COE211: Digital Logic Design

Introduction

Digital Computer Systems

- Digital systems consider *discrete* amounts of data
- Examples
  - 26 letters in the alphabet
  - 10 decimal digits
- Larger quantities can be built from discrete values:
  - Words made of letters
  - Numbers made of decimal digits (e.g. 239875.32)
- Computers operate on binary values (0 and 1)
- Easy to represent binary values electrically
  - Voltages and currents
  - Can be implemented using circuits
  - Create the building blocks of modern computers

Understanding Decimal Numbers

- **Decimal numbers are made of decimal digits: (0,1,2,3,4,5,6,7,8,9) ⇒ Base = 10**
- **But how many items does a decimal number represent?**
  - **$8653 = 8 \times 10^3 + 6 \times 10^2 + 5 \times 10^1 + 3 \times 10^0$**
  - **Number = $d_3 \times B^3 + d_2 \times B^2 + d_1 \times B^1 + d_0 \times B^0$ = Value**
- **What about fractions?**
  - **$97654.35 = 9\times10^4 + 7\times10^3 + 6\times10^2 + 5\times10^1 + 4\times10^0 + 3\times10^{-1} + 5\times10^{-2}$**
  - **In formal notation → $(97654.35)_{10}$**

Understanding Binary Numbers

- **Binary numbers are made of binary digits (bits):**
  - **0 and 1**
- **How many items does a binary number represent?**
  - **8 4 2 1 = Weights**
  - $(1011)_2 = 1x2^3 + 0x2^2 + 1x2^1 + 1x2^0 = (11)_{10}$
- **What about fractions?**
  - $(110.10)_2 = 1x2^2 + 1x2^1 + 0x2^0 + 1x2^{-1} + 0x2^{-2}$
- **Groups of eight bits are called a byte**
  - $(11001001)_2$
- **Groups of four bits are called a nibble**
  - $(1101)_2$

Understanding Octal Numbers

- Octal numbers are made of octal digits: (0,1,2,3,4,5,6,7)
- How many items does an octal number represent?
  - 512 64 8 1 = Weights
  - $(4536)_8 = 4x8^3 + 5x8^2 + 3x8^1 + 6x8^0 = (1362)_{10}$
- What about fractions?
  - $(465.27)_8 = 4x8^2 + 6x8^1 + 5x8^0 + 2x8^{-1} + 7x8^{-2}$
- Octal numbers don't use digits 8 or 9

Understanding Hexadecimal Numbers

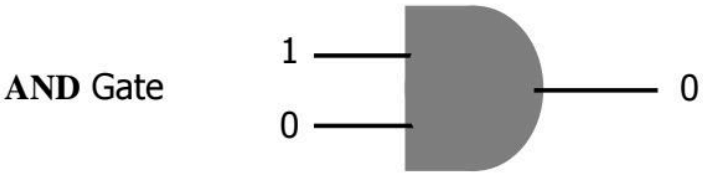
- **Hexadecimal numbers are made of 16 digits:**
  - **(0,1,2,3,4,5,6,7,8,9,A, B, C, D, E, F)**
- **How many items does a hex number represent?**

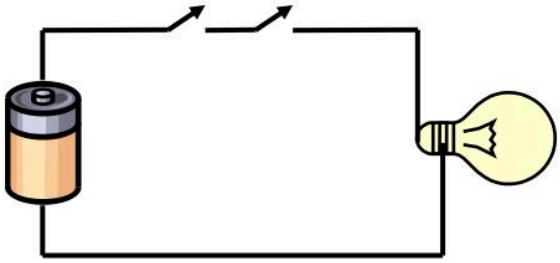
4096 256 16 1 = Weights

  - $(3A9F)_{16} = 3x16^3 + 10x16^2 + 9x16^1 + 15x16^0 = 14999_{10}$
- **What about fractions?**
  - $(2D3.5)_{16} = 2x16^2 + 13x16^1 + 3x16^0 + 5x16^{-1} = 723.3125_{10}$
- **Note that each hexadecimal digit can be represented with four bits**
  - $(1110)_2 = (E)_{16}$

Why Use Binary Numbers?

- **Easy to represent 0 and 1 using electrical values**
- **Possible to tolerate noise**
- **Easy to transmit data**
- **Easy to build binary circuits**





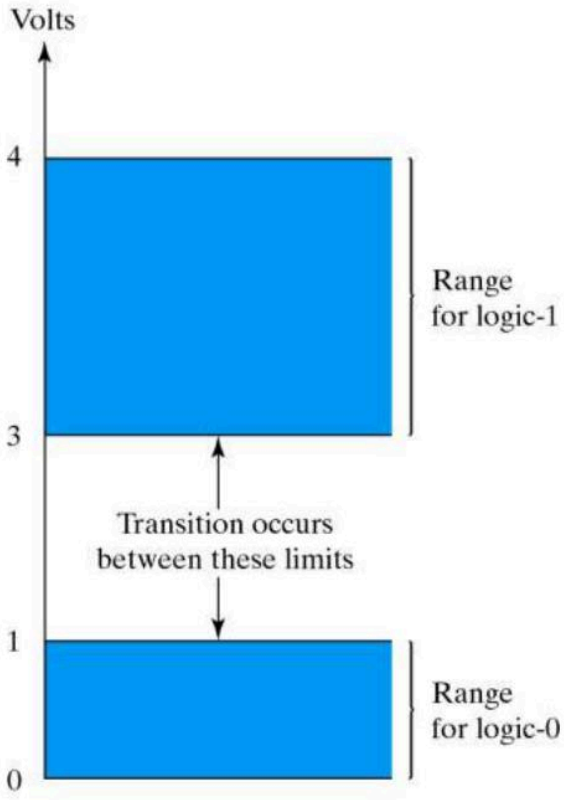


Fig. 1-3 Example of binary signals

Convert an Integer *from* Decimal *to* Another Base

For each digit position:

1. **Divide decimal number by the base (e.g. 2)**
2. **The *remainder* is the lowest-order digit**
3. **Repeat first two steps until no *divisor* remains**

Example for $(13)_{10}$:

	Quotient		Remainder	Coefficient
13/2 =	6	,	1	$a_0 = 1$
6/2 =	3	,	0	$a_1 = 0$
3/2 =	1	,	1	$a_2 = 1$
1/2 =	0	,	1	$a_3 = 1$

Answer $(13)_{10} = (a_3\, a_2\, a_1\, a_0)_2 = (1101)_2$

MSB LSB

Convert a Fraction *from* Decimal *to* Another Base

For each digit position:

1. **Multiply decimal number by the base (e.g. 2)**
2. **The *integer* is the highest-order digit**
3. **Repeat first two steps until fraction becomes zero**

Example for $(0.625)_{10}$:

	Integer		Fraction	Coefficient
0.625 x 2 =	1	+	0.250	$a_{-1} = 1$
0.250 x 2 =	0	+	0.500	$a_{-2} = 0$
0.500 x 2 =	1	+	0	$a_{-3} = 1$

Answer $(0.625)_{10} = (0.a_{-1}\, a_{-2}\, a_{-3})_2 = (0.101)_2$

MSB LSB

Conversion Between Base 16 and Base 2

- **Conversion is easy!**

  Determine the 4-bit value for each hex digit
- **Note that there are 16 different values of four bits**
- **Easier to read and write in hexadecimal**
- **Representations are equivalent!**

$$3A9F_{16} = \underline{0011}\ \underline{1010}\ \underline{1001}\ \underline{1111}_2$$

3 A 9 F

The Growth of Binary Numbers

n	2^n
0	$2^0=1$
1	$2^1=2$
2	$2^2=4$
3	$2^3=8$
4	$2^4=16$
5	$2^5=32$
6	$2^6=64$
7	$2^7=128$

n	2^n	
8	$2^8=256$	
9	$2^9=512$	
10	$2^{10}=1024$	Kilo
11	$2^{11}=2048$	
12	$2^{12}=4096$	
20	$2^{20}=1M$	Mega
30	$2^{30}=1G$	Giga
40	$2^{40}=1T$	Tera

Understanding Binary Coded Decimal

- ***Binary Coded Decimal*** **(BCD) represents each decimal digit with four bits**

  Ex. $\underbrace{0011}_{3}\ \underbrace{0010}_{2}\ \underbrace{1001}_{9} = 329_{10}$

- **This is NOT the same as** $001100101001_2 = 809_{10}$

Digit	BCD Code
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

Putting It All Together

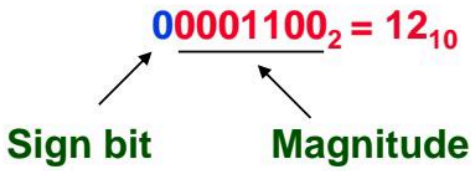
Decimal	Binary	Octal	Hexadecimal	BCD
0	0000	0	0	000
1	0001	1	1	0001
2	0010	2	2	0010
3	0011	3	3	0011
4	0100	4	4	0100
5	0101	5	5	0101
6	0110	6	6	0110
7	0111	7	7	0111
8	1000	10	8	1000
9	1001	11	9	1001
10	1010	12	A	0001 0000
11	1011	13	B	0001 0001
12	1100	14	C	0001 0010
13	1101	15	D	0001 0011
14	1110	16	E	0001 0100
15	1111	17	F	0001 0101

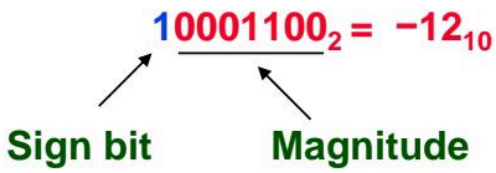
How To Represent Signed Numbers

- Plus and minus signs are used for decimal numbers:
  - 25 (or +25), −16, etc
- In computers, everything is represented as *bits*
- Three types of signed binary number representations:
  - signed magnitude
  - 1's complement
  - 2's complement
- In each case: left-most bit indicates the sign: '0' for positive and '1' for negative

Signed Magnitude Representation

- **The left most bit is designated as the *sign* bit while the remaining bits form the *magnitude***





One's Complement Representation

- The one's complement of a binary number is done by complementing (i.e. inverting) all bits

  1's comp of 00110011 is 11001100

  1's comp of 10101010 is 01010101

- For a n-bit number N the 1's complement is $(2^n - 1) - N$
- Called "*diminished radix complement*" by Mano
- To find the negative of a 1's complement number take its 1's complement

One's Complement Representation

4 bits

⇓

16 combinations

7	0111
6	0110
.	.
.	.
1	0001
0	0000
− 0	1111
− 1	1110
.	.
.	.
− 6	1001
− 7	1000

Two's Complement Representation

- **The two's complement of a binary number is done by complementing (inverting) all bits then adding 1**

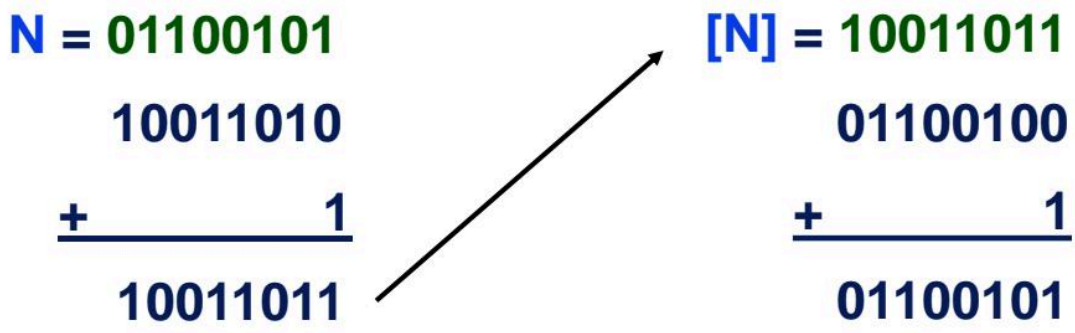
  2's comp of 00110011 is 11001101

  2's comp of 10101010 is 01010110

- **For an n-bit number N the 2's complement is $(2^n - 1) - N + 1$**
- **Called "*radix complement*" by Mano**
- **To find the negative of a 2's complement number take its 2's complement**

Two's Complement Shortcuts

- ***Algorithm 1***: **Complement each bit then add 1 to the result**



- ***Algorithm 2***: **Starting with the least significant bit, copy all of the bits up to and including the first '1' bit, then complement the remaining bits**

N = 0 1 1 0 0 1 1 0

[N] = 1 0 0 1 1 0 1 0

Two's Complement Representation

4 bits

⇓

16 combinations

7	0111
6	0110
.	.
2	0010
1	0001
0	0000
− 1	1111
− 2	1110
.	.
.	.
− 7	1001
− 8	1000

Putting All Together

Decimal	Signed-2's Complement	Signed-1's Complement	Signed Magnitude
+7	0111	0111	0111
+6	0110	0110	0110
+5	0101	0101	0101
+4	0100	0100	0100
+3	0011	0011	0011
+2	0010	0010	0010
+1	0001	0001	0001
+0	0000	0000	0000
−0	—	1111	1000
−1	1111	1110	1001
−2	1110	1101	1010
−3	1101	1100	1011
−4	1100	1011	1100
−5	1011	1010	1101
−6	1010	1001	1110
−7	1001	1000	1111
−8	1000	—	—

Finite-Precision Number Representation

- Machines that use 2's complement arithmetic can represent integers in the range

$$-2^{n-1} \le N \le 2^{n-1} - 1$$

n is the number of bits used for representing N

Note that $2^{n-1} - 1 = (011..11)_2$ and $-2^{n-1} = (100..00)_2$

- 2's complement code has more negative numbers than positive
- 1's complement code has 2 representations for zero
- For an n-bit number in base (i.e. radix) z there are z^n different unsigned values (combinations)

$$(0, 1, \ldots z^{n-1})$$

2's Complement Addition

- **Using 2's complement representation, adding numbers is easy**

  Step 1: Add binary numbers

  Step 2: Ignore the resulting carry bit

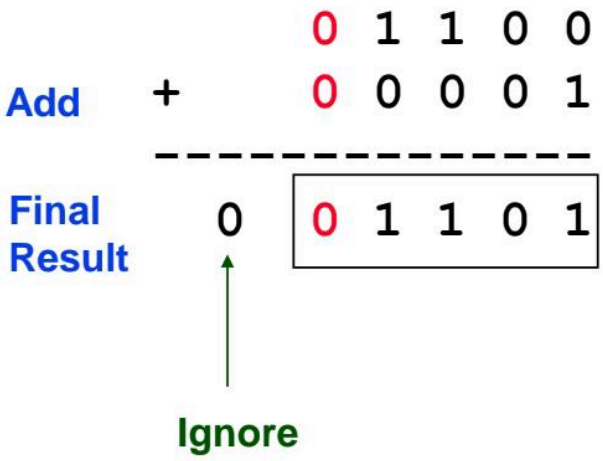
- **For example: $(12)_{10} + (1)_{10}$**

  $(12)_{10} = +(1100)_2$

  $= 01100_2$ *in 2's comp.*

  $(1)_{10} = +(0001)_2$

  $= 00001_2$ *in 2's comp.*



2's Complement Subtraction

- **Using 2's complement representation, subtracting numbers is also easy**

  Step 1: Take 2's complement of 2nd operand

  Step 2: Add binary numbers

  Step 3: Ignore the resulting carry bit

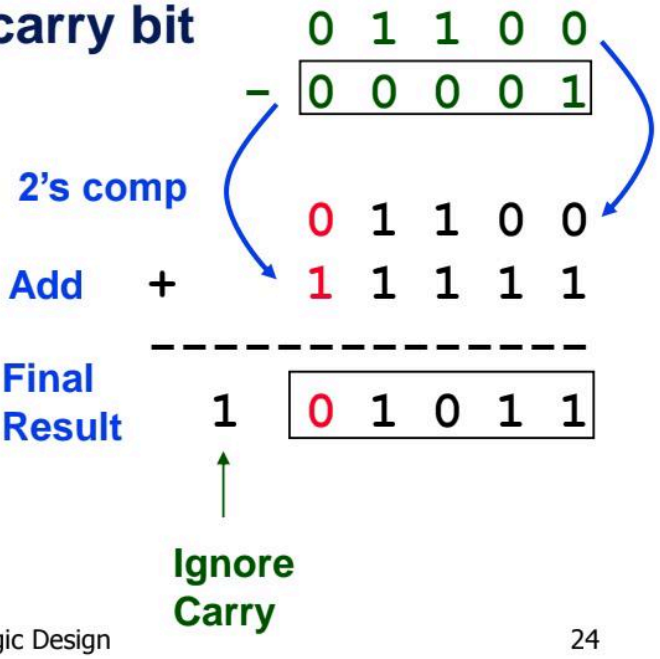
- **For example: $(12)_{10} - (1)_{10}$**

  $(12)_{10} = +(1100)_2$

  $= 01100_2$ *in 2's comp.*

  $(-1)_{10} = -(0001)_2$

  $= 11111_2$ *in 2's comp.*



2's Complement Subtraction (Cont'd)

- **Example 2: $(13)_{10} - (5)_{10}$**

  $(13)_{10} = +(1101)_2 = (01101)_2$

  $(-5)_{10} = -(0101)_2 = (11011)_2$

- **Adding these two 5-bit codes:**

$$\begin{array}{rr} & 0\ 1\ 1\ 0\ 1 \\ + & 1\ 1\ 0\ 1\ 1 \\ \hline \text{Carry} \longrightarrow & \boxed{1}\ 0\ 1\ 0\ 0\ 0 \end{array}$$

- **Discarding the carry bit, the sign bit is seen to be zero, indicating a positive result**

  Indeed: $(01000)_2 = +(8)_{10}$

2's Complement Subtraction (Cont'd)

- **Example 3: $(5)_{10} - (12)_{10}$**

  $(5)_{10} = +(0101)_2 = (00101)_2$

  $(-12)_{10} = -(1100)_2 = (10100)_2$

- **Adding these two 5-bit codes:**

$$\begin{array}{rl} & 0\ 0\ 1\ 0\ 1 \\ + & 1\ 0\ 1\ 0\ 0 \\ \hline \text{Carry} \longrightarrow & \boxed{0}\ 1\ 1\ 0\ 0\ 1 \end{array}$$

- **Here, there is no carry bit and the sign bit is 1. This indicates a negative result, which is what we expect: $(11001)_2 = -(7)_{10}$**

Gray Code

- Gray code is not a number system

  It is an alternate way to represent four bit data

- Only one bit changes from one decimal digit to the next
- Useful for reducing errors in communication
- Can be scaled to larger numbers

Digit	Binary	Gray Code
0	0000	0000
1	0001	0001
2	0010	0011
3	0011	0010
4	0100	0110
5	0101	0111
6	0110	0101
7	0111	0100
8	1000	1100
9	1001	1101
10	1010	1111
11	1011	1110
12	1100	1010
13	1101	1011
14	1110	1001
15	1111	1000

Text: ASCII Characters

- ASCII: Maps 128 characters to 7-bit code.
  - both printable and non-printable (ESC, DEL, …) characters
    http://www.asciitable.com/

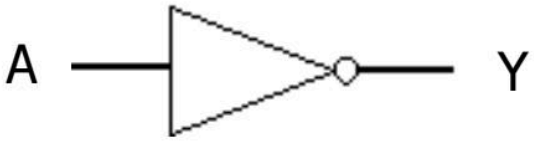
00	nul	10	dle	20	sp	30	0	40	@	50	P	60	`	70	p
01	soh	11	dc1	21	!	31	1	41	A	51	Q	61	a	71	q
02	stx	12	dc2	22	"	32	2	42	B	52	R	62	b	72	r
03	etx	13	dc3	23	#	33	3	43	C	53	S	63	c	73	s
04	eot	14	dc4	24	$	34	4	44	D	54	T	64	d	74	t
05	enq	15	nak	25	%	35	5	45	E	55	U	65	e	75	u
06	ack	16	syn	26	&	36	6	46	F	56	V	66	f	76	v
07	bel	17	etb	27	'	37	7	47	G	57	W	67	g	77	w
08	bs	18	can	28	(	38	8	48	H	58	X	68	h	78	x
09	ht	19	em	29	)	39	9	49	I	59	Y	69	i	79	y
0a	nl	1a	sub	2a	*	3a	:	4a	J	5a	Z	6a	j	7a	z
0b	vt	1b	esc	2b	+	3b	;	4b	K	5b	[	6b	k	7b	{
0c	np	1c	fs	2c	,	3c	<	4c	L	5c	\	6c	l	7c	|
0d	cr	1d	gs	2d	-	3d	=	4d	M	5d	]	6d	m	7d	}
0e	so	1e	rs	2e	.	3e	>	4e	N	5e	^	6e	n	7e	~
0f	si	1f	us	2f	/	3f	?	4f	O	5f	_	6f	o	7f	del

COE211: Digital Logic Design

Boolean Algebra and Logic Gates

Describing Circuit Functionality: Inverter

- **Basic logic functions have symbols**
- **The same functionality can be represented with a truth table**
  - **Truth table completely specifies outputs for all input combinations**
- **This is an inverter**
  - **An input of 0 is inverted to a 1**
  - **An input of 1 is inverted to a 0**



Symbol

Truth Table

A	Y
0	1
1	0

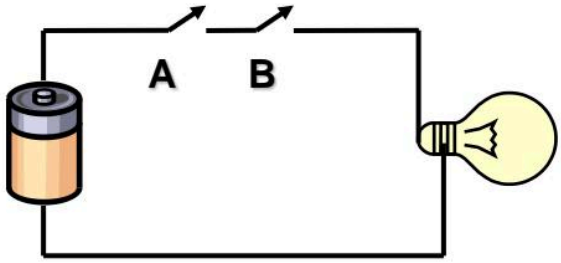
Input **Output**

The AND Gate

- This is an AND gate
- If the two input signals are *asserted* (i.e. high) the output will also be asserted. Otherwise, the output will be *deasserted* (i.e. low)

Truth Table

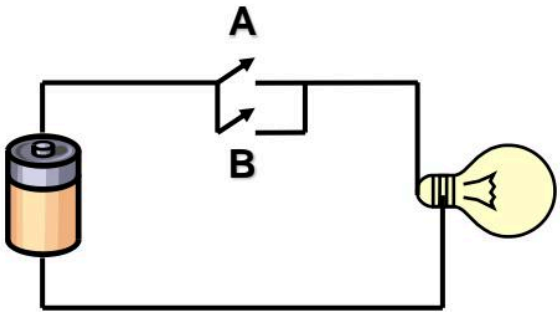
A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1



The OR Gate

- This is an OR gate
- If either of the two input signals is *asserted*, or both of them are, the output will be *asserted*

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1



The NAND Gate

- The NAND gate is a combination of an AND gate followed by an inverter
  - NAND(A, B) → (A AND B)'

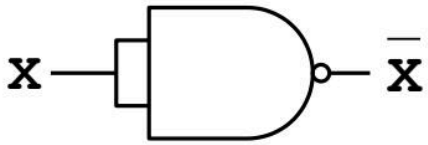
$$Y = \overline{A \cdot B}$$

A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

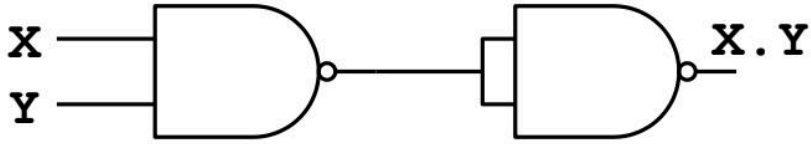
The Universal Property of NAND

- You can implement any function using: NOT, AND, and OR. They represent a **logically complete** set.
- You can use only NAND gates to implement the above three gates. Therefore, NAND alone is a **logically complete** set.

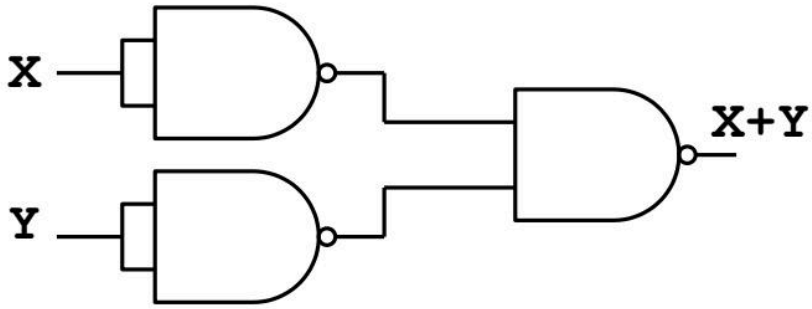
NOT



AND



OR



The NOR Gate

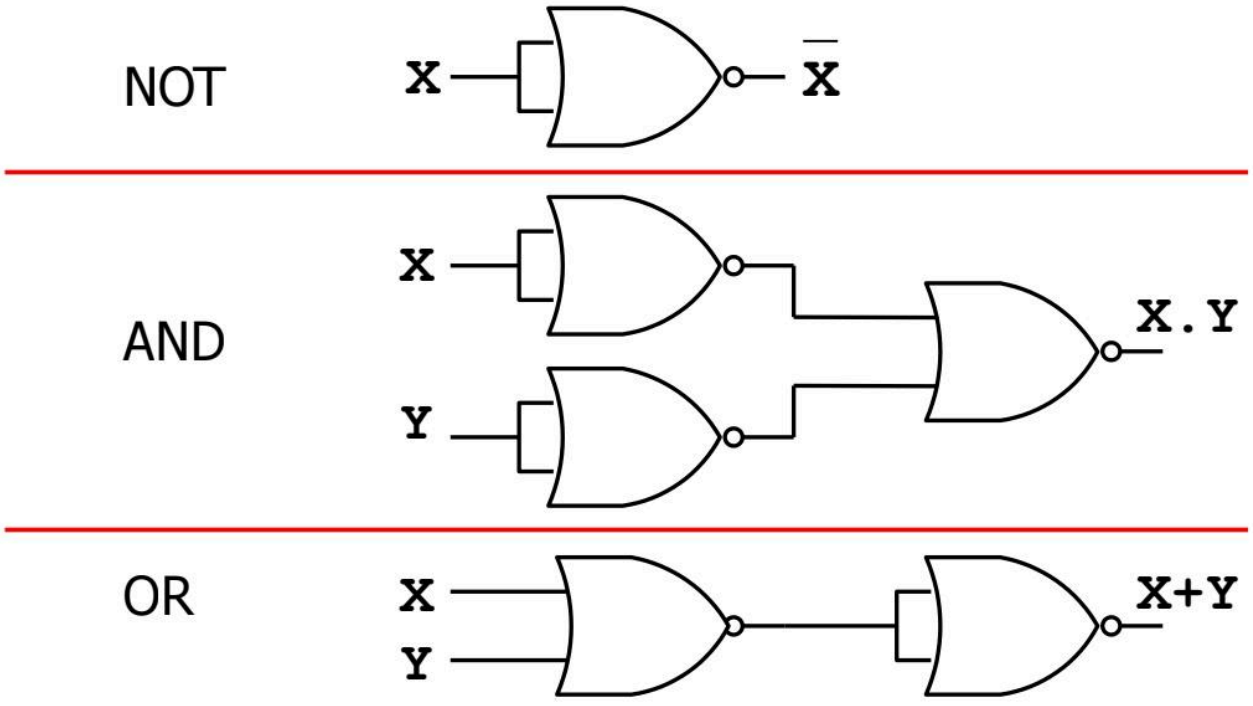
- A NOR gate is a combination of an OR gate followed by an inverter
  - NOR(A , B) → (A+B)'

$$Y = \overline{A + B}$$

A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0

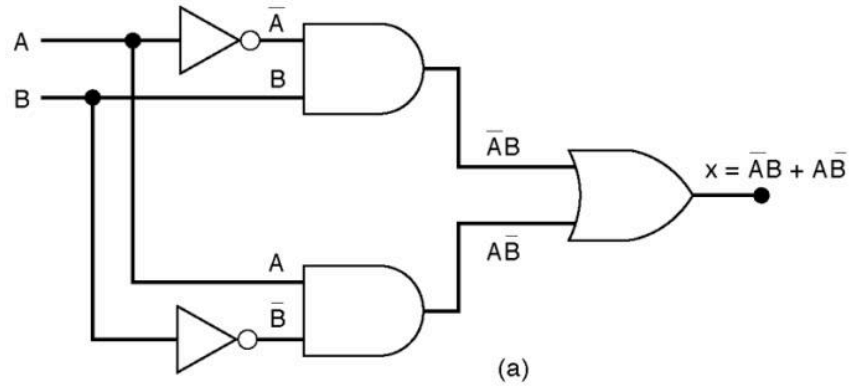
The Universal Property of NOR

- Similarly, you can use only NOR gates to implement NOT, AND, and OR. Therefore, NOR alone is a **logically complete** set.



Exclusive-OR and Exclusive-NOR Circuits

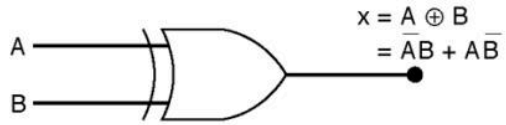
Exclusive-OR (XOR) produces a HIGH output whenever the two inputs are at opposite levels



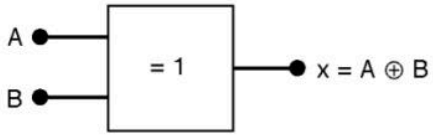
(a)

A	B	x
0	0	0
0	1	1
1	0	1
1	1	0

XOR gate symbols



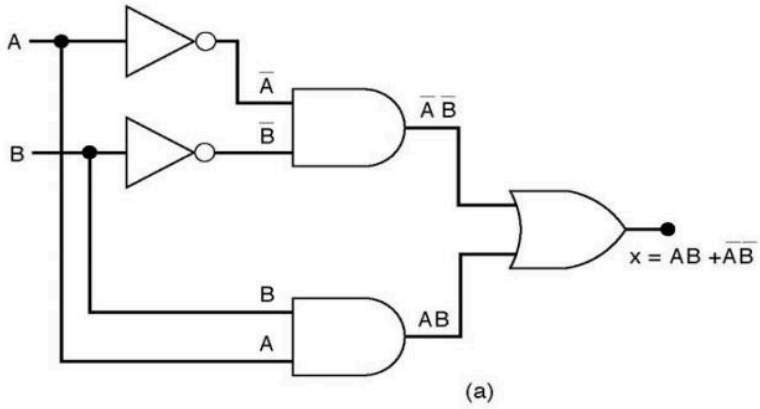
(b)



(c)

Exclusive-NOR Circuits

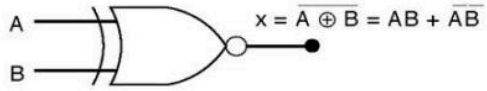
Exclusive-NOR (XNOR) produces a HIGH output whenever the two inputs are at the same level



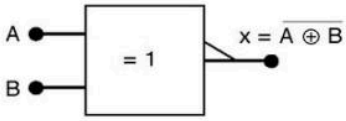
A	B	x
0	0	1
0	1	0
1	0	0
1	1	1

(a)

XNOR gate symbols



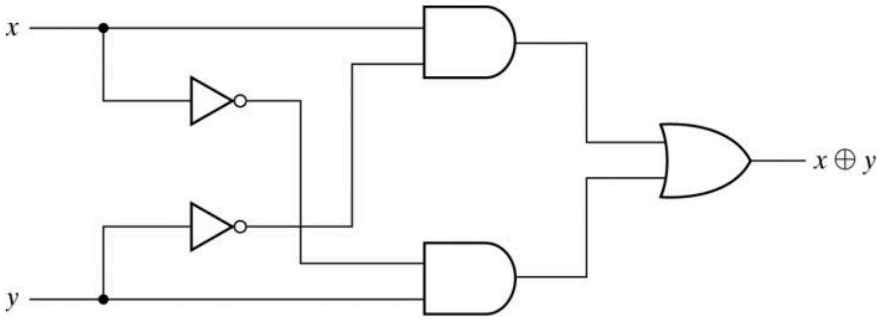
(b)



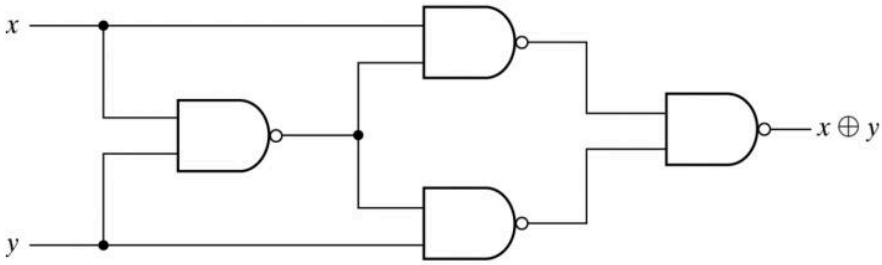
(c)

XOR Function

XOR function can also be implemented with AND/OR gates (also NANDs)



(a) With AND-OR-NOT gates

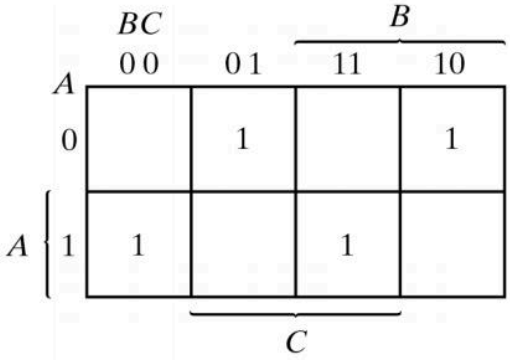


(b) With NAND gates

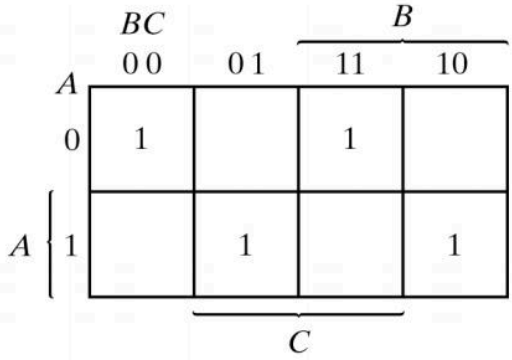
Fig. 3-32 Exclusive-OR Implementations

XOR Function

- Even function – even number of inputs are 1
- Odd function – odd number of inputs are 1



(a) Odd function
$F = A \oplus B \oplus C$

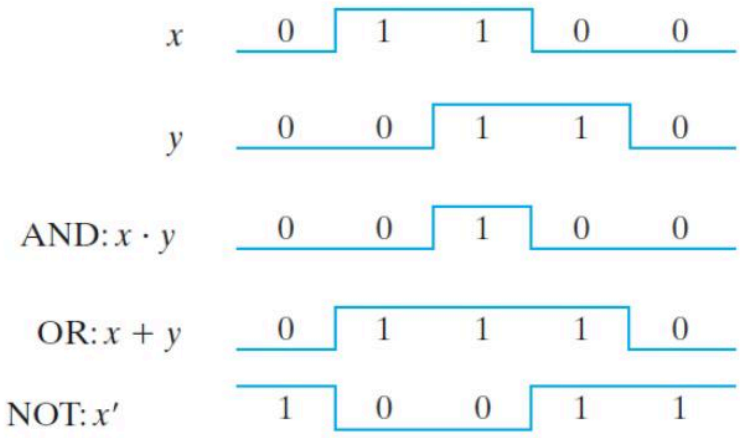


(a) Even function
$F = (A \oplus B \oplus C)'$

Fig. 3-33 Map for a Three-variable Exclusive-OR Function

Describing Circuit Functionality: Waveforms

- **Waveforms provide another approach for representing functionality**
- **Values are either high (logic 1) or low (logic 0)**
- **Can you create a truth table from the waveforms?**

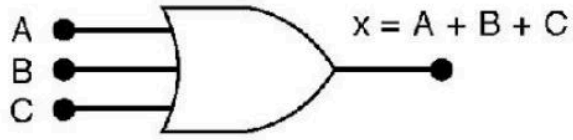


AND Gate

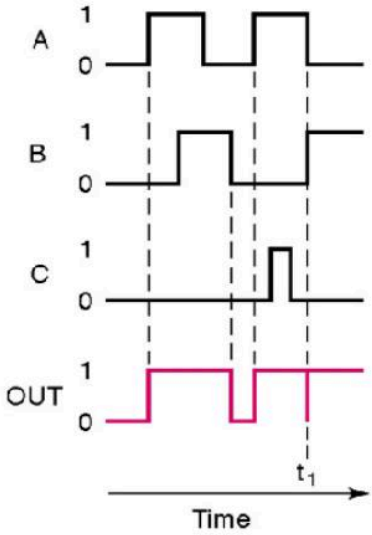
x	y	f
0	0	0
0	1	0
1	0	0
1	1	1

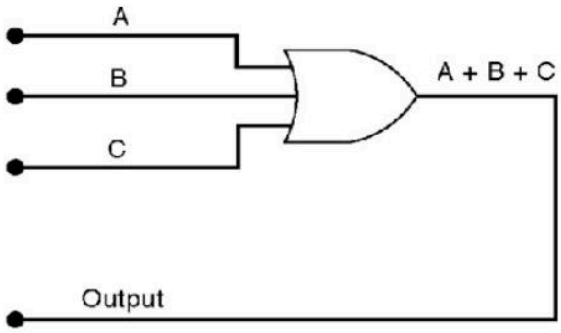
Consider Three-input Gates

3 Input OR Gate



A	B	C	x = A + B + C
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1





Boolean Algebra

- Useful for identifying and *minimizing* circuit functionality
- Identity elements
  - a + 0 = a
  - a • 1 = a
  - 0 is the identity element for the + operation
  - 1 is the identity element for the • operation
- The Complement: for every element 'a', there exists a unique element called a' (or ā) (complement of a) such that :
  - a + a' = 1
  - a • a' = 0

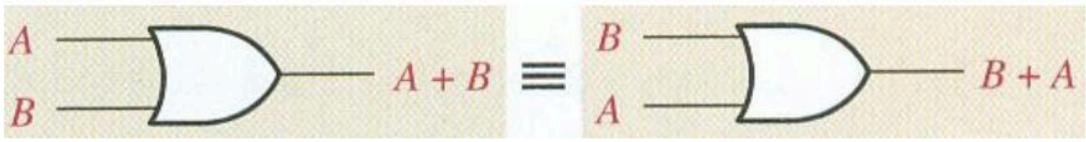
George Boole (1815 - 1864)

- Father of Boolean algebra
- Boole's system (*detailed in his 'An Investigation of the Laws of Thought, on Which Are Founded the Mathematical Theories of Logic and Probabilities', 1854*) was based on a binary approach, processing only two objects - the yes-no, true-false, on-off, zero-one approach.
- Surprisingly, given his standing in the academic community, Boole's idea was either criticized or completely ignored by the majority of his peers.
- Eventually, one bright student, **Claude Shannon** (1916-2001), picked up the idea and ran with it.

Laws of Boolean Algebra

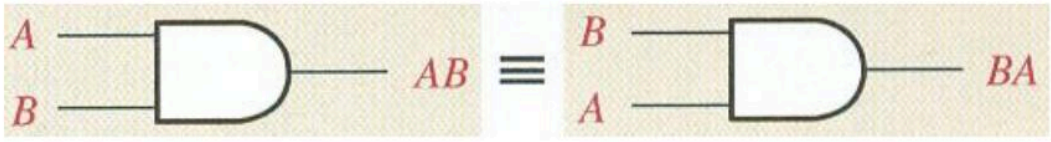
- Commutative Law of ORing:

$$A + B = B + A$$



- Commutative Law of ANDing:

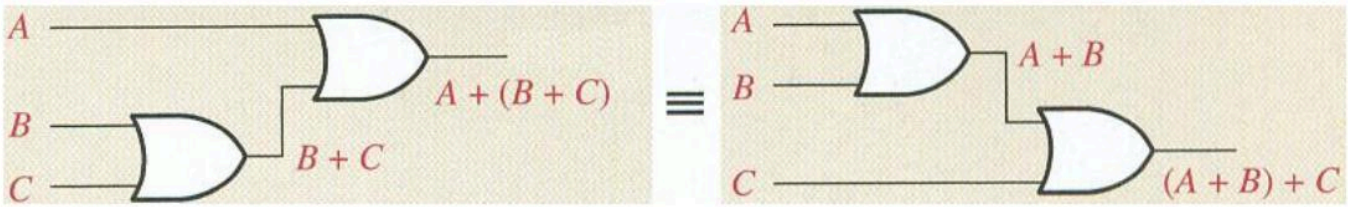
$$A \cdot B = B \cdot A$$



Laws of Boolean Algebra (Cont'd)

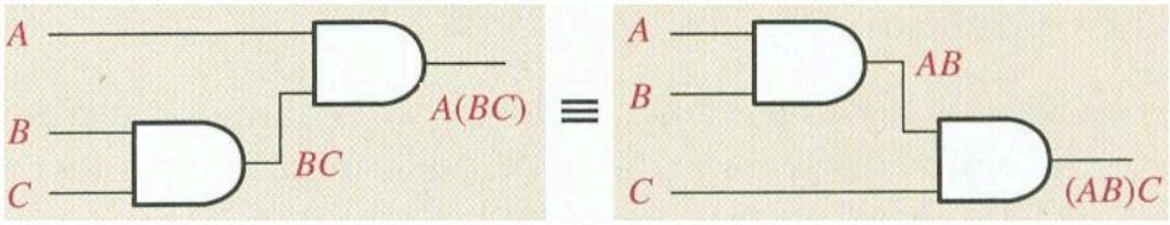
- Associative Law of ORing:

$$A + (B + C) = (A + B) + C$$



- Associative Law of ANDing:

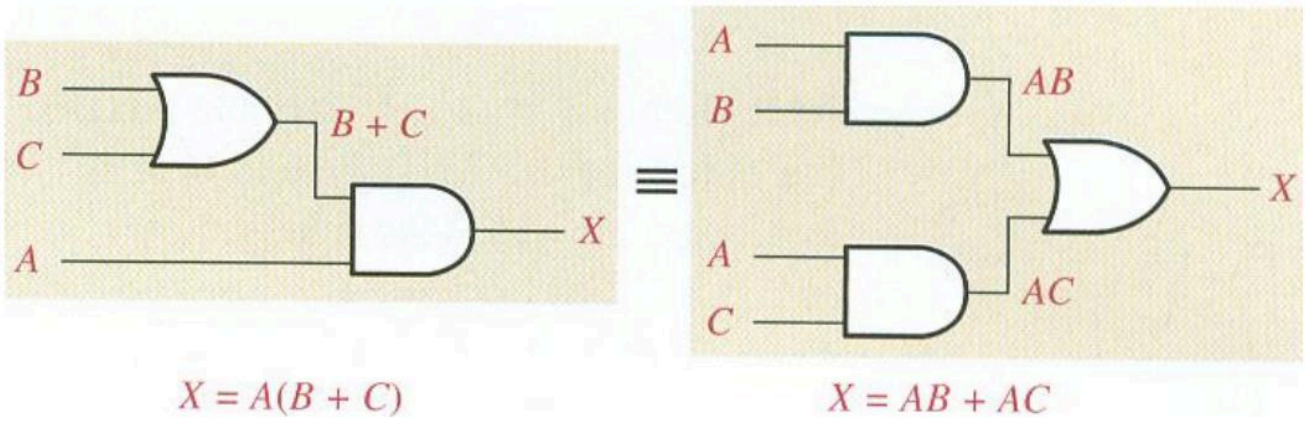
$$A \cdot (B \cdot C) = (A \cdot B) \cdot C$$



Laws of Boolean Algebra (Cont'd)

- Distributive Law:
  - *Note:* To simplify notation, the • operator is frequently omitted. When two elements are written next to each other, the AND (•) operator is implied

$$A(B + C) = AB + AC$$

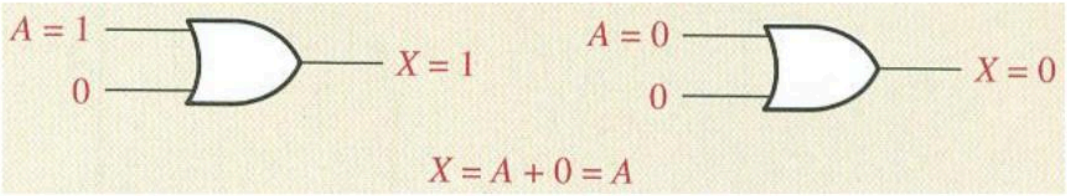


Rules of Boolean Algebra

1. $A + 0 = A$	**7.** $A \cdot A = A$
2. $A + 1 = 1$	**8.** $A \cdot \overline{A} = 0$
3. $A \cdot 0 = 0$	**9.** $\overline{\overline{A}} = A$
4. $A \cdot 1 = A$	**10.** $A + AB = A$
5. $A + A = A$	**11.** $A + \overline{A}B = A + B$
6. $A + \overline{A} = 1$	**12.** $(A + B)(A + C) = A + BC$

Rules of Boolean Algebra Cont'd

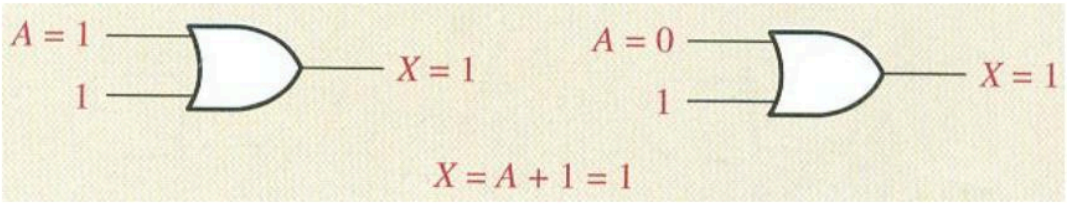
- Rule 1



A	B	X
0	0	0
0	1	1
1	0	1
1	1	1

OR Truth Table

- Rule 2

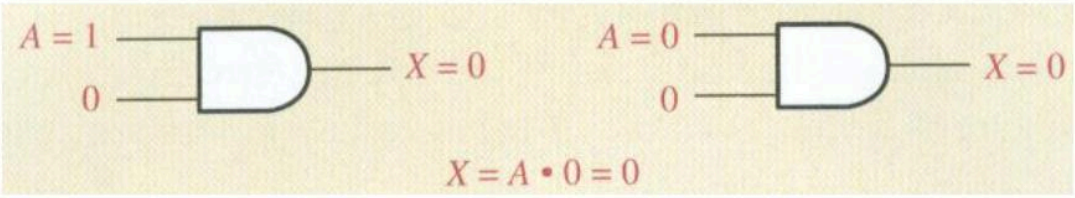


A	B	X
0	0	0
0	1	1
1	0	1
1	1	1

OR Truth Table

Rules of Boolean Algebra Cont'd

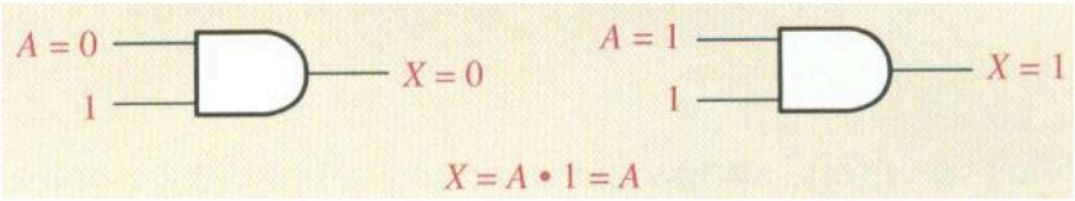
- Rule 3



A	B	X
0	0	0
0	1	0
1	0	0
1	1	1

AND Truth Table

- Rule 4

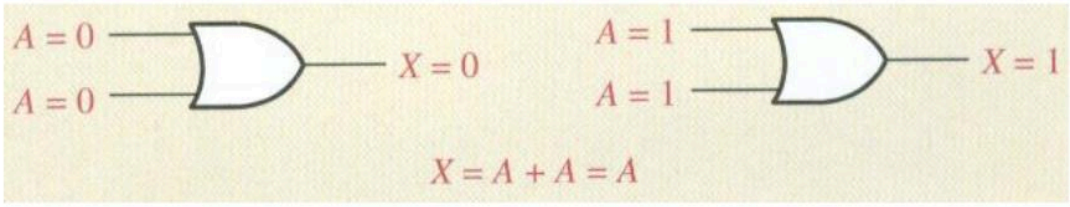


A	B	X
0	0	0
0	1	0
1	0	0
1	1	1

AND Truth Table

Rules of Boolean Algebra Cont'd

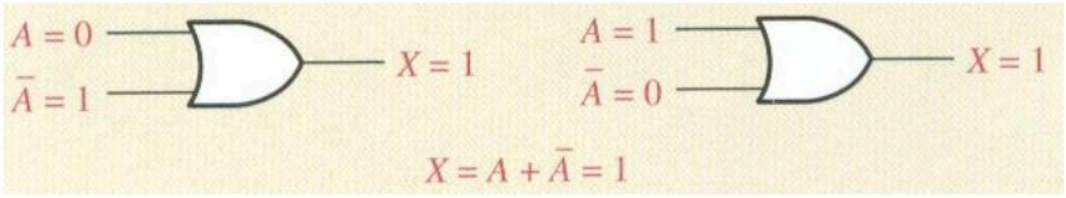
- Rule 5



A	B	X
0	0	0
0	1	1
1	0	1
1	1	1

OR Truth Table

- Rule 6

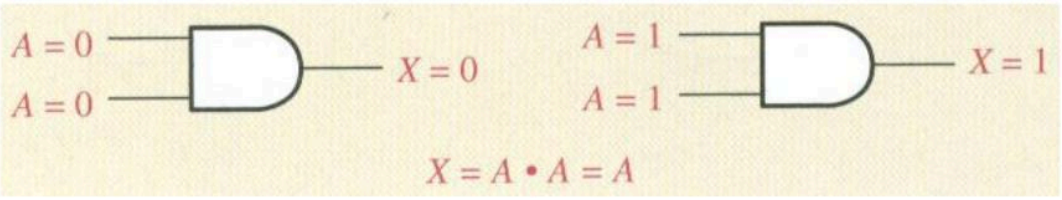


A	B	X
0	0	0
0	1	1
1	0	1
1	1	1

OR Truth Table

Rules of Boolean Algebra Cont'd

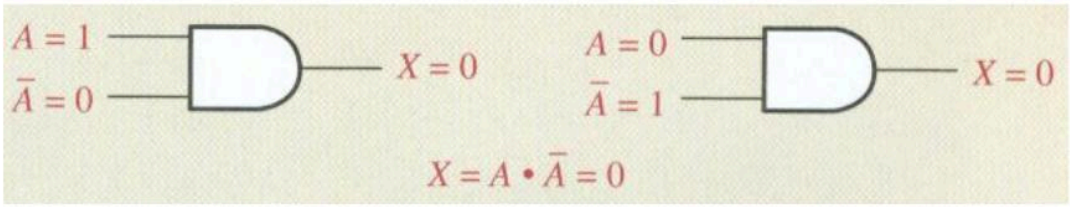
- Rule 7



A	B	X
0	0	0
0	1	0
1	0	0
1	1	1

AND Truth Table

- Rule 8

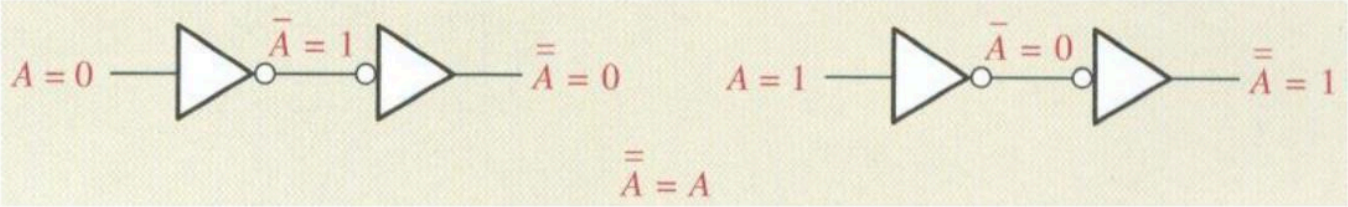


A	B	X
0	0	0
0	1	0
1	0	0
1	1	1

AND Truth Table

Rules of Boolean Algebra Cont'd

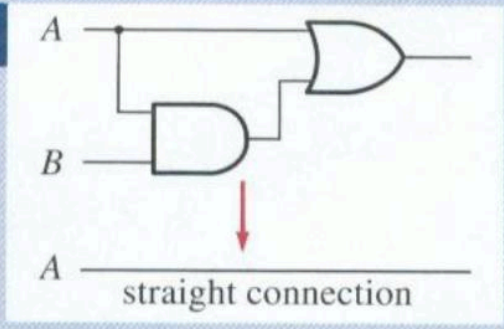
- Rule 9



- Rule 10 (the absorption rule): A + AB = A

A	B	AB	A + AB
0	0	0	0
0	1	0	0
1	0	0	1
1	1	1	1

equal

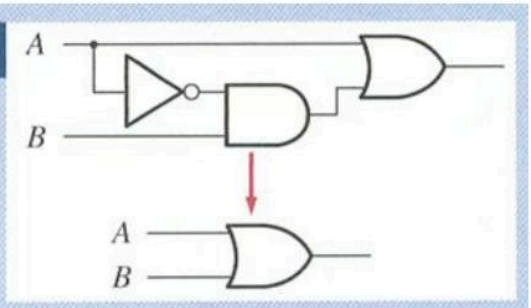


Rules of Boolean Algebra Cont'd

- Rule 11: $A + \overline{A}B = A + B$

A	B	$\overline{A}B$	$A + \overline{A}B$	A + B
0	0	0	0	0
0	1	1	1	1
1	0	0	1	1
1	1	0	1	1

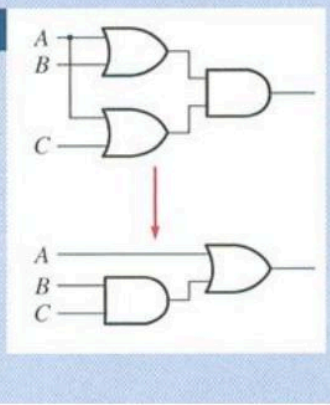
equal



- Rule 12: (A + B)(A + C) = A + BC

A	B	C	A + B	A + C	(A + B)(A + C)	BC	A + BC
0	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0
0	1	0	1	0	0	0	0
0	1	1	1	1	1	1	1
1	0	0	1	1	1	0	1
1	0	1	1	1	1	0	1
1	1	0	1	1	1	0	1
1	1	1	1	1	1	1	1

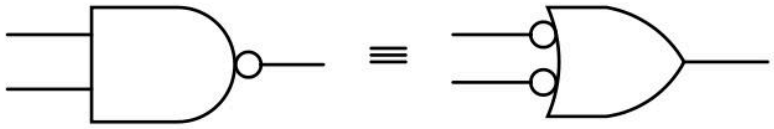
equal



De Morgan's Theorems

(by Augustus De Morgan (1806 - 1871), an English mathematician and logician)

$$\overline{A.B} = \overline{A} + \overline{B}$$



$$\overline{A+B} = \overline{A}.\overline{B}$$

General:

$$\overline{A.B.C.D} = \overline{A} + \overline{B} + \overline{C} + \overline{D}$$

$$\overline{A+B+C+D} = \overline{A}.\overline{B}.\overline{C}.\overline{D}$$

De Morgan's Theorems Example

$$\overline{(A.B+C)\,(A+B.C)}=$$

$$\overline{(A.B+C)}+\overline{(A+B.C)}=$$

$$(\overline{A.B}.\overline{C})+(\overline{A}.\overline{B.C})=$$

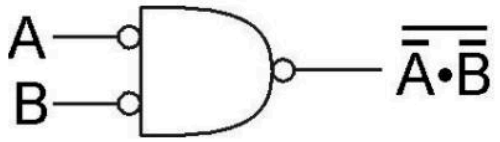
$$(\overline{A}+\overline{B}).\overline{C}+\overline{A}.(\overline{B}+\overline{C})=$$

$$\overline{A}.\overline{C}+\overline{B}.\overline{C}+\overline{A}.\overline{B}+\overline{A}.\overline{C}=$$

$$\overline{A}.\overline{C}+\overline{B}.\overline{C}+\overline{A}.\overline{B}$$

Converting AND to OR

- Using De Morgan's Theorems, AND can be converted to OR (with some help from NOT)
- Consider the following gate:



A	B	$\overline{A}$	$\overline{B}$	$\overline{A} \cdot \overline{B}$	$\overline{\overline{A} \cdot \overline{B}}$
0	0	1	1	1	0
0	1	1	0	0	1
1	0	0	1	0	1
1	1	0	0	0	1

Same as A+B

To convert AND to OR (or vice versa), invert inputs and output.

Standard Forms of Boolean Expressions

- The sum-of-product (SOP) form

  Example: X = AB + CD + EF

- The product of sum (POS) form

  Example: X = (A + B)(C + D)(E + F)

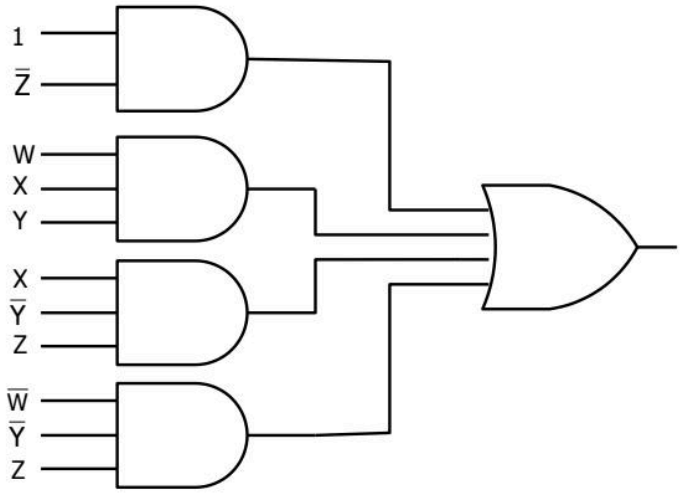
Sum-of-Products Expression

- A *literal* is a variable or the complement of a variable.
  Examples: X, Y, $\overline{X}$, $\overline{Y}$

- A *product term* is a single literal or a logical product of two or more literals. Examples: $\overline{Z}$, $W.X.Y$, $X.\overline{Y}.Z$, $\overline{W}.\overline{Y}.Z$

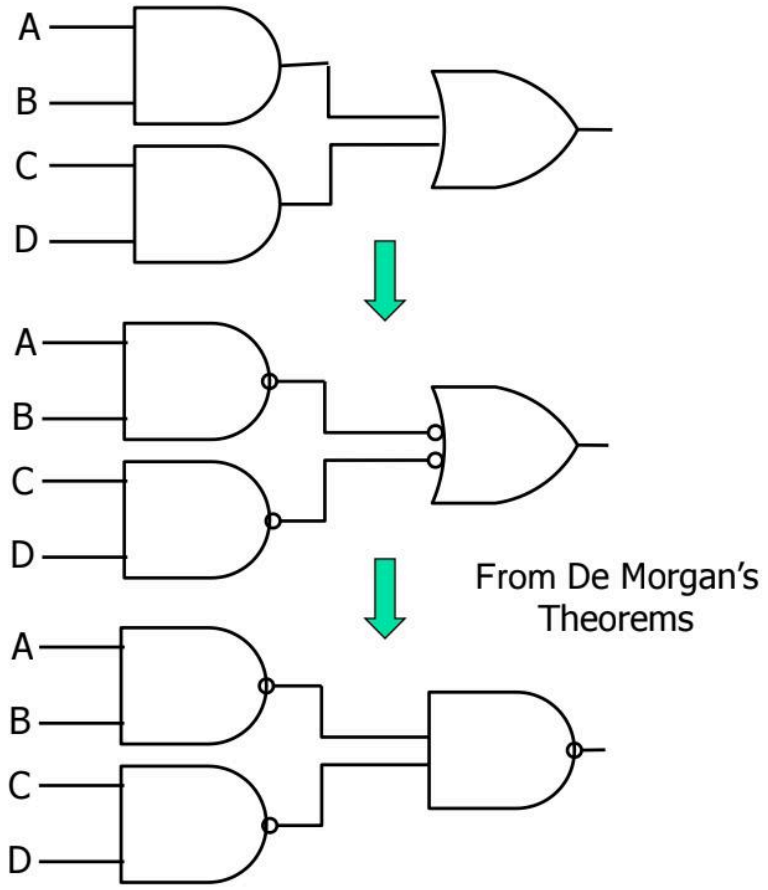
- A *sum-of-products* (SOP) expression is a logical sum of product terms.
  Example: $\overline{Z} + W.X.Y + X.\overline{Y}.Z + \overline{W}.\overline{Y}.Z$

- Every SOP expression can be realized by a two-level circuit containing AND gates followed by an OR gate



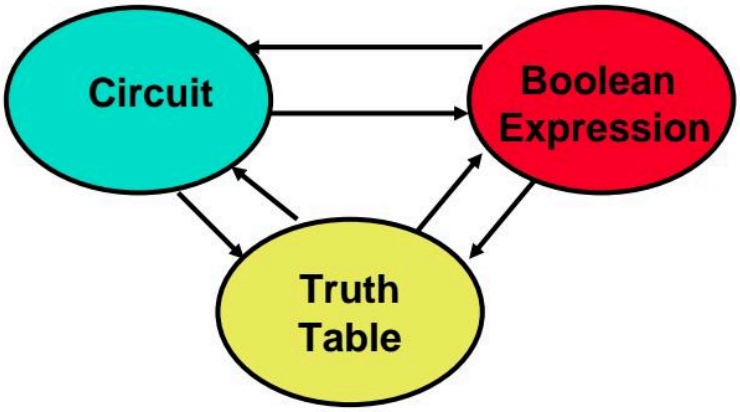
SOP With NANDs

- A SOP expression can be implemented with only NAND gates:



Function Representation Conversion

- **Need to transit between a Boolean expression, a truth table, and a circuit (symbols)**
- **All three formats are equivalent**



Minterm

- For each truth-table row a product term can be defined that evaluates to 1 only when the inputs have the values listed in that row.
- If the product term contains each input variable exactly once it is called a minterm

Row	A	B	C	Minterm
0	0	0	0	$\overline{A}.\overline{B}.\overline{C}$
1	0	0	1	$\overline{A}.\overline{B}.C$
2	0	1	0	$\overline{A}.B.\overline{C}$
3	0	1	1	$\overline{A}.B.C$
4	1	0	0	$A.\overline{B}.\overline{C}$
5	1	0	1	$A.\overline{B}.C$
6	1	1	0	$A.B.\overline{C}$
7	1	1	1	$A.B.C$

From Truth Table to Boolean Expression

- Any logic function can be expressed algebraically by taking the **OR** of all those **minterms** corresponding to the truth-table rows for which the function produces a 1 output.
- *Example:* Majority detector.

Row	A	B	C	Minterm	R
0	0	0	0	$\bar{A}.\bar{B}.\bar{C}$	0
1	0	0	1	$\bar{A}.\bar{B}.C$	0
2	0	1	0	$\bar{A}.B.\bar{C}$	0
3	0	1	1	$\bar{A}.B.C$	1
4	1	0	0	$A.\bar{B}.\bar{C}$	0
5	1	0	1	$A.\bar{B}.C$	1
6	1	1	0	$A.B.\bar{C}$	1
7	1	1	1	$A.B.C$	1

$$R=\bar{A}.B.C+ A.\bar{B}.C+ A.B.\bar{C}+ A.B.C$$

- Alternate forms:

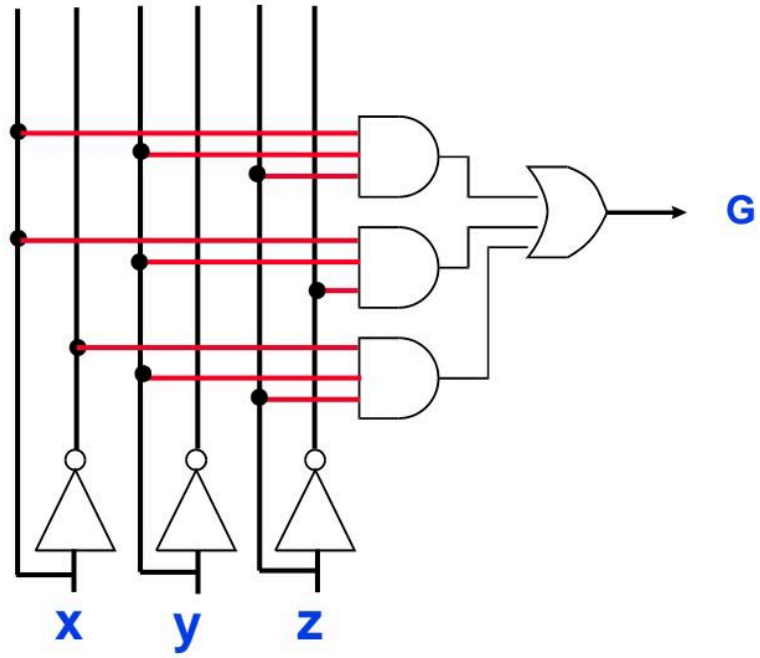
$$R = m_3 + m_5 + m_6 + m_7$$

$$R = \Sigma (3, 5, 6, 7)$$

Converting to a Circuit

- **Number of 1's in truth table output column equals AND terms for Sum-of-Products (SOP)**

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



G = xyz + xyz' + x'yz

Canonical Form of a Function

- A form is *canonical* if representation of a function in this form is unique.

- Examples:
  - Truth table representation of a function.
  - Sum of minterms representation of a function.

Boolean vs. Ordinary Algebra

- Differences
  - Distributivity of + over • holds for Boolean, but not for ordinary algebra

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

  - Boolean algebra does not have inverse elements for + or •

    Thus, no subtraction or division operators

  - Complement is not defined in ordinary algebra

COE211: Digital Logic Design

Simplification of Boolean Functions

Overview

- Introduction to Karnaugh Maps
- Karnaugh Maps Rules and Methods
- Two and Three-Variable K-Maps
- Four-Variable K-Maps
- Simplification Techniques
- Don’t Cares Conditions

Karnaugh Maps

- Alternate way of representing Boolean functions
  - Each row in the truth table is represented by a square
  - Each square represents a *minterm*
- Easy to convert between truth table, K-map, and SOP
  - Un-optimized form: number of 1's in K-map equals number of minterms (products) in SOP
  - Optimized form: reduced number of minterms

$F = \Sigma(m_0, m_1) = x'y + x'y'$

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

x \ y	0	1
0	x'y'	x'y
1	xy'	xy

x \ y	0	1
0	1	1
1	0	0

Karnaugh Maps (cont'd)

- A Karnaugh map is a graphical tool for assisting in the general simplification procedure
- Two variable maps

A \ B	0	1
0	0	1
1	1	0

$F=A\overline{B}+\overline{A}B$

A \ B	0	1
0	0	1
1	1	1

$F=AB+\overline{A}B+A\overline{B}$

- Three variable maps

A \ BC	00	01	11	10
0	0	1	0	1
1	1	1	1	1

A	B	C	F
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

$$F=A\overline{B}\,\overline{C}+A\overline{B}C+ABC+AB\overline{C}+\overline{A}\,\overline{B}C+\overline{A}B\overline{C}$$

Rules for K-Maps

- We can reduce functions by circling 1's in the K-map
- Each circle represents minterm reduction
- After circling, deduce a minimized AND-OR form

Rules to consider

- Every cell containing a 1 must be included at least once
- The largest possible "power of 2 rectangle" should be used
- Use the smallest possible number of rectangles

Karnaugh Maps Examples

- Two variable maps

A \ B	0	1
0	0	1
1	1	0

$F=A\overline{B}+\overline{A}B$

A \ B	0	1
0	0	1
1	1	1

$F=AB+\overline{A}B+A\overline{B}$

$F=A+B$

- Three variable maps

A \ BC	00	01	11	10
0	0	1	0	1
1	1	1	1	1

$F=A+\overline{B}C+B\overline{C}$

$$F=A\overline{B}\overline{C}+A\overline{B}C+ABC+AB\overline{C}+\overline{A}\overline{B}C+\overline{A}B\overline{C}$$

Karnaugh Maps Examples (cont'd)

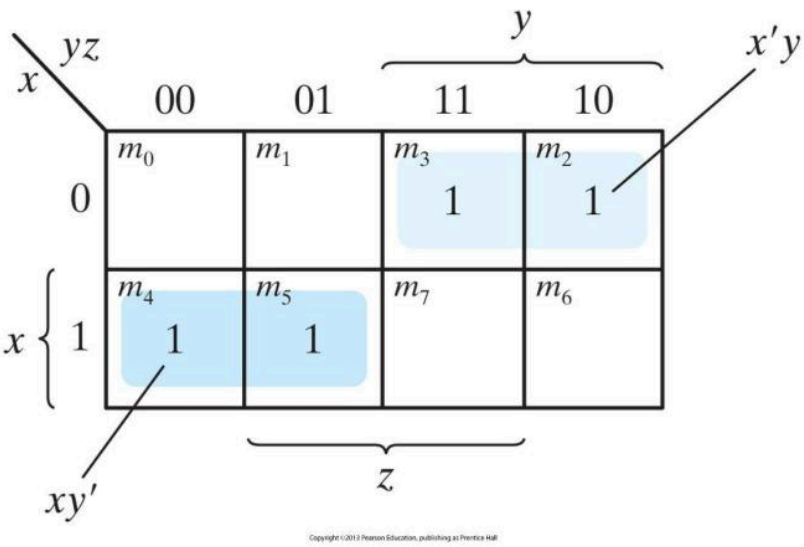
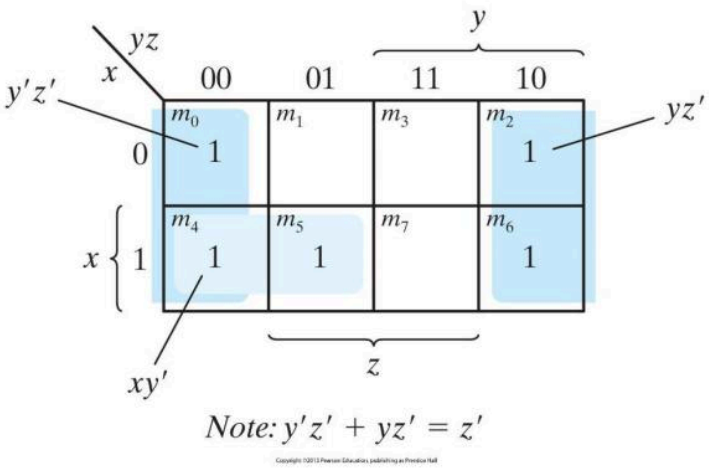


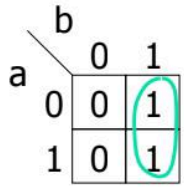
FIGURE 3.4 Map for Example 3.1, $F(x, y, z) = \Sigma(2, 3, 4, 5) = x'y + xy'$



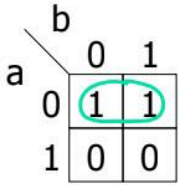
Note: $y'z' + yz' = z'$

FIGURE 3.6 Map for Example 3.3, $F(x, y, z) = \Sigma(0, 2, 4, 5, 6) = z' + xy'$

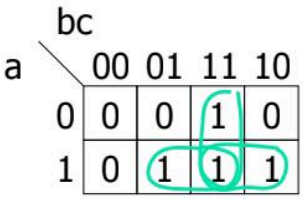
Karnaugh Maps Examples (cont'd)



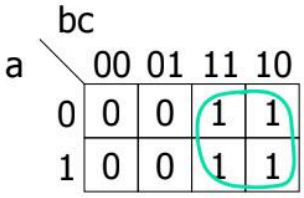
f = b



g = a'



cout = bc + ac + ab

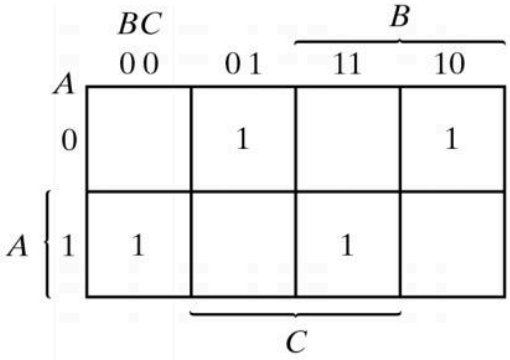


f = b

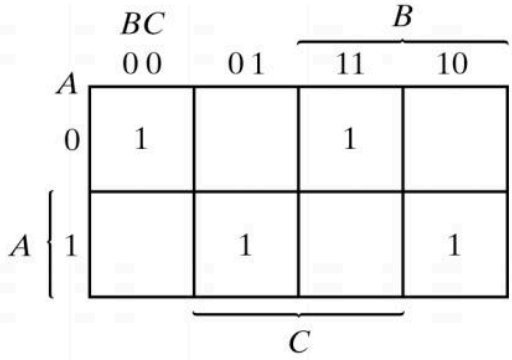
1. Circle the largest groups possible
2. Group dimensions must be a power of 2
3. Remember what circling means!

K-Maps of 3-variable XOR Function

- Odd function – odd number of inputs are 1
- Even function – even number of inputs are 1



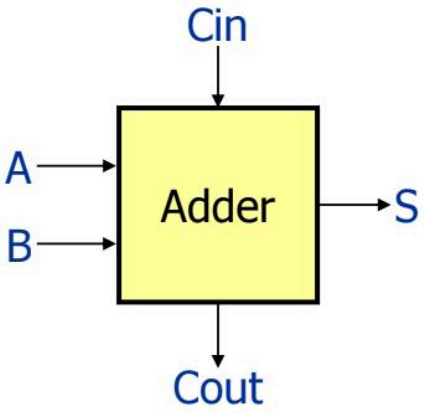
(a) Odd function
$F = A \oplus B \oplus C$



(a) Even function
$F = (A \oplus B \oplus C)'$

Fig. 3-33 Map for a Three-variable Exclusive-OR Function

Example using 1-bit Adder



A	B	Cin	S	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

How to use a Karnaugh Map instead of the Algebraic simplification?

$S = A'B'Cin + A'BCin' + A'BCin + ABCin$

$Cout = A'BCin + A\,B'Cin + ABCin' + ABCin$

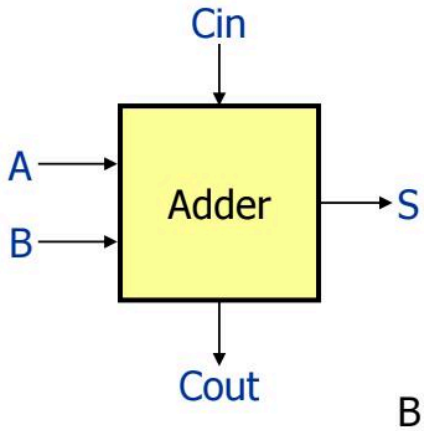
$= A'BCin + ABCin + AB'Cin + ABCin + ABCin' + ABCin$

$= (A' + A)BCin + (B' + B)ACin + (Cin' + Cin)AB$

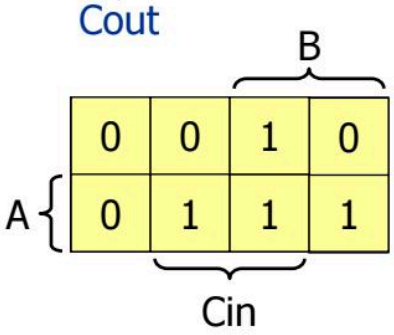
$= 1 \cdot BCin + 1 \cdot ACin + 1 \cdot AB$

$= BCin + ACin + AB$

Example using 1-bit Adder (cont'd)



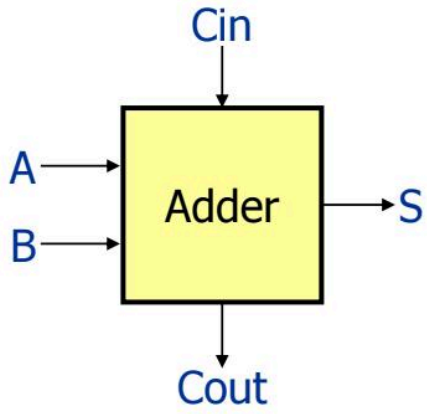
A	B	Cin	S	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1



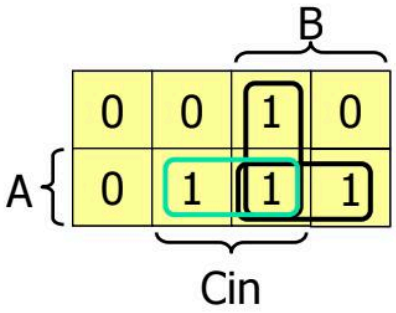
Karnaugh Map for Cout

Now we have to cover all the 1s in the Karnaugh Map using the largest rectangles and as few rectangles as we can.

Example using 1-bit Adder (cont'd)



A	B	Cin	S	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

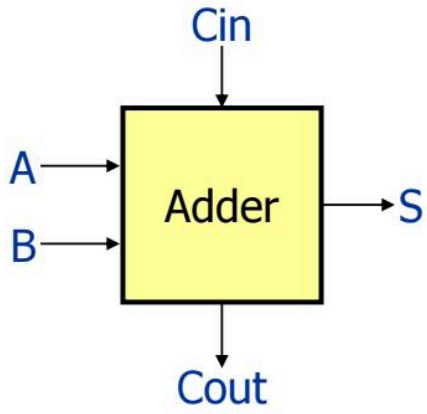


Karnaugh Map for Cout

Now we have to cover all the 1s in the Karnaugh Map using the largest rectangles and as few rectangles as we can.

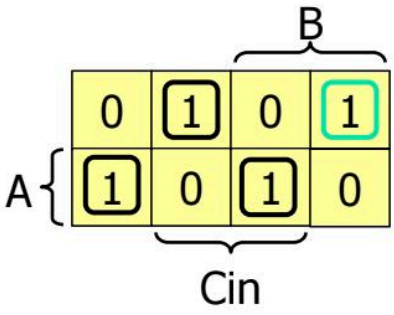
Cout = BCin + AB + ACin

Example using 1-bit Adder (cont'd)



Can you draw the circuit diagrams?

A	B	Cin	S	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1



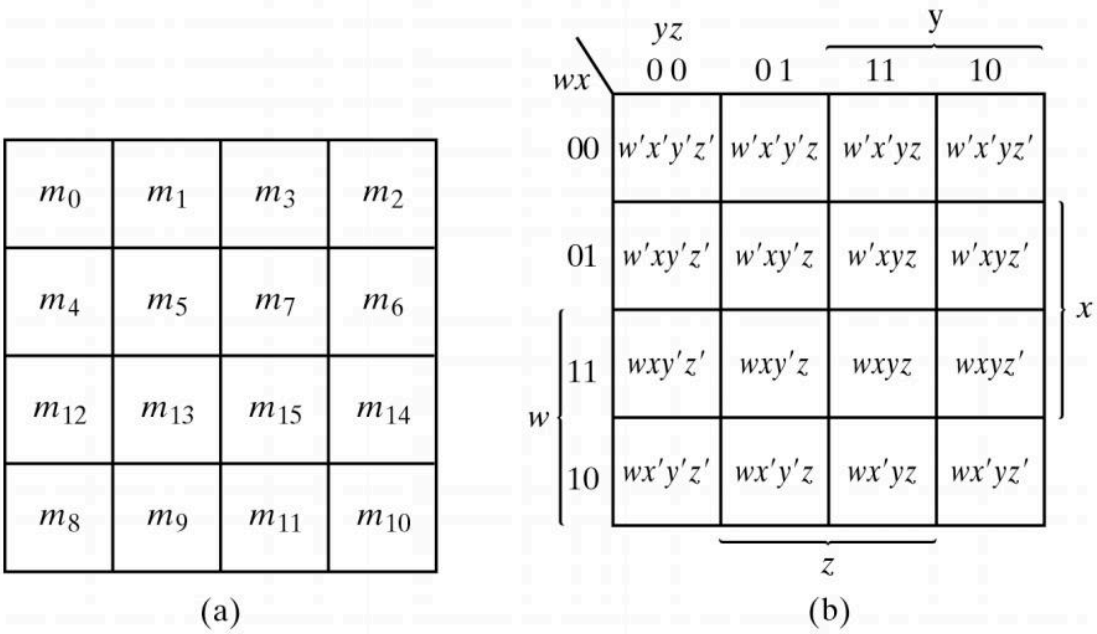
Karnaugh Map for S

$S = A\,\overline{B}\,\overline{Cin} + \overline{A}\,\overline{B}\,Cin + ABCin + \overline{A}B\overline{Cin}$

No Possible Reduction!

Karnaugh Maps for 4-Input Functions

- Represent functions of 4 inputs with 16 minterms
- Use same rules developed for 3-input functions



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

(a)

wx \ yz	00	01	11	10
00	$w'x'y'z'$	$w'x'y'z$	$w'x'yz$	$w'x'yz'$
01	$w'xy'z'$	$w'xy'z$	$w'xyz$	$w'xyz'$
11	$wxy'z'$	$wxy'z$	$wxyz$	$wxyz'$
10	$wx'y'z'$	$wx'y'z$	$wx'yz$	$wx'yz'$

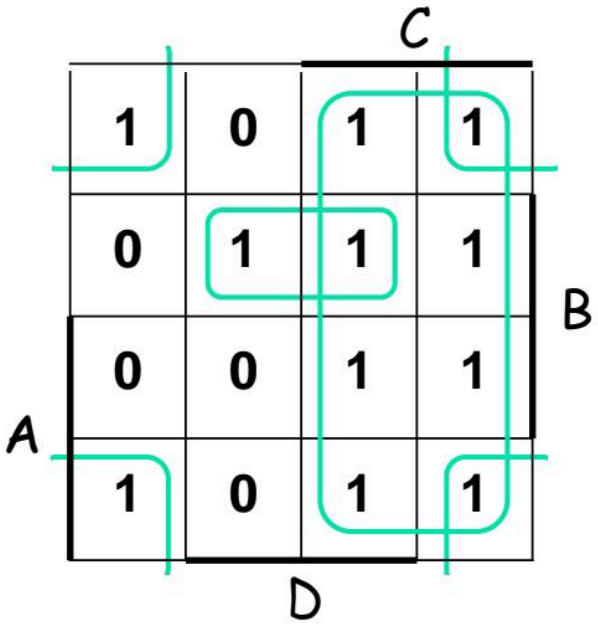
y (11, 10); x (01, 11); w (11, 10); z (00, 01)

(b)

Fig. 3-8 Four-variable Map

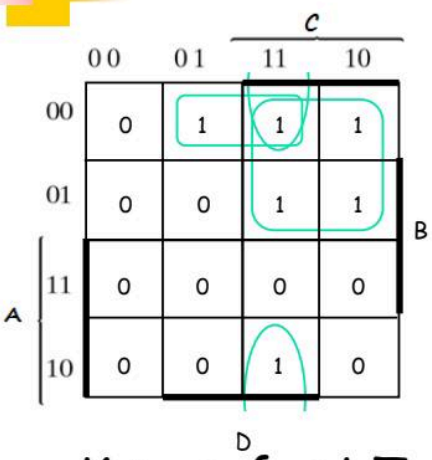
Karnaugh Map: 4-Variable Example

$F(A,B,C,D) = \Sigma m(0, 2, 3, 5, 6, 7, 8, 10, 11, 14, 15)$

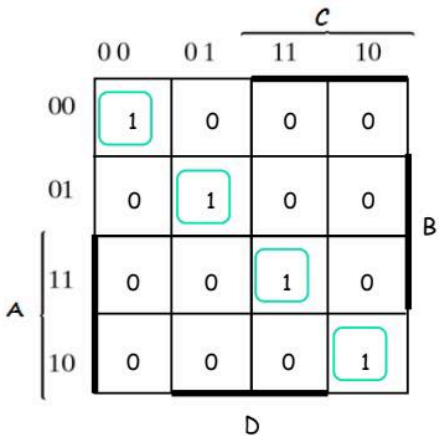


$F = C + A'BD + B'D'$

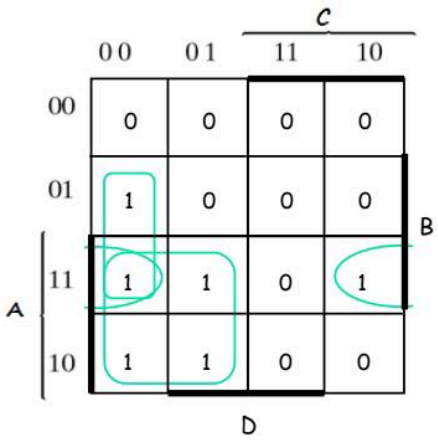
Design Examples



K-map for LT



K-map for EQ



K-map for GT

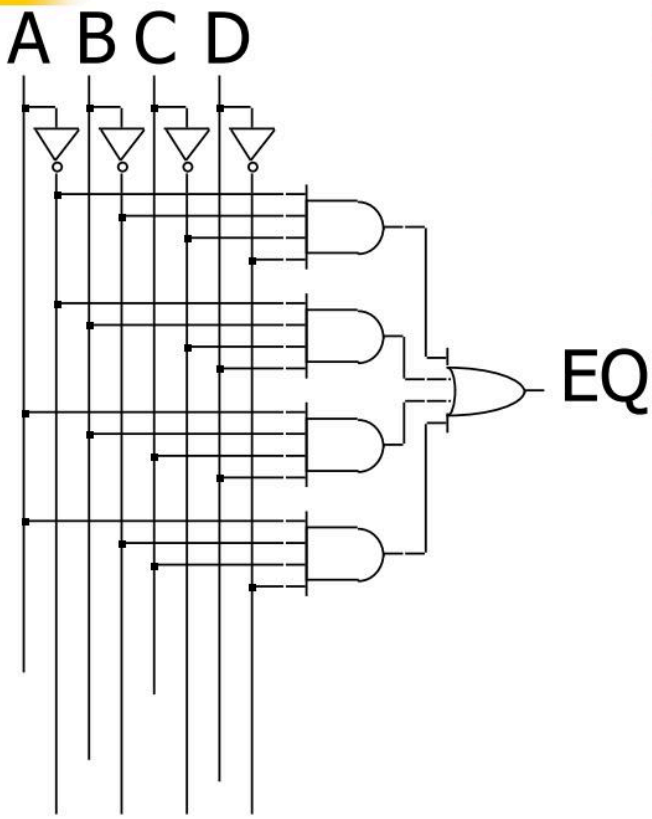
LT = A' C + A' B' D + B' C D

EQ = A'B'C'D' + A'BC'D + ABCD + AB'CD'

GT = A C' + B C' D' + A B D'

Can you draw the truth table for these examples?

Physical Implementation

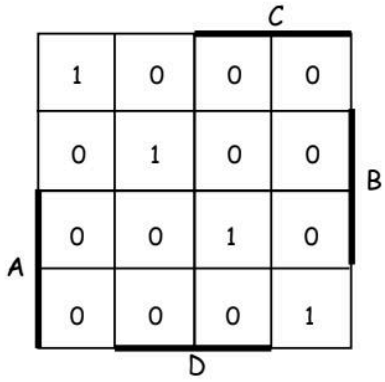


Step 1: Truth table

Step 2: K-map

Step 3: Minimized sum-of-products

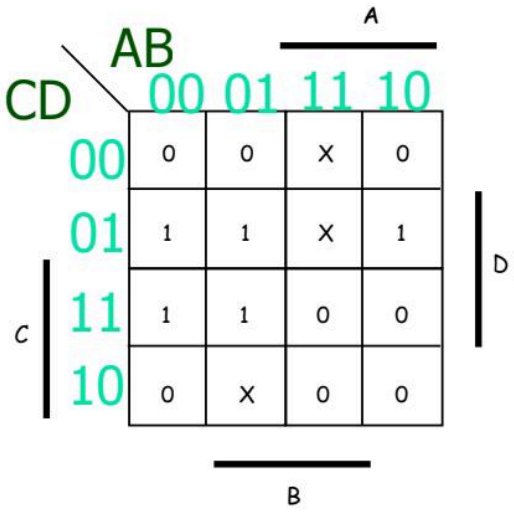
Step 4: Physical implementation with gates



K-map for EQ

Karnaugh Maps: Don't Cares

- In some cases, outputs are undefined
- We "don't care" if the circuit produces a '0' or a '1'
- This knowledge can be used to simplify functions



- Treat X's like either 1's or 0's
- Very useful
- OK to leave some X's uncovered

Don't Care Conditions

- In some situations, we don't care about the value of a function for certain combinations of the variables
  - these combinations may be impossible in certain contexts
  - or the value of the function may not matter when the combinations occur
- In such situations we say the function is incompletely specified and there are multiple (completely specified) logic functions that can be used in the design
  - so we can select a function that gives the simplest circuit
- When constructing the terms in the simplification procedure, we can choose to either cover or not cover the don't care conditions

Don't Cares Examples

AB \ CD	00	01	11	10
00	0	1	0	0
01	x	x	x	1
11	1	1	1	x
10	x	0	1	1

$F=A'C'D+B+AC$

Alternative covering:

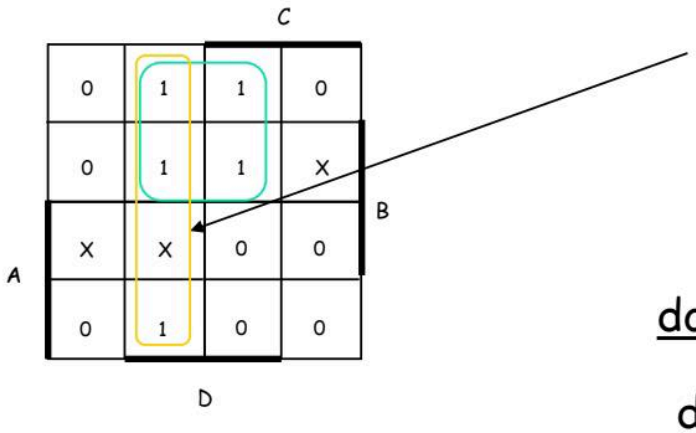
AB \ CD	00	01	11	10
00	0	1	0	0
01	x	x	x	1
11	1	1	1	x
10	x	0	1	1

$F=A'B'C'D+ABC'+BC+AC$

Don’t Cares Examples (cont’d)

$f(A,B,C,D) = \Sigma\, m(1,3,5,7,9) + d(6,12,13)$

- $f = A'D + B'C'D$ without don't cares
- $f = A' D + C' D$ with don't cares



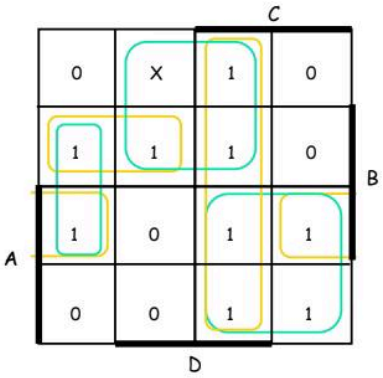
by using don't care as a "1"
a 2-cube can be formed
rather than a 1-cube to cover
this node

don't cares can be treated as
1s or 0s
depending on which is more
advantageous

Definition of Terms

- Implicant
  - Single product term of the ON-set (terms that create a logic 1)
- Prime implicant
  - Implicant that can't be combined with another to form an implicant with fewer literals
- Essential prime implicant
  - Prime implicant is essential if it alone covers a minterm in the K-map
  - Remember that all squares marked with 1 must be covered
- Objective:
  - Grow implicants into prime implicants (minimize literals per term)
  - Cover the K-map with as few prime implicants as possible (minimize number of product terms)

Examples to Illustrate Terms



6 prime implicants:

ABD', A'D, AC, A'BC', CD, BC'D'

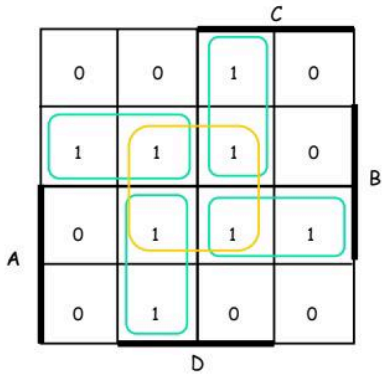
essential

minimum cover: AC + A'D + BC'D'

5 prime implicants:

BD, ABC, AC'D, A'BC', A'CD

essential



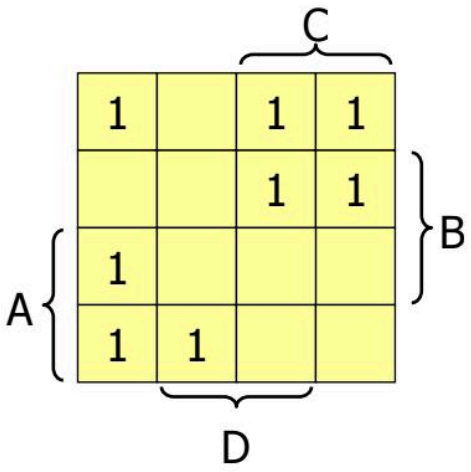
minimum cover: 4 essential implicants

Prime Implicants

Any single 1 or group of 1s in the Karnaugh map of a function F is an implicant of F.

A product term is called a prime implicant of F if it cannot be combined with another term to eliminate a variable.

Example:



If a function F is represented by this Karnaugh Map. Which of the following terms are implicants of F, and which ones are prime implicants of F?

(a) A'B'C
(b) BD
(c) A'B'C'D'
(d) A'C
(e) A'B'D'

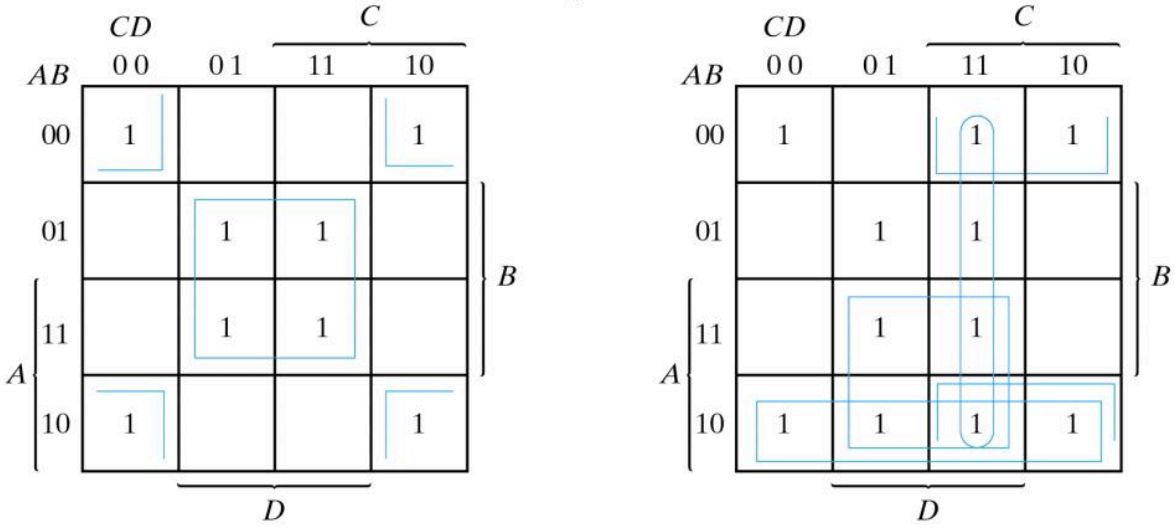
Implicants: (a),(c),(d),(e)

Prime Implicants: (d),(e)

Essential Prime Implicants

A product term is an essential prime implicant if there is a minterm that is only covered by that prime implicant

The minimal sum-of-products form of F must include all the essential prime implicants of F



(a) Essential prime implicants BD and B′D′

(b) Prime implicants CD, B′C AD, and AB′

Fig. 3-11 Simplification Using Prime Implicants

Summary

- Karnaugh map allows us to represent functions with new notation
- Representation allows for logic reduction
  - Implement same function with less logic
- Each square represents one minterm
- Each circle leads to one product term
- Not all functions can be reduced
- K-maps of four literals were considered (Larger examples exist)
- Don't care conditions help minimize functions
- Result of minimization is a minimal sum-of-products
- Result contains prime implicants
- Essential prime implicants are required in the implementation

COE211: Digital Logic Design

Combinational Circuits

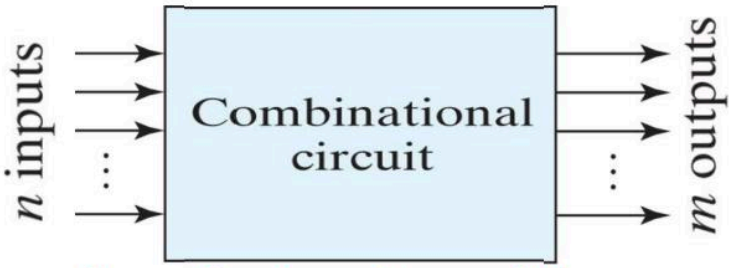
Outlines

- Introduction
- Combinational Circuit Analysis
- Combinational Circuit Design
- Binary Adders and Subtractors
- Binary Decoders and Encoders
- Magnitude Comparator
- Binary Multiplexers
- Three-state Gates

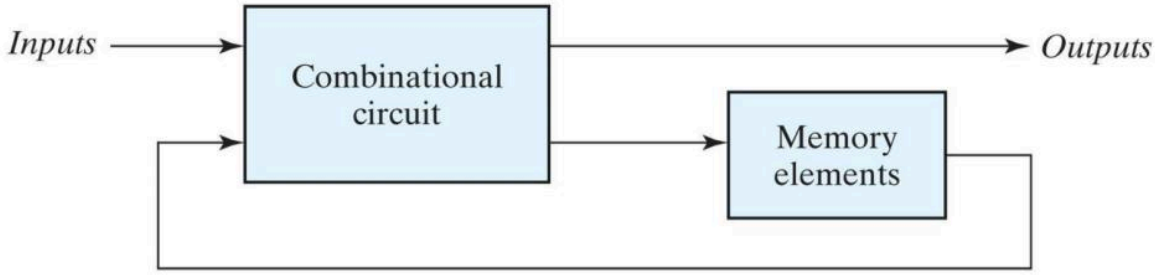
Introduction

Digital Circuits have two types of circuits:

1. Combinational Circuits
   - Outputs depend on the inputs only
   - Memoryless circuits



2. Sequential Circuits
   - Current outputs depend on the inputs and the previous state (outputs).
   - Use memory to store the previous state.



Introduction (Cont.)

Combinational Circuits:

- Combinational Circuit analysis:
  - In the analysis we are interested in determining the function of a given combination circuit.
- Combinational Circuit design:
  - In the design we are interested in developing a combinational circuit based on a given function.

Combinational Circuit Analysis

Combinational Circuit Analysis

Circuit analysis can be achieved by either one of the following two methods:

- Determining the output functions as algebraic expressions.

- Determining the truth table of the output functions.

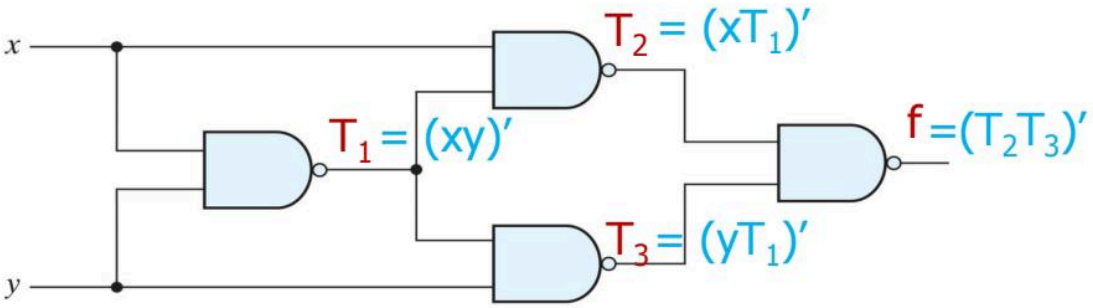
Circuit Analysis steps

1. Label all gate outputs with symbols.
2. Determine Boolean function at the output of each gate.
3. Express functions in terms of input variables
4. Simplify

- **Example:** Find the output function of the following circuit.

Step$_1$

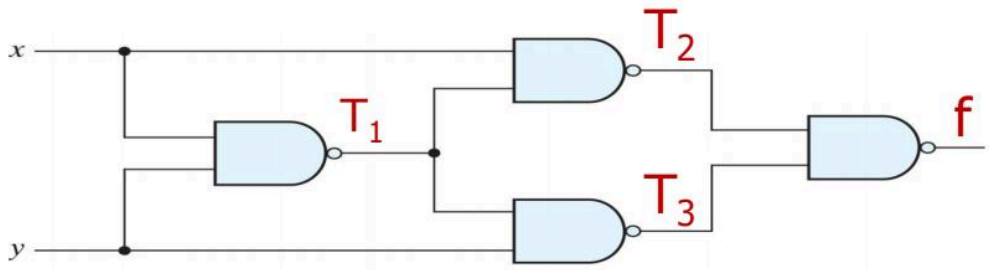
Step$_2$



Step$_3$ $f = (T_2T_3)' = ((xT_1)'\,(yT_1)')' = (xT_1)+(yT_1) = x(xy)'+y(xy)'$

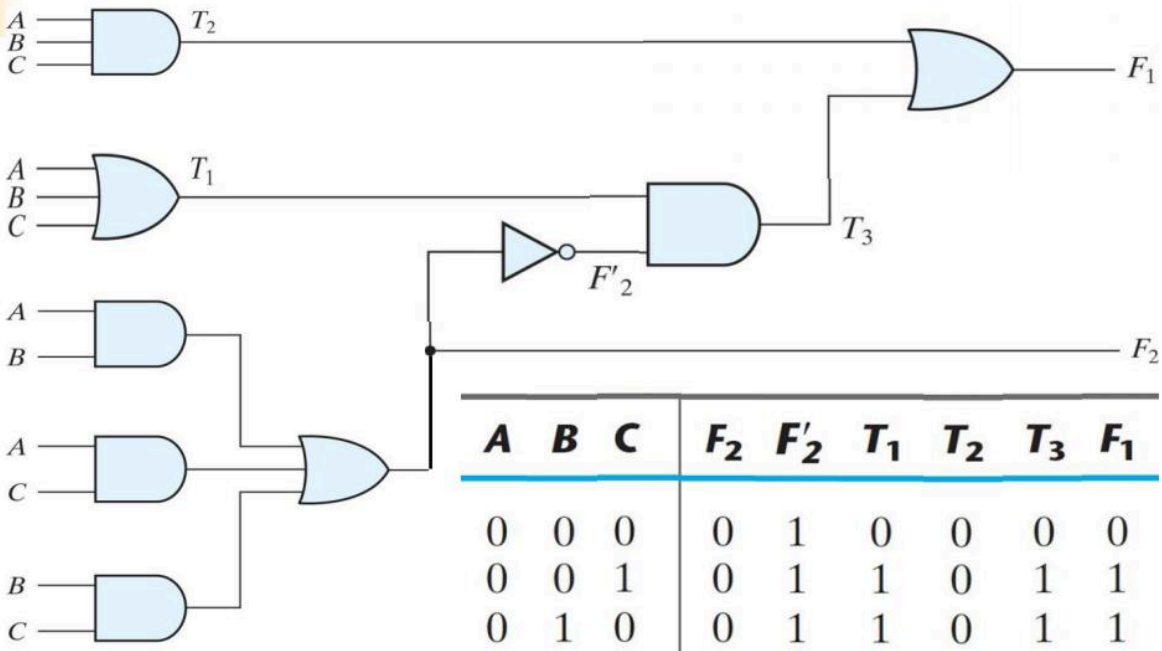
$= x(x'+y')+y(x'+y') = xy' + x'y = x \oplus y$

Solution using Truth Table



x	y	$T_1=(xy)'$	$T_2=(xT_1)'$	$T_3=(yT_1)'$	$f=(T_2T_3)'$
0	0	1	1	1	0
0	1	1	1	0	1
1	0	1	0	1	1
1	1	0	1	1	0

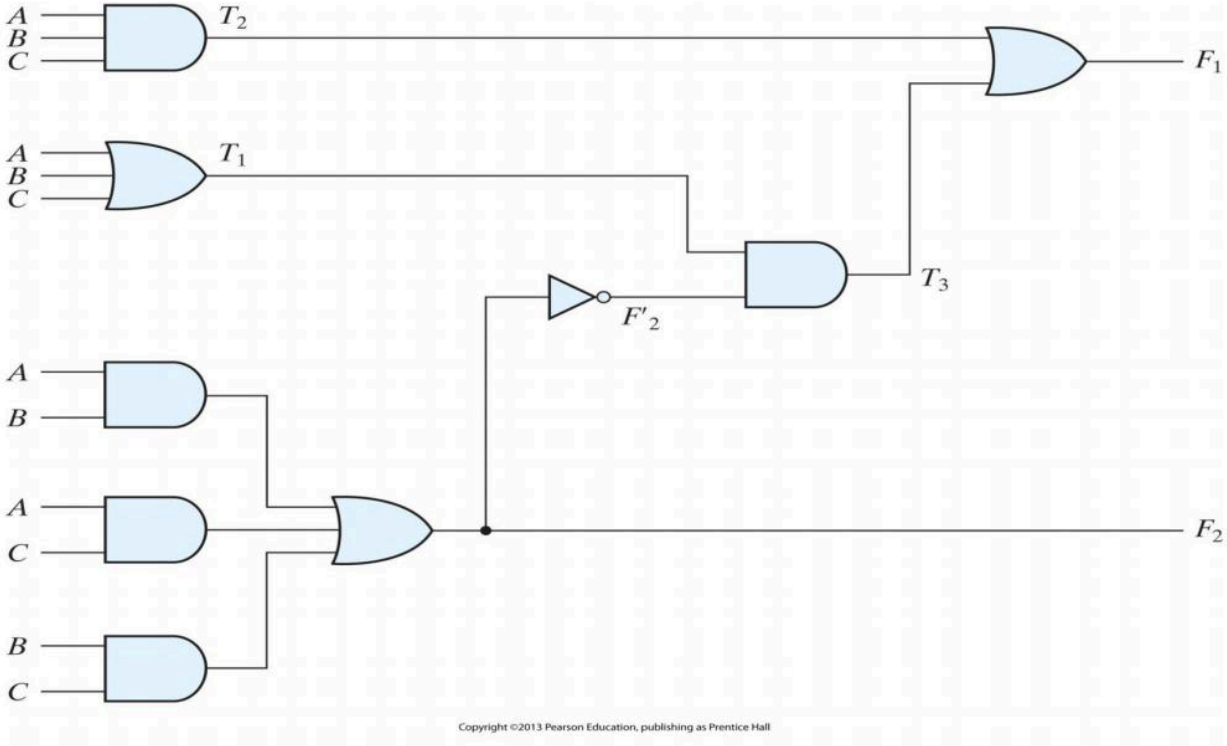
Example 2: Find the functions of the following Circuit using Truth Table



A	B	C	F_2	F'_2	T_1	T_2	T_3	F_1
0	0	0	0	1	0	0	0	0
0	0	1	0	1	1	0	1	1
0	1	0	0	1	1	0	1	1
0	1	1	1	0	1	0	0	0
1	0	0	0	1	1	0	1	1
1	0	1	1	0	1	0	0	0
1	1	0	1	0	1	0	0	0
1	1	1	1	0	1	1	0	1

Exercise

What are the output functions F_1 and F_2 of the following logic circuit?



Combinational Circuit Design

Design of Combinational Circuits

Four main steps in designing a combinational circuit:

1. Determine the number of inputs and outputs and give them symbols.
2. Derive the related Truth Table.
3. Simplify each output.
4. Draw logic diagram and verify correctness.

Design of Combinational Circuits (Cont.)

Example 1: Design a combinational logic circuit that detects odd inputs which are in the rang from 0 to 7.

Solution:

Step 1: 3 inputs (A, B, and C) and one output (F).

Step 2: Truth table

A	B	C	F
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Step 3: Simplify

A\BC	00	01	11	10
0	0	1	1	0
1	0	1	1	0

$F = C$

Design of Combinational Circuits (Cont.)

Example 2: Develop a combinational circuit that converts a BCD digit into a Excess-3 code.

Solution:

Step 1:

- Inputs: BCD digit
  (4 inputs: A, B, C, D)
- Outputs: Excess-3 code
  (4 outputs: w, x, y, z).

Step 2: Truth table

A	B	C	D	w	x	y	z
0	0	0	0	0	0	1	1
0	0	0	1	0	1	0	0
0	0	1	0	0	1	0	1
0	0	1	1	0	1	1	0
0	1	0	0	0	1	1	1
0	1	0	1	1	0	0	0
0	1	1	0	1	0	0	1
0	1	1	1	1	0	1	0
1	0	0	0	1	0	1	1
1	0	0	1	1	1	0	0

Example 2 (Cont.)

Step 3: Outputs' Simplification

AB\CD	00	01	11	10
00	0	0	0	0
01	0	1	1	1
11	x	x	x	x
10	1	1	x	x

$w = A + BC + BD$

AB\CD	00	01	11	10
00	0	1	1	1
01	1	0	0	0
11	x	x	x	x
10	0	1	x	x

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

AB\CD	00	01	11	10
00	1	0	1	0
01	1	0	1	0
11	x	x	x	x
10	1	0	x	x

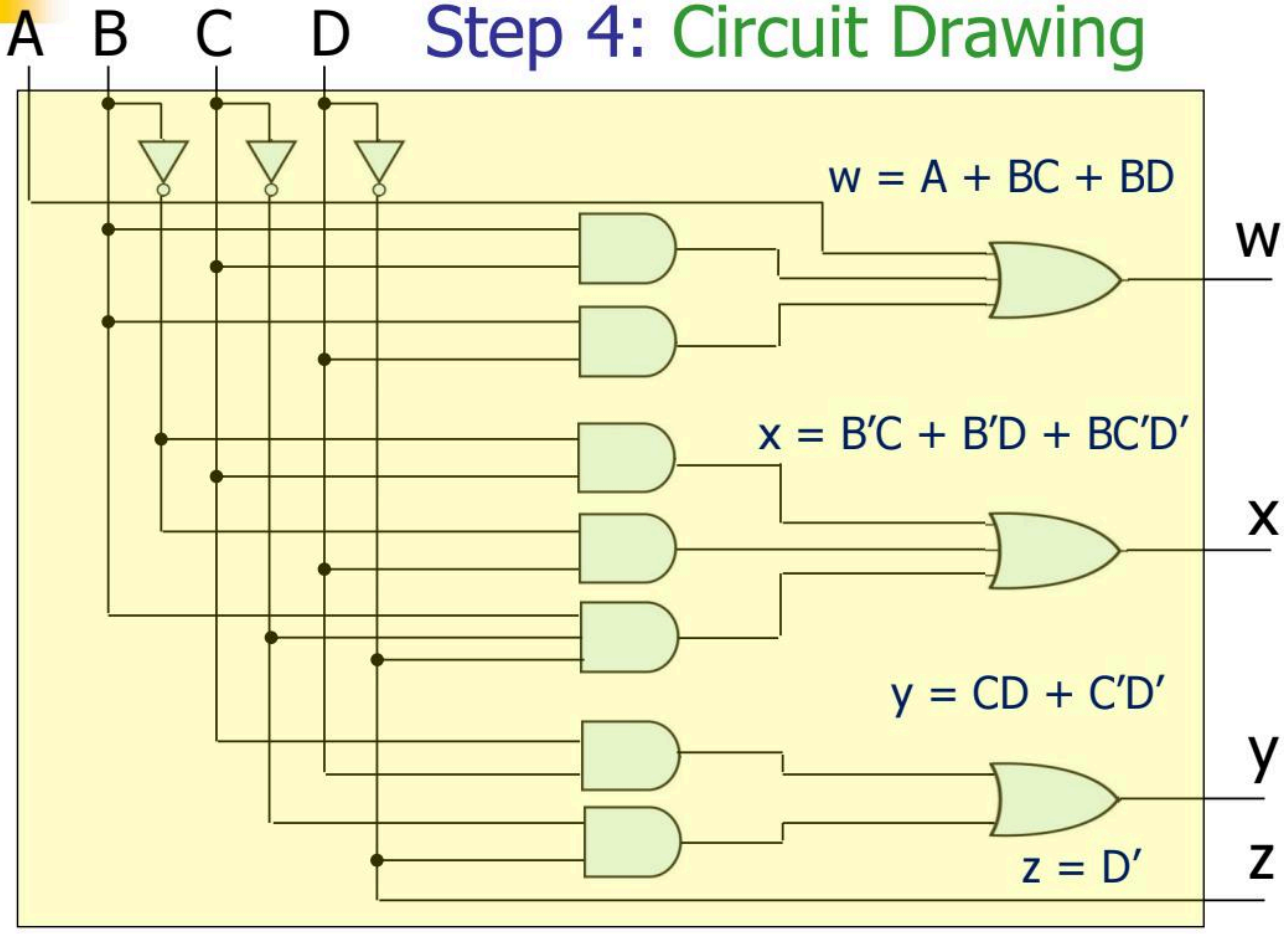
$y = C'D' + CD$

AB\CD	00	01	11	10
00	1	0	0	1
01	1	0	0	1
11	x	x	x	x
10	1	0	x	x

$z = D'$

Example 2 (Cont.)

Step 4: Circuit Drawing



Binary Adders and Subtractors

Half Adder

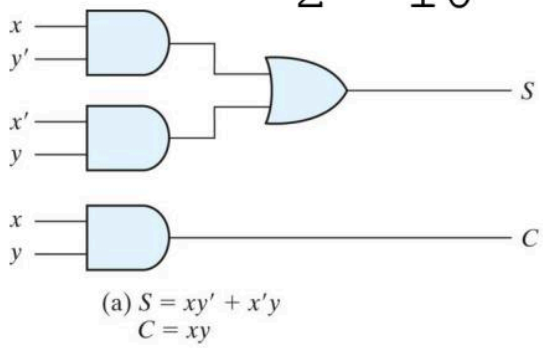
Add two binary numbers

- x , y → single bit inputs
- S → single bit sum
- C → carry out

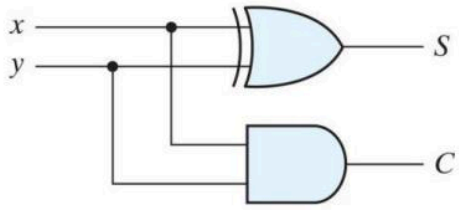
Dec	Binary
1	1
+1	+1
2	10

Half Adder

x	y	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0



(a) $S = xy' + x'y$
$C = xy$



(b) $S = x \oplus y$
$C = xy$

Multiple-bit Addition

Consider adding a 4-bit number to a 4-bit number:

(single-bit adder are used for each bit position)

	A_3	A_2	A_1	A_0
A	0	1	0	1

	B_3	B_2	B_1	B_0
B	0	1	1	1

	1	1	1	
A	0	1	0	1
B	0	1	1	1
	1	1	0	0

$$\begin{array}{r} C_{i+1} \quad C_i \\ A_i \\ +B_i \\ \hline S_i \end{array}$$

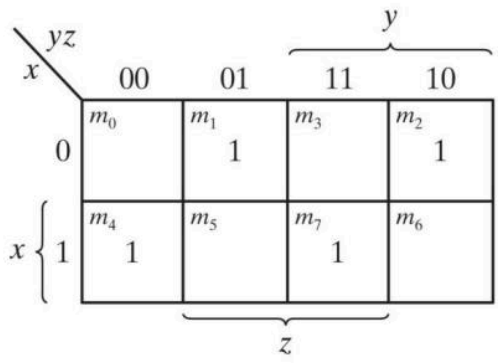
Each bit position creates a sum and carry

Full Adder

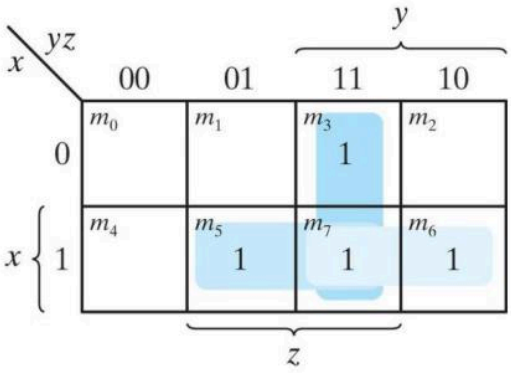
Full adder includes a carry-in z

Full Adder

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

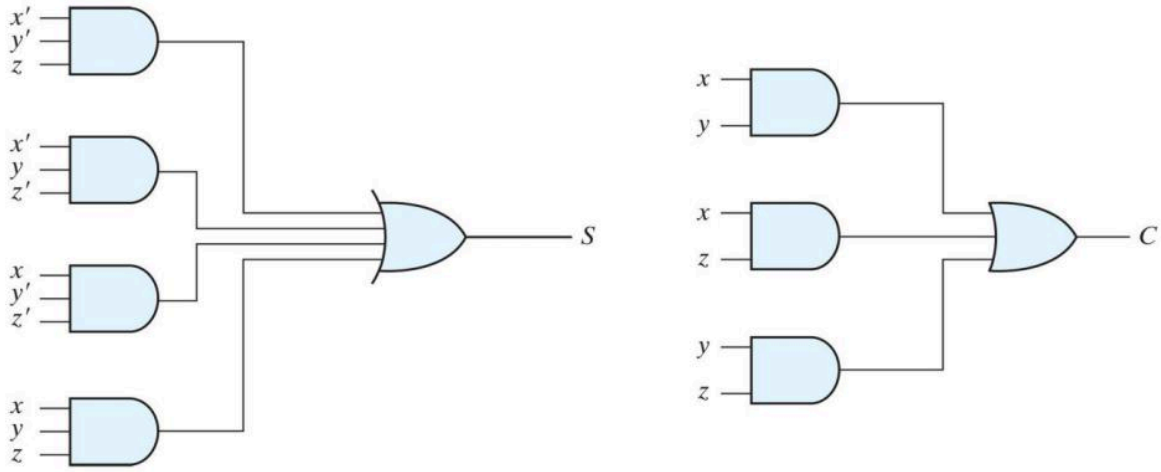


$$S = x'y'z + x'yz' + xy'z' + xyz$$



$$C = xy + xz + yz$$

Full Adder (cont.)



Full Adder (cont.)

The AND/OR representations of *S* and *C* can be reduced into XORs as follows:

$$S = x'y'z + x'yz' + xy'z' + xyz$$
$$= z\,(x'y' + xy) + z'\,(x'y + xy') \quad = z\,(x \odot y) + z'\,(x \oplus y)$$
$$= z\,(x \oplus y)' + z'\,(x \oplus y) \quad = z \oplus x \oplus y$$

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

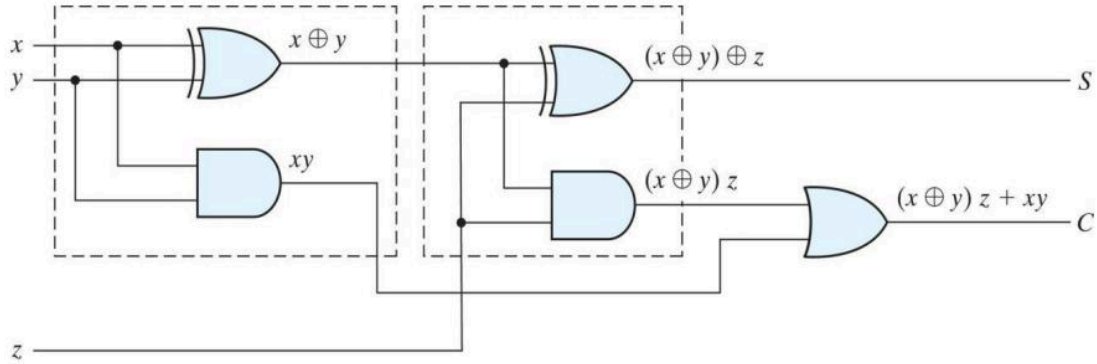
Therefore, the final equations are:

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

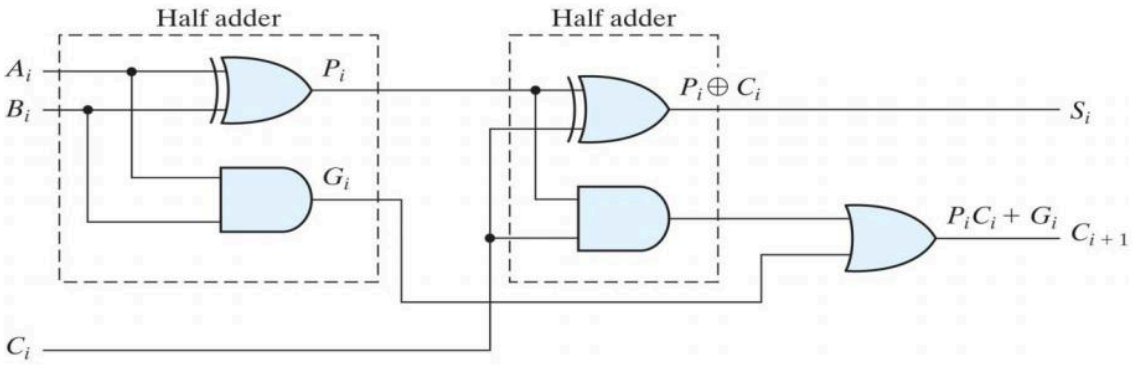
$$C = xy + z\,(x \oplus y)$$

Full Adder (cont.)

$$S = z \oplus x \oplus y \quad \text{and} \quad C = xy + z\,(x \oplus y)$$



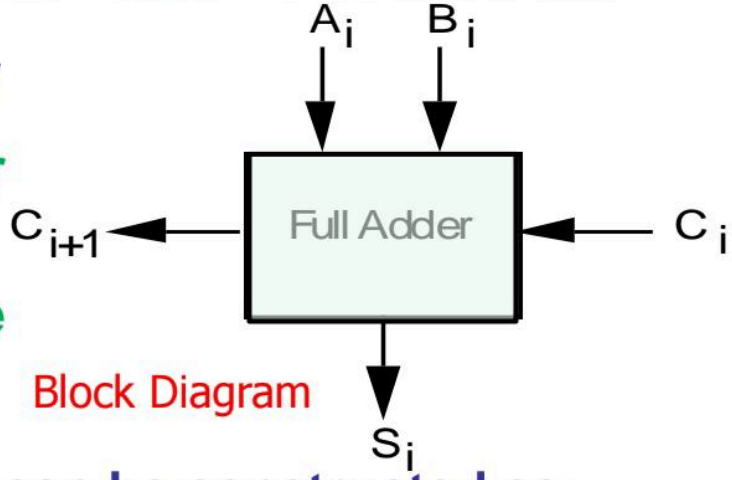
Therefore, the full adder can be constructed from two half adders and an OR gate.



Full Adder (cont.)

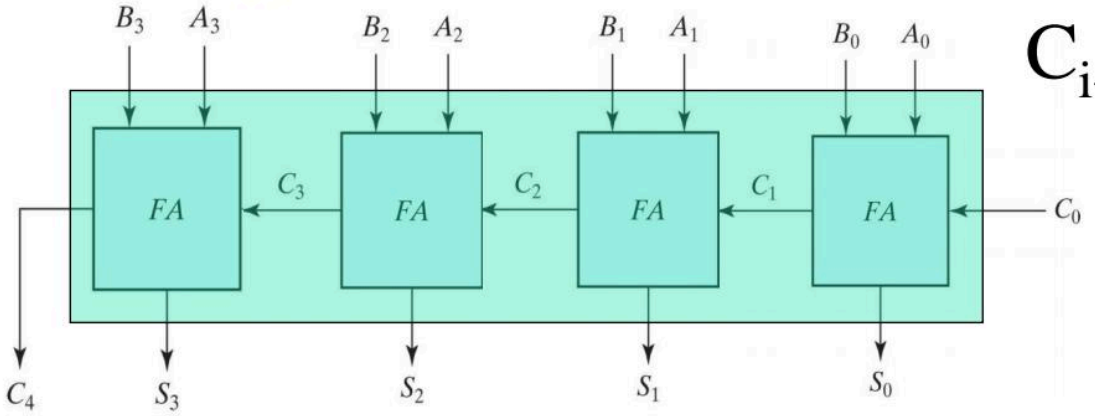
Putting it all together

- Single-bit full adder
- Common piece of computer hardware



Block Diagram

Accordingly, 4-bit Adder can be constructed as:



C_{i+1} C_i

A_i

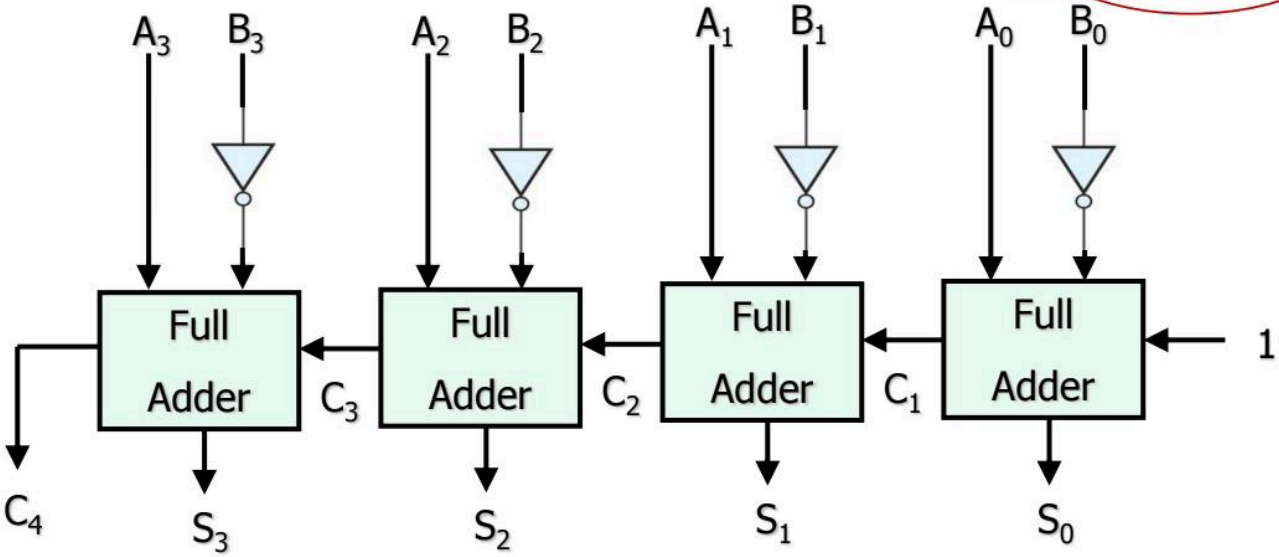
$+B_i$

S_i

Subtractor

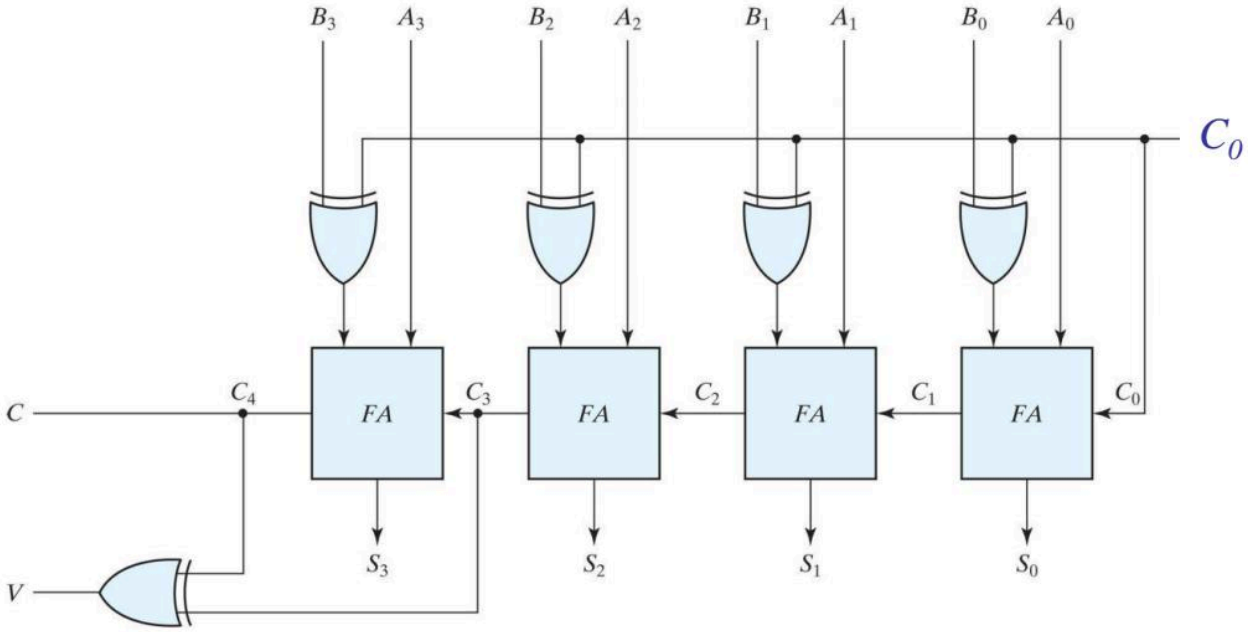
- **Subtracting a number is similar**
  - **Perform 2's complement**
  - **Perform addition**
- **Can we augment the adder with 2's complement hardware?**

$$\begin{array}{rl} & A_3\ A_2\ A_1\ A_0 \\ - & B_3\ B_2\ B_1\ B_0 \\ \hline \text{2's complement} & A_3\ A_2\ A_1\ A_0 \\ + & B'_3\ B'_2\ B'_1\ B'_0 \\ & +\qquad\quad 1 \\ \hline \end{array}$$



Adder-Subtractor

- **4-bit Adder-subtractor (with overflow detection).**
  - **When $C_0 = 0$, the circuit is used as an Adder.**
  - **When $C_0 = 1$, the circuit is used as a Subtractor.**



Overflow in 2's Complement Addition

When two values of the same sign are added:

- Result won't fit in the number of bits provided
- Result has the opposite sign

00	01	11	10	00	11
0010	0011	1110	1101	0010	1110
0011	0110	1101	1010	1100	0100
0101	1001	1011	0111	1110	0010
+2	+3	-2	-3	2	-2
+3	+6	-3	-6	-4	+4
5	-7	-5	7	-2	2
	OFL		OFL		

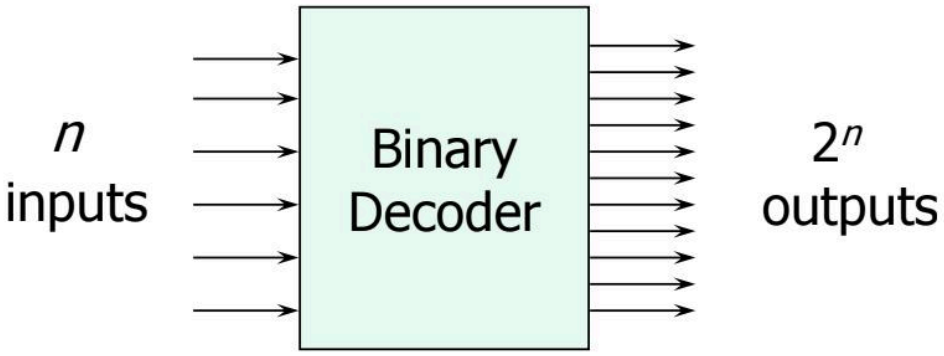
Summary

- Addition and subtraction are fundamental to computer systems
- Create a single bit adder/subtractor
  - Chain the single-bit hardware together to create bigger designs
- The approach is called ripple-carry addition
  - Can be slow for large designs
- Overflow is an important issue for computers
  - Processors often have hardware to detect overflow

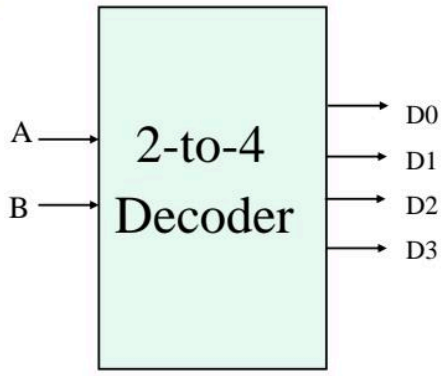
Binary Decoder and Encoder

Binary Decoder

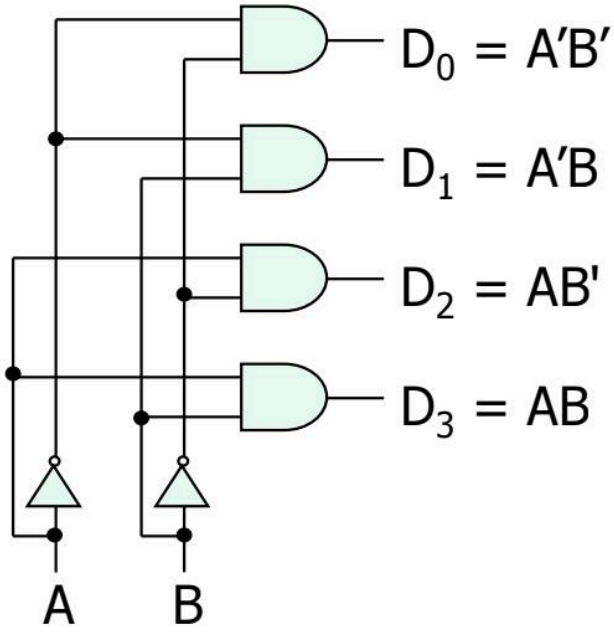
- Black box with *n* input lines and 2^n output lines.
- Only one output is a 1 for any given input.



2-to-4 Binary Decoder

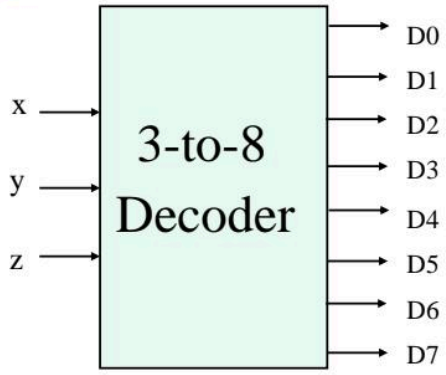


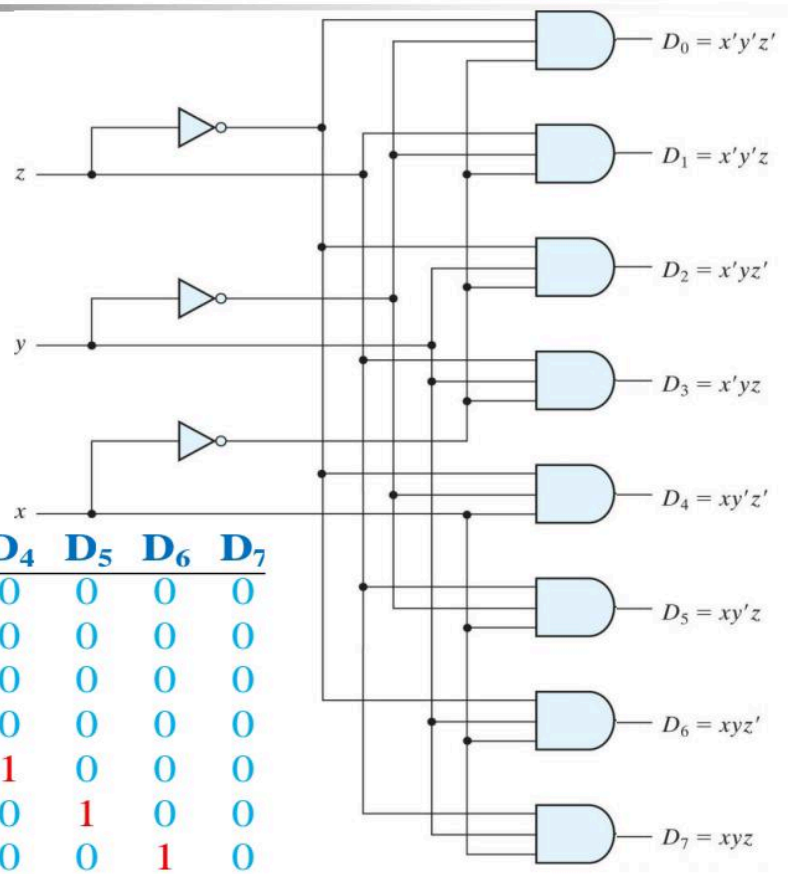
A	B	D_0	D_1	D_2	D_3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1



Note: Each output is a 2-variable minterm (A'B', A'B, AB' or AB)

3-to-8 Binary Decoder



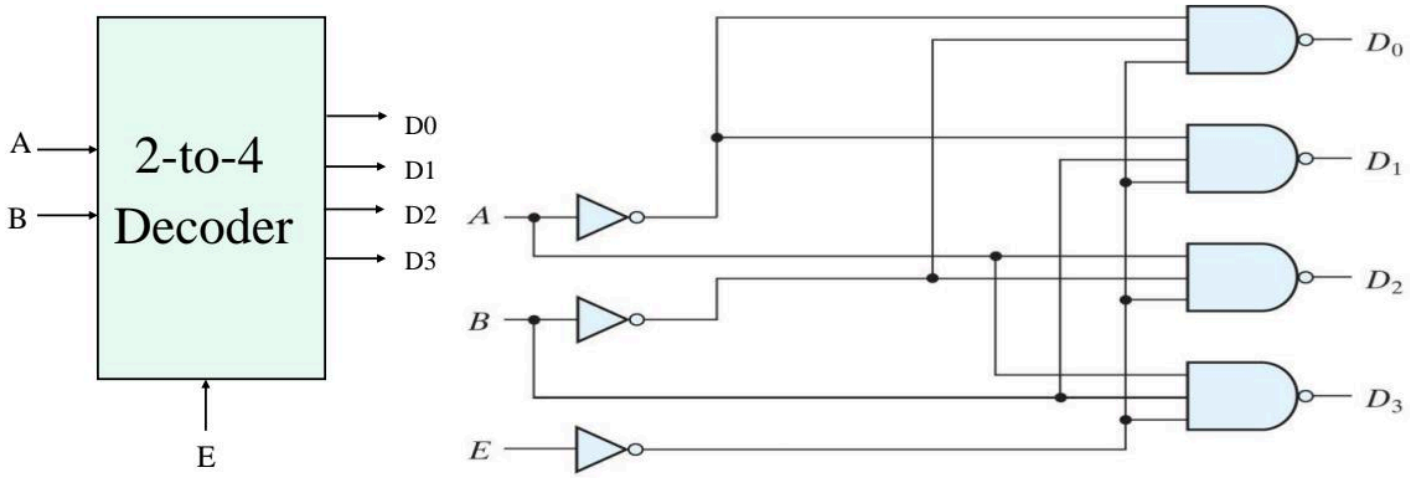


x	y	z	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

Building a binary decoder with NAND gates

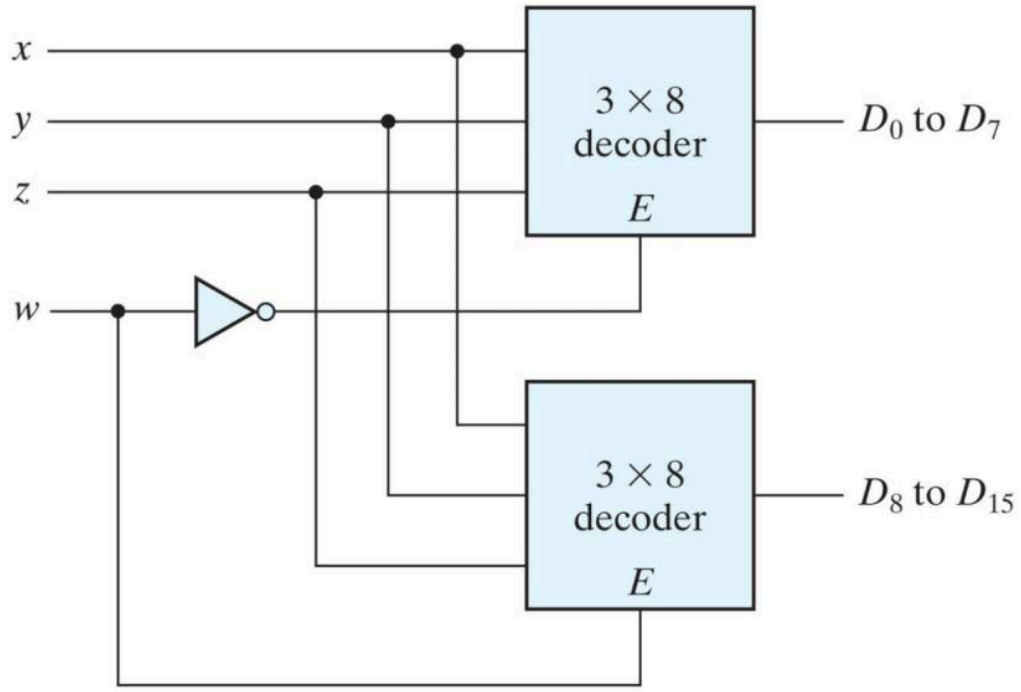
- 2-to-4 decoder with enable
- Add enable signal (E)
- Note: using NAND Gates makes only one 0 be active if the enable E = 0.

E	A	B	D_0	D_1	D_2	D_3
1	X	X	1	1	1	1
0	0	0	0	1	1	1
0	0	1	1	0	1	1
0	1	0	1	1	0	1
0	1	1	1	1	1	1



Use two 3-to-8 decoders to make 4-to-16 decoder

- The enable is used as the 4th line.
- In this case, only one decoder can be active at a time.



Implementing Functions Using Decoders

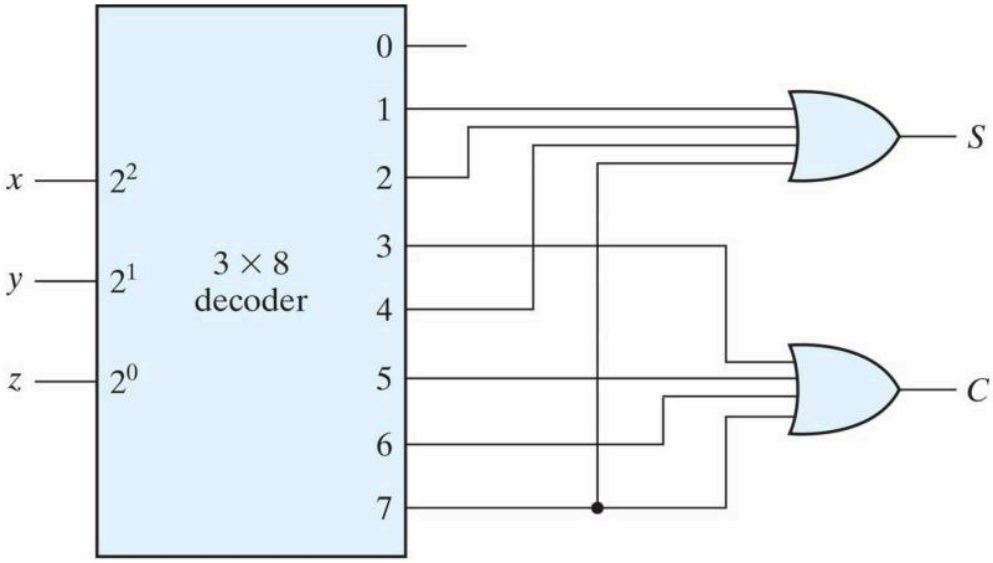
- Any n-variable logic function can be implemented using a single *n-to-*2^n decoder to generate the minterms.
  - OR gate forms the sum
  - The output lines of the decoder corresponding to the minterms of the function are used as inputs to the OR gate
- Any combinational circuit with n inputs and m outputs can be implemented with an *n-to-*2^n decoder with *m* OR gates.
- Suitable when a circuit has many outputs, and each output function is expressed with few minterms.

Example

Implement the function of the Full adder using a decoder.

Sum: S(x, y, z) = Σ (1,2,4,7) Carry: C(x, y, z) = Σ (3,5,6,7)

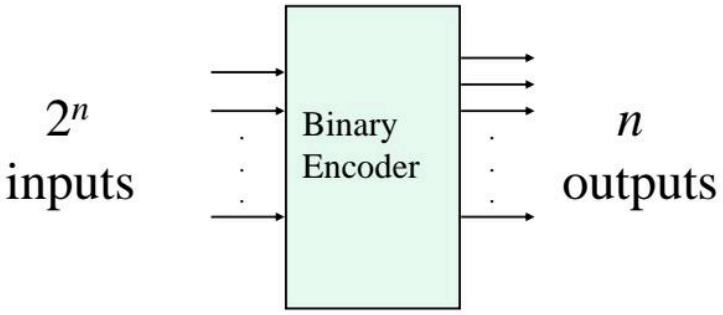
x	y	z	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1



Note: if you have the function, you do not need to construct the truth table.

Binary Encoder

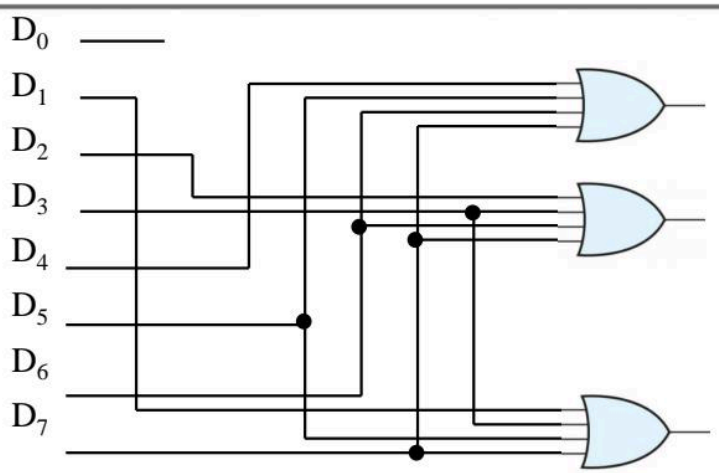
- If the a decoder's output code has fewer bits than the input code, the device is usually called an encoder.
  - e.g. 2^n-to-n
- The simplest encoder is a 2^n-to-n binary encoder
  - One of 2^n inputs = 1
  - Output is an n-bit binary number



8-to-3 Encoder

Truth Table of an Octal-to-Binary Encoder

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



$x = D_4 + D_5 + D_6 + D_7$

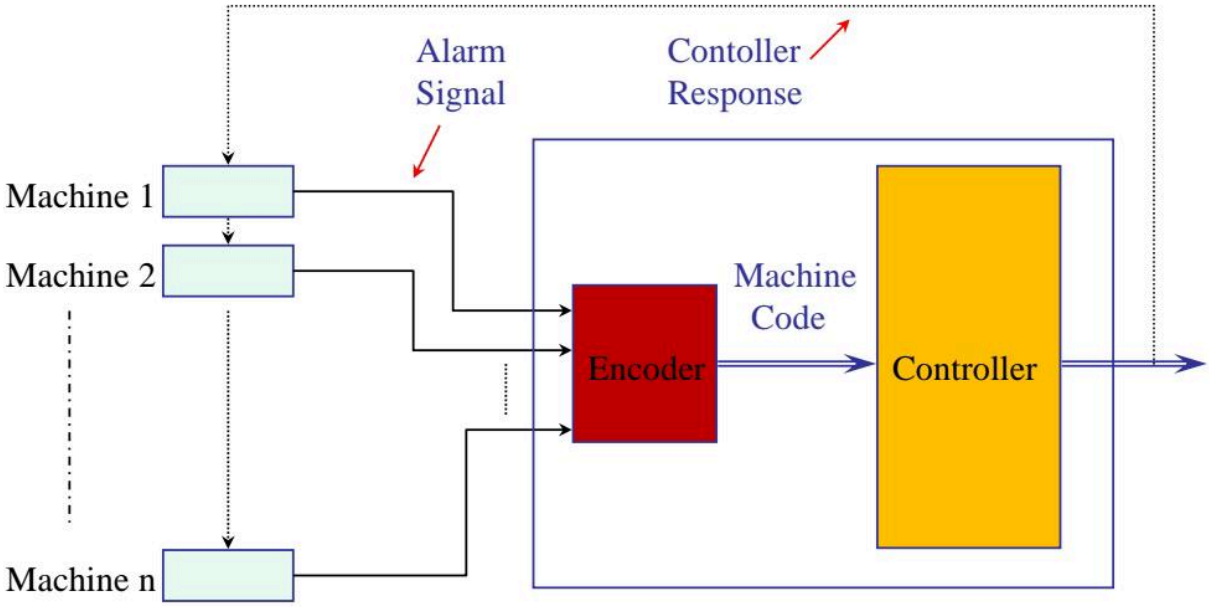
$y = D_2 + D_3 + D_6 + D_7$

$z = D_1 + D_3 + D_5 + D_7$

Note: At any one time, only one input line has a value of 1.

Encoder Application (Monitoring Unit)

- Encoder identifies the requester and encodes the value.
- Controller accepts digital inputs.



Summary

- Decoders allow for generation of a single binary output from an input binary code.
- For an n-input binary decoder there are 2^n outputs.
- Decoders are widely used with memory.
- Encoders can be used for data compression.

Binary Magnitude Comparators

Magnitude Comparator

- Compares two numbers: *A* and *B*
  - Result: $A > B$, $A = B$, or $A < B$
- Truth table entries for *n*-bit numbers = 2^{2n} entries
  - Impractical for design
- How can we determine that two numbers are equal?
  - $A_3\,A_2\,A_1\,A_0$ and $B_3\,B_2\,B_1\,B_0$ are equal iff
  - $A_3 = B_3$ and $A_2 = B_2$ and $A_1 = B_1$ and $A_0 = B_0$
- New function: x_i indicates if $A_i = B_i$
  - $x_i = A_iB_i + A'_iB'_i$ (X-NOR)
  - Thus, $(A = B) = x_3\,x_2\,x_1\,x_0$
  - What about $A < B$ and $A > B$?

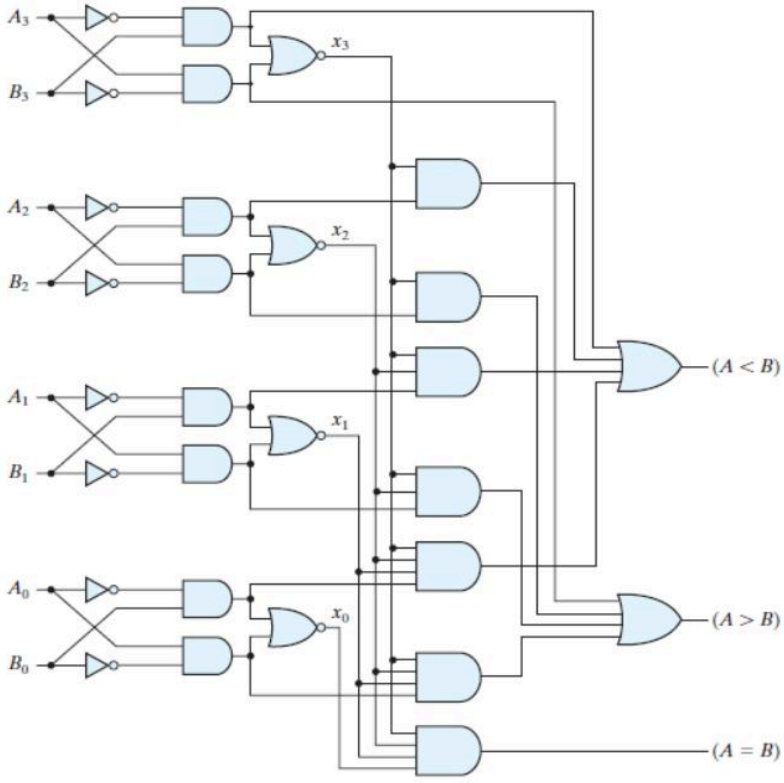
Magnitude Comparator (cont.)

- A > B
  - How can we tell that A > B?
  - Look at MSB where A and B differ
    - ✓ If A = 1 and B = 0, then A > B
    - ✓ If not, then A ≤ B
  - Assume $n = 4$
  - $(A>B) = A_3 B'_3 + x_3 A_2 B'_2 + x_3 x_2 A_1 B'_1 + x_3 x_2 x_1 A_0 B'_0$
- A < B
  - The same as A > B but A and B are exchanged.
  - $(A<B) = A'_3 B_3 + x_3 A'_2 B_2 + x_3 x_2 A'_1 B_1 + x_3 x_2 x_1 A'_0 B_0$

Note: The comparasion must be started from MSB.

Magnitude Comparator (cont.)

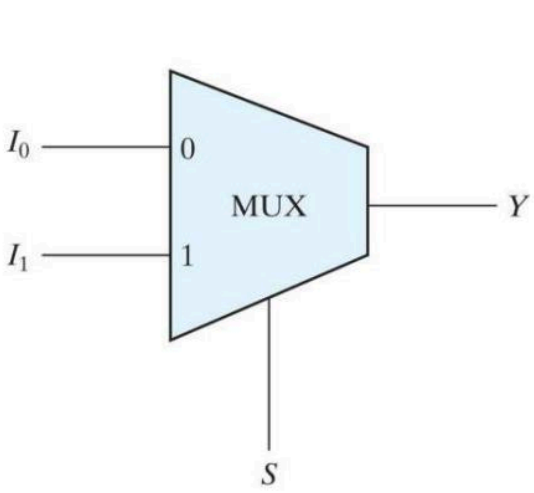
- $(A>B) = A_3 B'_3 + x_3 A_2 B'_2 + x_3 x_2 A_1 B'_1 + x_3 x_2 x_1 A_0 B'_0$
- $(A<B) = A'_3 B_3 + x_3 A'_2 B_2 + x_3 x_2 A'_1 B_1 + x_3 x_2 x_1 A'_0 B_0$
- $(A = B) = x_3 x_2 x_1 x_0$

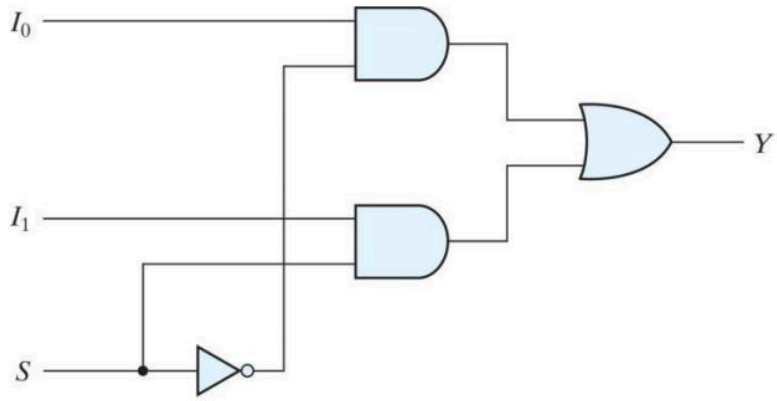


Binary Multiplexers

Binary Multiplexers

- Select an input value with one or more select bits
- Use for transmitting data
- Allow for conditional transfer of data
- Sometimes called a MUX
- Example: 2-to-1 MUX

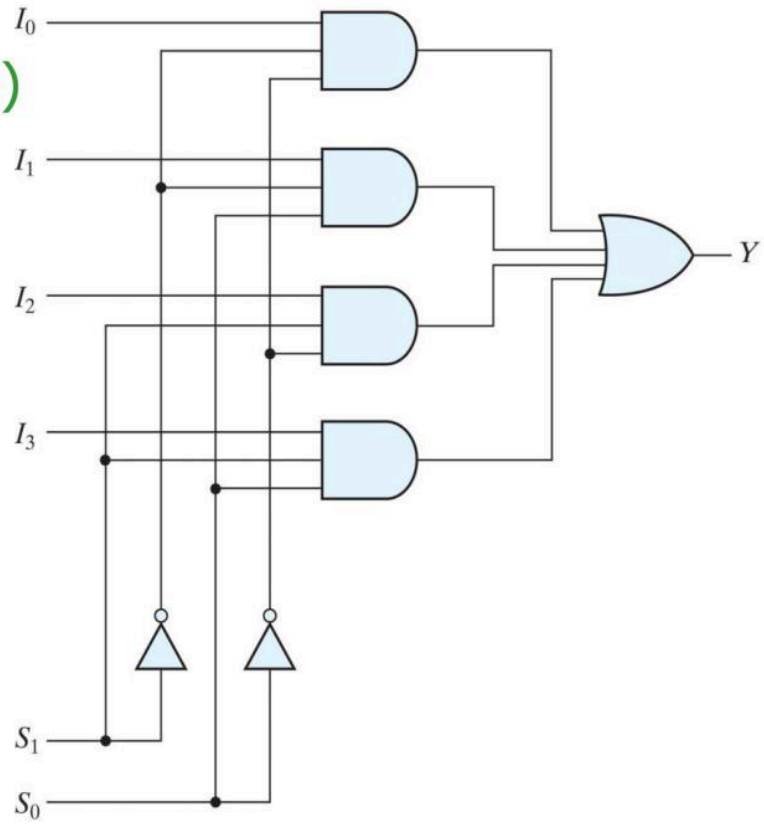




Multiplexers (cont.)

- 4-to-1 Multiplexer
  - 4 input lines (I_0, I_1, I_2, I_3)
  - 2 selection lines (S_0, S_1)
  - One output line (Y)
  - Function table

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

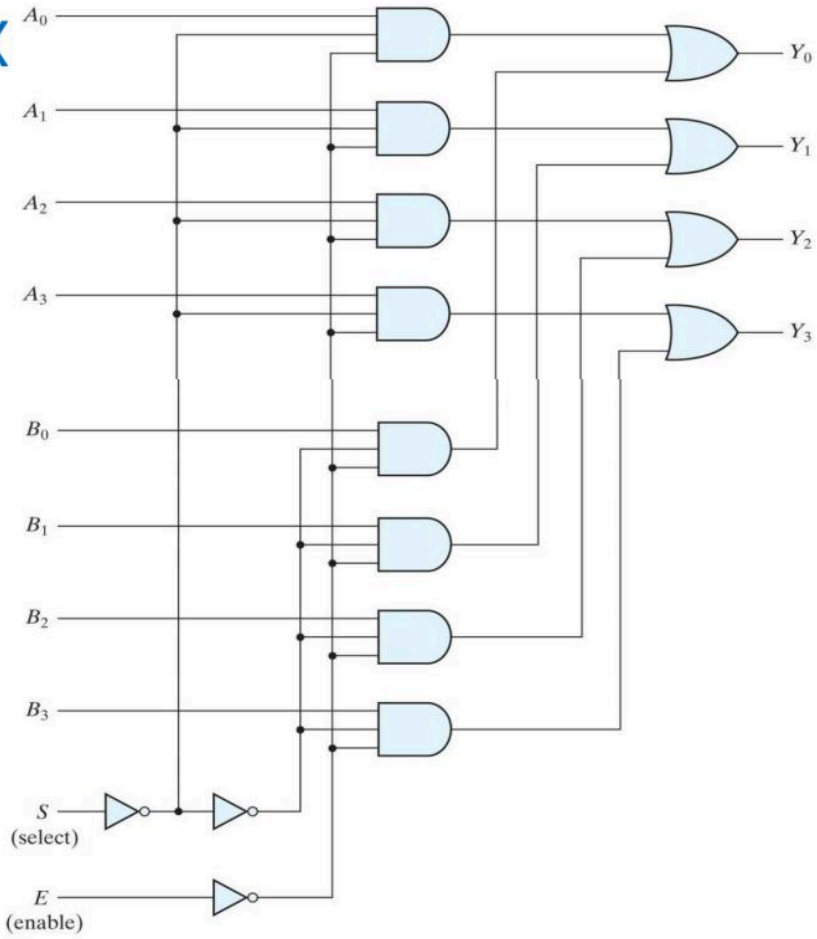


Multiplexers (cont.)

- Quadruple 2-to-1 MUX
  - 4-bit inputs
  - 4-bit outputs
  - one selection bit (*S*)
  - Enable bit (*E*)

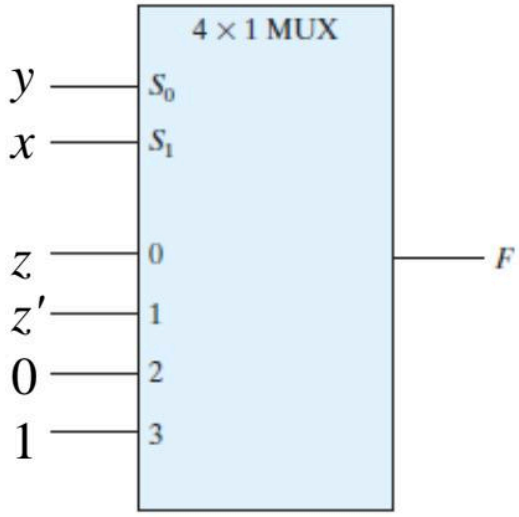
E	*S*	Output *Y*
1	X	all 0's
0	0	select *A*
0	1	select *B*

Function table



Implementing Boolean Functions with MUXs

- Example: Implement the Boolean function $F(x, y, z) = \Sigma(1, 2, 6, 7)$ using 4 x 1 MUX.



S_1	S_0		
x	y	z	F
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

$I_0 = z$

$I_1 = z'$

$I_2 = 0$

$I_3 = 1$

- Set "data" input lines of multiplexer to function values
- Connect input variable to "select" inputs
- Set the inputs of multiplexer.

Implementing a 4-Input Function with a MUX

- Example: Implement the Boolean function F(A, B, C, D) = ∑(1, 3, 4, 11, 12, 13, 14, 15) using 8 x 1 MUX.

S_2	S_1	S_0		
A	B	C	D	F
0	0	0	0	0
0	0	0	1	1
0	0	1	0	0
0	0	1	1	1
0	1	0	0	1
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

$I_0 = D$

$I_1 = D$

$I_2 = D'$

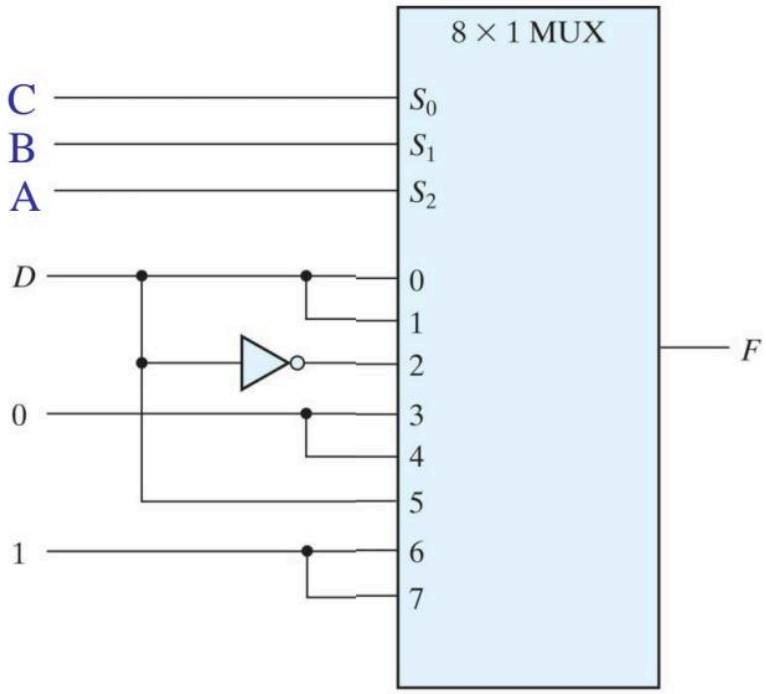
$I_3 = 0$

$I_4 = 0$

$I_5 = D$

$I_6 = 1$

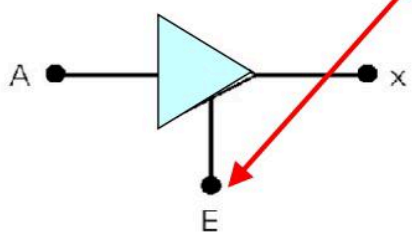
$I_7 = 1$



Three-state Gate

Three-state Gate

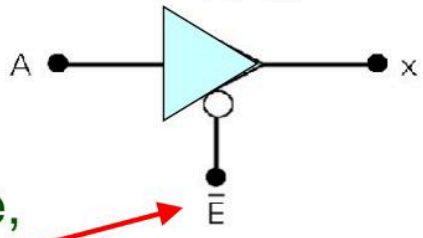
- Output: 0, 1, and high-impedance (open circuit)
- If the select input (E) is 0, the three-state gate has no output



E	x
0	Hi-Z
1	A

Opposite true here,

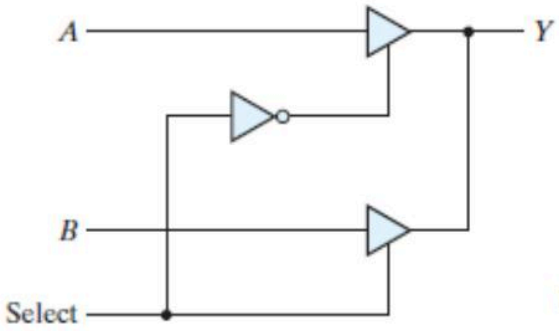
No output if E is 1



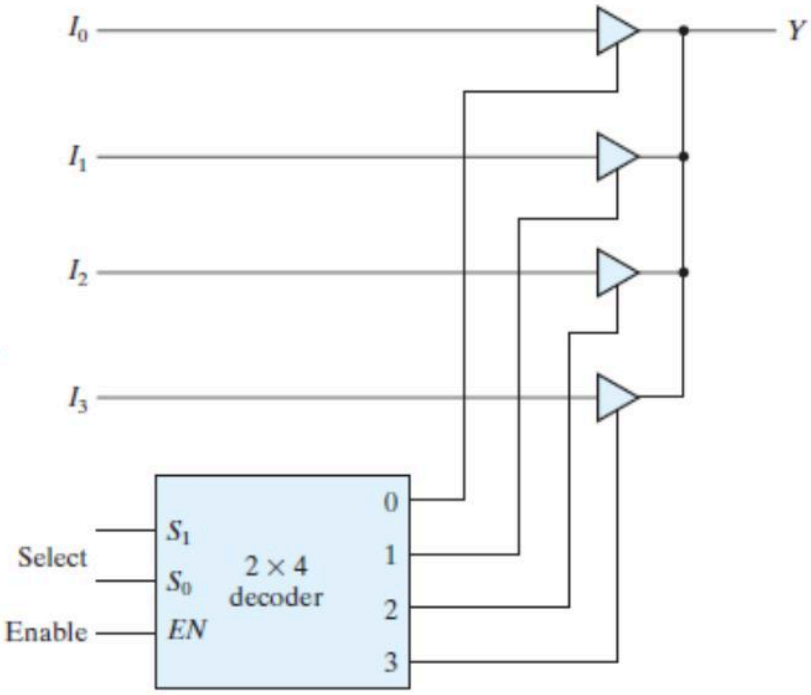
$\bar{E}$	x
0	A
1	Hi-Z

Three-state Gate (cont.)

- Multiplexers can be constructed with three-state gates.
- Two examples



(a) 2-to-1-line mux



(b) 4-to-1-line mux

Summary

- Magnitude comparators allow for data comparison
  - Can be built using AND-OR gates
  - Greater / Less than requires more hardware than equality
- Multiplexers are fundamental digital components
  - Can be used for implementing logic functions
  - Useful for datapaths
  - Scalable
- Tristate buffers have three types of outputs
  - 0, 1, high-impedence (Z)
  - Useful for datapaths

COE211: Digital Logic Design

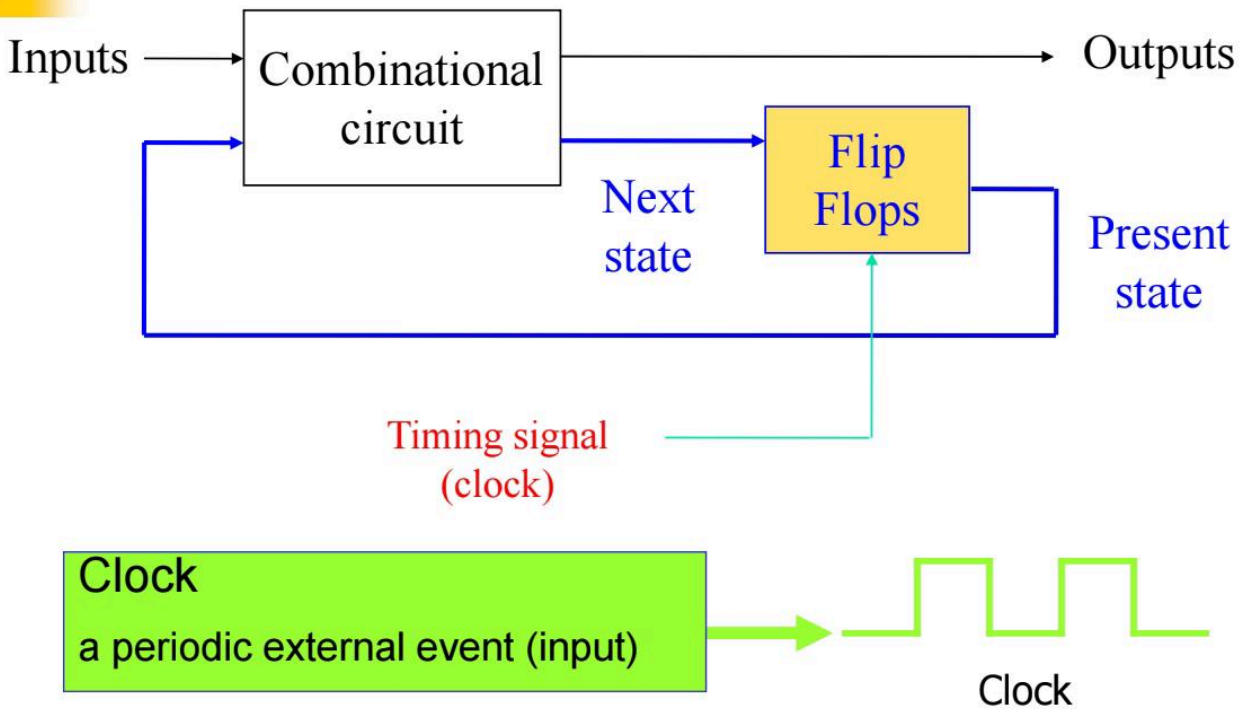
CH5: Synchronous Sequential Logic

Part 1: Latches and Flip-Flops

So Far ...

- Logical operations which respond to *combinations* of inputs to produce an output
  - Call these *combinational logic* circuits
- For example, we can add two numbers but:
  - No way to add two numbers, *then* add a third (a *sequential* operation)
  - No way to remember or store information after inputs have been removed
- To do this, we need *sequential logic* capable of storing intermediate (and final) results

Sequential Circuits



Clock

a periodic external event (input)

- synchronizes when current state changes should happen
- keeps system well-behaved
- makes it easier to design and build large systems

Storage Elements

- Binary storage device capable of storing one bit
- Latch = level-sensitive device
  - Control signal: Enable
  - State changes with input when enabled (e.g., when Enable = 1)
  - Holds last input value when disabled (when Enable = 0)
- Flip-flop = edge-triggered device
  - Control signal: periodic Clock
  - State of flip-flop can only change during clock transition
  - Example: Flip-flops change on rising/falling edge of clock
- Why change on an edge?
  - Couldn't we change state while clock is 1?
  - That would be a latch!
- Edge is moment in time, state is duration conditions

Level-sensitive vs Edge-triggered

- Latches are level-sensitive

(a) Response to positive level

- Flip-flops are edge-sensitive

(b) Positive-edge response

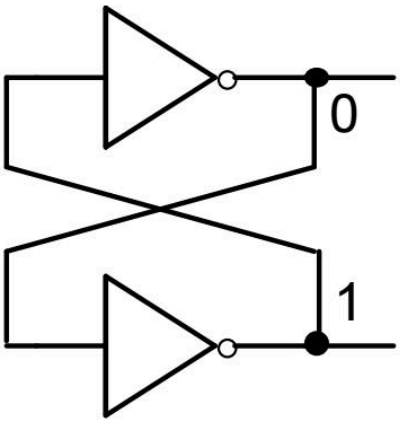
(c) Negative-edge response

Latchs

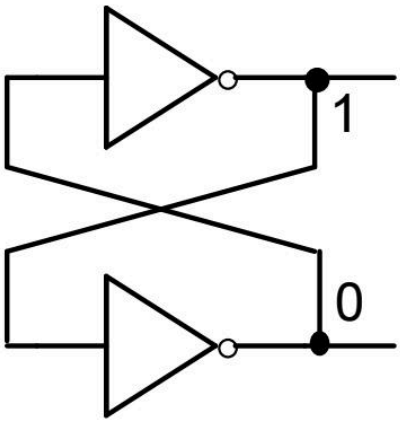
- Characteristics
  - Can store one bit of binary information
  - Level-sensitive devices, asynchronous
- SR Latch
  - Named after functionality: S = set, R = reset
  - Specification:
    - Inputs: S and R
    - Outputs: Q and Q'
- SR Latch Operation:
  - Q=1 and Q'=0 when in set state
  - Q=0 and Q'=1 when in reset state
  - Inputs should be 0 unless pulse on S or R sets or resets latch

Cross-coupled Inverters

A stable value can be stored at inverter outputs

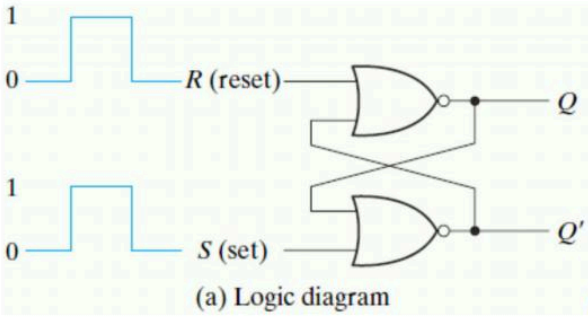


State 1



State 2

SR Latch with NORs

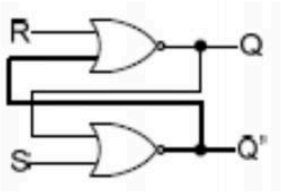


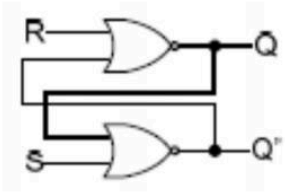
(a) Logic diagram

S	R	Q	Q'	
1	0	1	0	
0	0	1	0	(after $S = 1, R = 0$)
0	1	0	1	
0	0	0	1	(after $S = 0, R = 1$)
1	1	0	0	

(b) Function table

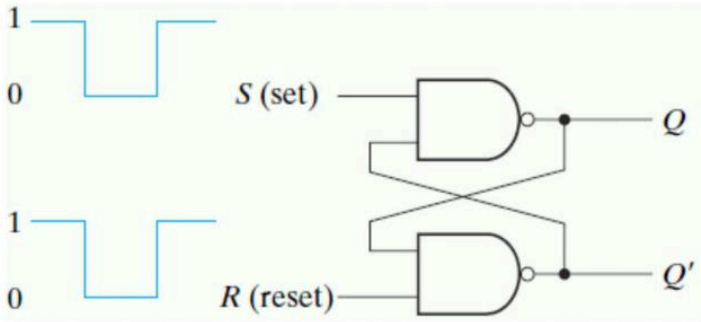
- *Set* and *Reset* are stable states
  - If *S=0* and *R=0*, then state will not change by itself





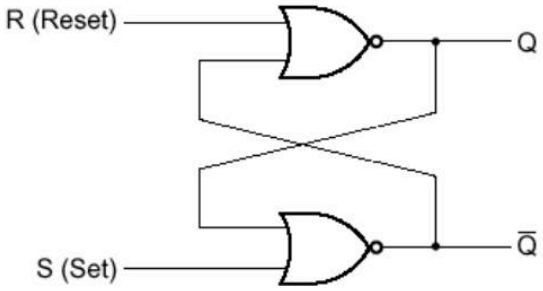
RS Latch with NANDs

- RS latch is made from two cross-coupled NANDs
- Sometimes called $\bar{R}\bar{S}$ latch
- Usually $\bar{S}$=1 and $\bar{R}$=1



S	R	Q	Q'	
1	0	0	1	
1	1	0	1	(after $S = 1, R = 0$)
0	1	1	0	
1	1	1	0	(after $S = 0, R = 1$)
0	0	1	1	

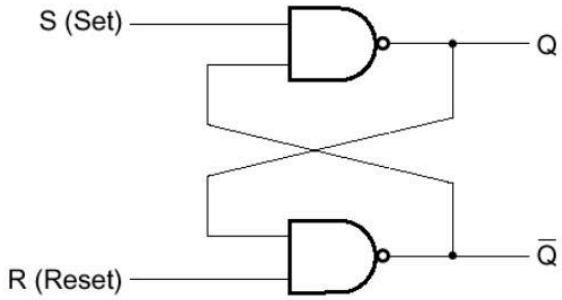
RS Latches



(a) Logic diagram

S	R	Q	$\overline{Q}$	
1	0	1	0	Set state
0	0	1	0	
0	1	0	1	Reset state
0	0	0	1	
1	1	0	0	Undefined

(b) Function table



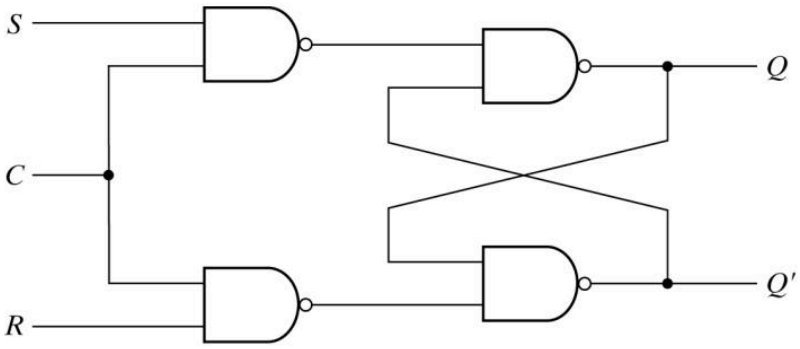
(a) Logic diagram

S	R	Q	$\overline{Q}$	
0	1	1	0	Set state
1	1	1	0	
1	0	0	1	Reset state
1	1	0	1	
0	0	1	1	Undefined

(b) Function table

RS Latch with control input

- **Avoid uncontrolled latch changes**
- **C = 0 disables all latch state changes**
- **Control signal enables data change when C = 1**



(a) Logic diagram

C	S	R	Next state of Q
0	X	X	No change
1	0	0	No change
1	0	1	$Q = 0$; Reset state
1	1	0	$Q = 1$; set state
1	1	1	Indeterminate

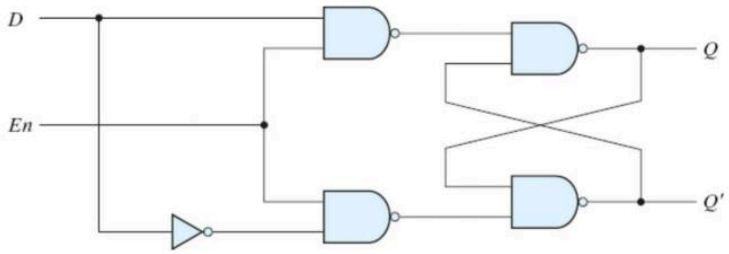
(b) Function table

Fig. 5-5 SR Latch with Control Input

D Latch

- How to remove state for S=1, R=1 ?
- Solution
  - Just use one input pin D to indicate set or reset
  - Enable bit (En) ensures that latch is only set when intended
- D latch
  - Inputs:
    - D (data)
    - En (enable)
- Circuit:

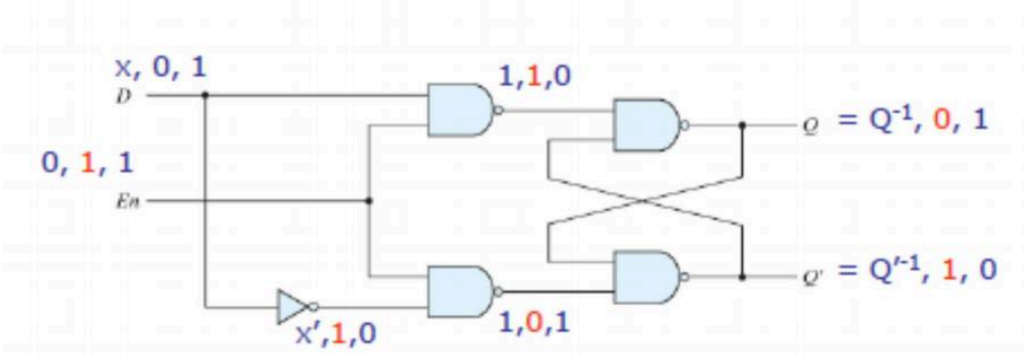
En	*D*	Next state of *Q*
0	X	No change
1	0	$Q = 0$; reset state
1	1	$Q = 1$; set state



D Latch in Operation

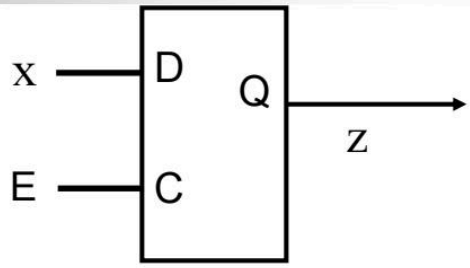
- Removes disallowed R,S combination
  - D latch forwards data while En=1
  - D latch holds data when En=0

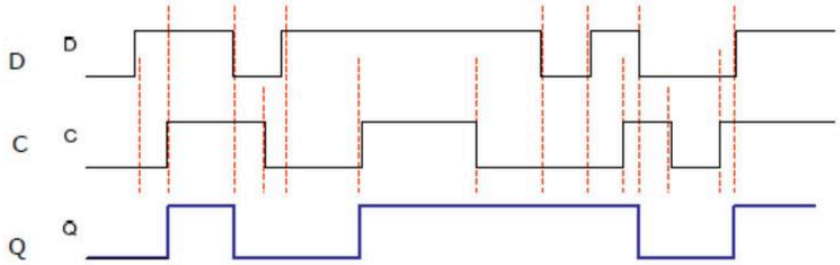
En	*D*	Next state of *Q*
0	X	No change
1	0	$Q = 0$; reset state
1	1	$Q = 1$; set state



D Latch

- Z only changes when E is high
- If E is high, Z follows X

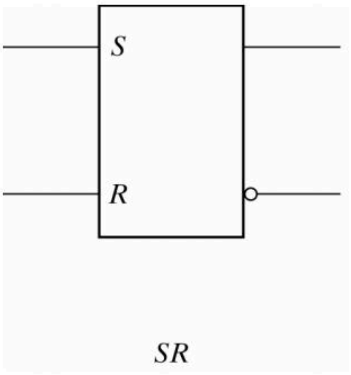




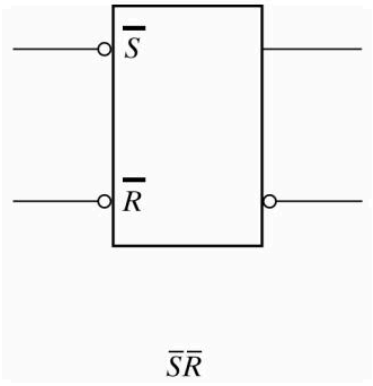
- If E = 0, the D latch stores data indefinitely regardless of input D values
- Forms basic storage element in computers

Symbols for Latches

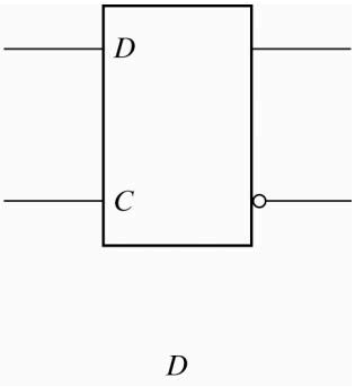
- RS latch, based on NOR gates
- R'S' latch, based on NAND gates
- D latch can be based on either NOR or NAND
- D latch, sometimes called transparent latch



SR



$\overline{S}\overline{R}$



D

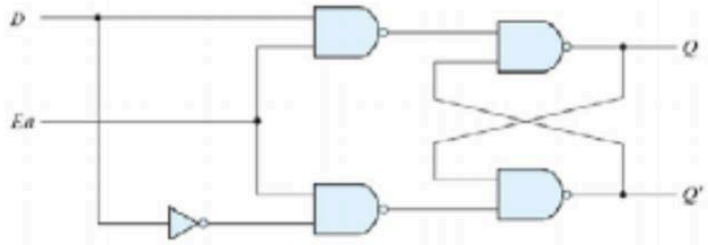
Summary

- Latches are based on combinational gates (e.g. NAND, NOR)
- Latches store data even after data input has been removed
- RS latches operate like cross-coupled inverters with control inputs (S = Set, R = Reset)
- With additional gates, a RS latch can be converted to a D latch (D stands for Data)
- D latch operation is simple
  - When C = 1, data input D stored in latch and Q = D
  - When C = 0, data input D is ignored and Q = previous latch value

Sec: 5.3
Sequential Circuits: Flip Flops

D Latch - Summary

- D latch circuit



En	*D*	Next state of *Q*
0	X	No change
1	0	$Q = 0$; reset state
1	1	$Q = 1$; set state

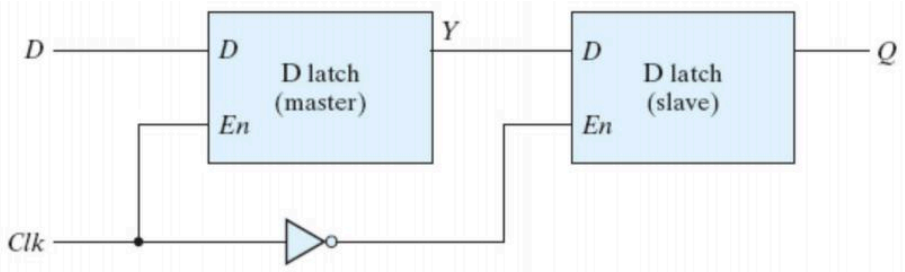
- Data is stored while clock is high

- How can we build a flip-flop that stores on edge transition?


or

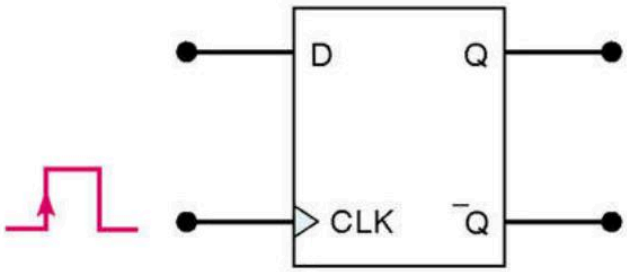

Edge-triggered D Flip-Flop

- Construct D flip-flop from two latches:

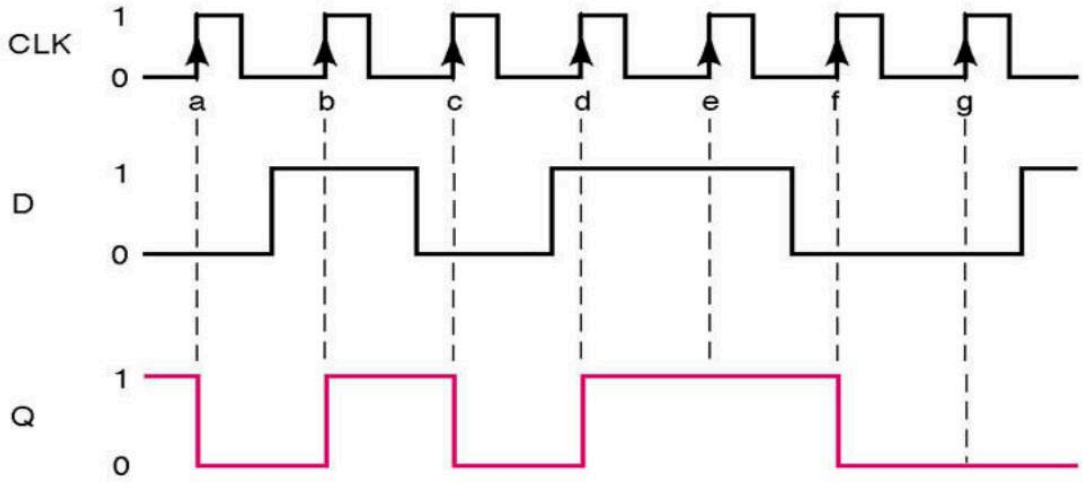


- Primary latch:
  - Reads value of D while CLK is high
  - Is disabled when clock is low
- Secondary latch:
  - Is disabled when CLK is high (i.e., holds previous value)
  - Takes value from master on negative edge of clock

Clocked D Flip-Flop

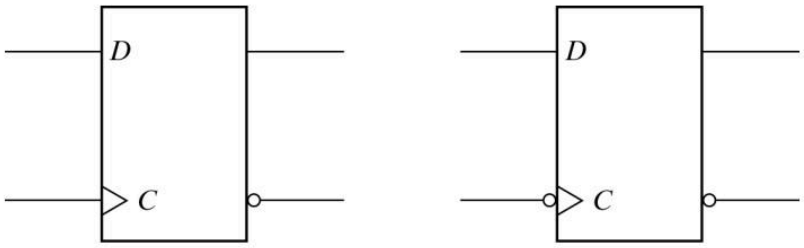


D	CLK	Q
0	↑	0
1	↑	1



Positive and Negative Edge D Flip-Flops

- **There exist positive and negative edge trigger D FF**
- **Bubbled *Clock (C) means* negative edge trigger**



(a) Positive-edge (a) Negative-edge

Fig. 5-11 Graphic Symbol for Edge-Triggered *D* Flip-Flop

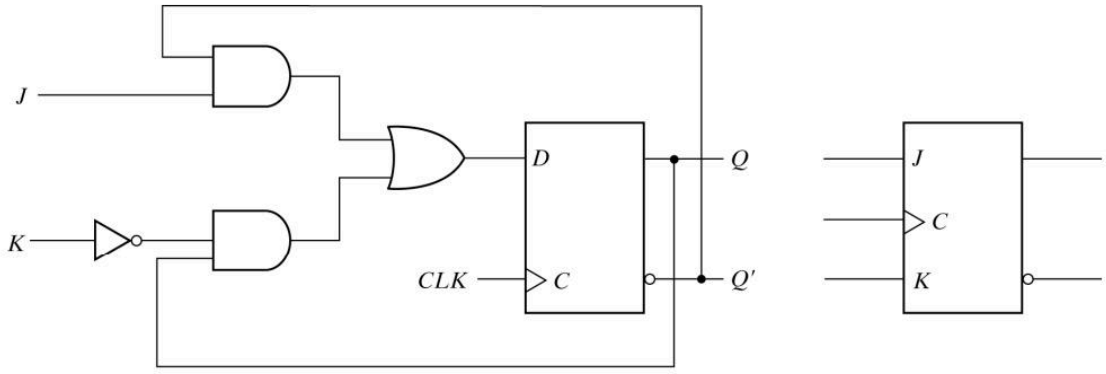
We may need other FFs to *synchronously* set or reset the FF state
and/or complement the previous state

Lo-Hi edge Hi-Lo edge

Positive Edge-Triggered J-K Flip-Flop

- Created from D FF
- K resets
- J sets
- J=K=1 inverts Q

J	K	CLK	Q	Q′
0	0		Q_0	Q_0'
0	1		0	1
1	0		1	0
1	1		Q_0'	Q_0

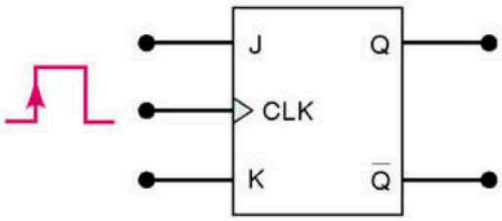


(a) Circuit diagram (b) Graphic symbol

Fig. 5-12 *JK* Flip-Flop

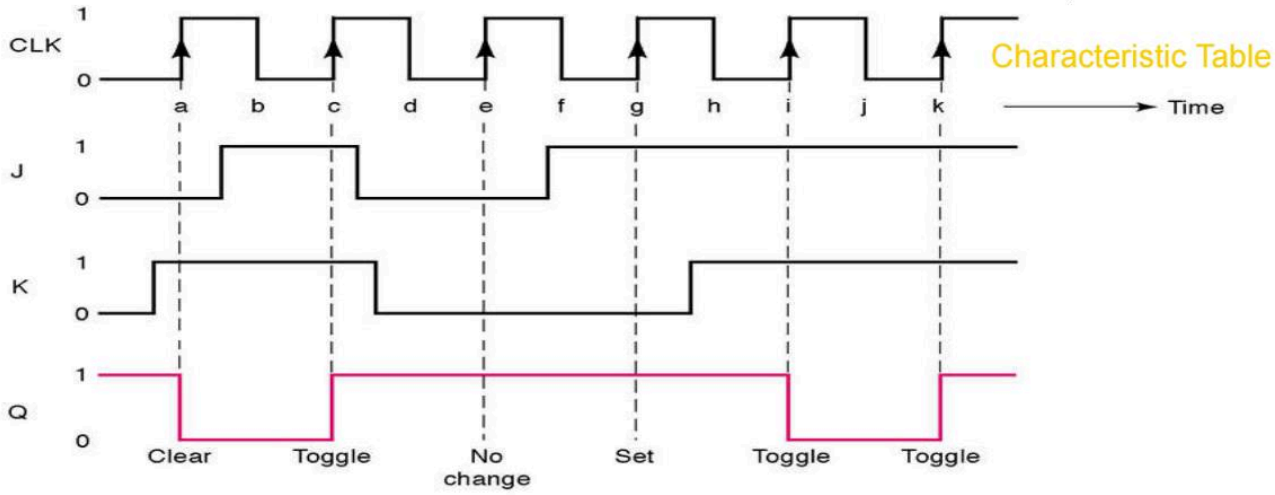
Clocked J-K Flip Flop

J: set, K: reset, if J=K=1 then toggle output



J	K	CLK	Q
0	0	↑	Q_0 (no change)
1	0	↑	1
0	1	↑	0
1	1	↑	$\overline{Q_0}$ (toggles)

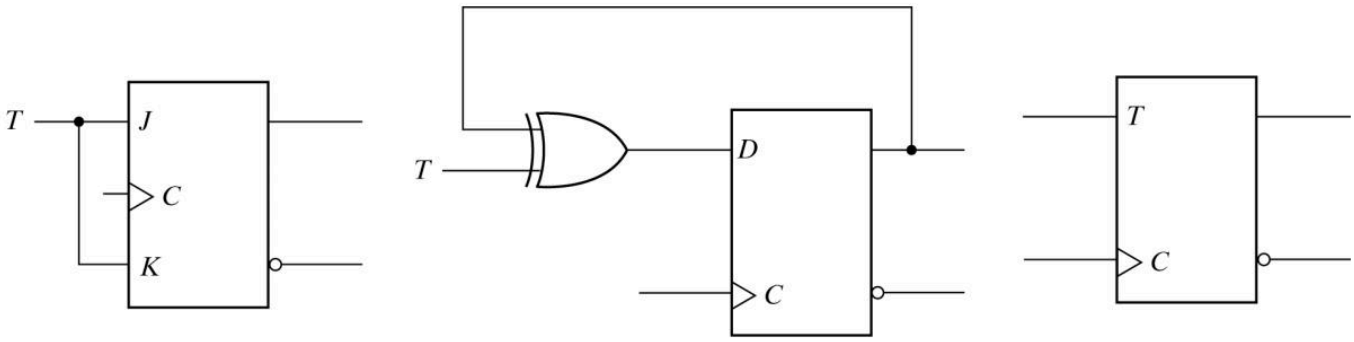
Characteristic Table



Positive Edge-Triggered T Flip-Flop

- Created from JK or D F.F.
- T=0 → No change
- T=1 → invert Q

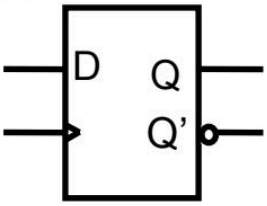
T	CLK	Q	Q′
0		Q_0	Q_0'
1		Q_0'	Q_0



(a) From *JK* flip-flop (b) From *D* flip-flop (c) Graphic symbol

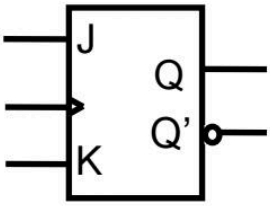
Fig. 5-13 T Flip-Flop

Characteristic Table and Equation



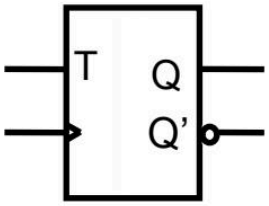
D	Q(t+1)
0	0
1	1

Q(t+1) = D



J	K	Q(t+1)
0	0	Q(t)
0	1	0
1	0	1
1	1	Q'(t)

Q(t+1) = JQ′(t) + K′Q(t)

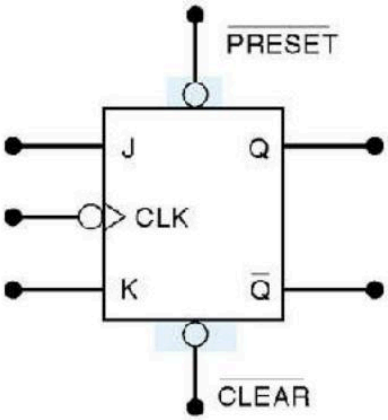


T	Q(t+1)
0	Q(t)
1	Q'(t)

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

Asynchronous Inputs

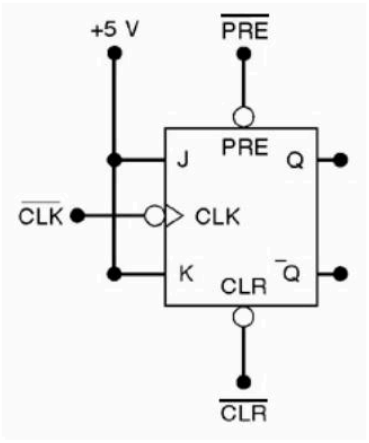
- J, K are *synchronous* inputs
  - Effects on the output are synchronized with the *CLK* input
- *Asynchronous* inputs operate independently of the clock and synchronous inputs
  - Set/reset the FF to 1/0 states *at any time*

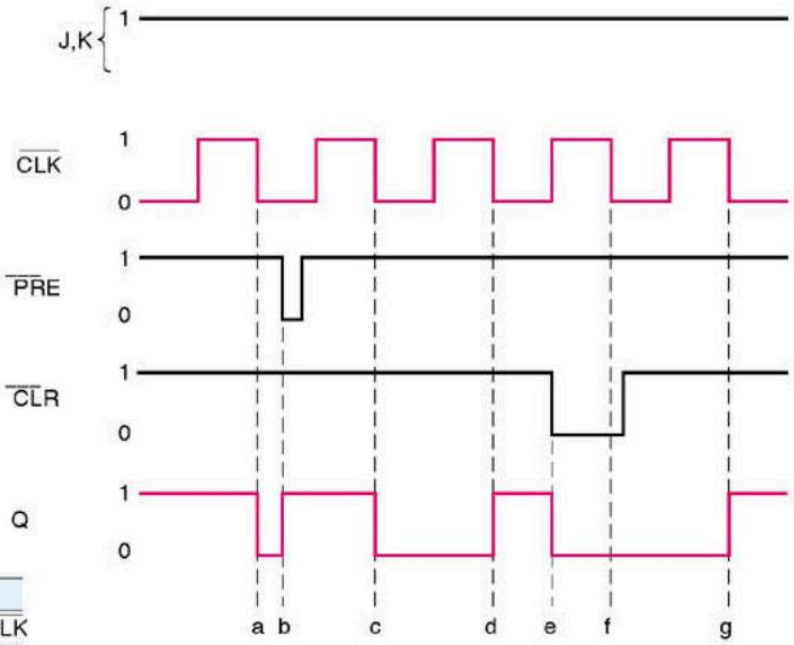


$\overline{\text{PRESET}}$	$\overline{\text{CLEAR}}$	FF response
1	1	Clocked operation*
0	1	Q = 1 (regardless of CLK)
1	0	Q = 0 (regardless of CLK)
0	0	Not used

*Q will respond to J, K, and CLK

Asynchronous Inputs

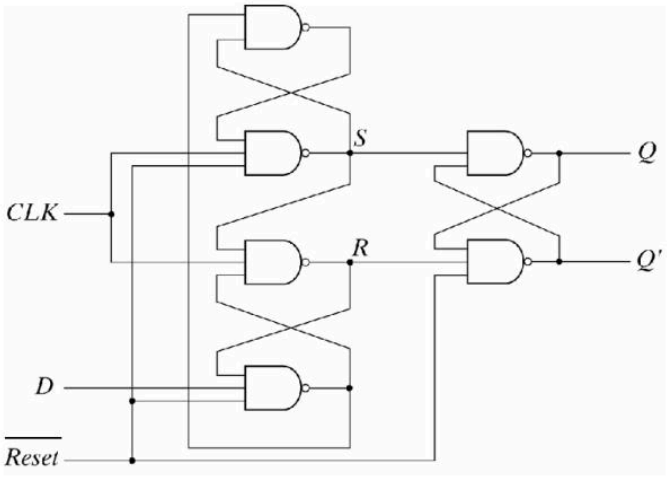


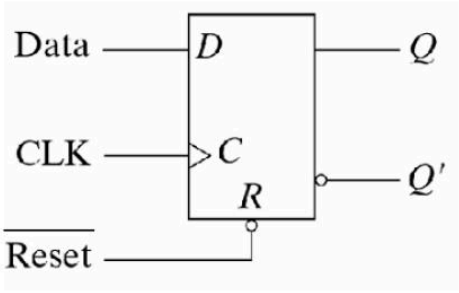


Point	Operation
a	Synchronous toggle on NGT of $\overline{CLK}$
b	Asynchronous set on $\overline{PRE}$ = 0
c	Synchronous toggle
d	Synchronous toggle
e	Asynchronous clear on $\overline{CLR}$ = 0
f	$\overline{CLR}$ over-rides the NGT of $\overline{CLK}$
g	Synchronous toggle

Asynchronous Inputs

- Reset signal (R) is active low
  - R = 0 clears the output Q
- This event can occur at any time, regardless of the value of the CLK

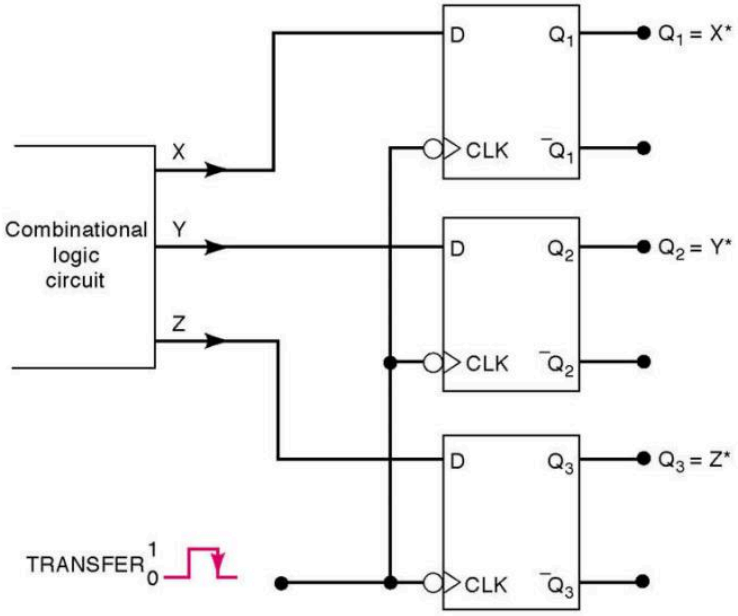




R	C	D	Q	Q′
0	X	X	0	1
1	↑	0	0	1
1	↑	1	1	0

Parallel Data Transfer

- Flip flops store outputs from combinational logic
- Multiple flops can store a collection of data



*After occurrence of NGT

Summary

- Flip flops are powerful storage elements
  - They can be constructed from gates and latches!
- D flip flop is simplest and most widely used
- Asynchronous inputs allow for *clearing* and *presetting* the flip flop output
- Multiple flip flops allow for data storage
  - The basis of computer memory!
- Combine storage and logic to make a computation circuit