$
00	00	01
01	01	10
10	10	11
11	11	00

Q_0^+	Q_1			
	0	1	3	2
X	4	5	7	6
	Q			
	0			

$$Q_0^+ = X \oplus Q_0$$

Q_1^+	Q_1			
	0	1	3	2
X	4	5	7	6
	Q			
	0			

$$Q_1^+ = \bar{X}Q_1 + Q_1\bar{Q}_0 + X\bar{Q}_1Q_0$$

Unused States in Sequential Circuits Design

- Unused states are states which the system **cannot** enter under normal operation.
- The system can enter an unused state due to:
 - Outside interference OR
 - Malfunction
- Three ways to accommodate unused states:
 - Assume the next state for the unused state to be don't care
 - Force the next state for the unused state to be one of the used states
 - Include a special output to indicate that the present state is unused. This output can change the state asynchronously through direct inputs of the state flip-flops

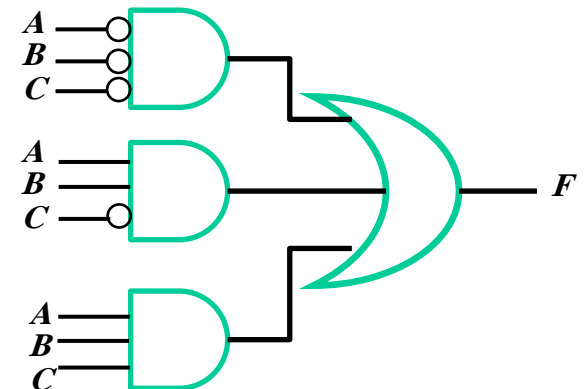
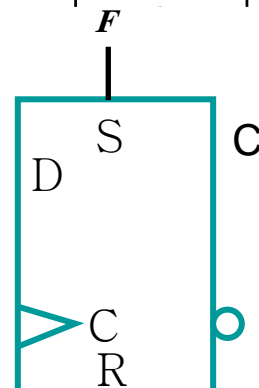
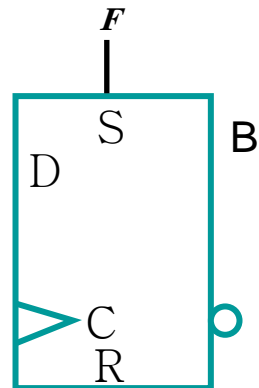
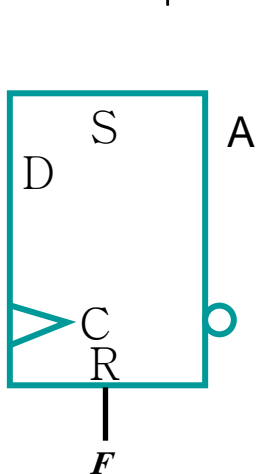
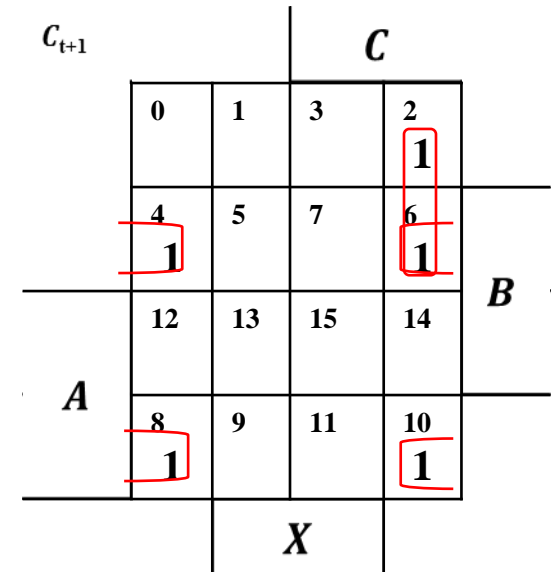
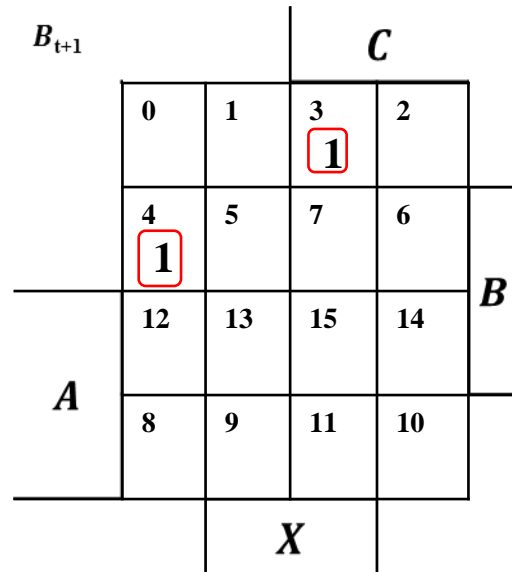
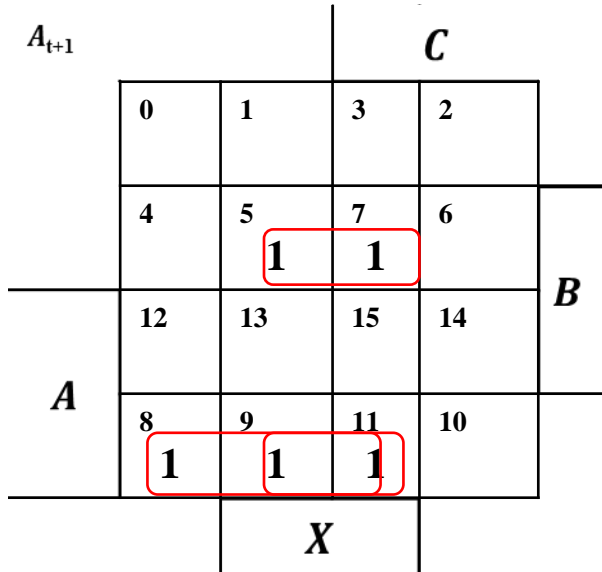
Exercise

- Use D-FFs to design the sequential circuit that implements the following state table. Note that there are three unused states (000, 110 and 111).

Present State			Input	Next State		
A	B	C	X	A	B	C
0	0	1	0	0	0	1
0	0	1	1	0	1	0
0	1	0	0	0	1	1
0	1	0	1	1	0	0
0	1	1	0	0	0	1
0	1	1	1	1	0	0
1	0	0	0	1	0	1
1	0	0	1	1	0	0
1	0	1	0	0	0	1
1	0	1	1	1	0	0

Solution

- Asynchronously change the state to "011"

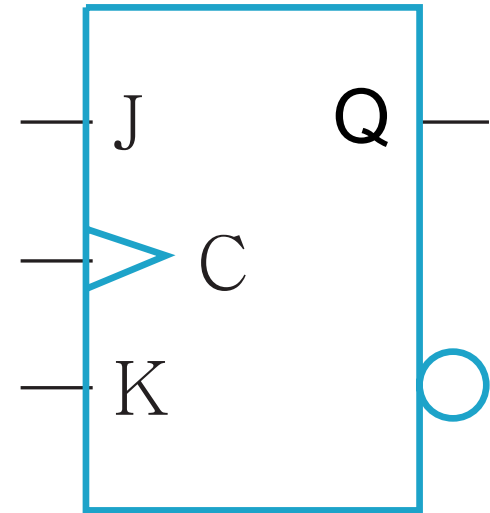


5-6 Other Flip-Flop Types

- J-K and T flip-flops
 - Behavior
 - Implementation
- Basic descriptors for understanding and using different flip-flop types
 - Characteristic tables
 - Defines the next state as a function of the present state and input
 - Characteristic equations
 - Excitation tables

J-K Flip-flop

- Behavior of JK flip-flop:
 - Same as S-R flip-flop with J analogous to S and K analogous to R
 - Except that $J = K = 1$ is **allowed**, and
 - For $J = K = 1$, the flip-flop changes to the *opposite state* (toggle)
- Behavior described by the **characteristic table** (function table):

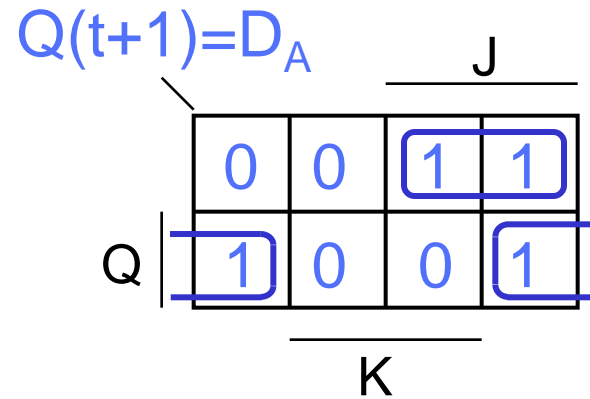


J	K	Q(t+1)	
0	0	Q(t)	no change
0	1	0	reset
1	0	1	set
1	1	$\overline{Q(t)}$	toggle

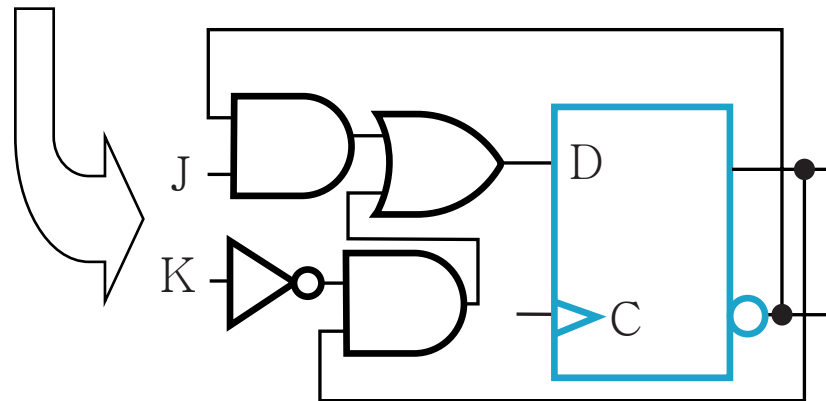
Design of an edge-triggered J-K Flip-Flop

State table of a JK FF:

Present state Q	Inputs		Next state Q(t+1)
	J	K	
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0



$Q(t+1) = D_A = JQ' + K'Q$
 Called the characteristic equation



J-K Flip-Flop Excitation Table

Q(t)	Q(t +1)	J	K	Operation
0	0	0	X	No change
0	1	1	X	Set
1	0	X	1	Reset
1	1	X	0	No Change

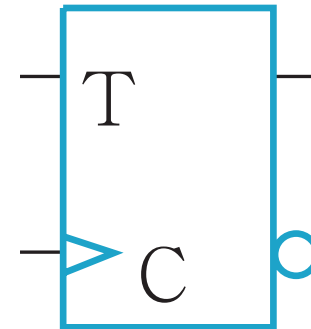
T Flip-Flop

- Behavior described by its characteristic table:
 - Has a single input T
 - For $T = 0$, no change to state
 - For $T = 1$, changes to opposite state
- Same as a J-K flip-flop with $J = K = T$

T	Q(t+1)
0	Q(t) no change
1	$\overline{Q}(t)$ complement

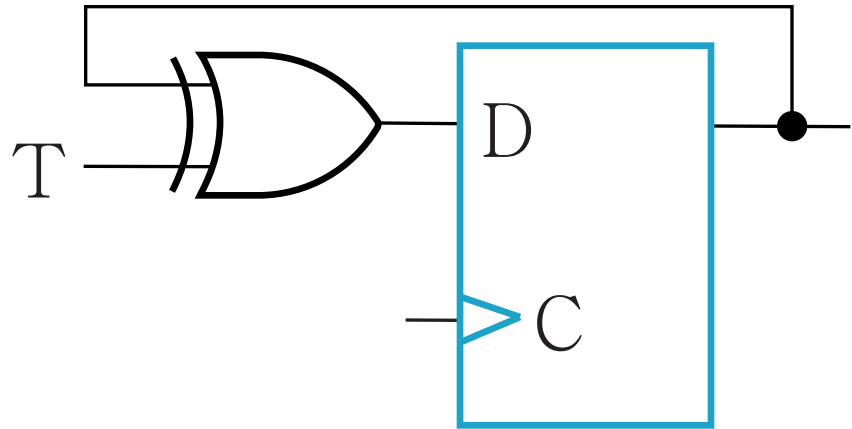
Characteristic equation:

$$Q(t+1) = T'Q(t) + TQ'(t) \\ = T \oplus Q(t)$$



T Flip-Flop Realization

- Using a D Flip-flop: $D = T \oplus Q(t)$



- Cannot be initialized to a known state using the T input
 - Reset (asynchronous or synchronous) essential

T Flip-Flop Excitation Table

Q(t +1)	T	Operation
$Q(t)$	0	No change
$\bar{Q}(t)$	1	Complement

Flip-Flops Characteristics

For analysis

- *Characteristic table* - defines the next state of the flip-flop in terms of flip-flop inputs and current state
- *Characteristic equation* - defines the next state of the flip-flop as a Boolean function of the flip-flop inputs and the current state.

For design

- *Excitation table* - defines the flip-flop input variable values as function of the current state and next state. In other words, the table tells us what input is needed to cause a transition from the current state to a specific next state.

D Flip-Flop Descriptors

- Characteristic Table

D	Q(t+1)	Operation
0	0	Reset
1	1	Set

- Characteristic Equation

$$Q(t+1) = D$$

- Excitation Table

Q(t + 1)	D	Operation
0	0	Reset
1	1	Set

S-R Flip-Flop Descriptors

- Characteristic Table

S	R	Q(t+1)	Operation
0	0	Q(t)	No change
0	1	0	Reset
1	0	1	Set
1	1	?	Undefined

- Characteristic Equation

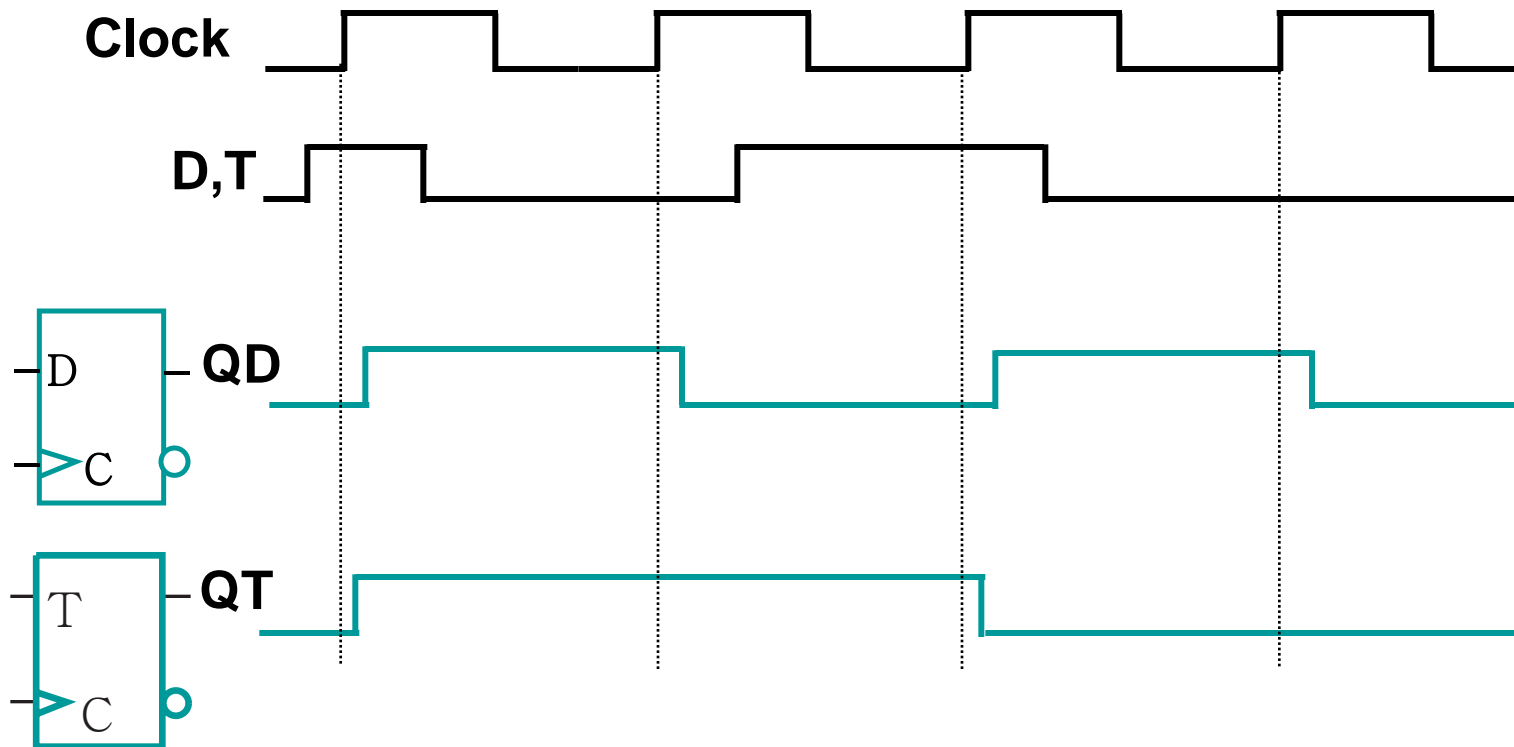
$$Q(t+1) = S + \bar{R} Q, S \cdot R = 0$$

- Excitation Table

Q(t)	Q(t+1)	S	R	Operation
0	0	0	X	No change
0	1	1	0	Set
1	0	0	1	Reset
1	1	X	0	No change

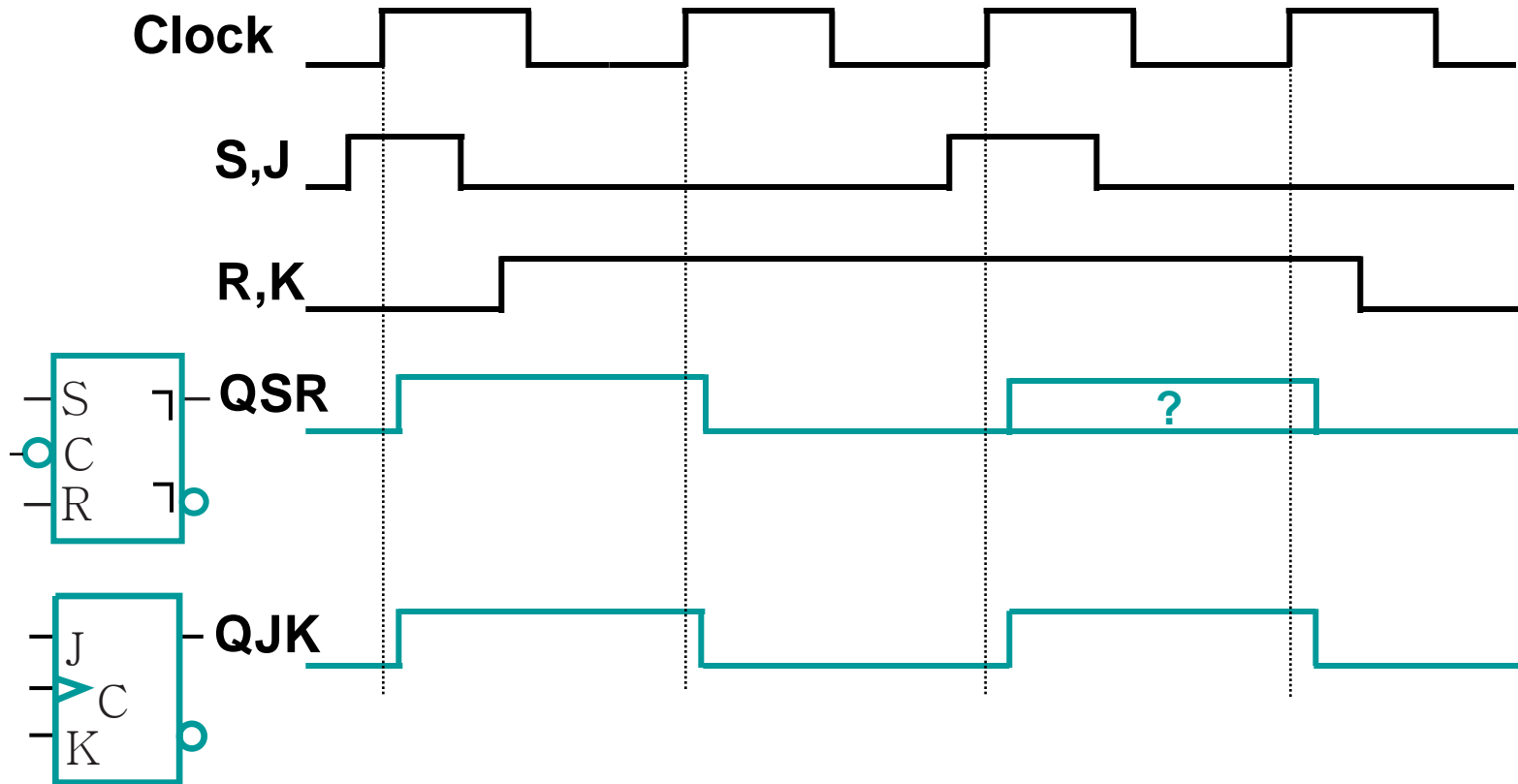
Flip-flop Behavior Example

- Use the characteristic tables to find the output waveforms for the flip-flops shown:

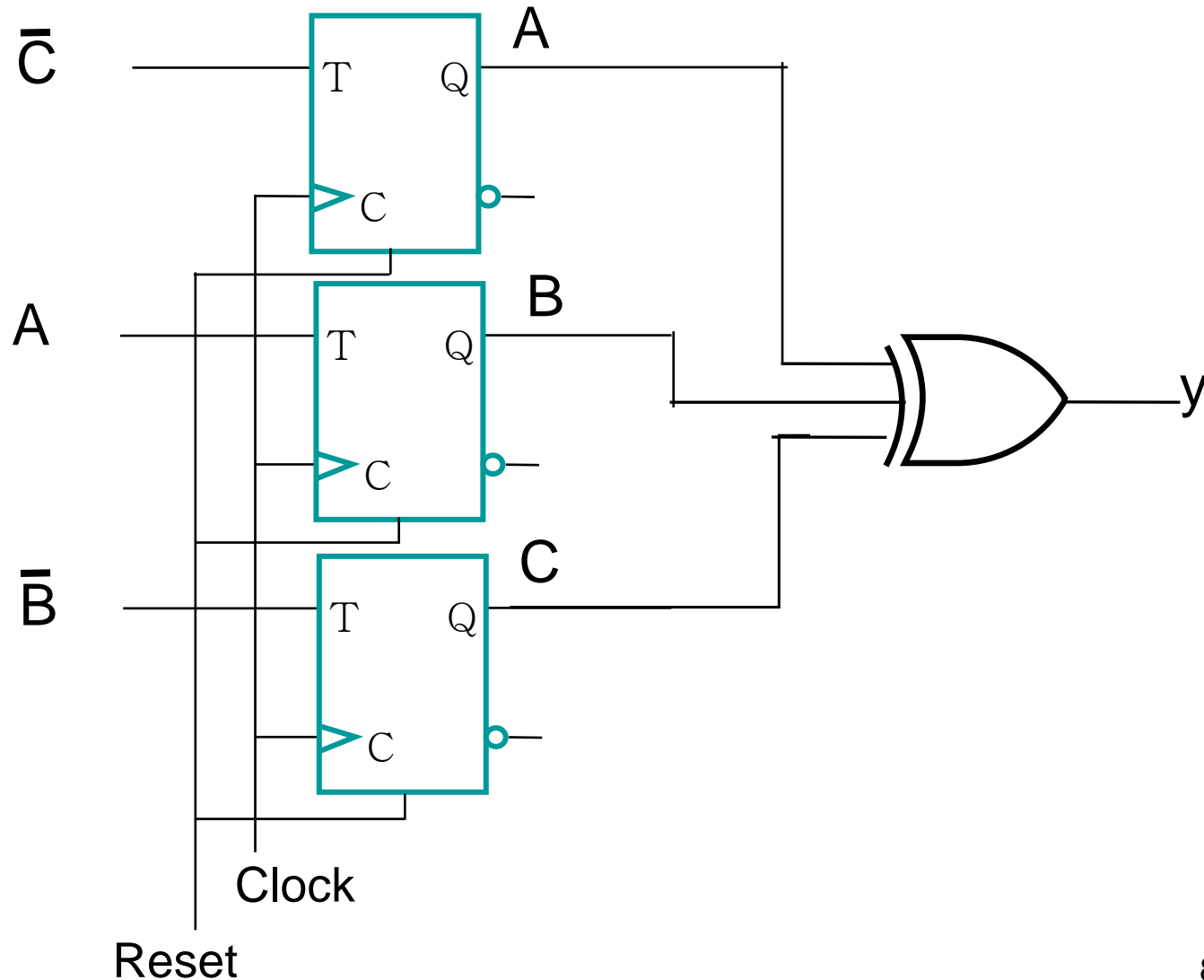


Flip-Flop Behavior Example (continued)

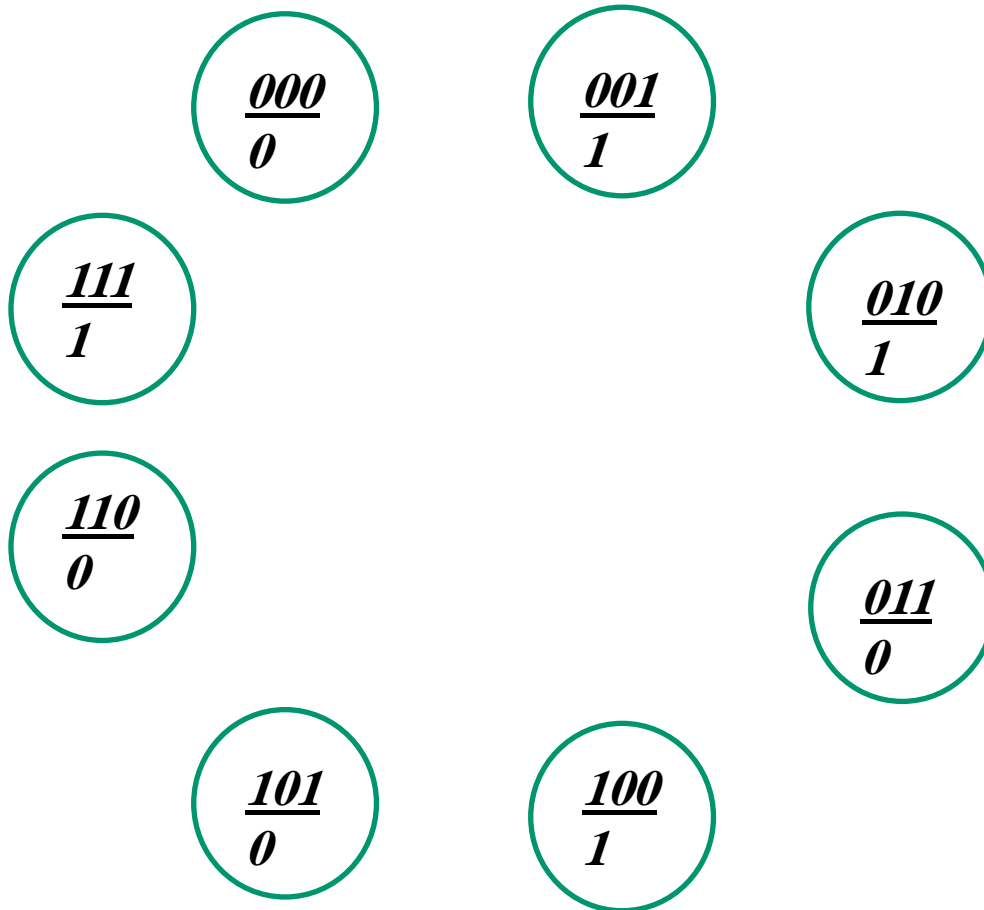
- Use the characteristic tables to find the output waveforms for the flip-flops shown:



Exercise: Find State Diagram

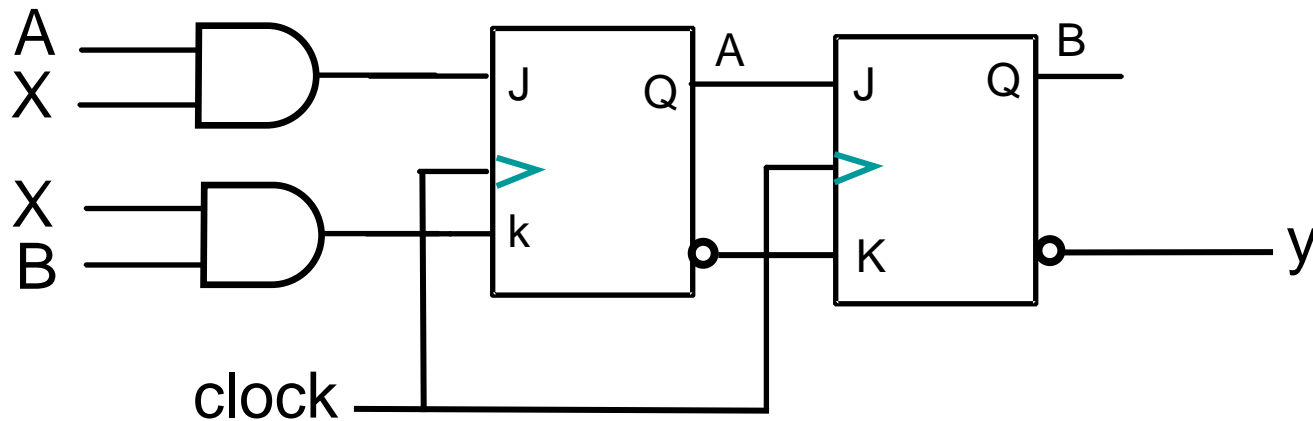


Present state						Next State			Y
A	B	C	T _A	T _B	T _C	A+	B+	C+	y
0	0	0	1	0	1	1	0	1	0
0	0	1	0	0	1	0	0	0	1
0	1	0	1	0	0	1	1	0	1
0	1	1	0	0	0	0	1	1	0
1	0	0	1	1	1	0	1	1	1
1	0	1	0	1	1	1	1	0	0
1	1	0	1	1	0	0	0	0	0
1	1	1	0	1	0	1	0	1	1

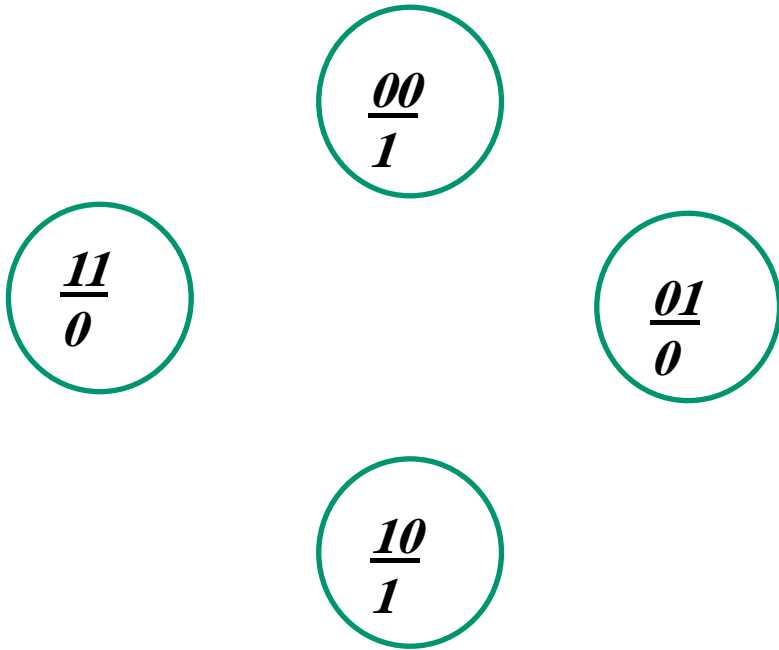




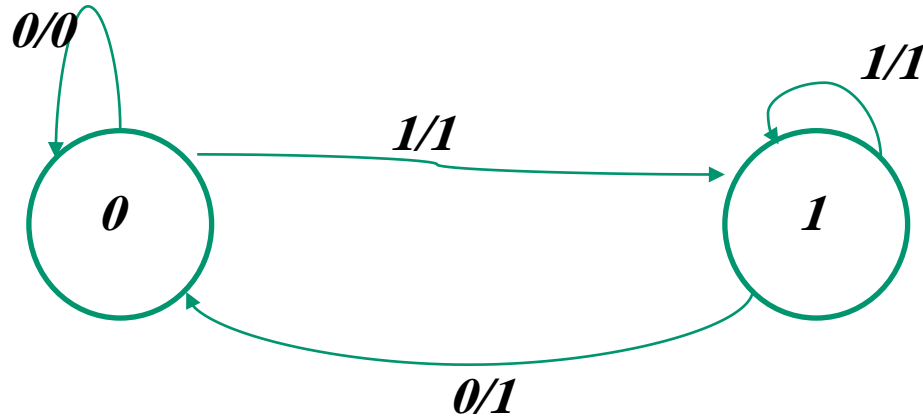
Exercise: Find State Diagram



Present state							Next State		Y
A	B	X	J_A	K_A	J_B	K_B	A+	B+	y
0	0	0	0	0	0	1	0	0	1
0	0	1	0	0	0	1	0	0	1
0	1	0	0	0	0	1	0	0	0
0	1	1	0	1	0	1	0	0	0
1	0	0	0	0	1	0	1	1	1
1	0	1	1	0	1	0	1	1	1
1	1	0	0	0	1	0	1	1	0
1	1	1	1	1	1	0	0	1	0



-
- Given the following state diagram design the sequential circuit that implements it. Compare the design when TFF & DFF is used.



PS	input	NS	out
A	X	A ⁺	y
0	0	0	0
0	1	1	1
1	0	0	1
1	1	1	1

Exercise

- Design the sequential circuit that implements the following state table using
 - JK Flip-Flops
 - T Flip-Flops
 - SR Flip-Flops
 - D Flip-Flops

Present State		Input	Next State		Output
A(t)	B(t)	X	A(t+1)	B(t+1)	Y
0	0	0	0	0	0
0	0	1	0	1	1
0	1	0	0	1	0
0	1	1	1	0	1
1	0	0	1	0	1
1	0	1	1	1	1
1	1	0	1	1	1
1	1	1	0	0	1

Using JK FF

Present state							Next State		Y
A	B	X	J _A	K _A	J _B	K _B	A+	B+	y
0	0	0	0	X	0	X	0	0	0
0	0	1	0	X	1	X	0	1	1
0	1	0	0	X	X	0	0	1	0
0	1	1	1	X	X	1	1	0	1
1	0	0	X	0	0	X	1	0	1
1	0	1	X	0	1	X	1	1	1
1	1	0	X	0	X	0	1	1	1
1	1	1	X	1	X	1	0	0	1

Using TFF

Present state					Next State		Y
A	B	X	T_A	T_B	A+	B+	y
0	0	0	0	0	0	0	0
0	0	1	0	1	0	1	1
0	1	0	0	0	0	1	0
0	1	1	1	1	1	0	1
1	0	0	0	0	1	0	1
1	0	1	0	1	1	1	1
1	1	0	0	0	1	1	1
1	1	1	1	1	0	0	1

Using SR FF

Present state							Next State		Y
A	B	X	S_A	R_A	S_B	R_B	A+	B+	y
0	0	0	0	X	0	X	0	0	0
0	0	1	0	X	1	0	0	1	1
0	1	0	0	X	X	0	0	1	0
0	1	1	1	0	0	1	1	0	1
1	0	0	X	0	0	X	1	0	1
1	0	1	X	0	1	0	1	1	1
1	1	0	X	0	X	0	1	1	1
1	1	1	0	1	0	1	0	0	1

Exercise

- Design the sequential circuit that implements the following state table using
 - JK Flip-Flops
 - T Flip-Flops
 - SR Flip-Flops
 - D Flip-Flops

Present State		Input	Next State		Output
A(t)	B(t)	X	A(t+1)	B(t+1)	Y
0	0	0	0	1	1
0	0	1	0	0	1
0	1	0	1	0	0
0	1	1	1	1	0
1	0	0	0	1	1
1	0	1	0	0	1
1	1	0	1	1	0
1	1	1	1	0	0

Topics Covered

- Storage Elements and Analysis
 - Introduction to sequential circuits
 - Types of sequential circuits
 - Storage elements
 - Latches
 - Flip-flops
- Sequential circuit analysis
 - State tables
 - State diagrams
 - Equivalent states
 - Moore and Mealy Models
- Sequential Circuit Design
- Other Flip-Flops Types

Terms of Use

- **All (or portions) of this material © 2008 by Pearson Education, Inc.**
- **Permission is given to incorporate this material or adaptations thereof into classroom presentations and handouts to instructors in courses adopting the latest edition of Logic and Computer Design Fundamentals as the course textbook.**
- **These materials or adaptations thereof are not to be sold or otherwise offered for consideration.**
- **This Terms of Use slide or page is to be included within the original materials or any adaptations thereof.**



Updates of Chapter5 Slides

By Dr. Waleed Dweik

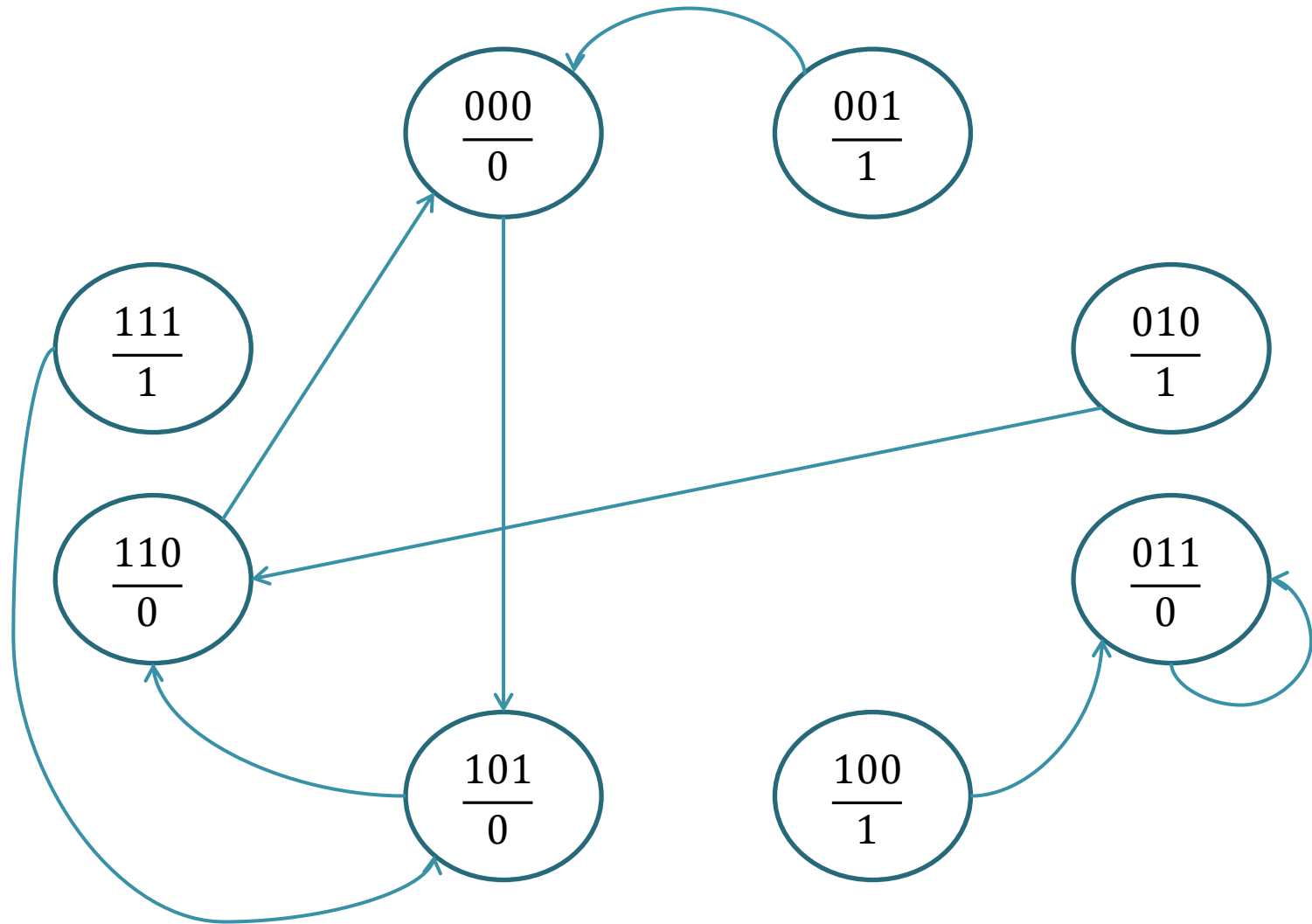
Updates on Slide 45

- Input equations:
 - $S_A = \overline{Q_A} \cdot X$
 - $R_A = Q_A + \overline{X}$
 - $S_B = Q_A \cdot \overline{Q_B}$
 - $R_B = \overline{Q_A} + Q_B$
- Output equation:
 - $Y = (Q_A \cdot \overline{Q_B}) + X \rightarrow \text{Mealy}$

Updates on Slide 88

- Input equations:
 - $T_A = \bar{C}$
 - $T_B = A$
 - $T_C = \bar{B}$
- Output equation:
 - $Y = A \oplus B \oplus C \rightarrow \text{Moore}$

Updates on Slide 90

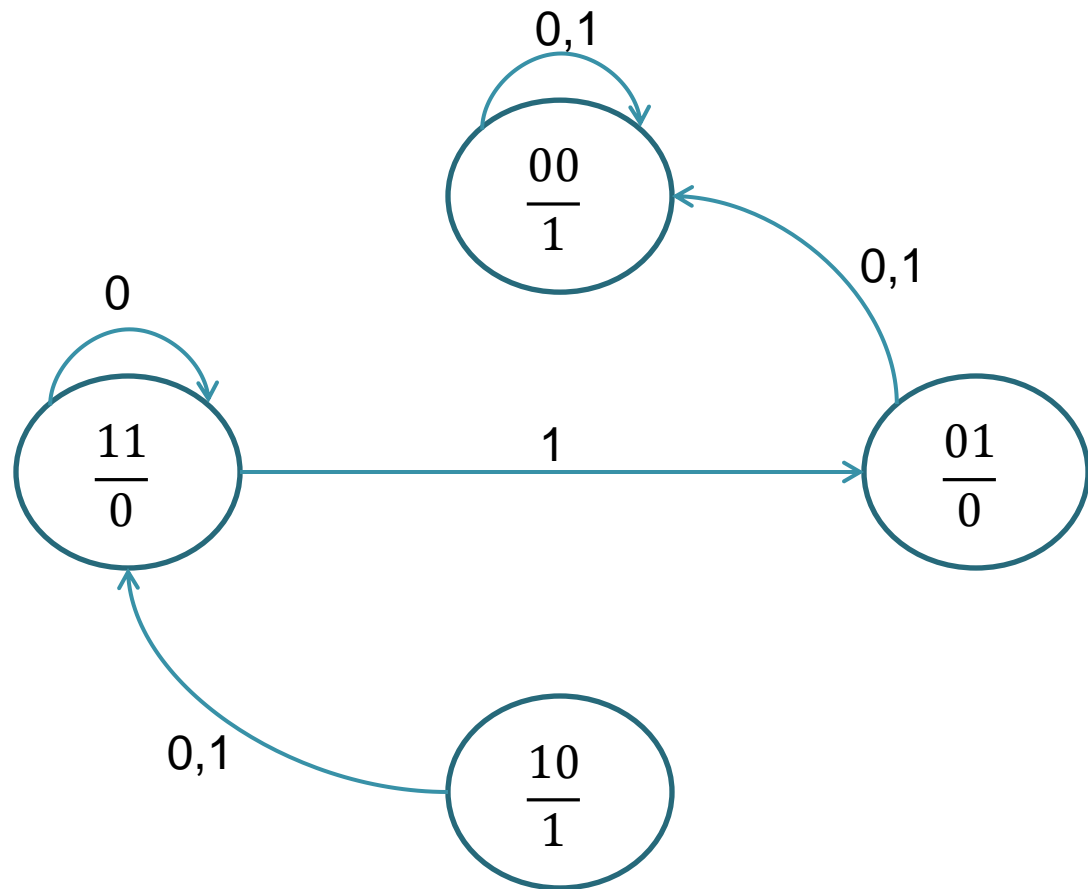


Updates on Slide 92

- Input equations:
 - $J_A = A.X$
 - $K_A = B.X$
 - $J_B = A$
 - $K_B = \bar{A}$

- Output equation:
 - $Y = \bar{B} \rightarrow \text{Moore}$

Updates on Slide 94



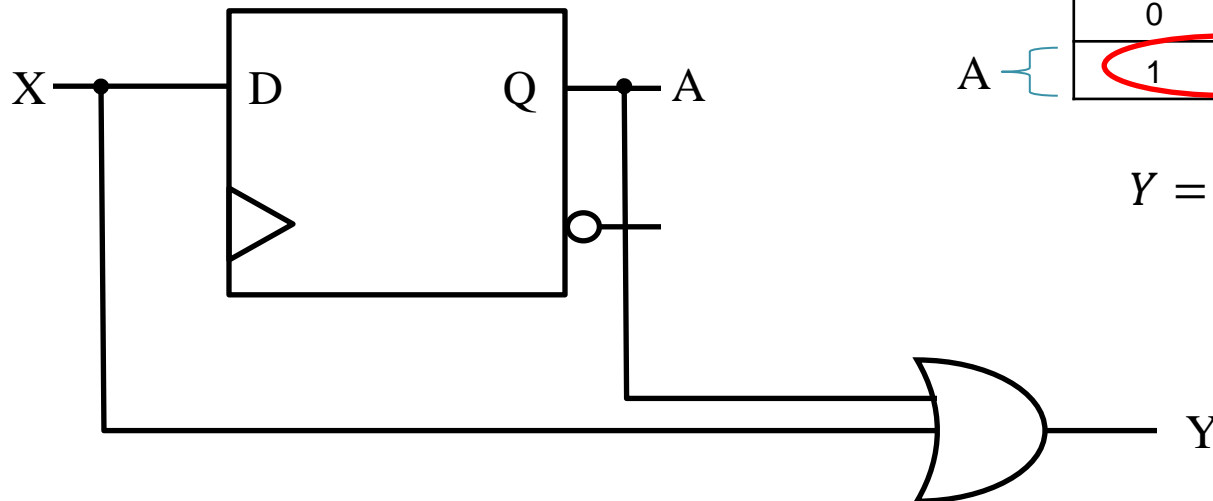
Updates on Slide 96

- Implementation using D-FF:

Present State	Input	Next State	Output
A	X	$A+ = D_A$	Y
0	0	0	0
0	1	1	1
1	0	0	1
1	1	1	1

		X
		1
A		1

$$D_A = X$$



		X
		1
A	1	1

$$Y = A + X$$

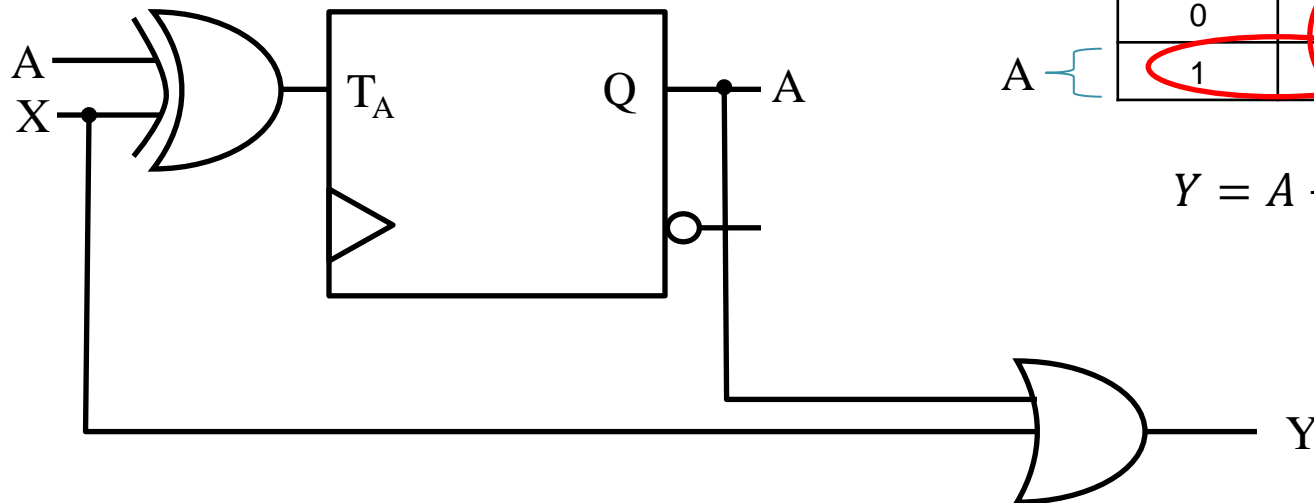
Updates on Slide 96

- Implementation using T-FF:

Present State	Input	Next State	Output	
A	X	A+	Y	T_A
0	0	0	0	0
0	1	1	1	1
1	0	0	1	1
1	1	1	1	0

		X	
A		0	1
1	0	1	0

$$T_A = \bar{A}X + A\bar{X} = A \oplus X$$



		X	
A		0	1
1	0	1	1

$$Y = A + X$$

Updates on Slide 98

	B			
	0	0	1	0
A	x	x	x	x
	X			

$$J_A = B \cdot X$$

	B			
	x	x	x	x
A	0	0	1	0
	X			

$$K_A = B \cdot X$$

	B			
	0	1	x	x
A	0	1	x	x
	X			

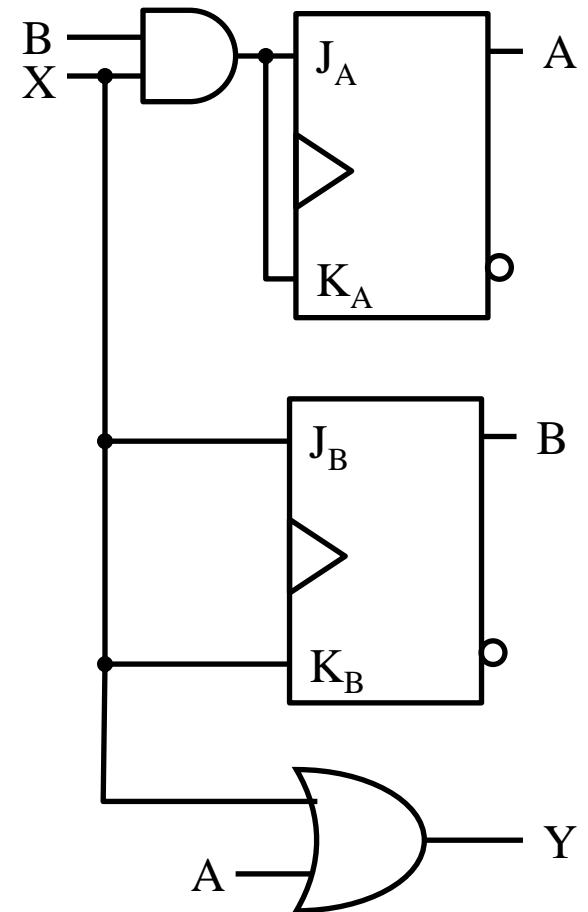
$$J_B = X$$

	B			
	x	x	1	0
A	x	x	1	0
	X			

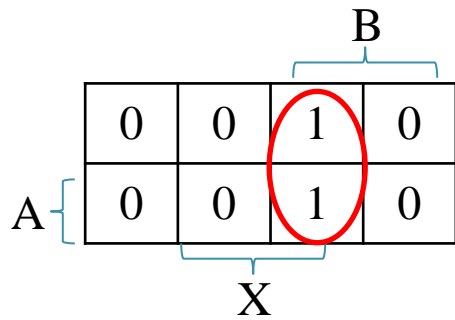
$$K_B = X$$

	B			
	0	1	1	0
A	1	1	1	1
	X			

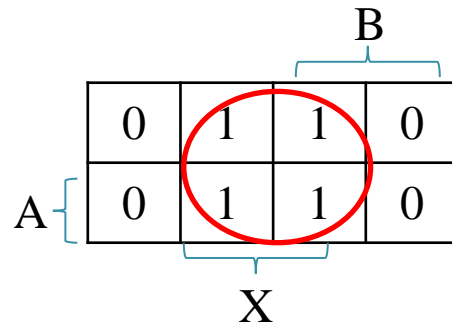
$$Y = A + X$$



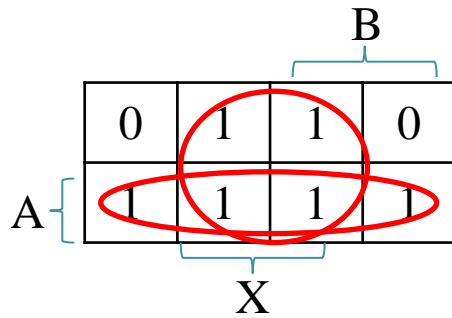
Updates on Slide 99



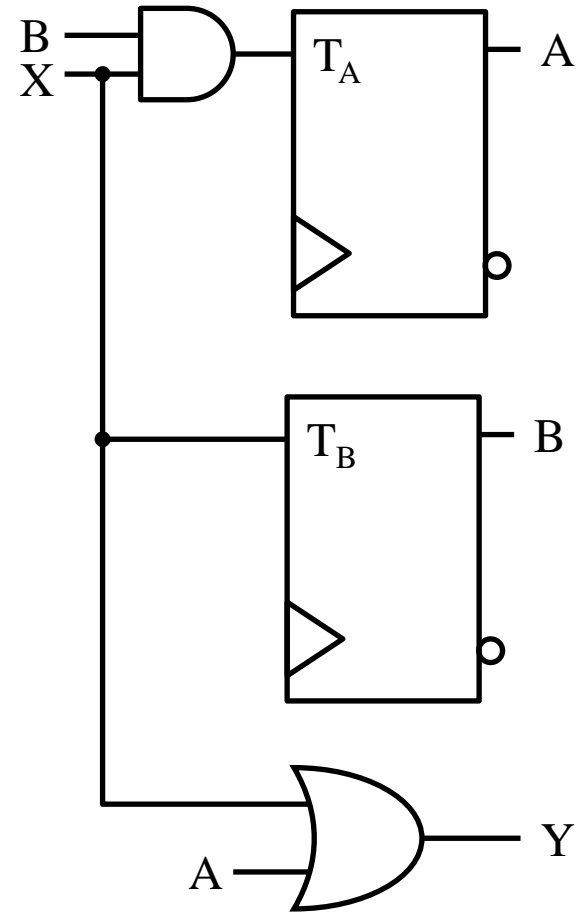
$$T_A = B.X$$



$$T_B = X$$



$$Y = A + X$$



Updates on Slide 100

	B			
	0	0	1	0
A	x	x	0	x
	X			

$$S_A = \bar{A} \cdot B \cdot X$$

	B			
	x	x	0	x
A	0	0	1	0
	X			

$$R_A = A \cdot B \cdot X$$

	B			
	0	1	0	x
A	0	1	0	x
	X			

$$S_B = \bar{B} \cdot X$$

	B			
	x	0	1	0
A	x	0	1	0
	X			

$$R_B = B \cdot X$$

	B			
	0	1	1	0
A	1	1	1	1
	X			

$$Y = A + X$$

