

# Lecture 1 – Introduction and Number systems

Dr. Ubaidulla,  
Assistant Professor, SPCRC

Chapter 1 (first half)

# About the course

---

- Name: Digital Systems and Microcontrollers (DSM)
- Textbook:
  - M. Morris Mano and Michael D. Ciletti, “*Digital Design*”
- Logistics:
  - Three 1-hour lectures per week
  - One 3-hour lab per week
  - One 1-hour tut per week
- Faculty: Dr. Ubaidulla (lectures)  
Dr. Harikumar Kandath (labs)

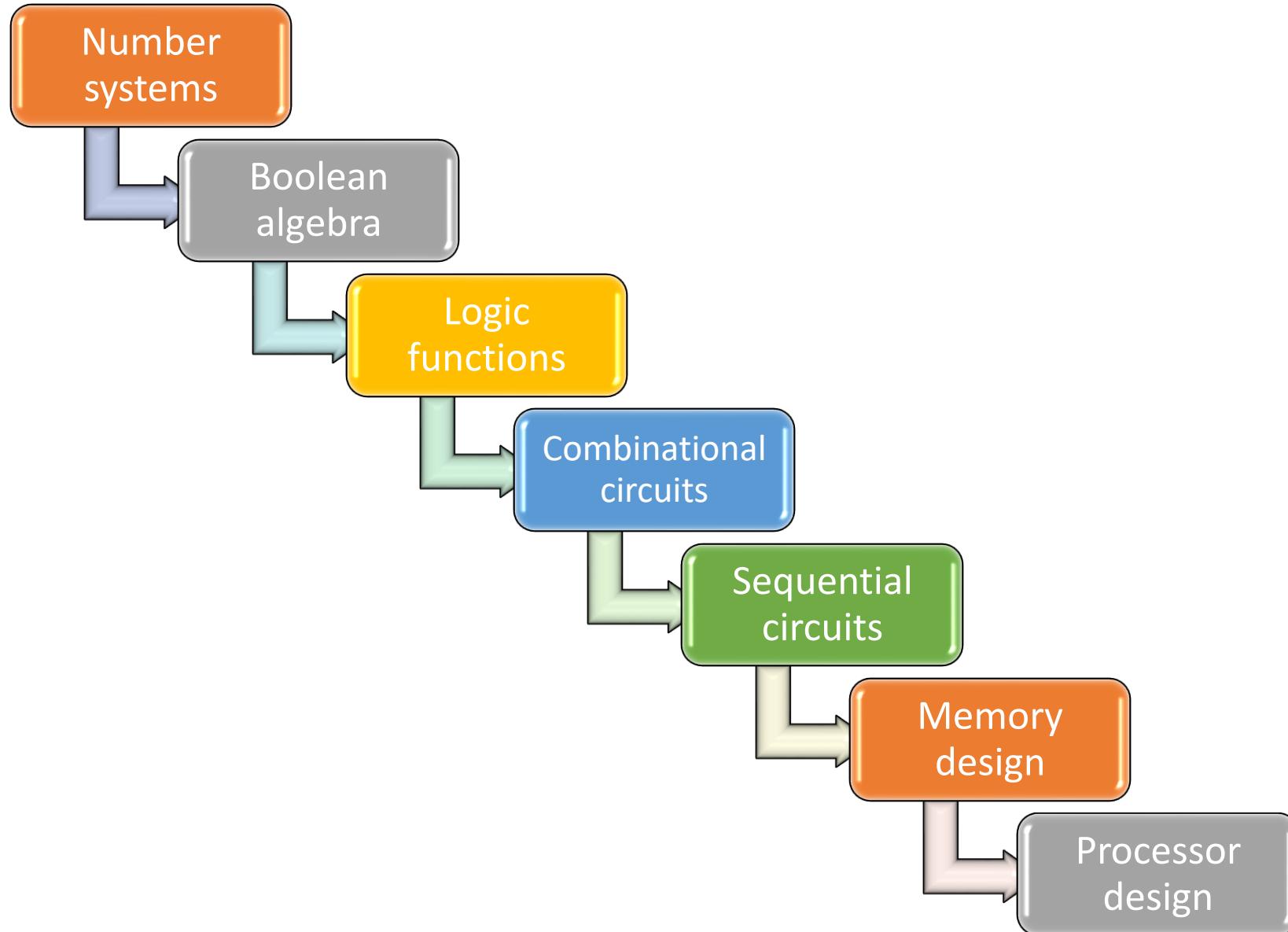
# About the course

---

- Grading scheme:

Quizzes (x2)	10
Midsem	20
Lab reports (x9)	15
Lab exam	20
End semester	35
Total	100

# About the course



# Counting

- Lets take a relook at counting...

0    1    2    3    4    5    6    7    8    9    **10**    **11** ...

- The number system:

- Put symbols in specific places/positions to denote their “power”
- The *base* or the *radix* of the decimal number system is 10

1 0 6 6

$10^3 \ 10^2 \ 10^1 \ 10^0$

1000 100 10 1

$$1 \times 1000 + 0 \times 100 + 6 \times 10 + 6 \times 1 = 1066$$

1 9 4 0

$10^3 \ 10^2 \ 10^1 \ 10^0$

1000 100 10 0

$$1 \times 1000 + 9 \times 100 + 4 \times 10 + 0 \times 1 = 1940$$

# Various number systems

---

- Octal number system

- The base or radix is 8
- The symbols are: 0, 1, 2, 3, 4, 5, 6, 7
- Counting in octal: 0, 1, 2, 3, 4, 5, 6, 7, ?

0, 1, 2, 3, 4, 5, 6, 7, **10, 11**, ...

- Hexadecimal number system

- The base or radix is 16
- The symbols are: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F
- Counting in Hex: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, **10, 11**, ...

# Various number systems

---

- **Hexadecimal number system**
  - The base or radix is 16
  - The symbols are: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F
- **Binary number system**
  - The base or radix is 2
  - The symbols are: 0, 1 (bit)
  - Counting in binary : 0, 1, **10**, **11**,...
- We denote the base of the number using a subscript:  $(10395)_{10}$
- In general a number  $(a_4a_3a_2a_1a_0)_r = a_4r^4 + a_3r^3 + a_2r^2 + a_1r^1 + a_0r^0$

# Conversion to decimal

---

In general a number  $(a_4a_3a_2a_1a_0)_r = a_4r^4 + a_3r^3 + a_2r^2 + a_1r^1 + a_0r^0$

- Octal to decimal:

- $(110)_8 = 1*8^2 + 1*8^1 + 0*8^0 = (72)_{10}$
- $(777)_8 =$
- $(777)_8 = 7*8^2 + 7*8^1 + 7*8^0 = (511)_{10}$

- Hex to decimal:

- $(110)_{16} = 1*16^2 + 1*16^1 + 0*16^0 = (272)_{10}$
- $(BAD)_{16} = ?$
- $(BAD)_{16} = 11*16^2 + 10*16^1 + 13*16^0 = (2989)_{10}$

# Conversions to decimal

---

- Hex to decimal:

- $(110)_{16} = 1*16^2 + 1*16^1 + 0*16^0 = (272)_{10}$
- $(BAD)_{16} = 11*16^2 + 10*16^1 + 13*16^0 = (2989)_{10}$

- Binary to decimal:

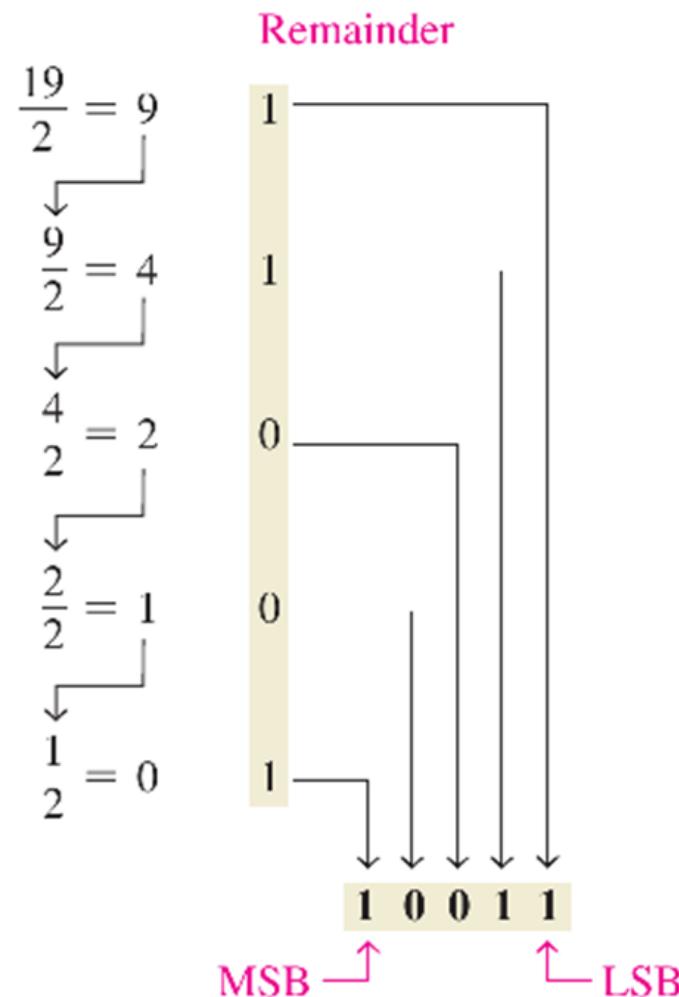
- $(110)_2 = 1*2^2 + 1*2^1 + 0*2^0 = (6)_{10}$
- $(101010)_2 =$
- $(101010)_2 = 1*2^5 + 0*2^4 + 1*2^3 + 0*2^2 + 1*2^1 + 0*2^0 = (42)_{10}$

# Conversions from decimal

- Algorithm:
  - Divide by radix
  - Save the remainder
  - Repeat till quotient '0'
  - Arrange remainders in reverse order

Eg: Convert  $(19)_{10}$  to binary

$$(19)_{10} = (10011)_2$$



MSB – most significant bit  
LSB – least significant bit

# Conversions from Oct/Hex to Binary

---

- From Oct/Hex to binary, we can take a short cut because the bases are  $8=2^3$  and  $16=2^4$  respectively
  - For octal: take each digit and convert it individually into *three* bits
  - For hex: take each digit and convert it individually into *four* bits
- 
- Octal number system
    - $(433)_8 = 100011011$
    - $(70)_8 = 111000$
  - Hexadecimal number system
    - $(DEAD)_{16} = 1101111010101101$
    - $(FEED)_{16} = 1111111011101101$

# Conversions from Binary to Oct/Hex

---

- The reverse course can be taken for converting binary to oct or hex
  - For octal: take *three* bits and convert it individually into a symbol
  - For hex: take *four* bits and convert it individually into a symbol
- 
- Octal number system
    - $(110\textcolor{brown}{101}011)_2 = (\textcolor{brown}{6}\textcolor{red}{5}3)_8$
    - $(1\textcolor{blue}{010}111\textcolor{red}{101})_2 = (\textcolor{blue}{1}2\textcolor{red}{7}5)_8$
  - Hexadecimal number system
    - $(\textcolor{red}{1110}1011)_2 = (\textcolor{red}{EB})_{16}$
    - $(1\textcolor{blue}{1000}0110)_2 = (\textcolor{blue}{1}8\textcolor{red}{6})_{16}$

# Lecture 2 – Binary numbers and representations

Chapter 1

# Recap

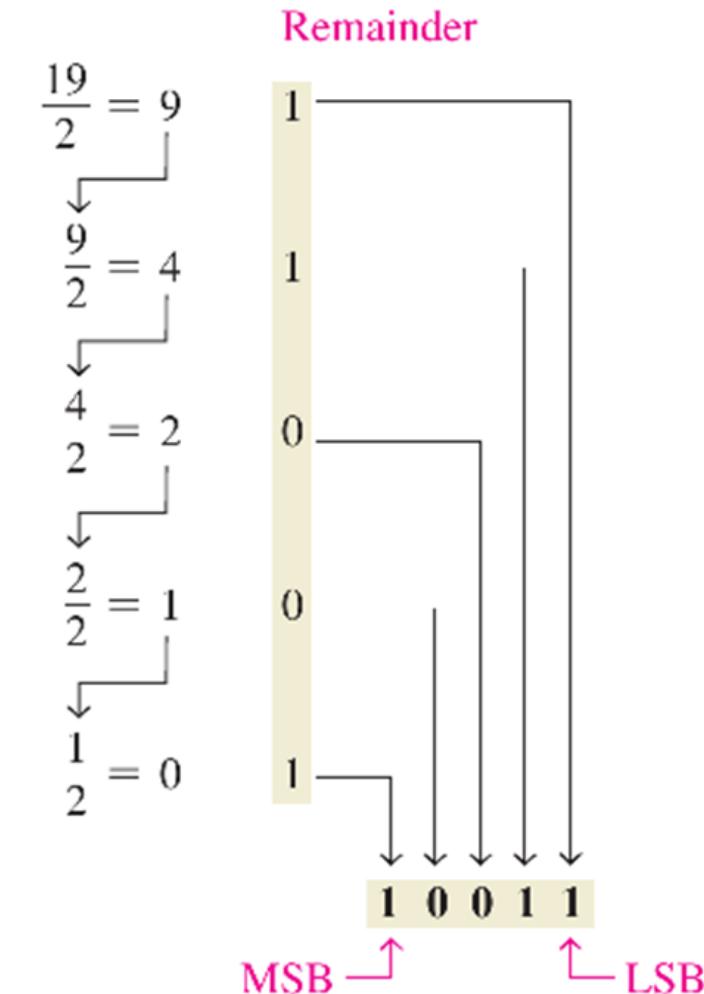
---

- Number Systems
  - Decimal
  - Octal
  - Hex
  - Binary
- Conversion from one base to another
- Need for various number systems
  - $(111111111111)_2 = (\text{FFF})_{16}$

# Recap: Conversions from decimal

- Algorithm:
  - Divide by radix
  - Save the remainder
  - Repeat till quotient '0'
  - Arrange remainders in reverse order

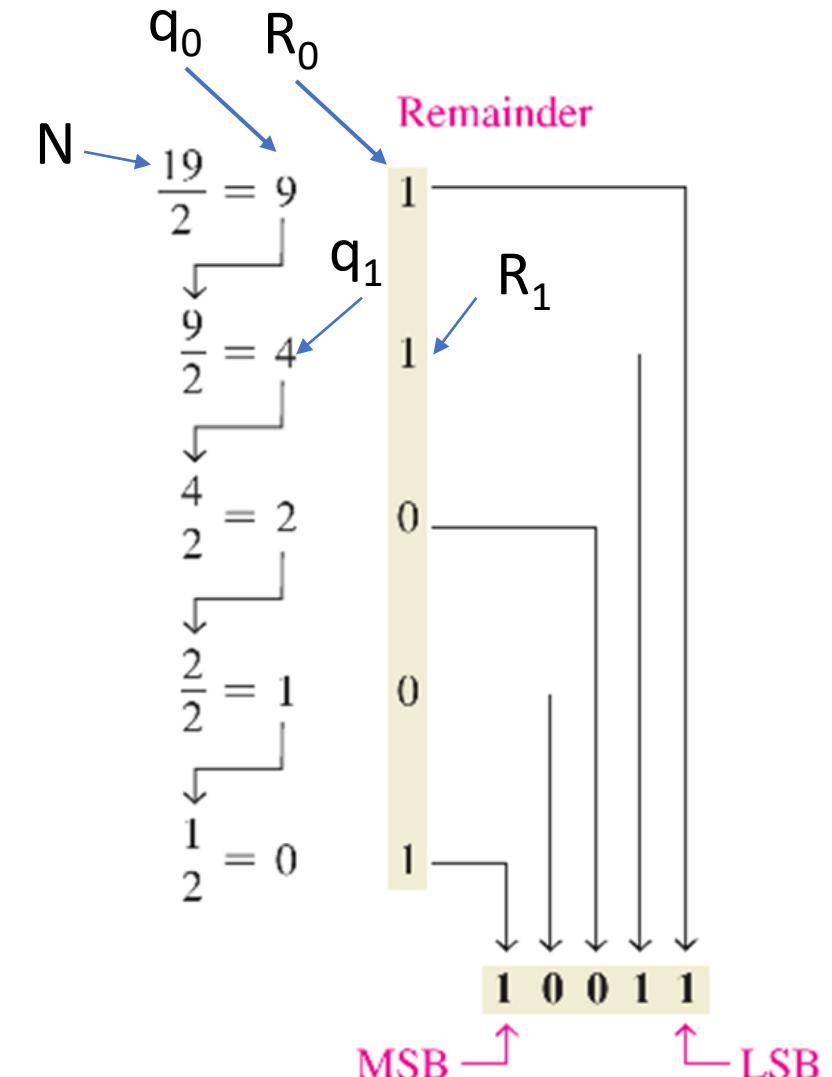
Eg: Convert  $(19)_{10}$  to binary



# Recap

Base conversion by repeated division – position of MSB and LSB

$$\begin{aligned}N &= q_0 r + R_0 \\&= (q_1 r + R_1) r + R_0 \\&= q_1 r^2 + R_1 r + R_0 \\&= (q_2 r + R_2) r^2 + R_1 r + R_0 \\&= q_2 r^3 + R_2 r^2 + R_1 r + R_0 \\&= \dots\dots \\&\quad \dots\dots \\&= 0 * r^{n+1} + R_n r^n + R_{n-1} r^{n-1} + \dots + R_0 r^0\end{aligned}$$



# The “decimal” point

---

$$\begin{aligned}568.23 &= (5 \times 10^2) + (6 \times 10^1) + (8 \times 10^0) + (2 \times 10^{-1}) + (3 \times 10^{-2}) \\&= (5 \times 100) + (6 \times 10) + (8 \times 1) + (2 \times 0.1) + (3 \times 0.01) \\&= \mathbf{500} + \mathbf{60} + \mathbf{8} + \mathbf{0.2} + \mathbf{0.03}\end{aligned}$$

In general:  $a_2a_1a_0.a_{-1}a_{-2}$  can be expressed as  $a_2r^2+a_1r^1+a_0r^0+a_{-1}r^{-1}+a_{-2}r^{-2}$

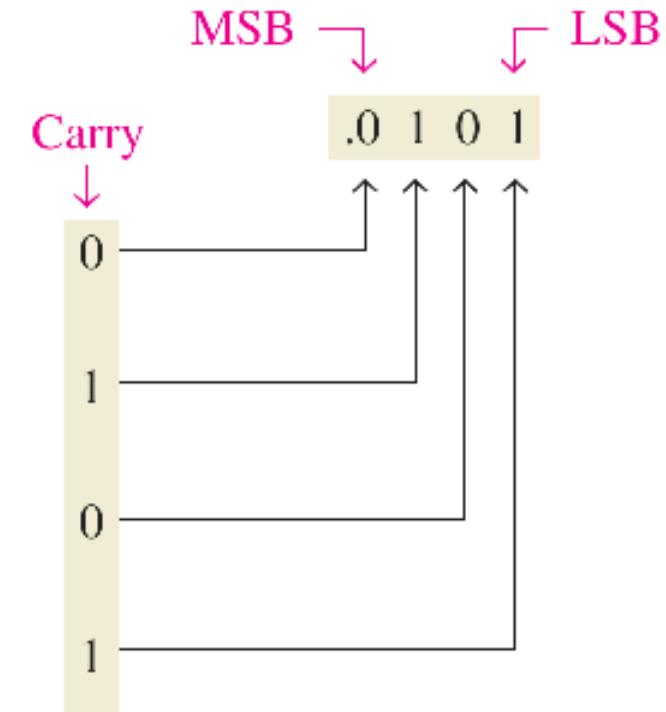
- Binary to decimal:
  - $(1.011)_2 = 1*2^0+0*2^{-1}+1*2^{-2}+1*2^{-3}$   
 $= 1+0.25+0.125$   
 $= (1.375)_{10}$

# The “decimal” point

Convert 0.3125 into binary:

$$\begin{aligned}0.3125 \times 2 &= 0.\underline{625} \\0.625 \times 2 &= 1.\underline{25} \\0.25 \times 2 &= 0.\underline{50} \\0.50 \times 2 &= 1.\underline{00}\end{aligned}$$

Continue to the desired number of decimal places  
or stop when the fractional part is all zeros.



# The “decimal” point

Convert  $(0.65626)_{10}$  to binary:

0.65626 * 2	1.31252	1
0.31252 * 2	0.62504	0
0.62504 * 2	1.25008	1
0.25008 * 2	0.50016	0
0.50016 * 2	1.00032	1
....	....	

$$(0.65626)_{10} = (0.10101 \dots)_2$$

Binary to Hex conversion:

$$(10 \quad 1100 \quad 0110 \quad 1011 \quad \cdot \quad 1111 \quad 0010)_2 = (2C6B.F2)_{16}$$

2      C      6      B      ·      F      2

# The “decimal” point

---

Convert  $(0.510)_{10}$  to octal:

$$0.513 * 8 = 4.104$$

$$0.104 * 8 = 0.832$$

$$0.832 * 8 = 6.656$$

$$0.656 * 8 = 5.248$$

$$0.248 * 8 = 1.984$$

$$0.984 * 8 = 7.872$$

$$(0.510)_{10} = (0.406517...)_8$$

# Addition in various number systems

- Octal number system

- (167)<sub>8</sub> + (765)<sub>8</sub>

$$\begin{array}{r} & 1 & 1 \\ & 1 & 6 & 7 \\ + & 7 & 6 & 5 \\ \hline & 1 & 1 & 5 & 4 \end{array}$$

- Hexadecimal number system

- (BA3)<sub>16</sub> + (5DE)<sub>16</sub>

$$\begin{array}{r} & 1 & 1 \\ & B & A & 3 \\ + & 5 & D & E \\ \hline & 1 & 1 & 8 & 1 \end{array}$$

- Binary number system

- (1101)<sub>2</sub> + (111)<sub>2</sub> = (10100)<sub>2</sub>

# Multiplication

- Binary number system

$$\begin{array}{r} 1010 \\ \times 1011 \\ \hline 1010 \\ 1010 \\ 0000 \\ 1010 \\ \hline 1101110 \end{array}$$

→      **Multiplicand**  
→      **Multiplier**  
→      **Partial product 1**  
→      **Partial product 2**  
→      **Partial product 3**  
→      **Partial product 4**

- Example:

- $(111)_2 * (110)_2 = (101010)_2$

# Complements of numbers

---

- *Complements* are used in digital computers to simplify the subtraction operation and for logical manipulation
- Simplifying operations leads to simpler, less expensive circuits to implement the operations
- There are two types of complements for each base- $r$  system:
  1. The *radix complement* [  $r$ 's complement] – called the 10's complement in decimal, 2's complement in binary and so on
  2. The *diminished radix complement* [  $(r-1)$ 's complement] – called the 9's complement in decimal, 1's complement in binary and so on

# Diminished radix complement

---

- Given an  $n$ -digit number  $N$  in base  $r$ , the  $(r - 1)$ 's complement of  $N$ , i.e., its **diminished radix complement**, is defined as  $(r^n - 1) - N$
- For decimal numbers, the 9's complement of  $N$  is  $(10^n - 1) - N$
- In this case,  $10^n - 1$  is a number represented by  $n$  9s
  - Eg: if  $n = 4$ , we have  $10^4 = 10,000$  and  $r^n - 1 = 10^4 - 1 = 9999$
  - If  $n=2$ , we have  $10^2 = 100$  and  $r^n - 1 = 10^2 - 1 = 99$
- It follows that the 9's complement of a decimal number is obtained by subtracting each digit from 9

# Diminished radix complement

---

It follows that the 9's complement of a decimal number is obtained by subtracting each digit from 9:

- 9's complement of 76 =  $99 - 76 = 23$
- 9's complement of 1242 =  $9999 - 1242 = 8757$
- 9's complement of 99981 is  $99999 - 99981 = 18$

# Diminished radix complement

---

- For n-bit binary numbers, the 1's complement of  $N$  is  $(2^n - 1) - N$ .
- Again,  $(2^n - 1)$  is a binary number represented by  $n$  1s
  - For example, if  $n = 4$ , we have  $2^4 = (10000)_2$  and  $2^4 - 1 = (1111)_2$ .
  - If  $n=2$ , we have  $2^2 = (100)_2$ , and  $2^2 - 1 = 11$
- 1's complement of a binary number can be obtained by subtracting each bit from 1

# Diminished radix complement

---

- 1's complement of a binary number can be obtained by subtracting each bit from 1
- However, when subtracting binary digits from 1, we can have either  $1 - 0 = 1$  or  $1 - 1 = 0$ , which causes the bit to change from 0 to 1 or from 1 to 0, respectively
- Therefore, **the 1's complement of a binary number is formed by changing 1's to 0's and 0's to 1's.**
- Examples of 1's complement:
  - 1's complement of 1011000 = 1111111 – 1011000 = **0100111**
  - 1's complement of 100 = 111 – 100 = **011**

# Lecture 3 – Binary Representation

Chapter 1

# Diminished radix complement

---

- Given an n-digit number  $N$  in base  $r$ , the  $(r - 1)$ 's complement of  $N$ , i.e., its **diminished radix complement**, is defined as  $(r^n - 1) - N$
- For decimal numbers, the 9's complement of  $N$  is  $(10^n - 1) - N$
- In this case,  $10^n - 1$  is a number represented by  $n$  9s
  - Eg: if  $n = 4$ , we have  $10^4 = 10,000$  and  $r^n - 1 = 10^4 - 1 = 9999$
  - If  $n=2$ , we have  $10^2 = 100$  and  $r^n - 1 = 10^2 - 1 = 99$
- It follows that the 9's complement of a decimal number is obtained by subtracting each digit from 9
  - Eg: 9's complement of 76 =  $99 - 76 = 23$
  - 9's complement of 1242 =  $9999 - 1242 = 8757$
  - 9's complement of 99981 is  $99999 - 99981 = 18$

# Diminished radix complement

---

- For n-bit binary numbers, the 1's complement of  $N$  is  $(2^n - 1) - N$ .
- Again,  $(2^n - 1)$  is a binary number represented by  $n$  1s
  - For example, if  $n = 4$ , we have  $2^4 = (10000)_2$  and  $2^4 - 1 = (1111)_2$ .
  - If  $n=2$ , we have  $2^2 = (100)_2$ , and  $2^2 - 1 = 11$ ,
- 1's complement of a binary number can be obtained by subtracting each bit from 1
- However, when subtracting binary digits from 1, we can have either  $1 - 0 = 1$  or  $1 - 1 = 0$ , which causes the bit to change from 0 to 1 or from 1 to 0, respectively
- Therefore, **the 1's complement of a binary number is formed by changing 1's to 0's and 0's to 1's.**
- Examples of 1's complement:
  - 1's complement of  $1011000 = 1111111 - 1011000 = 0100111$
  - 1's complement of  $100 = 111 - 100 = 011$

# Radix complement

---

- The  $r$ 's complement of an  $n$ -digit number  $N$  in base  $r$  is defined as  $r^n - N$  for  $N \neq 0$  and as  $0$  for  $N = 0$
- $r$ 's complement can be obtained by adding 1 to the  $(r - 1)$ 's complement, since  $r^n - N = [(r^n - 1) - N] + 1$
- Thus, the 10's complement of decimal 2389 is  $7610 + 1 = 7611$  and is obtained by adding 1 to the 9's complement value
- The 2's complement of binary 101100 is  $010011 + 1 = 010100$  and is obtained by adding 1 to the 1's-complement value
- Examples:
  - $(66772)_{10}$ 
    - $33227 + 1 = 33228$
  - $(10011)_2$ 
    - $01100 + 1 = 01101$

# Some notes on complements

---

- If the original number  $N$  contains a radix point, the point should be removed temporarily in order to form the  $r$ 's or  $(r - 1)$ 's complement
- The radix point is then restored to the complemented number in the same relative position
- Example: 9's complement and 10's complement of  $(82.314)_{10}$ 
  - 9's complement : 17.685
  - 10's complement: 17.686

# Some notes on Complements

---

- **The complement of the complement restores the number to its original value**
- $r$ 's complement of  $N$  is  $r^n - N$ , so that the complement of the complement is  $r^n - (r^n - N) = N$  and is equal to the original number
- $(r-1)$ 's complement of  $N$  is  $r^n - 1 - N$ , so that the complement of the complement is  $(r^n - 1) - (r^n - 1 - N) = N$  and is equal to the original number

# Subtraction with Radix complements

- Subtraction using method of borrowing is less efficient when implemented with digital hardware
- Consider subtraction  $M - N$  in base  $r$
- Here is the algorithm using Radix complement:
  1. Take radix complement of subtrahend  $N$ :  $r^n - N$
  2. Add this to  $M$ :  $(r^n - N) + M = r^n + (M - N) = r^n - (N - M)$
  3. If you get a carry in the  $(n+1)^{\text{th}}$  digit,
    - then the result is positive, discard the carry and you are done
  4. If you **do not** get a carry in the  $(n+1)^{\text{th}}$  digit,
    - then the result is **negative**. Take the radix complement of the number to get the answer, then put a negative sign

# Subtraction with Radix complements

---

## Subtraction using 10's complement

- $(4637)_{10} - (2579)_{10}$ 
  1. 10's complement of 2579 = 7421
  2.  $4637 + 7421 = 12058$  ( $r^n + (M - N)$ )
  3. Result after removing the end carry : 2058
  
- $(2579)_{10} - (4637)_{10}$ 
  - 10's complement of 4637 = 5363
  - $2579 + 5363 = 7942$  ( $r^n - (N - M)$ )
  - No end carry. Hence, answer is  $-(10's \text{ complement of } 7942) = -2058$

# Binary subtraction with complements

Perform the following subtractions using 2's complement method:

- $(110001)_2 - (010100)_2$  [  $(49)_{10} - (20)_{10}$  ]

- 2's complement of 010100 = 101100

$$\begin{array}{r} 110001 \\ + \underline{101100} \\ \hline 1011101 \end{array}$$

- Result obtained by dropping end carry : 011101  $[(29)_{10}]$

- $(010110)_2 - (100)_2$  [  $(22)_{10} - (4)_{10}$  ]

- 2's complement of 000100 = 111100

$$\begin{array}{r} 010110 \\ + \underline{111100} \\ \hline 1010010 \end{array}$$

- Result obtained by dropping end carry : 010010  $[(18)_{10}]$

# Binary subtraction with complements

Perform the following subtractions using 2's complement method:

- $(1000011)_2 - (1010100)_2$  [  $(67)_{10} - (84)_{10}$  ]

- 2's complement of  $1010100 = 0101100$

$$\begin{array}{r} 1000011 \\ + \underline{0101100} \\ 1101111 \end{array}$$

- No end carry  $\Rightarrow$  the answer is  $- (2\text{'s complement of } 1101111) = -0010001$  [  $-(17)_{10}$  ]

# Subtraction with Diminished radix complements

---

- Lets assume we have to perform  $M - N$  in base r
- Here is the algorithm using Diminished radix complement:
  1. Take diminished radix complement of N:  $r^n - 1 - N$
  2. Add this to M:  $r^n - 1 - N + M = r^n + (M - N - 1) = (r^n - 1) - (N - M)$
  3. If you get a carry in the  $(n+1)^{\text{th}}$  digit,
    - then the result is positive, ***add the carry to the result*** and you are done
  4. If you ***do not*** get a carry in the  $(n+1)^{\text{th}}$  digit,
    - then the result is **negative**. Take the diminished radix complement of the number to get the answer, then put a negative sign

# Subtraction with Diminished radix complements

---

9's complement subtraction:

- $(76425)_{10} - (28321)_{10}$ 
  - 9's complement of 28321 is 71678
  - $76425 + 71678 = 148103; [ r^n + (M - N - 1) ]$   
drop the carry and add 1 to 48103 ; sum is 48104
  - End carry => result is positive
- $(2124)_{10} - (9667)_{10}$ 
  - 9's complement of 9667 is 0332
  - $2124 + 0332 = 2456; [ r^n - 1 - (N - M) ]$
  - No end carry => answer is – (9's complement of 2456) = – 7543

# Subtraction with Diminished radix complements

Perform the following subtractions using 1's complement method:

- $(101011)_2 - (111001)_2$  [  $(43)_{10} - (57)_{10}$  ]

- 1's complement of 111001 = 000110

$$\begin{array}{r} 101011 \\ + \underline{000110} \\ 110001 \end{array}$$

- No end carry => answer is  $- (1\text{'s complement of } 110001) = - 1110$

- $(1)_2 - (10100)_2$

$$\begin{array}{r} 1 \\ + \underline{01011} \\ 01100 \end{array}$$

- No end carry => answer is  $- (1\text{'s complement of } 01100) = - 10011$

# Binary subtraction with complements

Perform the following subtractions using 2's complement method:

- $(110001)_2 - (010100)_2$  [  $(49)_{10} - (20)_{10}$  ]

- 2's complement of 010100 = 101100

$$\begin{array}{r} 110001 \\ + \underline{101100} \\ \hline 1011101 \end{array}$$

- Result obtained by dropping end carry : 011101  $[(29)_{10}]$

- $(010110)_2 - (100)_2$  [  $(22)_{10} - (4)_{10}$  ]

- 2's complement of 000100 = 111100

$$\begin{array}{r} 010110 \\ + \underline{111100} \\ \hline 1010010 \end{array}$$

- Result obtained by dropping end carry : 010010  $[(18)_{10}]$

# Binary subtraction with complements

Perform the following subtractions using 2's complement method:

- $(1000011)_2 - (1010100)_2$

- 2's complement of 1010100 = 0101100

$$\begin{array}{r} 1000011 \\ + \underline{0101100} \\ 1101111 \end{array}$$

- No end carry  $\Rightarrow$  the answer is  $- (2\text{'s complement of } 1101111) = - 0010001$

# Representation of negative numbers

---

- In ordinary arithmetic, a negative number is indicated by a minus sign and a positive number by a plus sign
- Computers must represent everything with binary digits
- It is customary to represent the sign with a bit placed in the leftmost position of the number
- The convention is to make the sign bit 0 for positive and 1 for negative
- This can be done using:
  1. Signed magnitude representation
  2. Signed complement representation
    1. Signed 1's complement representation
    2. Signed 2's complement representation

# Lecture 4 – Binary Representation

Chapter 1

# Representation of negative numbers

- In ordinary arithmetic, a negative number is indicated by a minus sign and a positive number by a plus sign
- Computers must represent everything with binary digits
- It is customary to represent the sign with a bit placed in the leftmost position of the number
- The convention is to make the sign bit 0 for positive and 1 for negative
- This can be done using:
  1. Signed magnitude representation
  2. Signed complement representation
    1. Signed 1's complement representation
    2. Signed 2's complement representation

# Signed-magnitude representation

- In this notation, the number consists of a magnitude and a symbol ( + or - ) or a bit (0 or 1) indicating the sign
- This is similar to the representation of signed numbers used in ordinary arithmetic
- Eg: 01001 represents +9, and 11001 represents -9 in signed magnitude representation
  - -9 is obtained from +9 by changing only the sign bit from 0 to 1
- Weird: +0 is represented as 0000 and minus 0 is represented as 1000. So, two representations for zero – inefficient and may cause errors

# Signed complement representation

- In digital hardware, it is more convenient to use *signed complement* system, for representing negative numbers
- In this system, a **negative number is indicated by its complement**
  - signed-complement system negates a number by taking its complement
- Since positive numbers always start with 0 (plus) in the leftmost position (in all representations), it follows that the complement will always start with a 1, indicating a negative number
- In signed-1's-complement, -9 is obtained by taking the 1's complement of all the bits of +9 (01001), including the sign bit, ie, **10110**
- The signed-2's-complement representation of -9 is obtained by taking the 2's complement of +9 , including the sign bit, ie **10111**

# Signed complement representation

- Find signed 2's complement representation in 4 bits:
  - +3  
 $+3 = 0011$
  - -7  
 $(7)_{10} = 0111 \Rightarrow -7 = 1001$
  - 0  
 $0 = 0000$
- -39 in 8 bit:
  - 10100111 (signed magnitude)
  - 11011000 (signed 1's complement)
  - 11011001 (signed 2's complement)

# Signed complement representation

Find the decimal numbers for the following signed 2's complement representation in 4 bits:

- $(1100)_2$ 
  - $1100 = -4$
- $(1111)_2$ 
  - $1111 = -1$
- $(0000)_2$ 
  - $0000 = 0$
- $(1000)_2$ 
  - $1000 = -8$

# Interpretations for 4 bit binary numbers

<b>Decimal</b>	<b>Signed-2's Complement</b>	<b>Signed-1's Complement</b>	<b>Signed Magnitude</b>
+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	—	—

# Signed addition

- *If the numbers are represented in memory in 2's complement form, the sign of the sum takes care of itself! The result is also in 2's complement representation*
- The sign bit is to be included in the addition. If there is a carry, it is discarded
- Examples in 4-bit signed 2's complement representation:

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

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

$$\begin{array}{r} - 6 \quad 11111010 \\ +13 \quad 00001101 \\ \hline + 7 \quad 00000111 \end{array}$$

$$\begin{array}{r} - 6 \quad 11111010 \\ -13 \quad 11110011 \\ \hline -19 \quad 11101101 \end{array}$$

# Signed addition

- In order to obtain a correct answer, we must ensure that the result has a sufficient number of bits to accommodate the sum
- If we start with two  $n$ -bit numbers and the sum occupies  $n + 1$  bits, we say that an overflow occurs

$$\begin{array}{r} 01111101 \\ + 00111010 \\ \hline 10110111 \end{array} \quad \begin{array}{r} 125 \\ + 58 \\ \hline 183 \end{array}$$

Sign incorrect \_\_\_\_\_  
Magnitude incorrect \_\_\_\_\_

# Signed subtraction

- Subtraction of two signed binary numbers when negative numbers are in 2's-complement form is simple and can be stated as follows:
  - Take the 2's complement of the subtrahend (including the sign bit) and add it to the minuend (including the sign bit)
  - A carry out is discarded
- This works because:  $M - N = M + (-N)$
- Example :  $8 - 3 = 8 + (-3)$

$$\begin{array}{r} 00001000 \\ + 11111101 \\ \hline \text{Discard carry} \longrightarrow 1\ 00000101 \end{array}$$

Minuend (+8)  
2's complement of subtrahend (-3)  
Difference (+5)

# Lecture 5 – Binary Codes

Chapter 1

# Range of signed integer numbers

- Byte – group of 8 bits
  - kilobyte (KB) =  $2^{10} = 1024$  bytes
  - megabyte (MB) =  $2^{20} = 1,048,576$  bytes
- For 2's complement signed numbers, the range of values for n-bit numbers is:  $-(2^{n-1})$  to  $+(2^{n-1} - 1)$ 
  - Eg: with 2 bytes, range is -32,768 to +32,767
- How to represent large integers and numbers with integer and fractional parts?
  - Fixed point Vs Floating point representation

# Representing fractions (real numbers)

- We need to operate with fractions all the time
- This means we need a method to store/represent them in binary
- The simplest way is to have a “fixed” point representation, where the binary point is assumed to be fixed at a certain location
  - For example, for an 4-bit system, if given fixed-point representation is **ii.ff**, then the smallest positive value is 00.01 and maximum value is 11.11
  - The point is not actually stored – it is assumed to be there
- There are three parts of a fixed-point number representation: **the sign field, integer field, and fractional field** – which means we can also have signed magnitude fixed point numbers and signed complement fixed point numbers

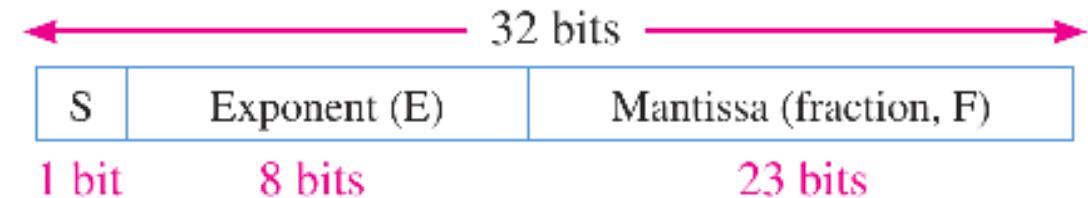
# Fixed point representation

- The advantage of using a fixed-point representation is performance and ease of arithmetic
- The disadvantage is relatively limited range of values that they can represent
- So, it is usually inadequate for numerical analysis as it does not allow enough numbers and accuracy
- For instance, using 32-bit format: the 1 bit sign bit, 15 bits for integer, and 16 bits for the fractional part, the smallest positive number is  $2^{-16} \approx 0.000015$ , and the largest positive number is  $\approx 32768$

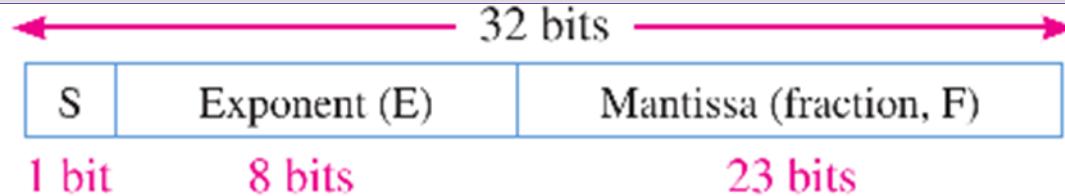
Smallest	0	0000000000000000	0000000000000001
	Sign bit	Integer part	Fractional part
Largest	0	1111111111111111	1111111111111111
	Sign bit	Integer part	Fractional part

# Floating point representation

- This representation does not reserve a specific number of bits for the integer part or the fractional part
- Number is first converted to the form  $N = M * r^E$  and store mantissa  $M$  and exponent  $E$  as binary
  - Eg:  $241,506,800 = 0.2415068 * 10^9$
- Clearly, a large mantissa and small exponent can give both high precision and high range
- For binary floating-point numbers, the format is defined by [IEEE Standard 754-1985](#) in three forms: single-precision (32 bits), double-precision (64 bits), and extended-precision (80 bits).

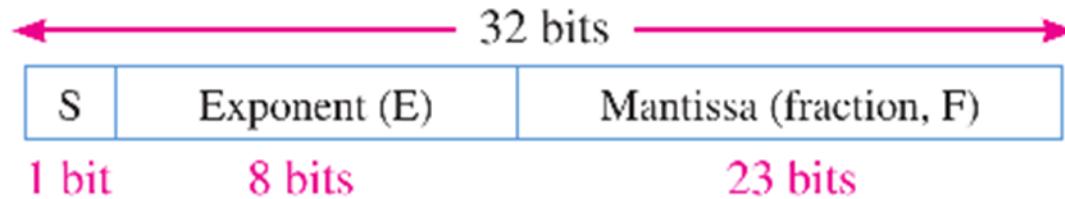


# Single-Precision floating point representation



- In the mantissa or fractional part, the binary point is understood to be to the left of the 23 bits.
- In any binary number the left-most (most significant) bit is always a 1. Therefore, this 1 is understood to be there although it does not occupy an actual bit position. So, effectively 24 bits
- The eight bits in the exponent represent a *biased exponent* obtained by adding 127 to the actual exponent. This is to allow very large or very small numbers without requiring a separate sign bit for the exponents

# Single-Precision floating point representation



- Eg: Represent 1011010010001 in single-precision floating point format
  - $1011010010001 = 1.011010010001 \times 2^{12}$
  - Positive number, hence sign bit  $S=0$
  - Mantissa = 011010010001 [ *MSB '1' is not explicitly represented*]
  - $E = \text{biased exponent} = \text{Actual exponent} + 127 = 12 + 127 = 139 = 10001011.$

S	E	F
0	10001011	01101001000100000000000

# Single-Precision floating point representation

Value of floating point number is given by

$$\text{Number} = (-1)^S(1 + F)2^{E-127}$$

Eg:

S	E	F
1	10010001	10001110001000000000000

$$\begin{aligned}\text{Number} &= (-1)^1 (1.10001110001)(2^{145-127}) \\ &= (-1)(1.10001110001)(2^{18}) = -110001110001000000\end{aligned}$$

# Binary codes

---

- Any discrete element of information that is distinct among a group of quantities can be represented with a binary code (i.e., a pattern of 0's and 1's)
- Binary codes merely change the symbols, not the meaning of the elements of information that they represent
- If we inspect the bits of a computer at random, we will find that most of the time they represent some type of coded information rather than binary numbers
- An  $n$ -bit binary code can represent up to  $2^n$  distinct elements of the set that is being coded

# Binary Coded Decimal (BCD)

- The code most commonly used for the decimal digits is the straight binary assignment and is called *binary-coded decimal* (BCD)
- A binary code will have some unassigned bit combinations if the number of elements in the set is not a multiple power of 2
- The 10 decimal digits form such a set
- A binary code that distinguishes among 10 elements must contain at least four bits, but 6 out of the 16 possible combinations remain unassigned

## Binary-Coded Decimal (BCD)

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

# Binary Coded Decimal (BCD)

- A number with  $k$  decimal digits will require  $4k$  bits in BCD
  - Eg: 396 is represented in BCD with 12 bits as 0011 1001 0110, with each group of 4 bits representing one decimal digit
- A BCD number greater than 10 looks different from its equivalent binary number, even though both contain 1's and 0's
- Moreover, **the binary combinations 1010 through 1111 are not used and have no meaning in BCD**

## *Binary-Coded Decimal (BCD)*

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

# American Standard Code for Information Interchange (ASCII)

- Many applications of digital computers require the handling not only of numbers, but also of other characters or symbols, such as the letters of the alphabet
- The standard binary code for the alphanumeric characters is the [American Standard Code for Information Interchange \(ASCII\)](#), which uses seven bits to code 128 characters
- The seven bits of the code are designated by  $b_1$  through  $b_7$ , with  $b_7$  the most significant bit
- The letter A, for example, is represented in ASCII as 1000001
- The ASCII code also contains 94 graphic characters that can be printed and 34 nonprinting characters used for various control functions
- The graphic characters consist of the 26 uppercase letters (A through Z), the 26 lowercase letters (a through z), the 10 numerals (0 through 9), and 32 special printable characters, such as %, \*, and \$

# ASCII code

$b_4b_3b_2b_1$	$b_7b_6b_5$							
	000	001	010	011	100	101	110	111
0000	NUL	DLE	SP	0	@	P	`	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	"	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EOT	DC4	\$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	'	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[	k	{
1100	FF	FS	,	<	L	\	l	
1101	CR	GS	-	=	M	]	m	}
1110	SO	RS	.	>	N	^	n	~
1111	SI	US	/	?	O	-	o	DEL

Eg: B is encoded as 1000010

# Lecture 6 – Boolean algebra

Chapter 2

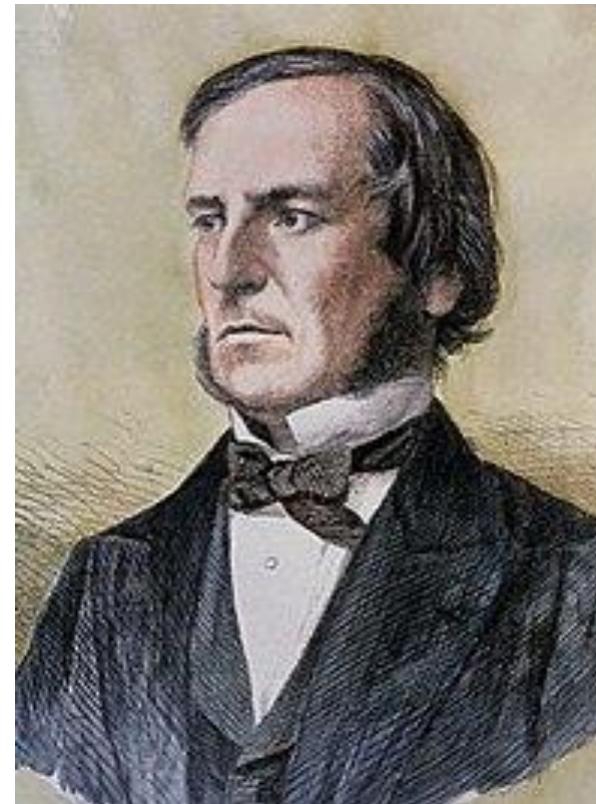
# Binary logic

---

- Binary logic deals with variables that take on two discrete values and with operations that assume logical meaning
- The two values the variables assume may be called by different names (*true* and *false*, *yes* and *no*, etc.), but for our purpose, it is convenient to think in terms of bits and assign the values 1 and 0
- Binary logic consists of binary variables and a set of logical operations
- The variables are designated by letters of the alphabet, such as  $A$ ,  $B$ ,  $C$ ,  $x$ ,  $y$ ,  $z$ , etc., with each variable having two and only two distinct possible values: 1 and 0

# Boolean algebra

- The system for formalization of binary logic came much before their applications in electronics/computers
- Boolean algebra was introduced by George Boole in his first book *The Mathematical Analysis of Logic* (1847)
- In the 1930s, while studying switching circuits, Claude Shannon observed that one could also apply the rules of Boole's algebra in this setting, and he introduced switching algebra as a way to analyze and design circuits by algebraic means in terms of logic gates.



George Boole



Claude Shannon

# Basic operations

- **NOT:** This operation is represented by a prime (sometimes by an overbar). For example,  $z = x'$  (or  $z = \bar{x}$  ); meaning that  $z$  is what  $x$  is not
- In other words, if  $x = 1$ , then  $z = 0$ , but if  $x = 0$ , then  $z = 1$
- The NOT operation is also referred to as the *complement* operation, since it changes a 1 to 0 and a 0 to 1, i.e., the result of complementing 1 is 0, and vice versa
- **AND:** This operation is represented by a dot or by the absence of an operator
- For example,  $z = x \cdot y$  or  $z = xy$
- The logical operation AND is interpreted to mean that  $z = 1$  if and only if  $x = 1$  and  $y = 1$ ; otherwise  $z = 0$
- **OR:** This operation is represented by a plus sign. For example,  $z = x + y$ , meaning that  $z = 1$  if  $x = 1$  or if  $y = 1$  or if both  $x = 1$  and  $y = 1$ . If both  $x = 0$  and  $y = 0$ , then  $z = 0$

# Basic operations

- We make a table of all possible values of the variables and the results of these operations (truth table)

AND		
$x$	$y$	$x \cdot y$
0	0	0
0	1	0
1	0	0
1	1	1

OR		
$x$	$y$	$x + y$
0	0	0
0	1	1
1	0	1
1	1	1

NOT		
$x$	$x'$	
0	1	
1	0	

# Binary logic

---

- *Binary logic* is different from binary numbers although it uses some of the same symbols
- In binary logic, we assume that variables can have ONLY two values – no other values are possible
- In binary numbers variables can have higher values or fraction or negative values, however, that is not the case in binary logic
- For example: in binary numbers,  $(1+1 = 10)_2$ , however, in binary logic,  $1+1 = 1$  because two trues make a true
- There are formal rules and proofs for many of the statements we make in binary logic
- In modern circuits, logic gates are used to perform binary logic using a variety of complex architectures

# Formalization of Boolean algebra

---

- Boolean algebra, like any other deductive mathematical system, may be defined with a set of elements, a set of operators, and a number of unproved axioms or postulates
- A *operator* defined on a set  $S$  of elements is a rule that assigns, to each pair of elements from  $S$ , a unique element from  $S$
- As an example, consider the relation  $a * b = c$ . We say that  $*$  is an operator if it specifies a rule for finding  $c$  from the pair  $(a, b)$  and also if  $a, b, c \in S$
- However,  $*$  is not an operator if  $a, b \in S$ , and if  $c \notin S$ .

# Postulates of Boolean algebra – Closure

---

- A set  $S$  is closed with respect to an operator if, for every pair of elements of  $S$ , the operator specifies a rule for obtaining an element of  $S$
- For example, the set of natural numbers  $N = \{1, 2, 3, 4, \dots\}$  is closed with respect to the operator  $+$  by the rules of arithmetic addition, since, for any  $a, b \in N$ , there is a unique  $c \in N$  such that  $a + b = c$
- The set of natural numbers is *not* closed with respect to the operator  $-$  by the rules of arithmetic subtraction, because  $2 - 3 = -1$  and  $2, 3 \in N$ , but  $(-1) \notin N$
- *The Boolean logic structure is closed with respect to NOT, AND and OR logic operations*

# Postulates of Boolean algebra – Associative law

---

- The operator \* on a set  $S$  is said to be *associative* whenever  $(x * y) * z = x * (y * z)$  for all  $x, y, z, \in S$
- In case of real numbers, the multiplication and addition operations are associative while subtraction and division are not
- Similarly in binary logic, the operators AND and OR are associative
- Thus,  $x$  AND  $(y$  AND  $z)$  is the same as  $(x$  AND  $y)$  AND  $z$
- Also,  $x$  OR  $(y$  OR  $z)$  is the same as  $(x$  OR  $y)$  OR  $z$

# Postulates of Boolean algebra – Commutative law

---

- The operator \* on a set  $S$  is said to be *commutative* whenever  $x * y = y * x$  for all  $x, y \in S$
- In case of real numbers, the multiplication and addition operations are commutative, while subtraction and division are not
- Similarly in binary logic, the operators AND and OR are commutative
- Thus,  $x$  AND  $y$  is the same as  $y$  AND  $x$
- Also,  $x$  OR  $y$  is the same as  $y$  OR  $x$

# Postulates of Boolean algebra – Identity

- A set  $S$  is said to have an *identity element* with respect to an operation  $*$  on  $S$  if there exists an element  $e \in S$  with the property that  $e * x = x * e = x$  for every  $x \in S$
- *Example:* The element 0 is an identity element with respect to the operator  $+$  on the set of integers  $I = \{..., -3, -2, -1, 0, 1, 2, 3, ...\}$ , since  $x + 0 = 0 + x = x$  for any  $x \in I$
- The set of natural numbers,  $N$ , has no identity element w.r.t the operator  $+$ , since 0 is excluded from the set
- In Boolean logic, 0 is the identity element for OR operation and 1 is the identity element for AND operation

# Postulates of Boolean algebra – Distributive

- If  $*$  and  $\&$  are two operators on a set  $S$ ,  $*$  is said to be *distributive* over  $\&$  whenever  $x * (y \& z) = (x * y) \& (x * z)$
- In normal algebra, multiplication is distributive over addition
- In Boolean logic, the operator AND ( $\cdot$ ) is distributive over OR ( $+$ ); that is,  
$$x \cdot (y + z) = (x \cdot y) + (x \cdot z)$$
- Also, the operator OR ( $+$ ) is distributive over AND ( $\cdot$ ); that is,  
$$x + (y \cdot z) = (x + y) \cdot (x + z)$$
- This is counter intuitive!
- An easy way to prove the distributive law is to make a table of all possible values of the variables and their results

# Postulates of Boolean algebra – Distributive

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

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

y + z	x · (y + z)
0	0
1	0
1	0
1	0
0	0
1	1
1	1
1	1

x · y	x · z	(x · y) + (x · z)
0	0	0
0	0	0
0	0	0
0	0	0
0	0	0
0	1	1
1	0	1
1	1	1

# Lecture 7 – Binary Logic

Chapter 2

# Postulates of Boolean algebra – Duality principle

- The *duality principle* states that every algebraic expression deducible from the postulates of Boolean algebra remains valid if the operators and identity elements are interchanged, i.e., AND is changed to OR and vice versa
- In a two-valued Boolean algebra, the identity elements and the elements of the set  $S$  are the same: 1 and 0
- Thus, if the *dual* of an algebraic expression is desired, we simply interchange OR and AND operators and replace 1's by 0's and 0's by 1's

$$(a) \quad x + 0 = x$$

$$(b) \quad x \cdot 1 = x$$

$$(a) \quad x + x' = 1$$

$$(b) \quad x \cdot x' = 0$$

$$(a) \quad x + x = x$$

$$(b) \quad x \cdot x = x$$

$$(a) \quad x + 1 = 1$$

$$(b) \quad x \cdot 0 = 0$$

# Theorems of Boolean algebra

- **Theorem:**  $x + x = x$

$$\begin{aligned}x + x &= (x + x) \cdot 1 \\&= (x + x) \cdot (x + x') \\&= x + x \cdot x' \\&= x + 0 \\&= x\end{aligned}$$

- **Its dual:**  $x \cdot x = x$

# Theorems of Boolean algebra

- **Theorem:**  $x + 1 = 1$

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

- **Its dual:**  $x \cdot 0 = 0$

# Theorems of Boolean algebra

- **Theorem:**  $x + xy = x$  (absorption theorem)

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

- **Its dual:**  $x \cdot (x + y) = x$

# Theorems of Boolean algebra

- **Theorem:**  $(x + y)' = x' \cdot y'$  ( DeMorgan's theorem)
- A simple way to prove the theorem is to prove that  $(x+y)$  and  $x'y'$  are complements of each other so that  $(x+y)'$  is the same as  $x'y'$
- So, we need to prove that  $(x + y) + x'y' = 1$  and  $(x + y) \cdot x'y' = 0$ .  
If they are both true, then  $(x + y)' = x'y'$

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

# Theorems of Boolean algebra

- We proved that  $(x + y) + x'y' = 1$ .
- Let's now show that:  $(x + y) \cdot x'y' = 0$ .
- If they are both true, then  $(x + y)' = x'y'$  ([DeMorgan's theorem](#))

$$\begin{aligned}(x + y) \cdot x'y' &= x \cdot x' \cdot y' + y \cdot x' \cdot y' \\&= (x \cdot x') \cdot y' + x' \cdot (y \cdot y') \\&= 0 \cdot y' + x' \cdot 0 \\&= 0 + 0 \\&= 0\end{aligned}$$

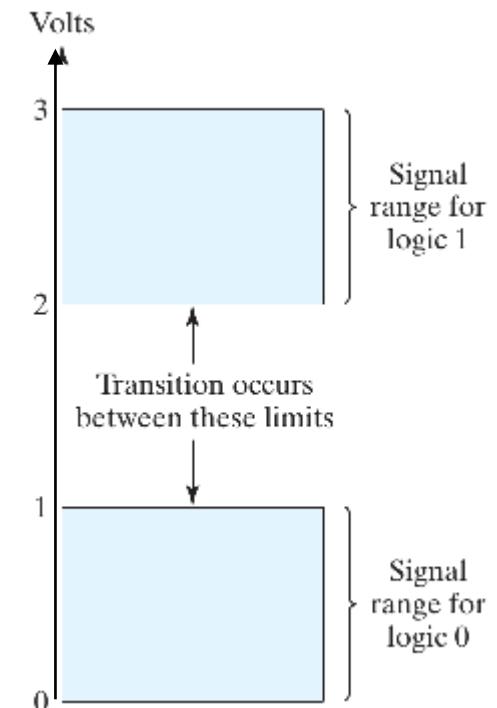
- Because both are true, the theorem is proved

# Operator precedence

- The operator precedence for evaluating Boolean expressions is :
  - 1) parentheses, 2) NOT, 3) AND, 4) OR.
- In other words, expressions inside parentheses must be evaluated before all other operations
- The next operation that holds precedence is the complement, and then follows the AND and, finally, the OR
- As an example, consider one of DeMorgan's theorems
  - The left side of the expression is  $(x + y)'$ . Therefore, the expression inside the parentheses is evaluated first and the result then complemented
  - The right side of the expression is  $x'y'$ , so the complement of  $x$  and the complement of  $y$  are both evaluated first and the result is then ANDed

# Logic gates!

- Logic gates are electronic circuits that operate on one or more input signals to produce an output signal
- Electrical signals such as voltages or currents exist as analog signals having values over a given continuous range, say, 0 to 3 V, but in a digital system these voltages are interpreted to be either of two recognizable values, 0 or 1
- Logic circuits respond to two separate voltage levels that represent a binary variable equal to logic 1 or logic 0
- For example, a particular digital system may define logic 0 as a signal equal to 0 V and logic 1 as a signal equal to 3 V

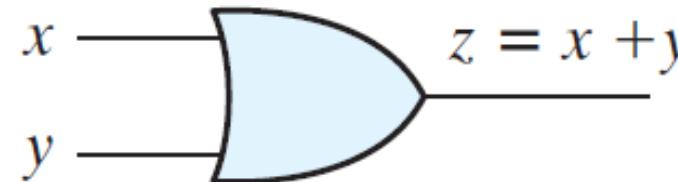


# Logic gates

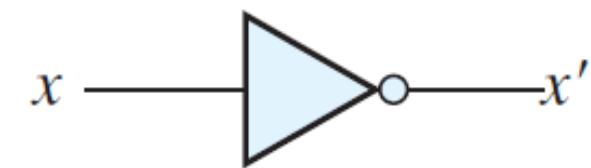
- We represent the logic gates for basic Boolean operations as shown
- Logic gates can have more than two inputs (except for NOT gate of course!)



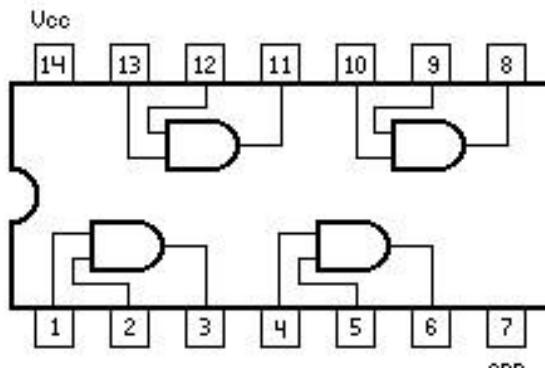
(a) Two-input AND gate



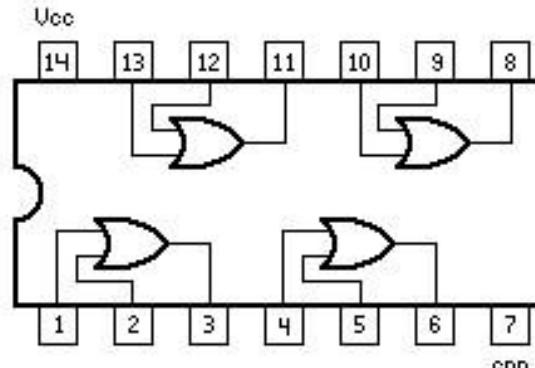
(b) Two-input OR gate



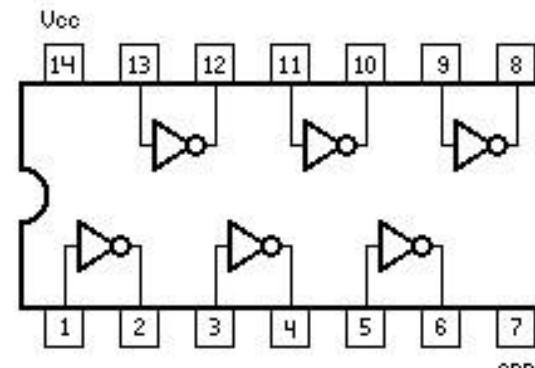
(c) NOT gate or inverter



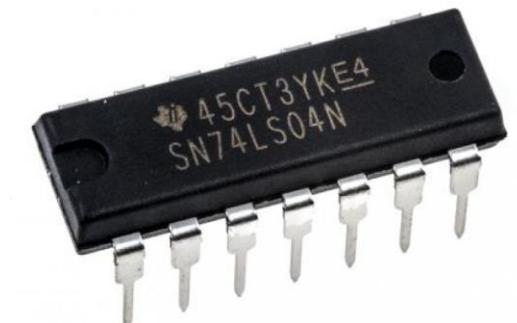
7408 (AND)



7432 (OR)

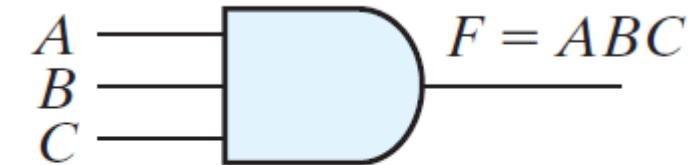


7404 (NOT)

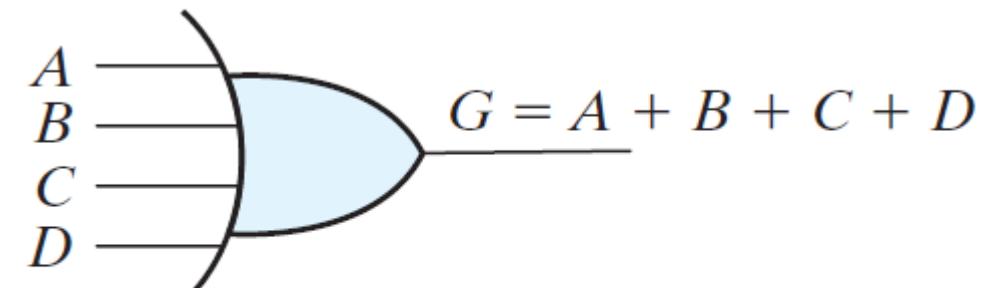


# Logic gates

- AND and OR gates may have more than two inputs
- The three-input AND gate responds with logic 1 output if all three inputs are logic 1. The output produces logic 0 if any input is logic 0
- The four-input OR gate responds with logic 1 if any input is logic 1; its output becomes logic 0 only when all inputs are logic 0



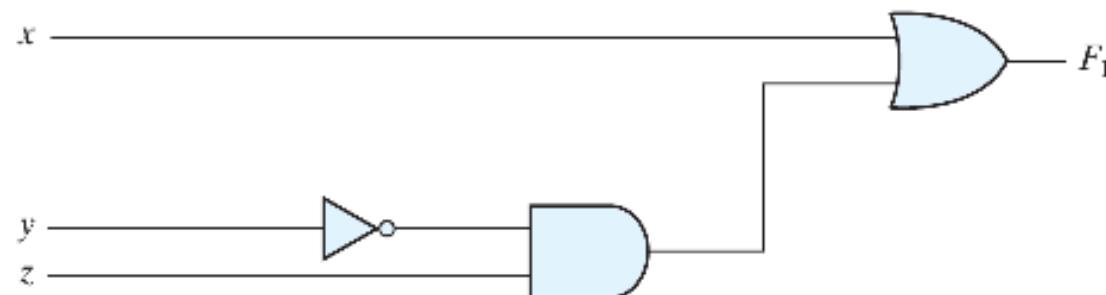
(a) Three-input AND gate



(b) Four-input OR gate

# Boolean functions

- In normal algebra, we define functions on real number variables using addition, subtraction, etc.
- A Boolean function described by a Boolean expression consists of binary variables, the constants 0 and 1, and the logic operation symbols
- For a given value of the binary variables, the function can be equal to either 1 or 0 (closure on the Boolean set)
- As an example, consider the Boolean function:  $F_1 = x + y'.z$



# Boolean functions

- A Boolean function can be represented in a *truth table*
- The number of rows in the truth table is  $2^n$ , where  $n$  is the number of variables in the function
- The binary combinations for the truth table are obtained from the binary numbers by counting from 0 through  $(2^n - 1)$
- For example, there are eight possible binary combinations for assigning bits to the three variables  $x$ ,  $y$ , and  $z$

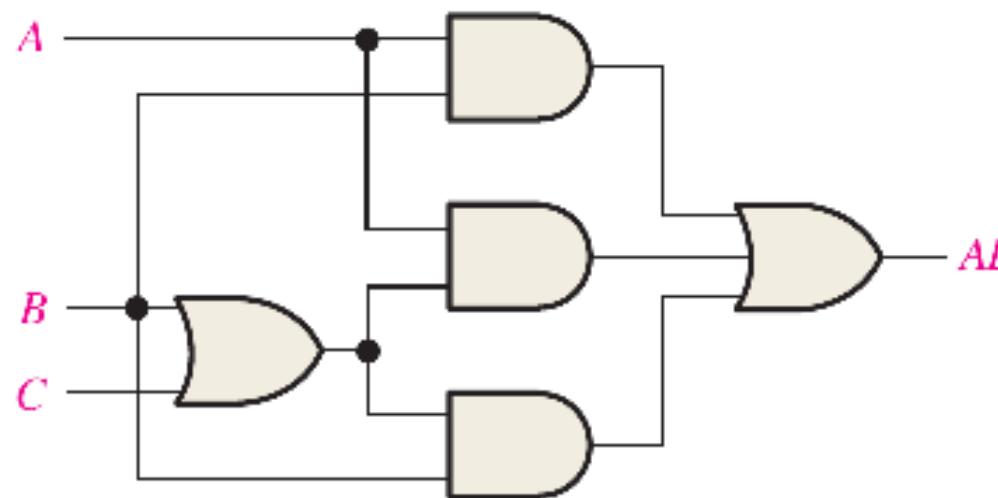
$$F_1 = x + y' \cdot z$$

$x$	$y$	$z$	$F_1$
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

# Boolean functions

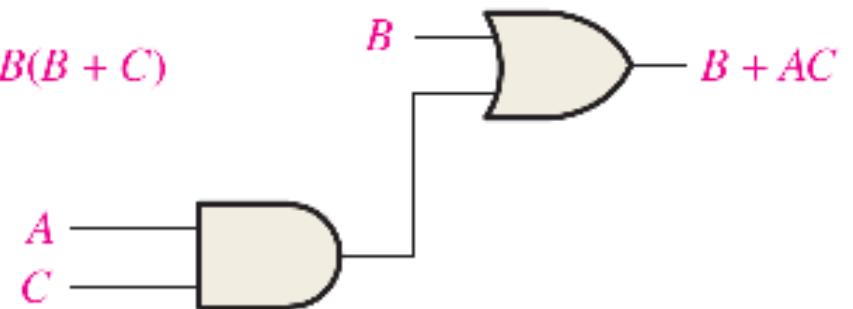
- There is only one way that a Boolean function can be represented in a truth table
- *However, when the function is in algebraic form, it can be expressed in a variety of ways, all of which have equivalent logic*
- Here is a key fact that motivates our use of Boolean algebra: By manipulating a Boolean expression according to the rules of Boolean algebra, it is sometimes possible to obtain a simpler expression for the same logic function, and thus reduce the complexity of the circuit representing that logic
- Consider, for example, the following Boolean functions:
$$F = AB + A(B + C) + B(B + C)$$
- Can we simplify this?

# Boolean functions



(a)

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



(b)

These two circuits are equivalent.

# Boolean functions

---

- A simplified function also has a simplified logic gate implementation
- Therefore, the two circuits have the same outputs for all possible binary combinations of inputs (truth table)
- Thus, each circuit implements the same identical function, but the one with fewer gates and fewer inputs to gates is preferable because it requires fewer wires and components
- In general, there are many equivalent representations of a logic function
- Finding the most economic representation of the logic is an important design task

# Boolean functions

- The manipulation of Boolean algebra consists mostly of reducing an expression for the purpose of obtaining a simpler circuit
- Functions of up to five variables can be simplified by the map method which will be discussed later
- For complex Boolean functions and many different outputs, designers of digital circuits use computer minimization programs that are capable of producing optimal circuits with millions of logic gates
- The manual method available is a procedure employing the basic relations and other manipulation techniques that become familiar with use, but remain, nevertheless, subject to human error
- Example: Simplify  $F = xy + x'z + yz$

# Lecture 8 – Boolean functions

# Complement of a function

- The complement of a function  $F$  is  $F'$  and is obtained from an interchange of 0's for 1's and 1's for 0's in the value of  $F$  (truth table method)
- The complement of a function may be derived algebraically through DeMorgan's theorems
- Example:  $F = x + y'z$ . What is  $F'$ ?
- But what if the function has more terms?
- DeMorgan's theorems can be extended to three or more variables
- The three-variable form of the DeMorgan's theorems:  
$$(x + y + z)' = x'y'z'$$
$$(xyz)' = x' + y' + z'$$
- The generalized form of DeMorgan's theorems states that the complement of a function is obtained by interchanging AND and OR operators and complementing each *literal*

# Minterms

- A binary variable may appear either in its **normal form** ( $x$ ) or in its **complement form** ( $x'$ )
- Now consider two binary variables  $x$  and  $y$  combined with an AND operation
- Since each variable may appear in either form, there are four possible combinations:  $xy$ ,  $x'y$ ,  $xy'$ ,  $x'y'$
- Each of these four AND terms is called a **minterm**, or a **standard product**
- In a similar manner,  $n$  variables can be combined to form  $2^n$  minterms
- The binary numbers from 0 to  $2^n - 1$  are listed under the  $n$  variables. Each minterm is obtained from an AND term of the  $n$  variables, with each variable being primed if the corresponding bit of the binary number is a 0 and unprimed if a 1
- A symbol for each minterm is  $m_j$ , where the subscript  $j$  denotes the decimal equivalent of the binary number of the minterm designated

x	y	z	Minterms	
			Term	Designation
0	0	0	$x'y'z'$	$m_0$
0	0	1	$x'y'z$	$m_1$
0	1	0	$x'yz'$	$m_2$
0	1	1	$x'yz$	$m_3$
1	0	0	$xy'z'$	$m_4$
1	0	1	$xy'z$	$m_5$
1	1	0	$xyz'$	$m_6$
1	1	1	$xyz$	$m_7$

# Maxterms

- In a similar fashion,  $n$  variables forming an OR term, with each variable being primed or unprimed, provide  $2^n$  possible combinations, called *maxterms*, or *standard sums*
- Each maxterm is obtained from an OR term of the  $n$  variables, with each variable being unprimed if the corresponding bit is a 0 and primed if a 1, and
- Maxterms are denoted by  $M_j$

$x$	$y$	$z$	Maxterms	Notation
0	0	0	$x + y + z$	$M_0$
0	0	1	$x + y + z'$	$M_1$
0	1	0	$x + y' + z$	$M_2$
0	1	1	$x + y' + z'$	$M_3$
1	0	0	$x' + y + z$	$M_4$
1	0	1	$x' + y + z'$	$M_5$
1	1	0	$x' + y' + z$	$M_6$
1	1	1	$x' + y' + z'$	$M_7$

# Minterms and Maxterms

## *Minterms and Maxterms for Three Binary Variables*

x	y	z	Minterms		Maxterms	
			Term	Designation	Term	Designation
0	0	0	$x'y'z'$	$m_0$	$x + y + z$	$M_0$
0	0	1	$x'y'z$	$m_1$	$x + y + z'$	$M_1$
0	1	0	$x'yz'$	$m_2$	$x + y' + z$	$M_2$
0	1	1	$x'yz$	$m_3$	$x + y' + z'$	$M_3$
1	0	0	$xy'z'$	$m_4$	$x' + y + z$	$M_4$
1	0	1	$xy'z$	$m_5$	$x' + y + z'$	$M_5$
1	1	0	$xyz'$	$m_6$	$x' + y' + z$	$M_6$
1	1	1	$xyz$	$m_7$	$x' + y' + z'$	$M_7$

It is important to note that:

1. Each maxterm is the complement of its corresponding minterm and vice versa
2. Minterms are 1 for a unique combination of the variables, ie,  $x'y$  is only one when x is 0 and y is 1, in all other cases, it is zero
3. Maxterms are 0 for a single unique combination of variables

# Boolean functions

- Any Boolean function can be expressed algebraically from a given truth table by forming a minterm for each combination of the variables that produces a 1 in the function and then taking the OR of all those terms

- Example:

$$\begin{aligned}f_1 &= x'y'z + xy'z' + xyz \\&= m_1 + m_4 + m_7 \\&= \Sigma(1,4,7)\end{aligned}$$

- Thus, any Boolean function can be expressed as a sum of minterms (with “sum” meaning the ORing of terms)

<b>x</b>	<b>y</b>	<b>z</b>	<b>Function <math>f_1</math></b>
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

# Boolean functions

- Now consider the complement of a Boolean function
- It may be read from the truth table by forming a minterm for each combination that produces a 0 in the function and then ORing those terms

$$f'_1 = m_0 + m_2 + m_3 + m_5 + m_6$$

- If we again take a complement, we get  $f_1$  back:

$$f_1 = (f'_1)' = (m_0 + m_2 + m_3 + m_5 + m_6)'$$

$$f_1 = m'_0 m'_2 m'_3 m'_5 m'_6$$

$$f_1 = M_0 M_2 M_3 M_5 M_6 = \prod_{(0,2,3,5,6)}$$

x	y	z	Function $f_1$	$f'_1$
0	0	0	0	1
0	0	1	1	0
0	1	0	0	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	0

# Boolean functions

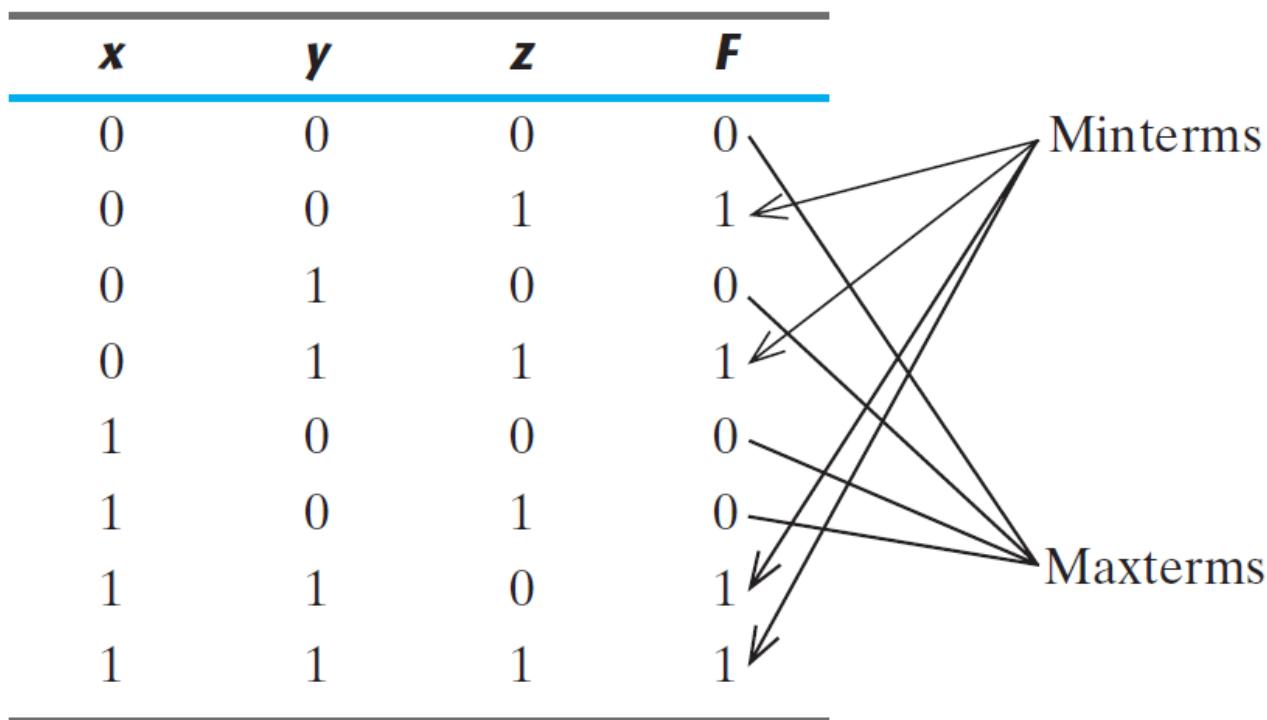
- This shows a second property of Boolean algebra: Any Boolean function can be expressed as a product of maxterms (with “product” meaning the ANDing of terms)
- The procedure for obtaining the product of maxterms directly from the truth table is as follows: Form a maxterm for each combination of the variables that produces a 0 in the function, and then form the AND of all those maxterms
- Boolean functions expressed as a sum of minterms or product of maxterms are said to be in *canonical form*

<b>x</b>	<b>y</b>	<b>z</b>	<b>Function <math>f_1</math></b>
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

# Canonical form

- To convert from one canonical form to another, interchange the symbols  $\Sigma$  and  $\Pi$  and list those numbers missing from the original form
- In order to find the missing terms, one must realize that the total number of minterms or maxterms is  $2^n$ , where  $n$  is the number of binary variables in the function
- This is because the function can either have 0 (maxterm) or 1 (minterm) as the output
- $F = \Sigma(1,3,6,7) = \Pi(0,2,4,5)$

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



# Standard form

- Another way to express Boolean functions is in *standard form*
- In this configuration, the terms that form the function may contain one, two, or any number of literals
- There are two types of standard forms: the **sum of products** and **products of sums**

$$F = x' + y'z + xz$$

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

- The logic diagram of a sum-of-products expression consists of a group of AND gates followed by a single OR gate
- Each product term requires an AND gate, except for a term with a single literal
- The logic sum is formed with an OR gate whose inputs are the outputs of the AND gates
- It is assumed that the input variables are directly available in their complements, so inverters are not included in the diagram
- This circuit configuration is referred to as a *two-level implementation*

# Lecture 9 – Logic gates

# Boolean functions

---

- When the binary operators AND and OR are placed between two variables,  $x$  and  $y$ , they form two Boolean functions,  $x.y$  and  $x + y$ , respectively
- **However, we can have  $2^{2^n}$  functions for  $n$  binary variables!**
- Thus, for two variables,  $n = 2$ , and the number of possible Boolean functions is 16
- Therefore, the AND and OR functions are only 2 of a total of 16 possible functions formed with two binary variables
- Let us find the other 14 functions and investigate their properties

# Boolean functions

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

- Constant functions: 0 and 1 ( $F_0, F_{15}$ )
- AND and OR –the well-known logic functions
- Transfer functions:  $x$  and  $y$

# Boolean functions

<b>x</b>	<b>y</b>	<b><math>F_0</math></b>	<b><math>F_1</math></b>	<b><math>F_2</math></b>	<b><math>F_3</math></b>	<b><math>F_4</math></b>	<b><math>F_5</math></b>	<b><math>F_6</math></b>	<b><math>F_7</math></b>	<b><math>F_8</math></b>	<b><math>F_9</math></b>	<b><math>F_{10}</math></b>	<b><math>F_{11}</math></b>	<b><math>F_{12}</math></b>	<b><math>F_{13}</math></b>	<b><math>F_{14}</math></b>	<b><math>F_{15}</math></b>
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	1	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

- NAND and NOR –these are complementary functions to the usual AND and OR functions
- Take the AND/OR and then take the complement
- NAND is represented by  $\uparrow$  and NOR is represented by  $\downarrow$
- $x \uparrow y = (x \cdot y)'$  and  $x \downarrow y = (x + y)'$

# Boolean functions

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

- Exclusive OR (XOR) returns 1 only if one of x or y is 1, it is 0 if both are one
- This is represented by the symbol  $\oplus$
- $x \oplus y = x'y + y'x$
- The complement of this is XNOR or Equivalence (is x=y?)

# Logic gates

- Since Boolean functions are expressed in terms of AND, OR, and NOT operations, it is easier to implement a Boolean function with these type of gates
- Still, the possibility of constructing gates for the other logic operations is of practical interest
- Factors to be weighed in considering the construction of other types of logic gates are:
  1. The feasibility and economy of producing the gate with physical components
  2. The possibility of extending the gate to more than two inputs
  3. The basic properties of the binary operator such as commutativity and associativity
  4. The ability of the gate to implement Boolean functions alone or in conjunction with other gates

# Logic gates

AND



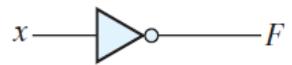
x	y	F
0	0	0
0	1	0
1	0	0
1	1	1

OR



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

Inverter



$$F = x'$$

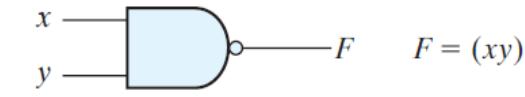
x	F
0	1
1	0

Buffer



x	F
0	0
1	1

NAND



$$F = (xy)'$$

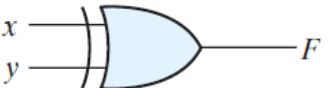
x	y	F
0	0	1
0	1	1
1	0	1
1	1	0

NOR



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

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

Exclusive-OR  
(XOR)

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

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

Exclusive-NOR  
or  
equivalence

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

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

# Gate interconversions

- For a given function, many logic implementations can be done using the gates discussed
- But can we do ALL these implementations with a single logic function implemented over and over?
- These are the NAND and NOR gates –they are referred to as *Universal gates* for this property
- How do we prove this?
- If we can prove that we can get NOT, AND and OR from these gates, we can generate other logic functions using these basic functions

# Gate interconversions

---

- Obtain NOT, AND and OR from NAND
- Obtain NOT, AND and OR from NOR

# Multiple inputs

- A gate can be extended to have multiple inputs if the binary operation it represents is commutative and associative
- The AND and OR operations, defined in Boolean algebra, possess these two properties
- The NAND and NOR functions are commutative
- The difficulty is that the NAND and NOR operators are not associative  
$$(x \downarrow y) \downarrow z \neq x \downarrow (y \downarrow z)$$
- To overcome this difficulty, we define the multiple NOR (or NAND) gate as a complemented OR (or AND) gate. Thus, by definition, we have:

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

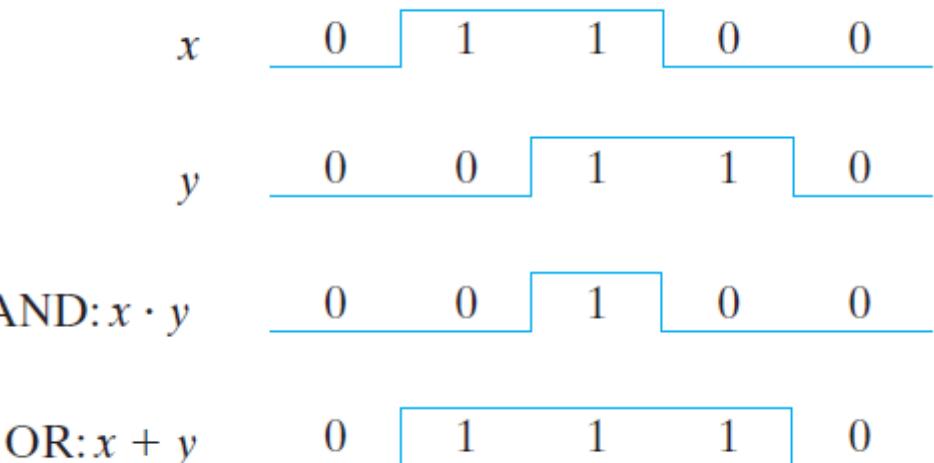
# Multiple inputs

- The exclusive-OR and equivalence gates are both commutative and associative and can be extended to more than two inputs
- However, multiple-input exclusive-OR gates are difficult to make from the hardware standpoint
- The definition of the function must be modified when extended to more than two variables
- Multi-input Exclusive-OR is an *odd* function (i.e., it is equal to 1 if the input variables have an odd number of 1's), and XNOR is its complement

# Timing diagrams

- With logic functions, it is always nice to visualize the various conditions giving a particular output
- One way of visualizing is the truth table, another way can be the timing diagram of a particular function
- Timing diagrams are useful to indicate the sequence of events in large combinational circuits

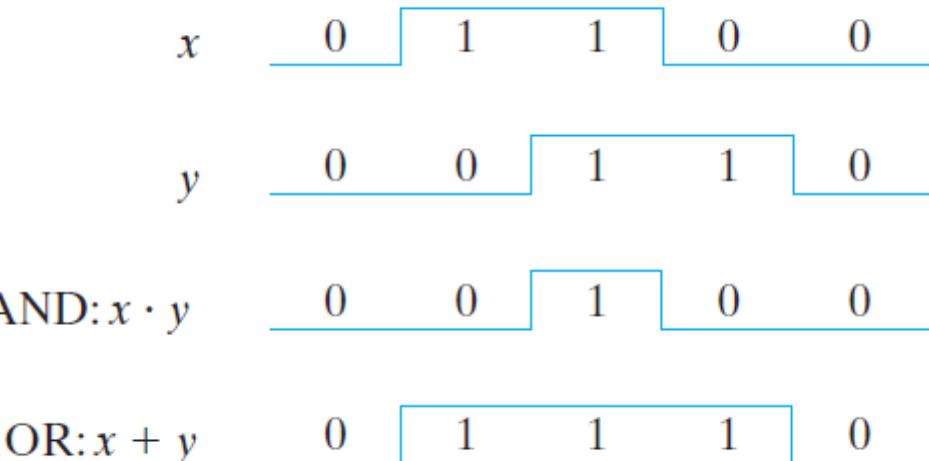
		AND		OR	
$x$	$y$	$x \cdot y$		$x$	$y$
0	0	0		0	0
0	1	0		0	1
1	0	0		1	0
1	1	1		1	1



# Timing diagrams

- The timing diagrams illustrate the idealized response of each gate to the four input signal combinations
- The horizontal axis of the timing diagram represents the time, and the vertical axis shows the signal as it changes between the two possible voltage levels
- In reality, the transitions between logic values occur quickly, but not instantaneously
- The low level represents logic 0, the high level logic 1

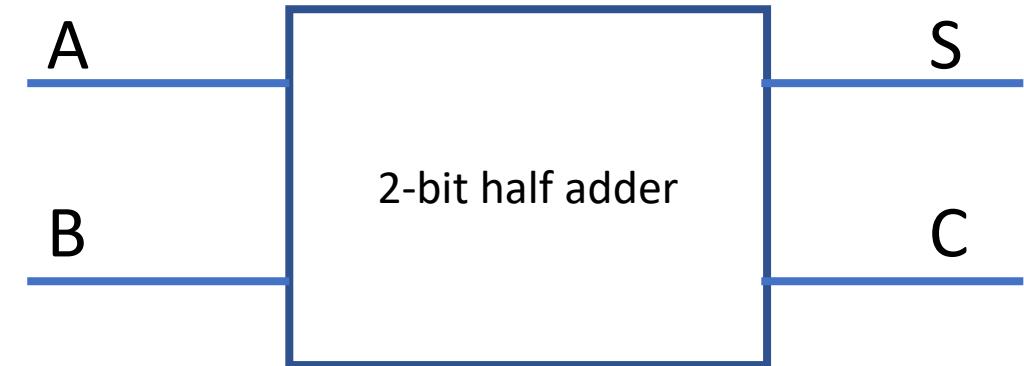
		AND		OR	
$x$	$y$	$x \cdot y$		$x$	$y$
0	0	0		0	0
0	1	0		0	1
1	0	0		1	0
1	1	1		1	1



# Logic circuits!

- Our first circuit...
- Logic circuits are combinations of logic gates to make a particular logic function of our choice
- The logic function can be uniquely represented by a truth-table
- However, many algebraic expressions give the same truth-table and the best or most optimum way of making the circuit is to be found by designers
- Consider a simple 2-bit adder:  $A + B$
- We go from having a logic function to getting the truth-table, then obtaining the circuit
- In this case, because of its simplicity, we know that

$$S = A \oplus B \text{ and } C = A \cdot B$$



A	B	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

# Logic circuits

- Let us say we wanted to continue adding, now we need to take care of the previous carry
- Thus, we need to add A, B and C to get a generic adding machine
- Still from observation, we can deduce that  $S = A \oplus B \oplus C_0$
- But what is the algebraic expression for  $C_1$ ?
- We know the sum of minterms, but is there an better way?



A	B	$C_0$	$C_1$	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

# Gate level minimization

- *Gate-level minimization* is the design task of finding an optimal gate-level implementation of the Boolean functions describing a digital circuit
- This task is well understood, but is difficult to execute by manual methods when the logic has more than a few inputs
- Fortunately, computer-based logic synthesis tools can minimize a large set of Boolean equations efficiently and quickly
- Nevertheless, it is important that a designer understand the underlying mathematical description and solution of the problem

# Lecture 10 – K-maps

# Gate level minimization

- Some history: although we use the term *Karnaugh map* (named after Maurice Karnaugh, 1953) for the map method, it dates back further
- In 1881, Allan Marquand published a paper criticizing “Mr. Venn” for his approach to visualization of logic problems using curvilinear shapes and proposed a method with non-intersecting squares
- Non-intersecting because he was concerned with two-state variables that are either *true* or *false*

XXXIII. *On Logical Diagrams for n terms.* By ALLAN MARQUAND, Ph.D., late Fellow of the Johns Hopkins University\*.

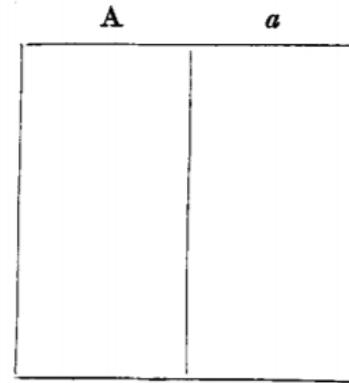
IN the Philosophical Magazine for July 1880 Mr. Venn has offered diagrams for the solution of logical problems involving three, four, and five terms. From the fact that he makes use of circles, ellipses, and other curvilinear figures, the construction of diagrams becomes more and more difficult as new terms are added. Mr. Venn stops with the five-term diagram, and suggests that for six terms “the best plan would be to take two five-term figures.”

It is the object of this paper to suggest a mode of constructing logical diagrams, by which they may be indefinitely extended to any number of terms, without losing so rapidly their special function, viz. that of affording visual aid in the solution of problems.

Conceiving the logical universe as always more or less limited, it may be represented by any closed figure. For convenience we take a square. If then we drop a perpendicular from the middle point of the upper to the lower side of the square, the universe is prepared for a classification of its contents by means of a single logical term.

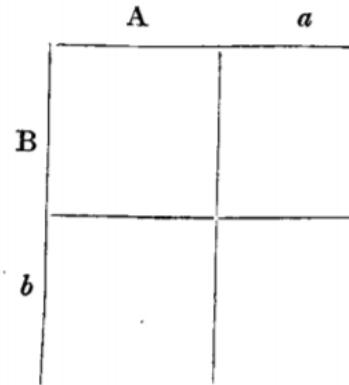
# Gate level minimization

- Some history: although we use the term *Karnaugh map* (named after Maurice Karnaugh, 1953) for the map method, it dates back further
- In 1881, Allan Marquand published a paper criticizing “Mr. Venn” for his approach to visualization of logic problems using curvilinear shapes and proposed a method with non-interesting squares
- Non-intersecting because he was concerned with two-state variables that are either *true* or *false*



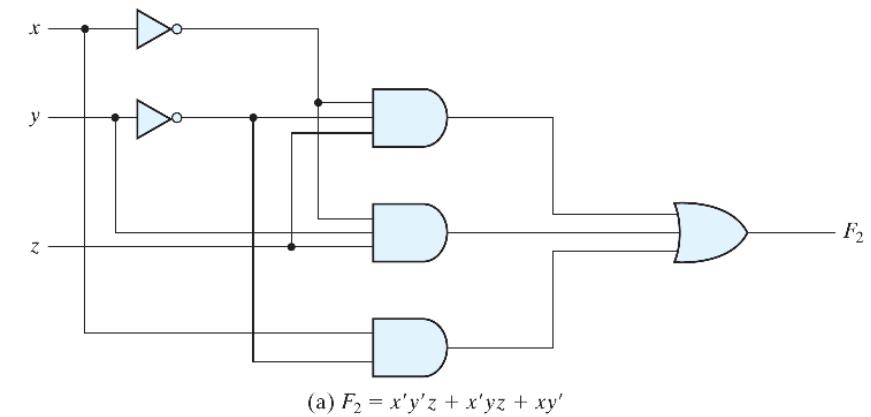
This represents a universe with its A and not-A “compartments.” The quantitative relation of the compartments being insignificant, they may for convenience be represented as equal.

The introduction of a second term divides each of the existing compartments. This may be done by a line drawn at right angles to our perpendicular and through its centre, thus :—

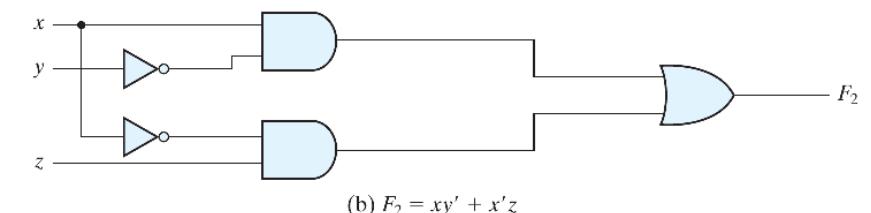


# Gate level minimization

- **Gate-level minimization** is the task of finding an optimal gate-level implementation of the Boolean functions describing a digital circuit.
- The complexity of logic gate implementation of a Boolean function is directly related to the complexity of the algebraic expression from which the function is implemented.
- Algebraic simplification of Boolean expressions can be cumbersome
- Karnaugh map (K-Map) provides a systematic method to simplify Boolean expressions



$$(a) F_2 = x'y'z + x'yz + xy'$$



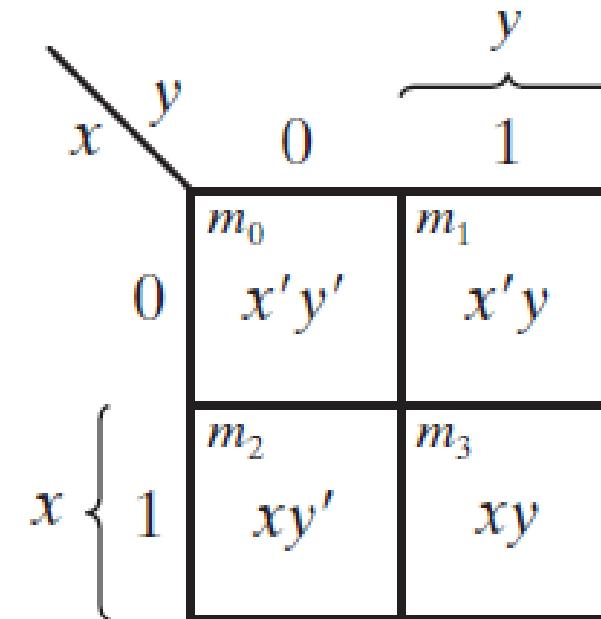
$$(b) F_2 = xy' + x'z$$

$$F_2 = x'y'z + x'yz + xy' = x'z(y' + y) + xy' = x'z + xy'$$

## 2 variable K-map

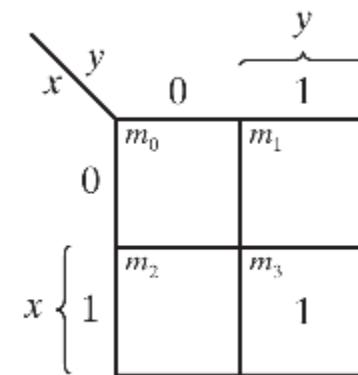
- There are four *minterms* for two variables –  $xy$ ,  $x'y$ ,  $xy'$ ,  $x'y'$ ; hence, the map consists of four squares, one for each *minterm*
- The map can be drawn to show the relationship between the squares and the two variables  $x$  and  $y$
- The 0 and 1 marked in each row and column designate the values of variables
- In *minterm* form, variable  $x$  appears primed in row 0 and unprimed in row 1. Similarly,  $y$  appears primed in column 0 and unprimed in column 1

$m_0$	$m_1$
$m_2$	$m_3$

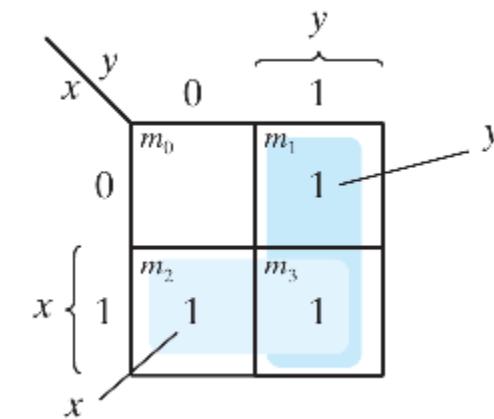


# 2 variable K-map

Example:



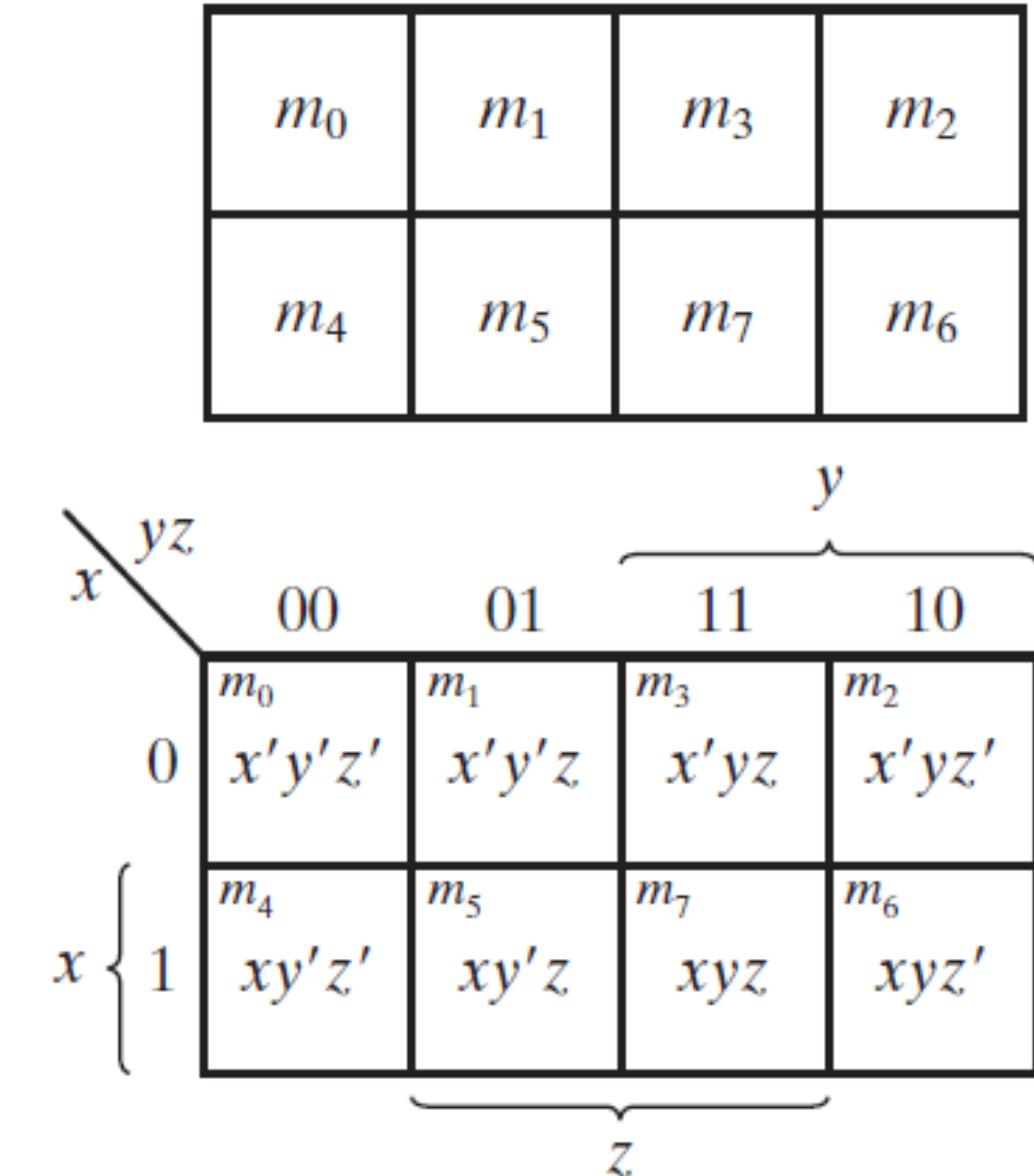
Map for  $F = xy$



Map for  $F = x+y$

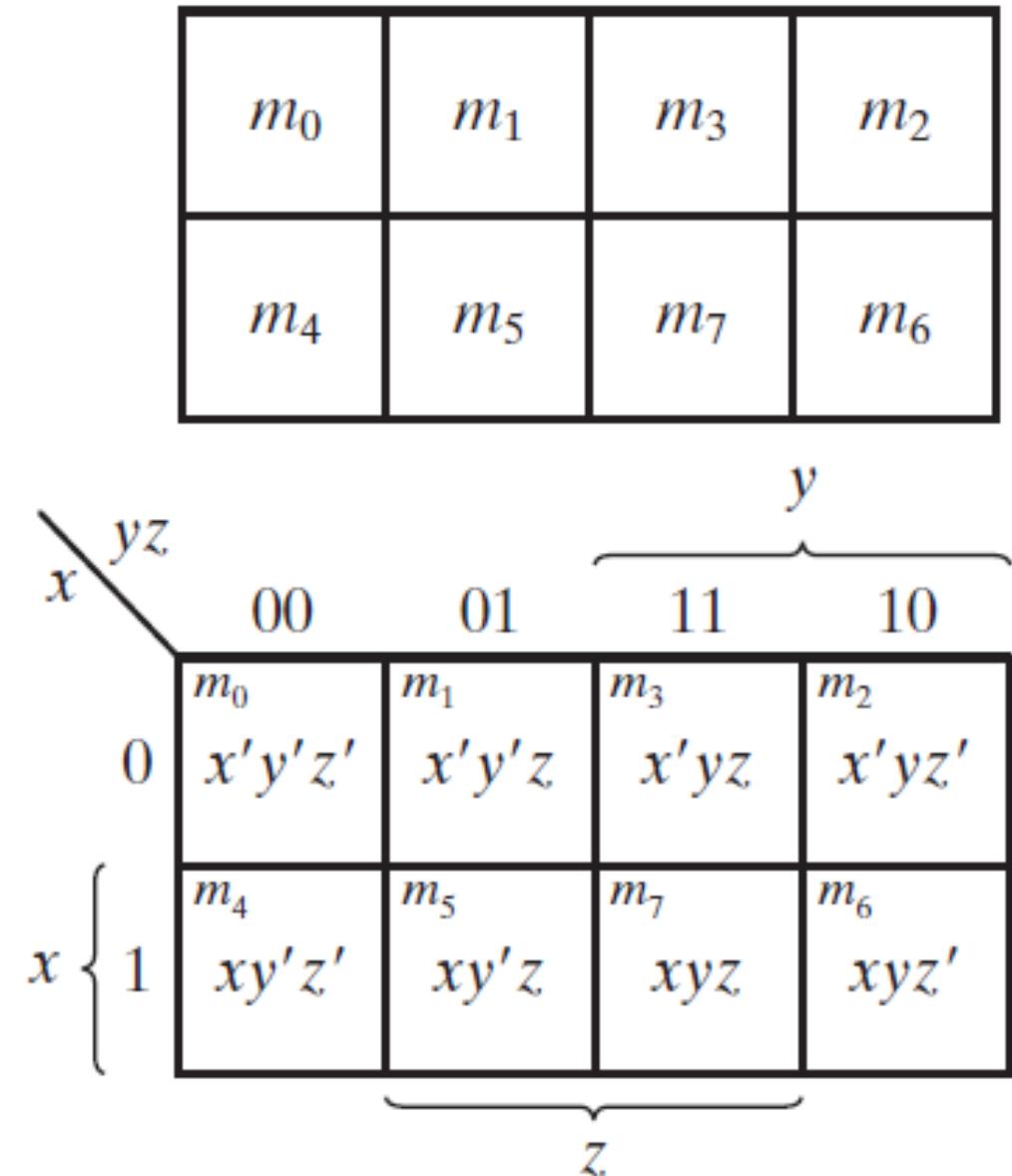
# 3 variable K-map

- In 3 variable problems, there are eight *minterms* for three binary variables; therefore, the map consists of eight squares
- The map is marked with numbers in each row and each column to show the relationship between the squares and the three variables
- For example, the square assigned to  $m_5$  corresponds to row 1 and column 01
- Each cell of the map corresponds to a unique *minterm*, so another way of looking at the square is  $m_5 = xy'z$
- Note that there are four squares in which each variable is equal to 1 and four in which each is equal to 0



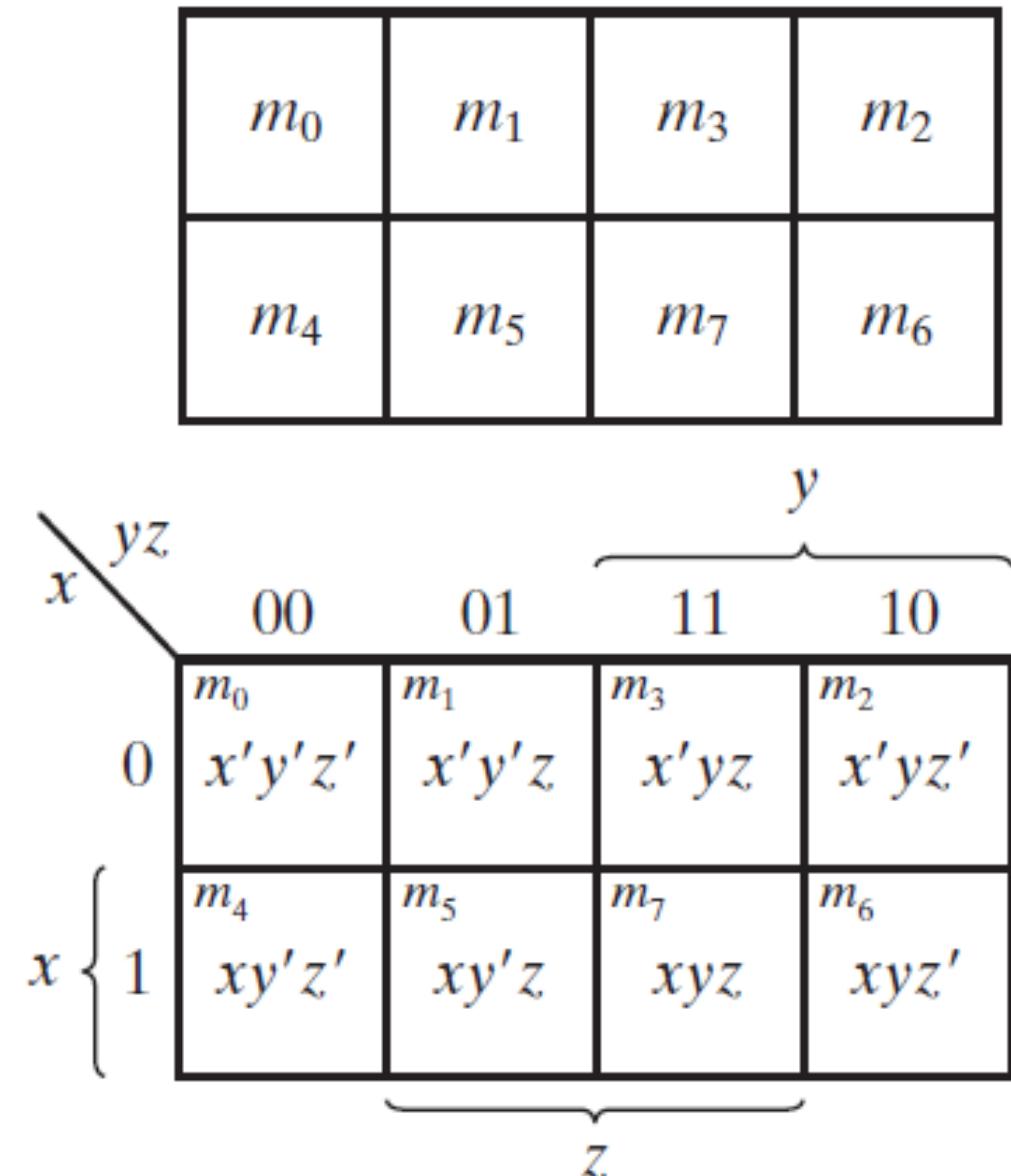
# 3 variable K-map

- Here is the magic: To understand the usefulness of the map in simplifying Boolean functions, we must recognize the basic property possessed by adjacent squares: **Any two adjacent squares in the map differ by only one variable**, which is primed in one square and unprimed in the other
- For example,  $m_5$  and  $m_7$  lie in two adjacent squares
- Variable  $y$  is primed in  $m_5$  and unprimed in  $m_7$ , whereas the other two variables are the same in both squares
- From the postulates of Boolean algebra, it follows that the sum of two **minterms** in adjacent squares can be simplified to a single product term consisting of only two literals- Eg:  $xy'z + xyz = xz(y'+y) = xz$



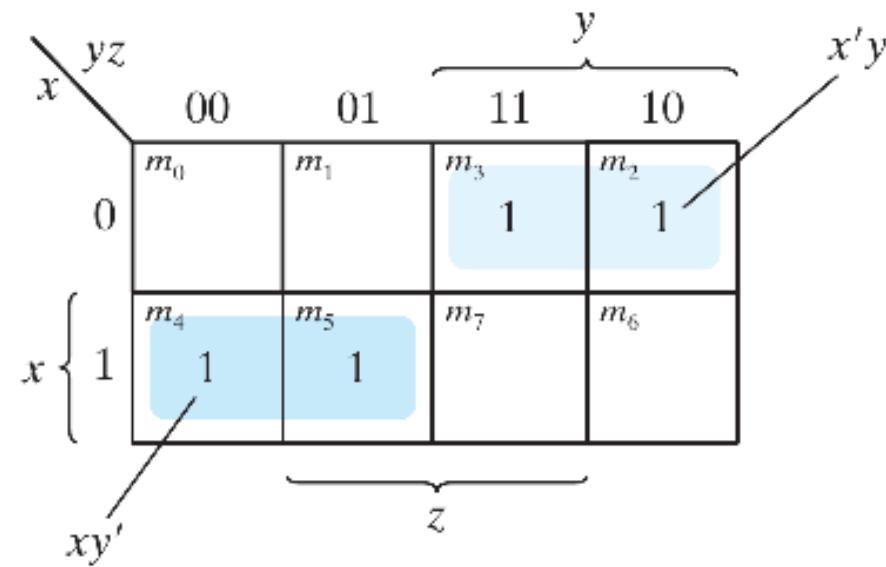
# 3 variable K-map

- Here is how we optimize the logic gate expression using maps:
  1. We look for a cluster of adjacent squares with the function value being 1 (or true):
    1. cluster of 8 squares (meaning the entire function is 1)
    2. A cluster of 4 squares (meaning one literal)  
– eg:  $xyz + xyz' + x'yz + x'yz' = xy + x'y = y$
    3. A cluster of 2 squares (meaning two literals ANDed)
    4. single square with all three variables ANDed (the minterm)
  2. We OR all the expressions related to the clusters
- Note that “adjacent” squares mean vertically or horizontally, not diagonally



# 3 variable K-map

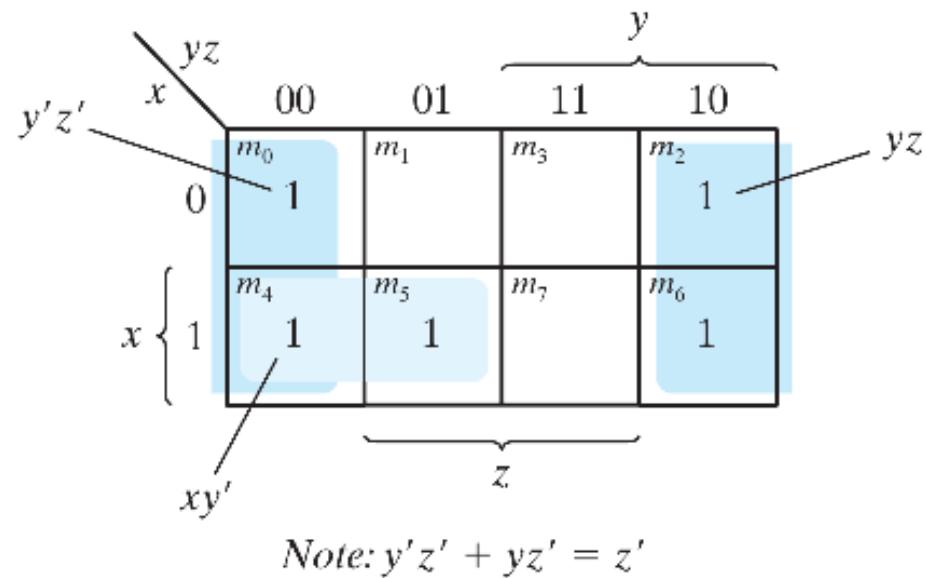
Simplify the Boolean function:  $F(x, y, z) = \Sigma(2, 3, 4, 5)$



$$F(x, y, z) = \Sigma(2, 3, 4, 5) = x'y + xy'$$

# 3 variable K-map

Simplify the Boolean function:  $F(x, y, z) = \Sigma(0, 2, 4, 5, 6)$



$m_2$  is adjacent to  $m_0$   
 $m_6$  is adjacent to  $m_4$

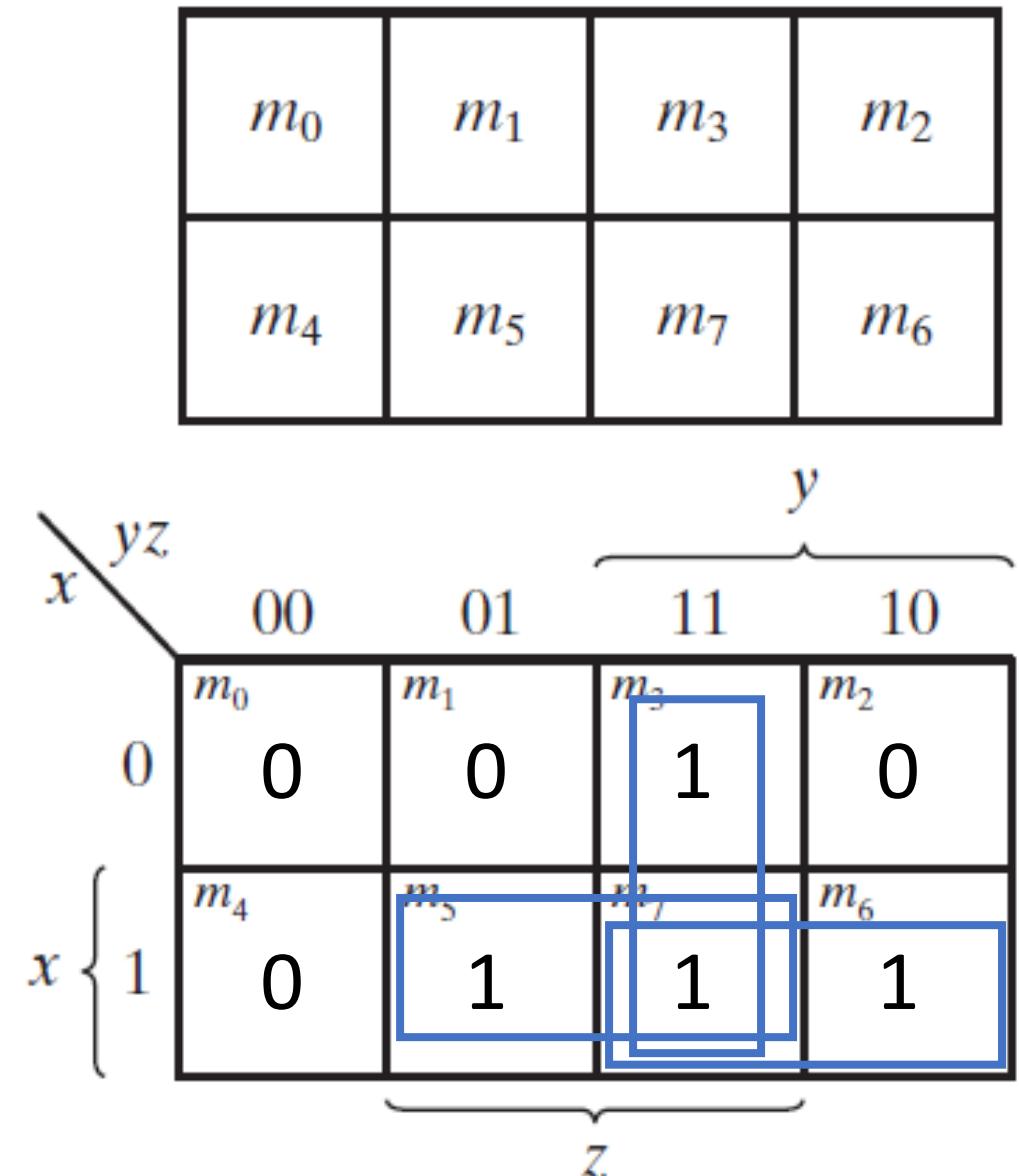
$$F(x, y, z) = \Sigma(0, 2, 4, 5, 6) = z' + xy'$$

# 3 variable K-map

- Full adder:

A	B	$C_0$	$C_1$	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

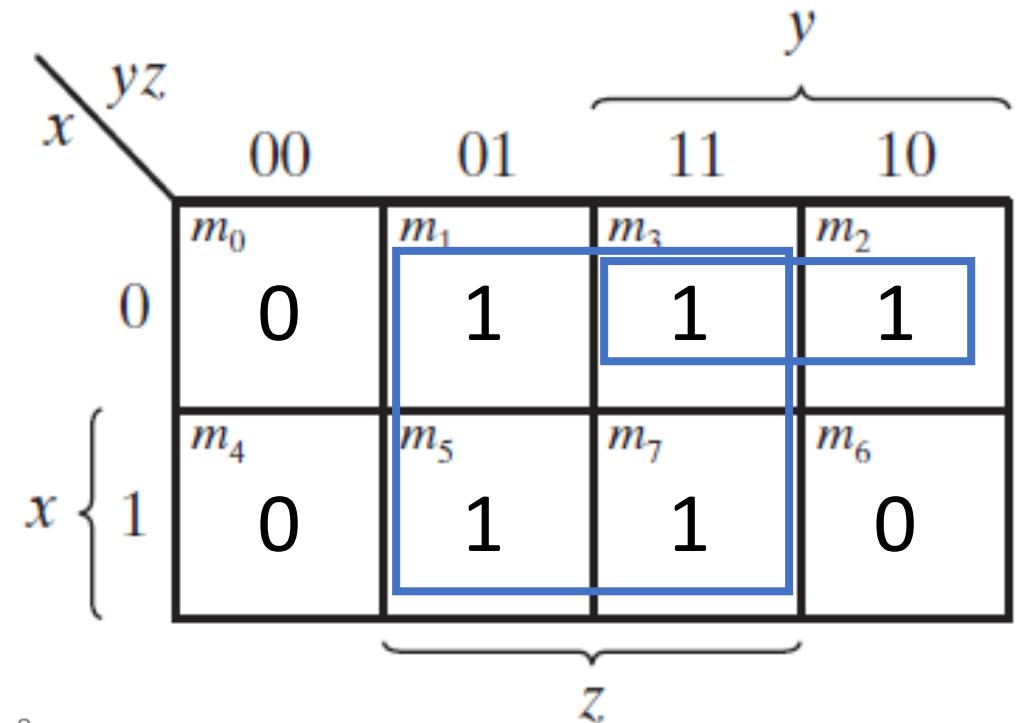
- Let  $x = A$ ,  $y = B$ ,  $z = C_0$
- Three clusters of 2 squares
- Thus,  
$$C_1 = xy + yz + zx$$
$$C_1 = AB + BC_0 + C_0A$$



## 3 variable K-map

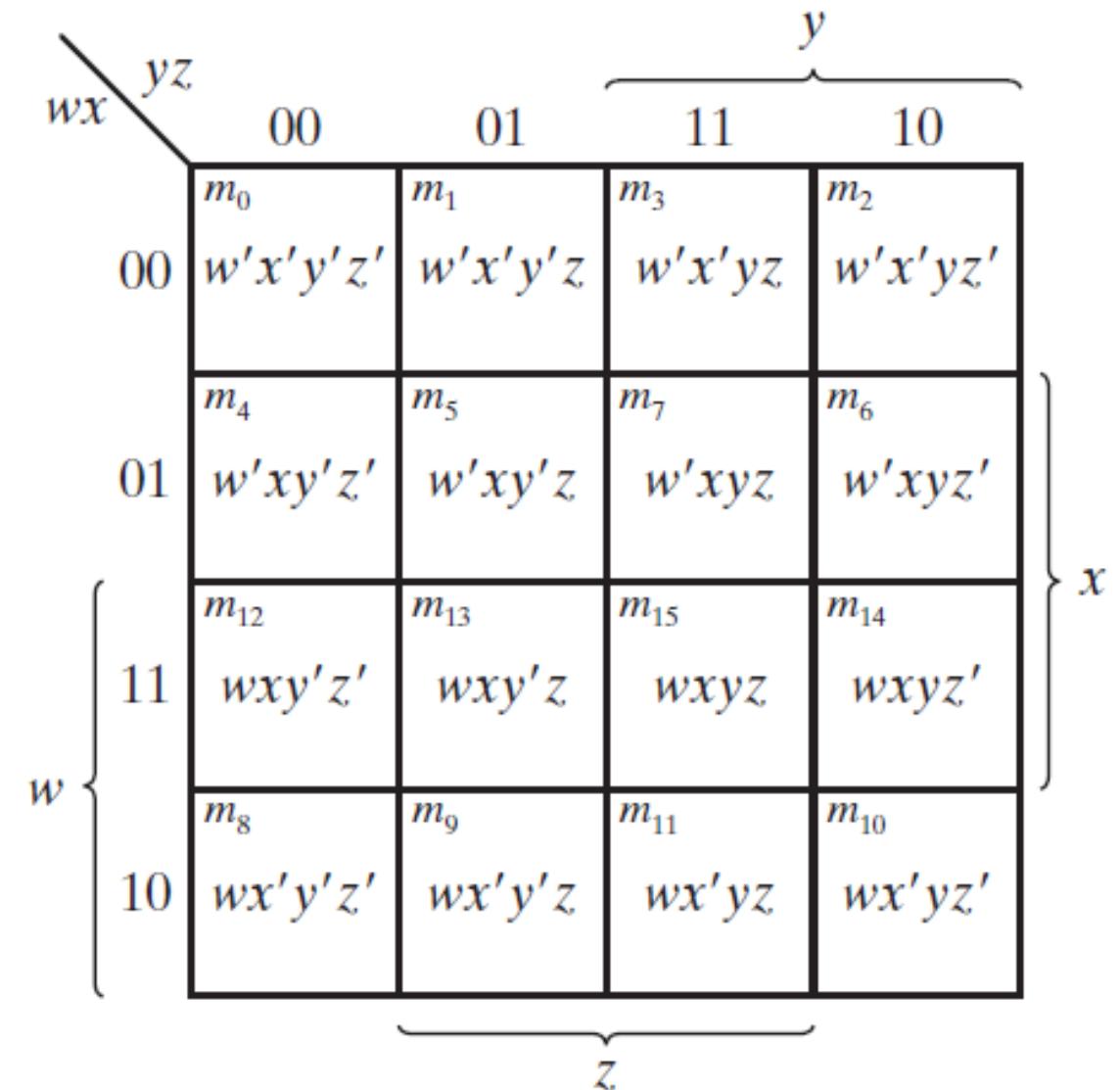
- $F(x, y, z) = \sum(1, 2, 3, 5, 7)$
- We have one cluster of 4 squares
- This represents  $z$
- One cluster of 2 squares
- This represents  $x'y$
- Thus,  $F = z + x'y$

$m_0$	$m_1$	$m_3$	$m_2$
$m_4$	$m_5$	$m_7$	$m_6$



# 4 variable K-map

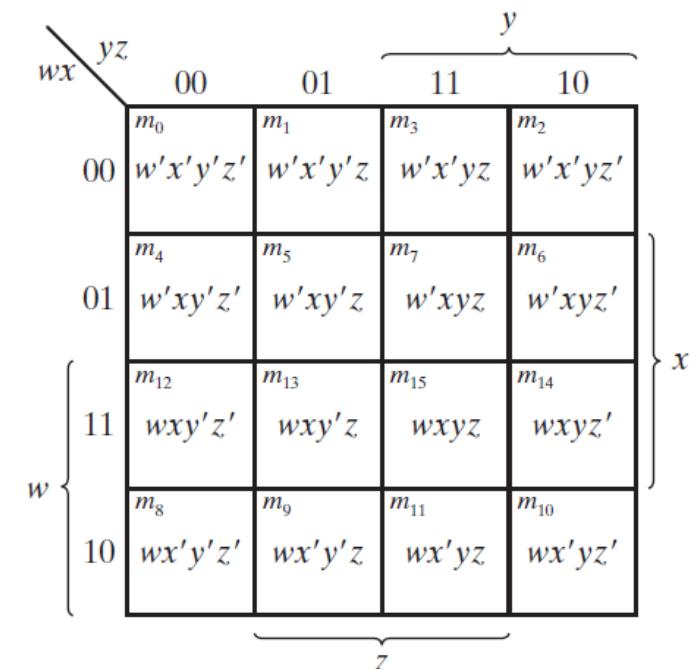
$m_0$	$m_1$	$m_3$	$m_2$
$m_4$	$m_5$	$m_7$	$m_6$
$m_{12}$	$m_{13}$	$m_{15}$	$m_{14}$
$m_8$	$m_9$	$m_{11}$	$m_{10}$



# 4 variable K-map

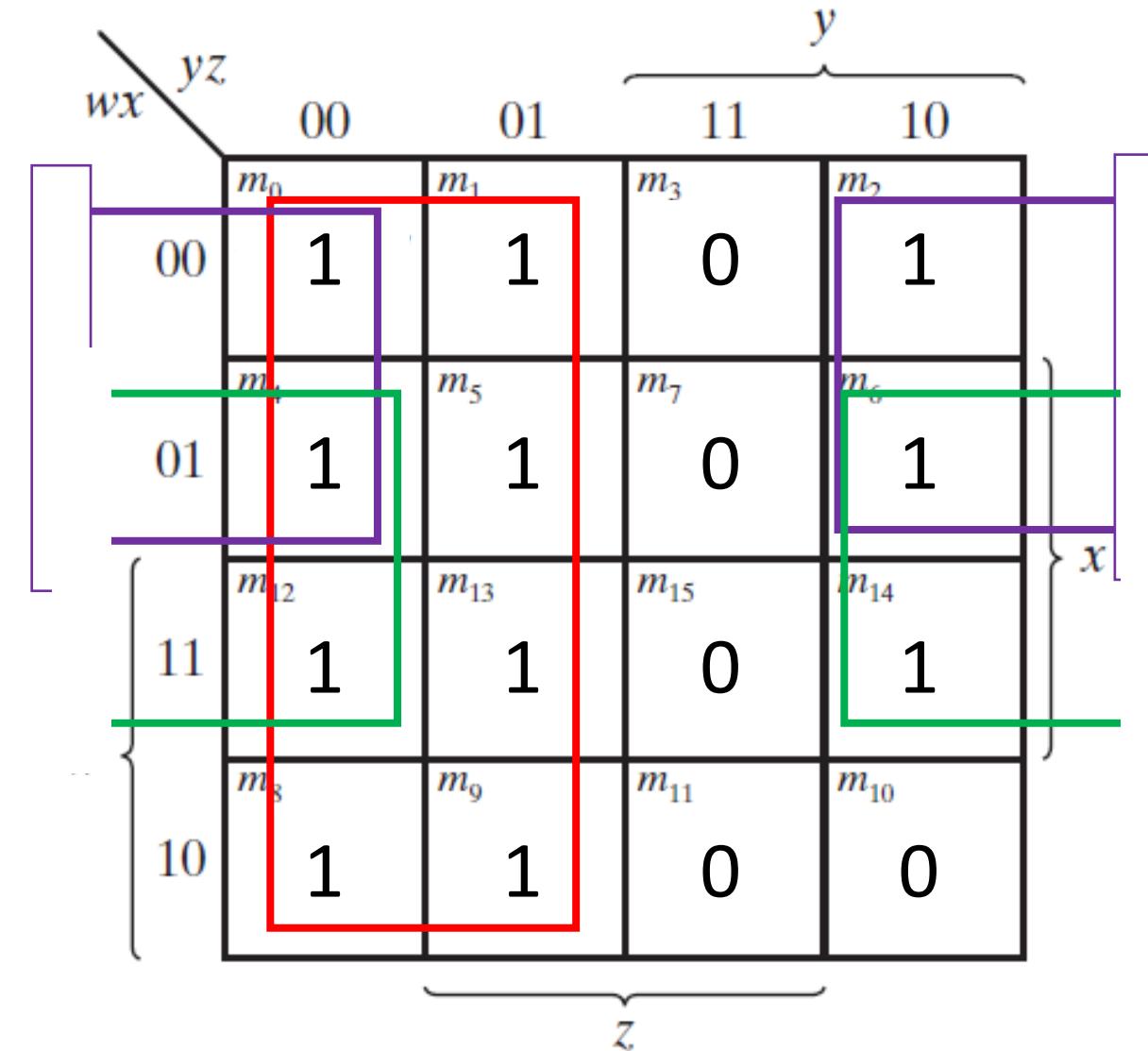
1. We look for a cluster of adjacent squares with the function value being 1 (or true):
  1. A cluster of 16 squares (meaning the entire function is 1)
  2. A cluster of 8 squares (meaning one literal)
  3. A cluster of 4 squares (meaning two literals ANDed)
  4. A cluster of 2 squares (meaning three literals ANDed)
  5. A single square with all the four variables ANDed (a minterm)
2. We OR all the expressions related to the clusters

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



## 4 variable K-map

- Simplify the function  
 $F(w, x, y, z) = \Sigma(0, 1, 2, 4, 5, 6, 8, 9, 12, 13, 14)$
- Cluster of 16? No
- Cluster of 8?
- It represents  $y'$
- Cluster of 4?
- This represents  $w'z'$
- This represents  $xz'$
- Thus, the function is  
 $F(w, x, y, z) = y' + w'z' + xz'$



# Product of Sums

---

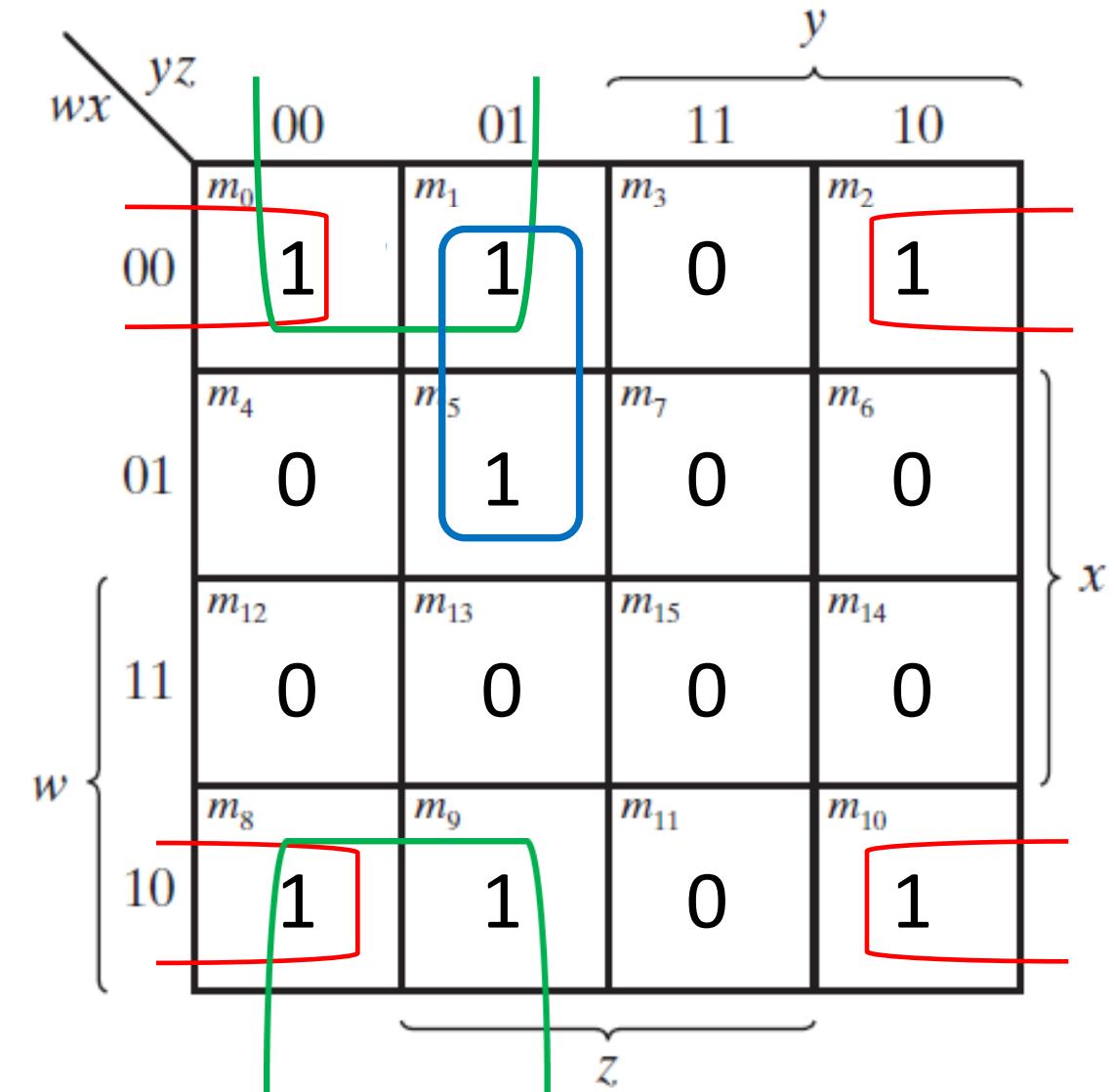
- The minimized Boolean functions derived from the K-map were expressed in *sum-of-products* form
- With a minor modification, the product-of-sums form can be obtained
- The 1's placed in the squares of the map represent the *minterms* of the function
- From this observation, we see that the complement of a function is represented in the map by the squares not marked by 1's
- *If we mark the empty squares by 0's and combine them into valid adjacent squares, we obtain a simplified sum-of-products expression of the complement of the function (i.e., of  $F'$ )*
- The complement of  $F'$  gives us back the function  $F$  in product-of-sums form (a consequence of DeMorgan's theorem)

# Product of Sums

- Simplify the following Boolean function into (a) sum-of-products form and (b) product-of-sums form:  
 $F(w, x, y, z) = \sum(0, 1, 2, 5, 8, 9, 10)$

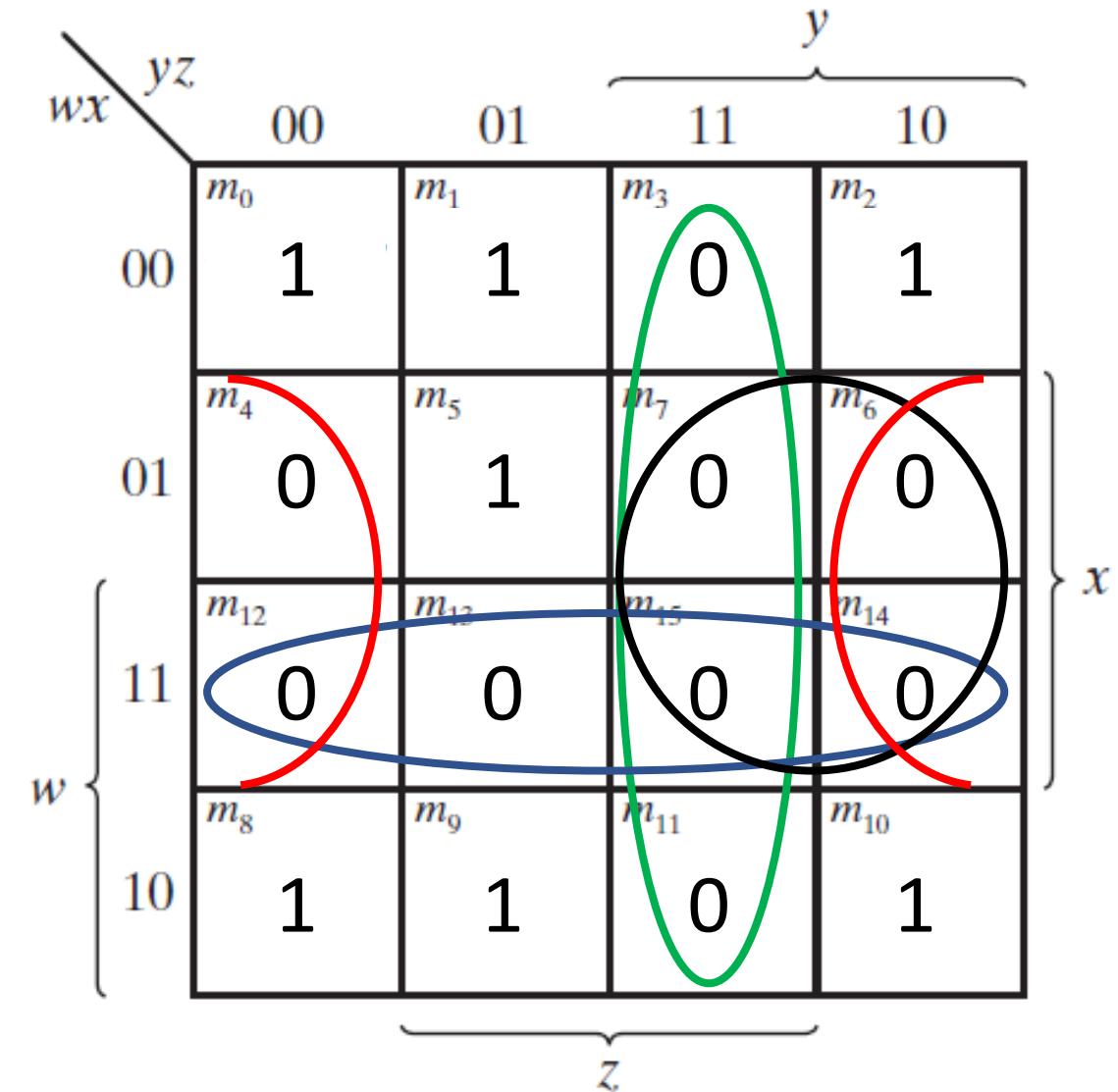
- Two clusters of four squares:  $x'z'$  and  $x'y'$
- One cluster of two squares:  $w'y'z$
- Thus, the function is:

$$F = x'z' + x'y' + w'y'z$$



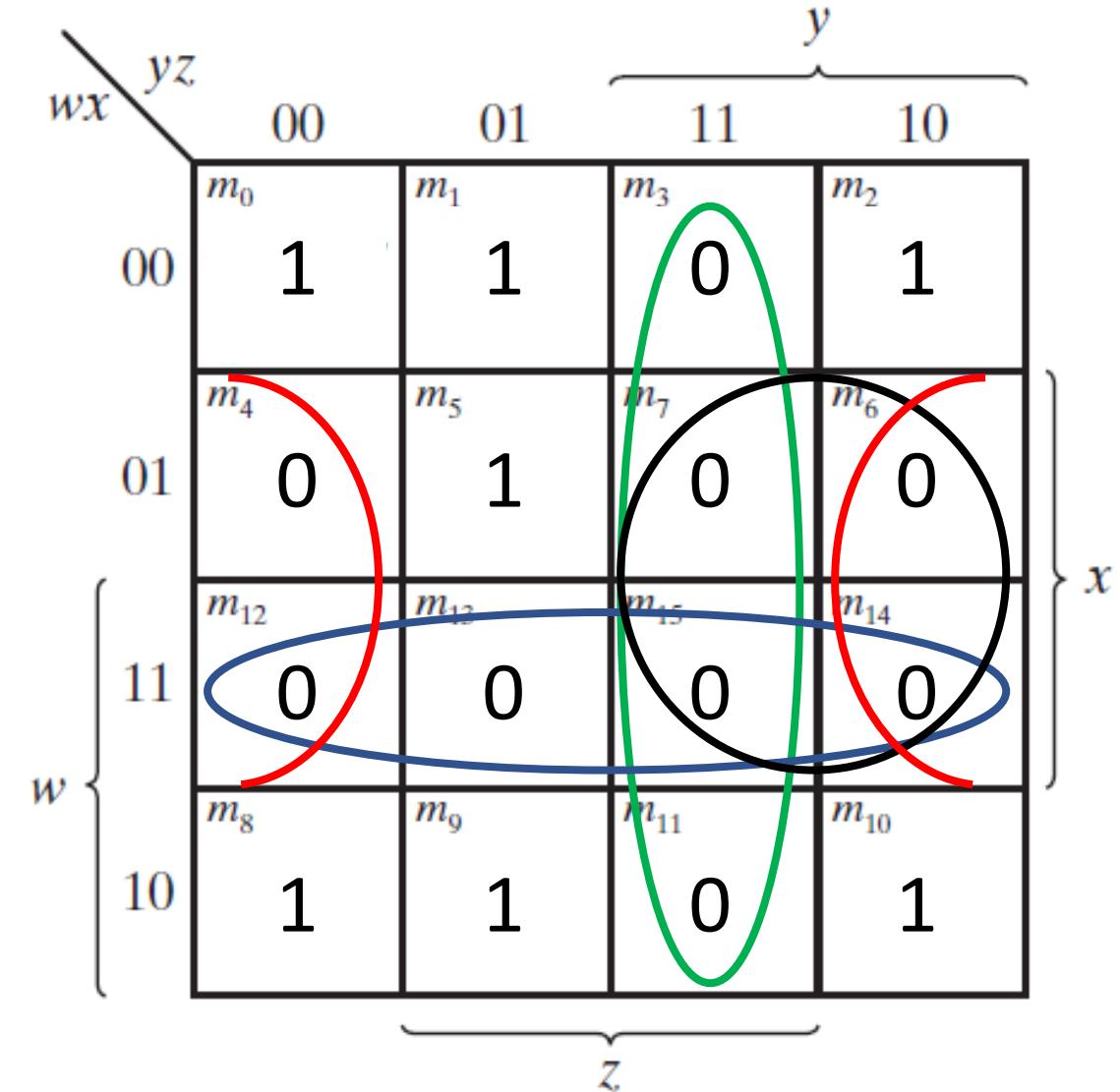
# Product of Sums

- The original function is:  
$$F = x'z' + x'y' + w'y'z$$
- Now, let us see the complement of the function:  $F'$
- This can be obtained from clustering all the 0s together
- We have four clusters of four (*prime implicants*)
- Of these, the *essential prime implicants* are:  $yz$ ,  $wx$ ,  $xz'$
- Thus,  $F' = wx + yz + xz'$



# Product of Sums

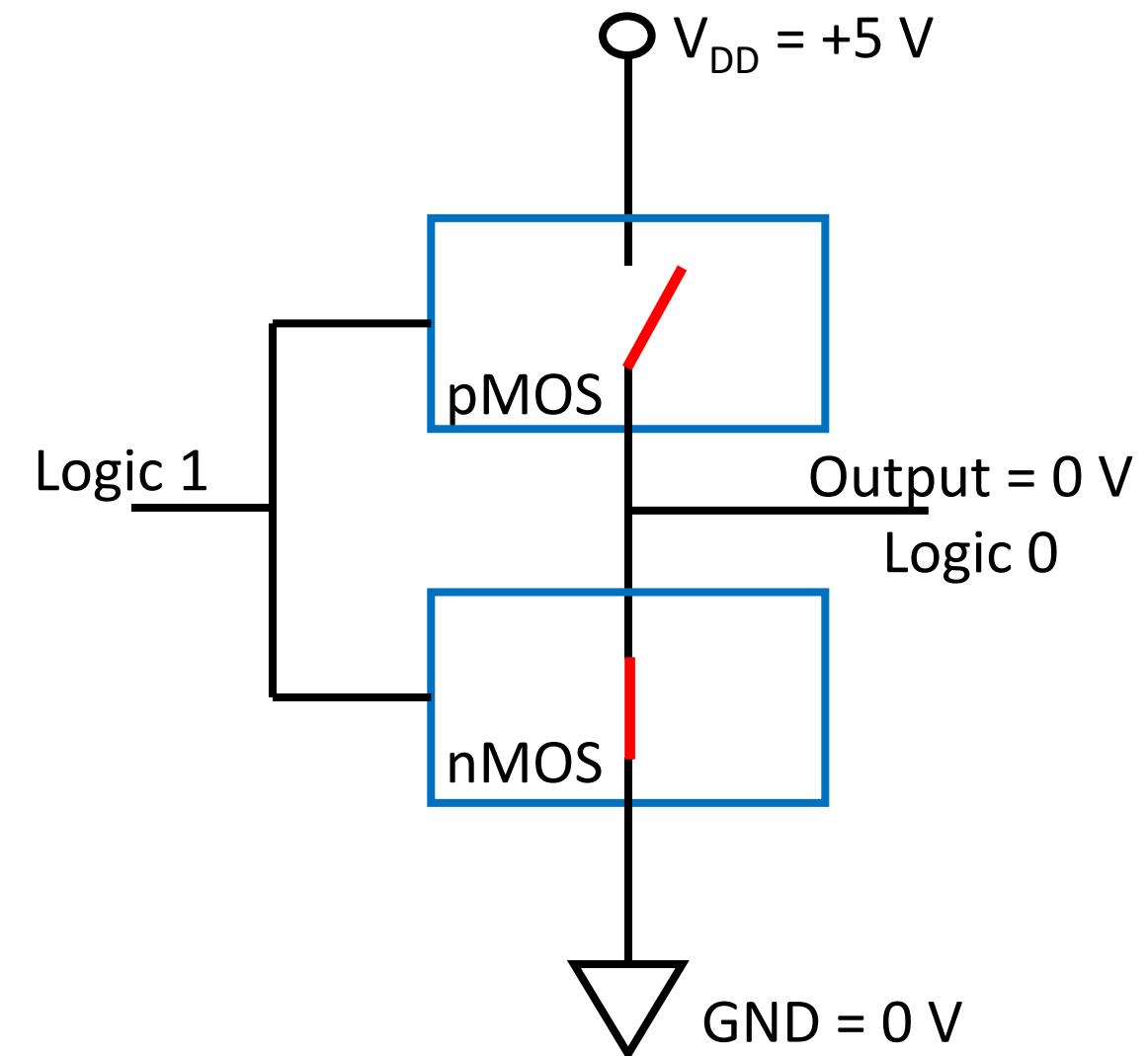
- The original function is:  
$$F = x'z' + x'y' + w'y'z$$
- Thus, the complement function is  
$$F' = wx + yz + xz'$$
- Now, we can obtain F back from F' using the DeMorgan's theorems
- Thus,  
$$F = (w' + x')(y' + z')(x' + z)$$
- Hence, the actual simplest implementation of a function can be through its complement (product of sum)



# Lecture 11 – Logic implementation

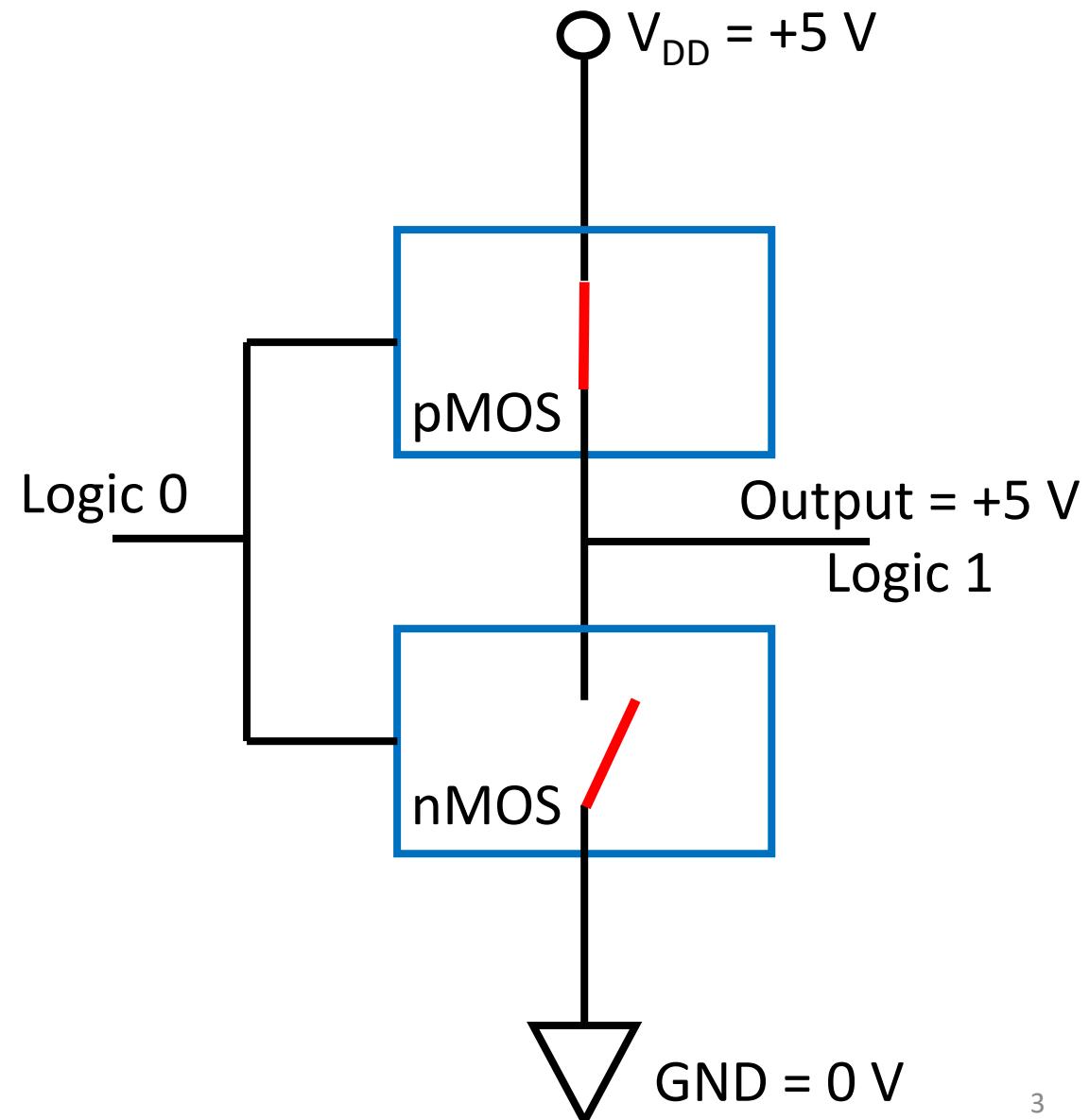
# Implementation of logic gates

- Typically, logic gates are implemented using transistors in CMOS architecture, where they act as switches, connecting  $V_{DD}$  or GND to the output
  - pMOS – p channel *metal-oxide semiconductor*
  - pMOS – n channel *metal-oxide semiconductor*
  - CMOS – complementary *metal-oxide semiconductor*
- The switches are themselves driven using the logic states as inputs:
  - nMOS is on for logic 1 input and off for logic 0
  - pMOS is on for logic 0 input and off for logic 1
- When GND is connected, the output goes to logic 0
- When  $V_{DD}$  is connected, the output goes to logic 1



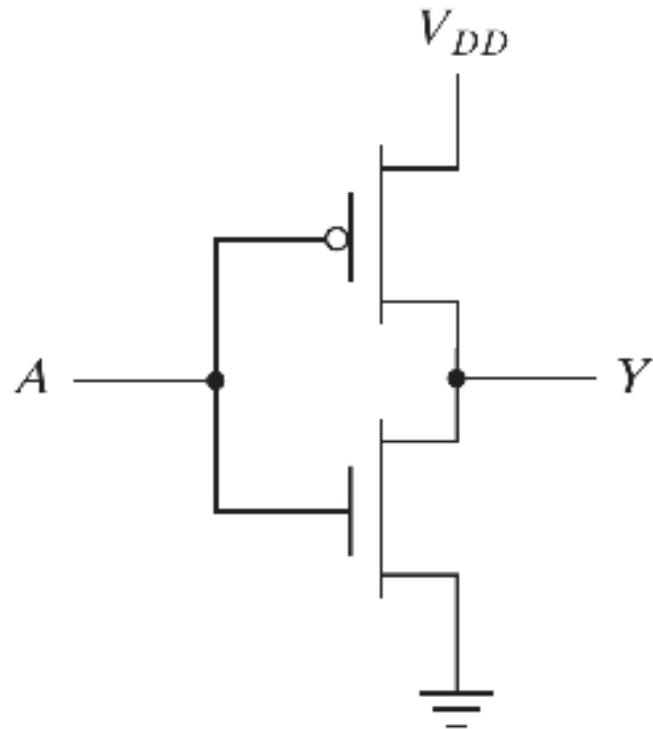
# Implementation of logic gates

- Typically, logic gates are implemented using transistors in CMOS architecture, where they act as switches, connecting  $V_{DD}$  or GND to the output
  - pMOS – p channel *metal-oxide semiconductor*
  - pMOS – n channel *metal-oxide semiconductor*
  - CMOS – complementary *metal-oxide semiconductor*
- The switches are themselves driven using the logic states as inputs:
  - nMOS is on for logic 1 input and off for logic 0
  - pMOS is on for logic 0 input and off for logic 1
- When GND is connected, the output goes to logic 0
- When  $V_{DD}$  is connected, the output goes to logic 1

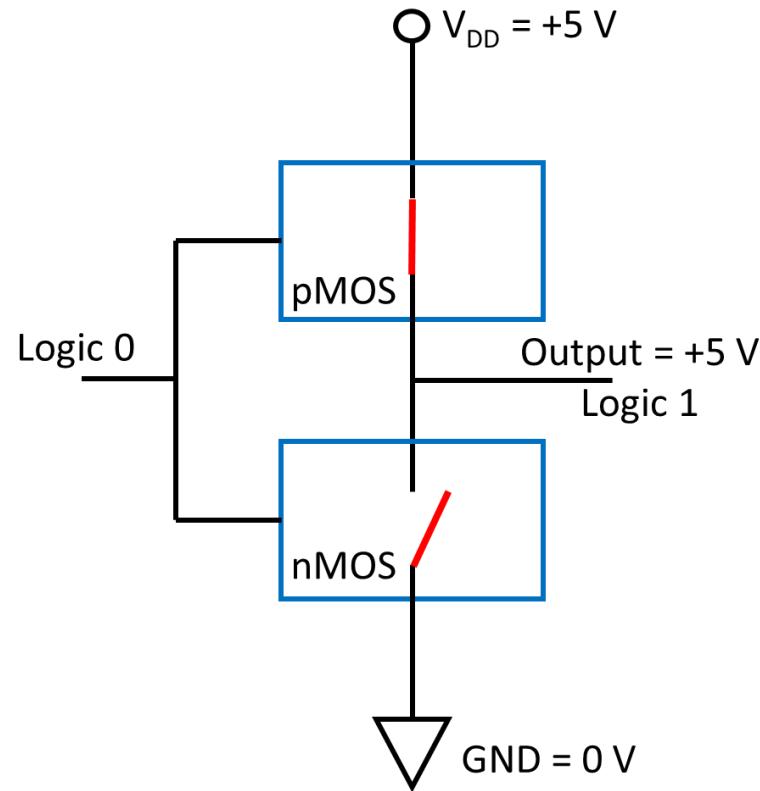


# Implementation of logic gates

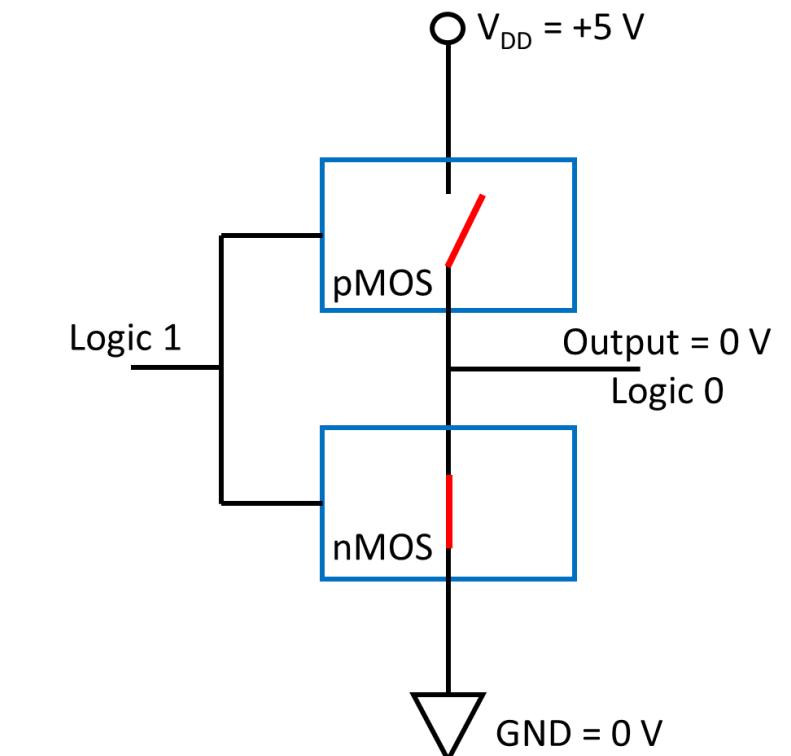
## CMOS Inverter



Logical model

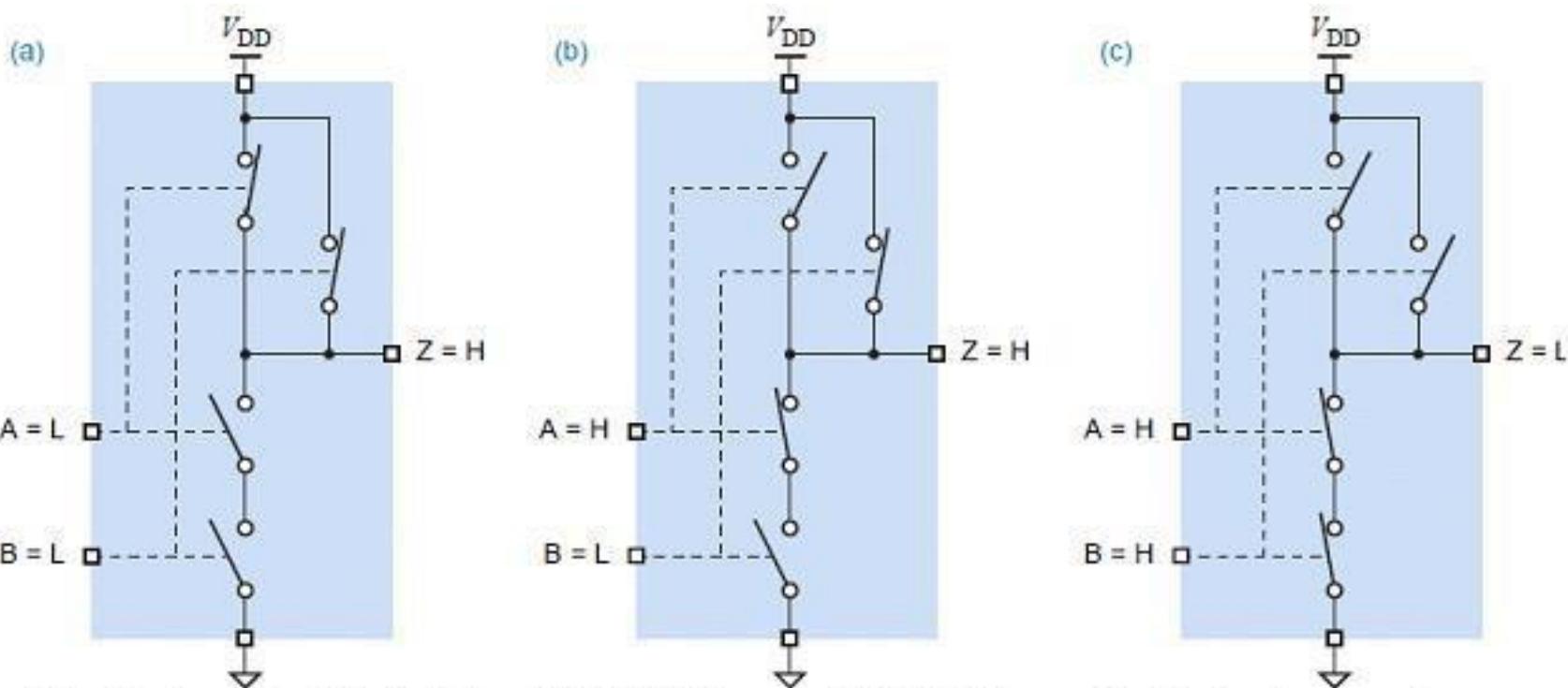
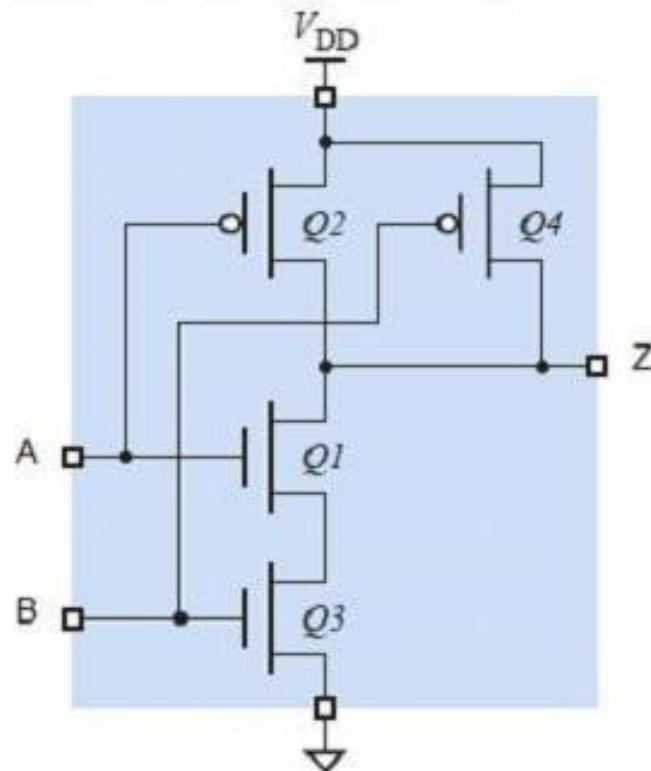


4



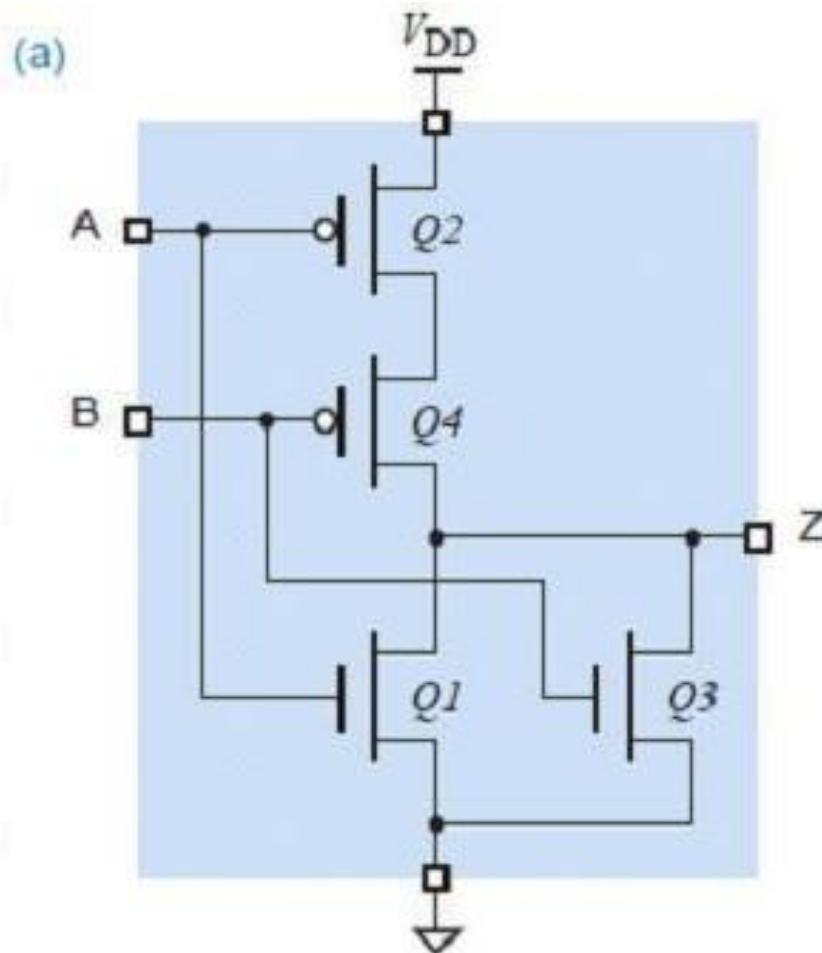
Switch model

# Implementation of logic gates



CMOS NAND Gate

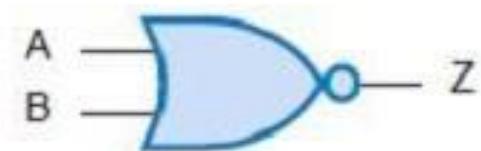
# Implementation of logic gates



(b)

A	B	$Q1$	$Q2$	$Q3$	$Q4$	Z
L	L	off	on	off	on	H
L	H	off	on	on	off	L
H	L	on	off	off	on	L
H	H	on	off	on	off	L

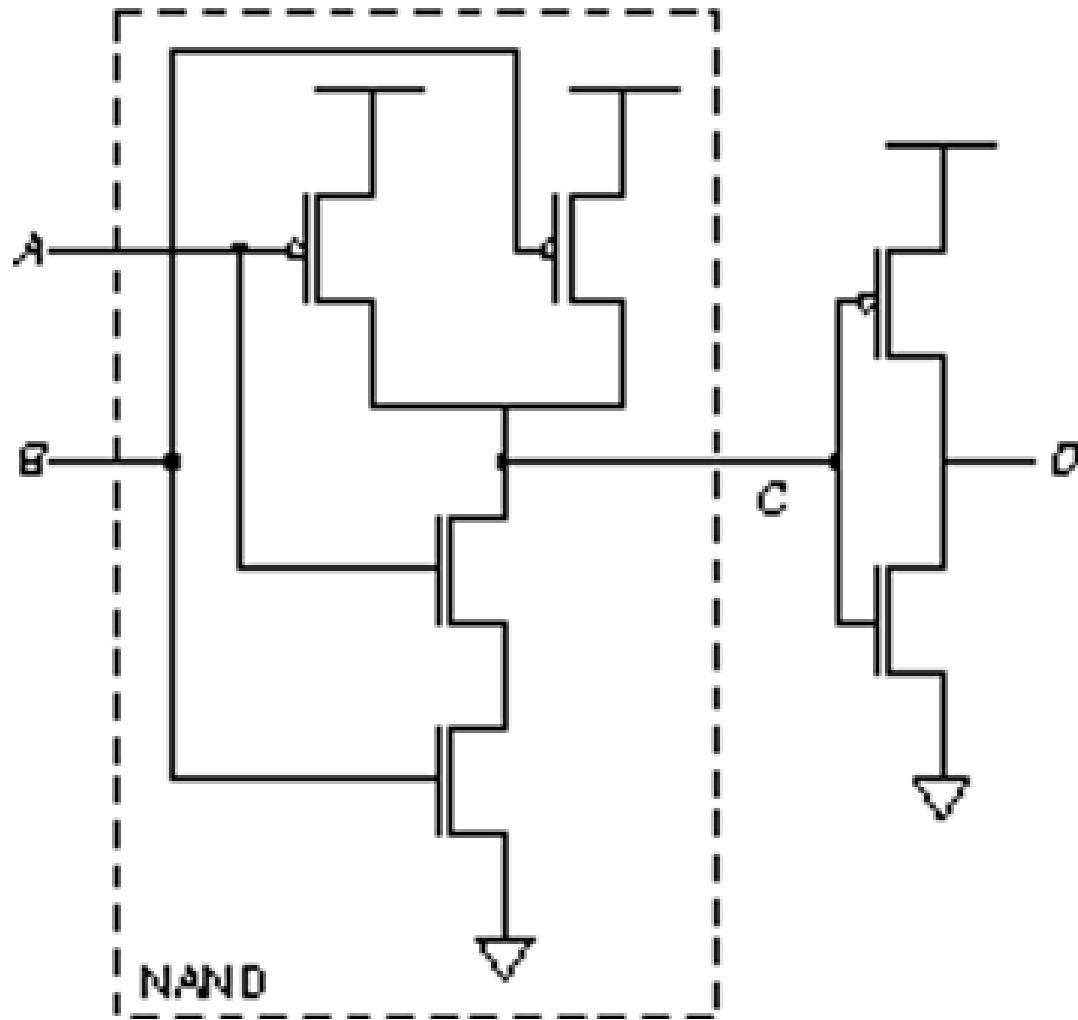
(c)



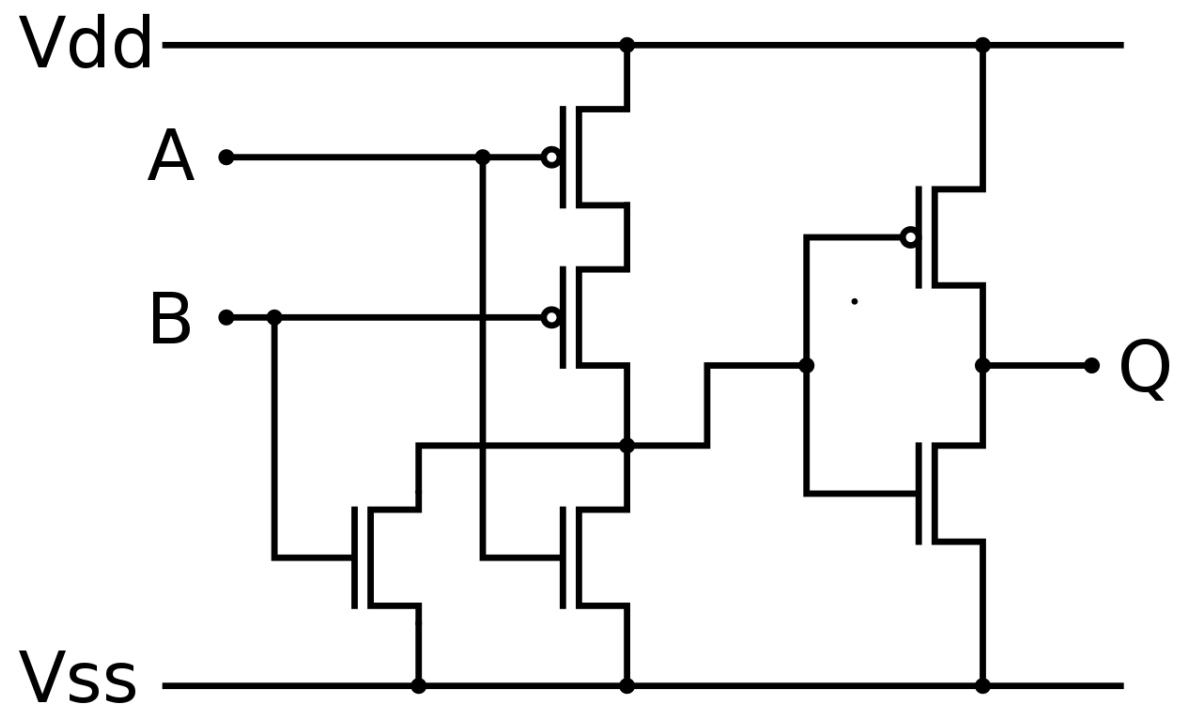
CMOS NOR gate

# Implementation of logic gates

AND gate

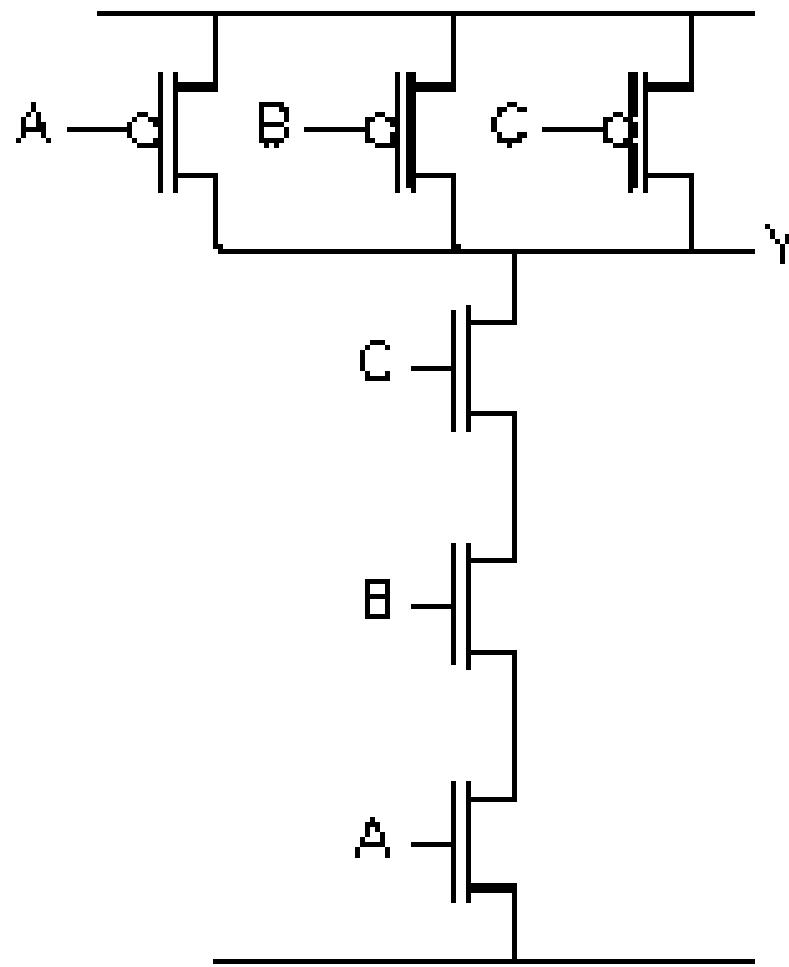


OR gate

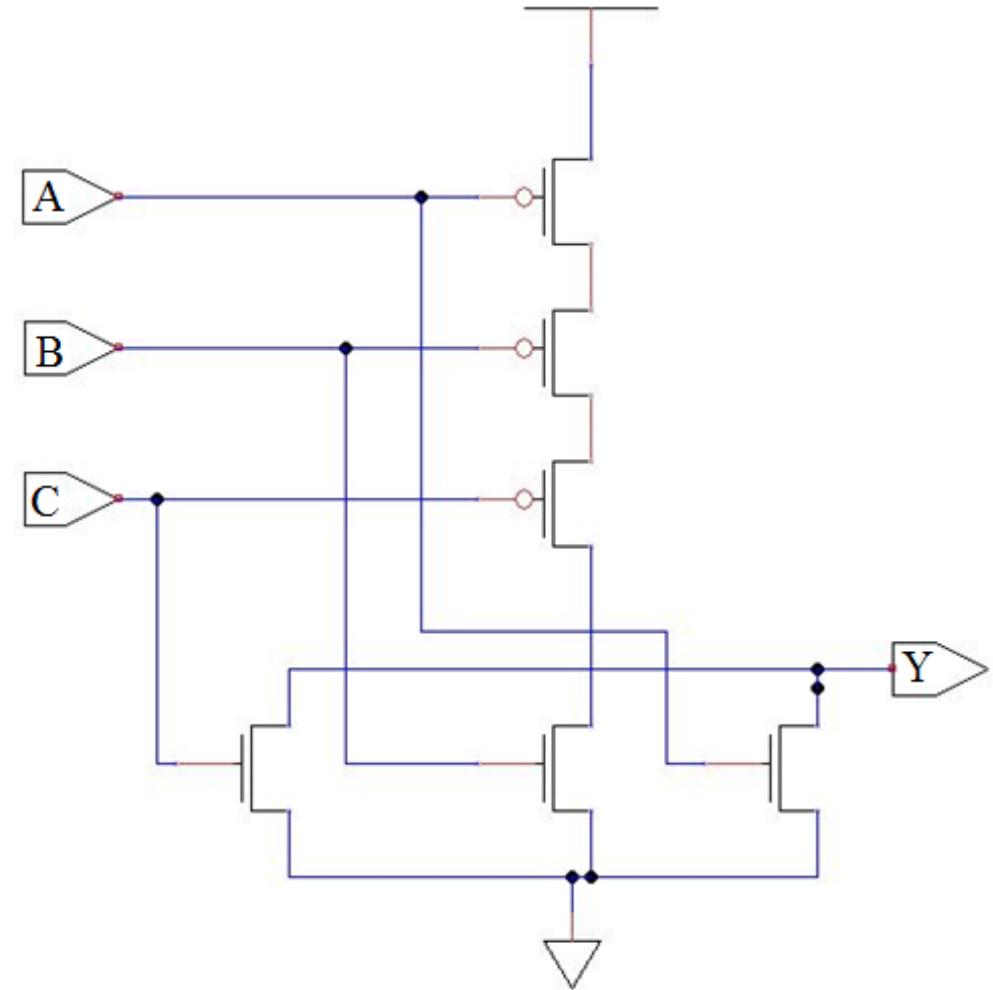


# Implementation of logic gates

3-input NAND gate



3-input NOR gate



# Circuit design

- The output binary functions listed in the truth table are simplified by any available method, such as algebraic manipulation, the map method, or a computer-based simplification program
- Frequently, there is a variety of simplified expressions from which to choose
- A designer must consider such constraints - *the number of gates, number of inputs to a gate, propagation time of the signal through the gates, number of interconnections, limitations of the driving capability of each gate (i.e., the number of gates to which the output of the circuit may be connected), and various other criteria that must be taken into consideration when designing integrated circuits*
- Since the importance of each constraint is dictated by the particular application, it is difficult to make a general statement about what constitutes an acceptable implementation
- In most cases, the simplification begins by satisfying an elementary objective, such as producing the simplified Boolean functions in a standard form
- Then the simplification proceeds with further steps to meet other performance criteria

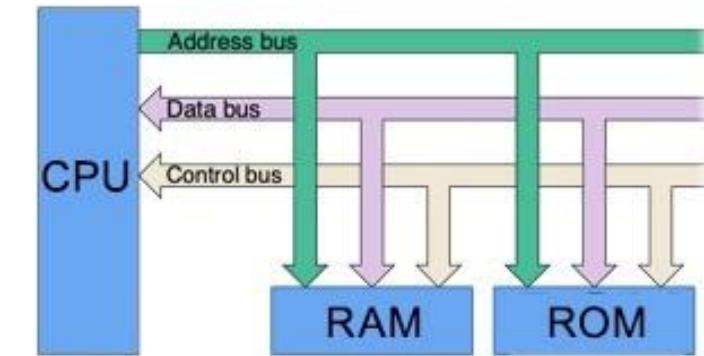
# Circuit design

- Say with everything else kept constant, we are most worried about the silicon area within which our design will fit
- To minimize the silicon real-estate needed, we have to reduce the number of transistors we use for a particular design – transistors are electronic switches that form the backbone of most of the modern day electronics
- A simple guide to remember:

Gate	Inputs	No of Transistors
NOT	1	2
NAND	N	2N
NOR	N	2N
AND	N	2N+2
OR	N	2N+2

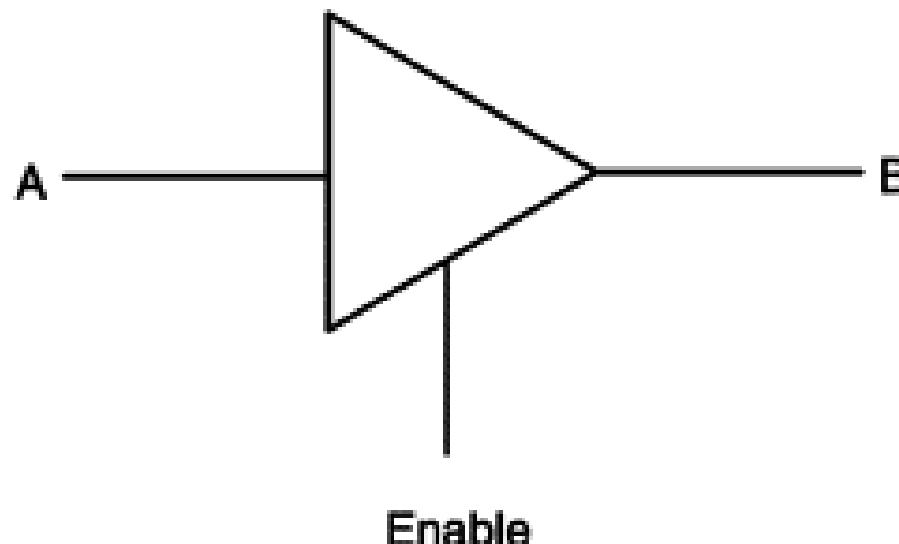
# The Other states

- This is REALLY important
- Apart from the two states of 0 and 1, there is a third state called the *high impedance state* denoted by Z
- When we say a particular pin is at HIGH or LOW state, we are assuming a driver behind it, i.e., the pin is *driven* to HIGH or LOW value
- This can be done through a transistor connecting the pin to either ground or Vcc
- However, if we do not connect a pin to either of HIGH or LOW, the pin is said to be in high impedance state or in Z state
- This concept is used extensively in the digital logic world to control buses



# The Other states

- Most logic gates only output HIGH or LOW, the third state is generally obtained using tristate buffers
- These are used just before the bus connections



Enable	A	B
0	0	z
0	1	z
1	0	0
1	1	1

# The Other states

---

- **Don't care state** : we do not care what the output in a particular case is, i.e., for a particular set of inputs - output can be either 1 or 0
- This is represented as X
- This can be either 0 or 1 and both are equally acceptable while forming logic circuits for a given function

# The Other states

- Consider a case where we need to know the value of B when A is TRUE, but when A is FALSE, we DON'T CARE!
- If this is the case, we can make the truth table for the function as shown
- In this case, because X can take either 0 or 1 value, we can simply make the desired function using an AND gate or as transfer of B

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

# The Other states

- Simplify the Boolean function

$$F(w, x, y, z) = \sum(1, 3, 7, 11, 15)$$

- We have one cluster of four:  $yz$  and one cluster of two:  $w'x'z$

- Thus, the function is

$$F = yz + w'x'z$$

		$wx$		$yz$			
		00	01	11	10		
		$m_0$	$m_1$	$m_3$	$m_2$		
w	00	0	1	1	0	$x$	$z$
	01	0	0	1	0 <th data-kind="ghost"></th> <th data-kind="ghost"></th>		
	11	0	0	1	0 <th data-kind="ghost"></th> <th data-kind="ghost"></th>		
	10	0	0	1	0 <th data-kind="ghost"></th> <th data-kind="ghost"></th>		

# The Other states

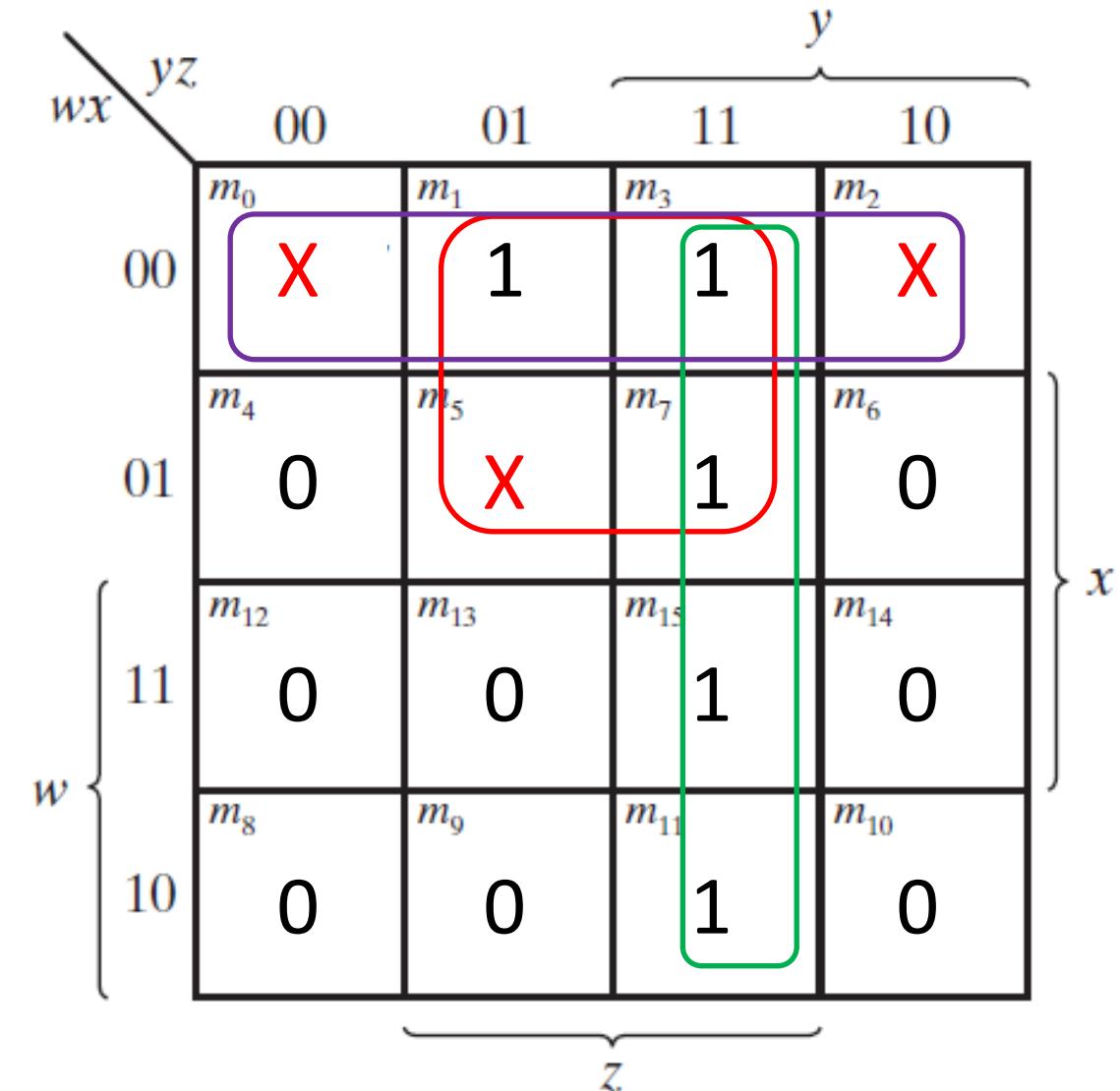
- Now, consider the same function

$$F(w, x, y, z) = \sum(1, 3, 7, 11, 15)$$

which has the don't-care conditions

$$d(w, x, y, z) = \sum(0, 2, 5)$$

- Now, we have two clusters of four squares:  $yz$  and  $w'z$
- This is because  $m_5$  can be 1, and it is ok if  $m_0$  and  $m_2$  are 0
- Thus, the function is  $F = yz + w'z$
- The function can also be simplified as  $F = yz + w'x'$



# The Other states

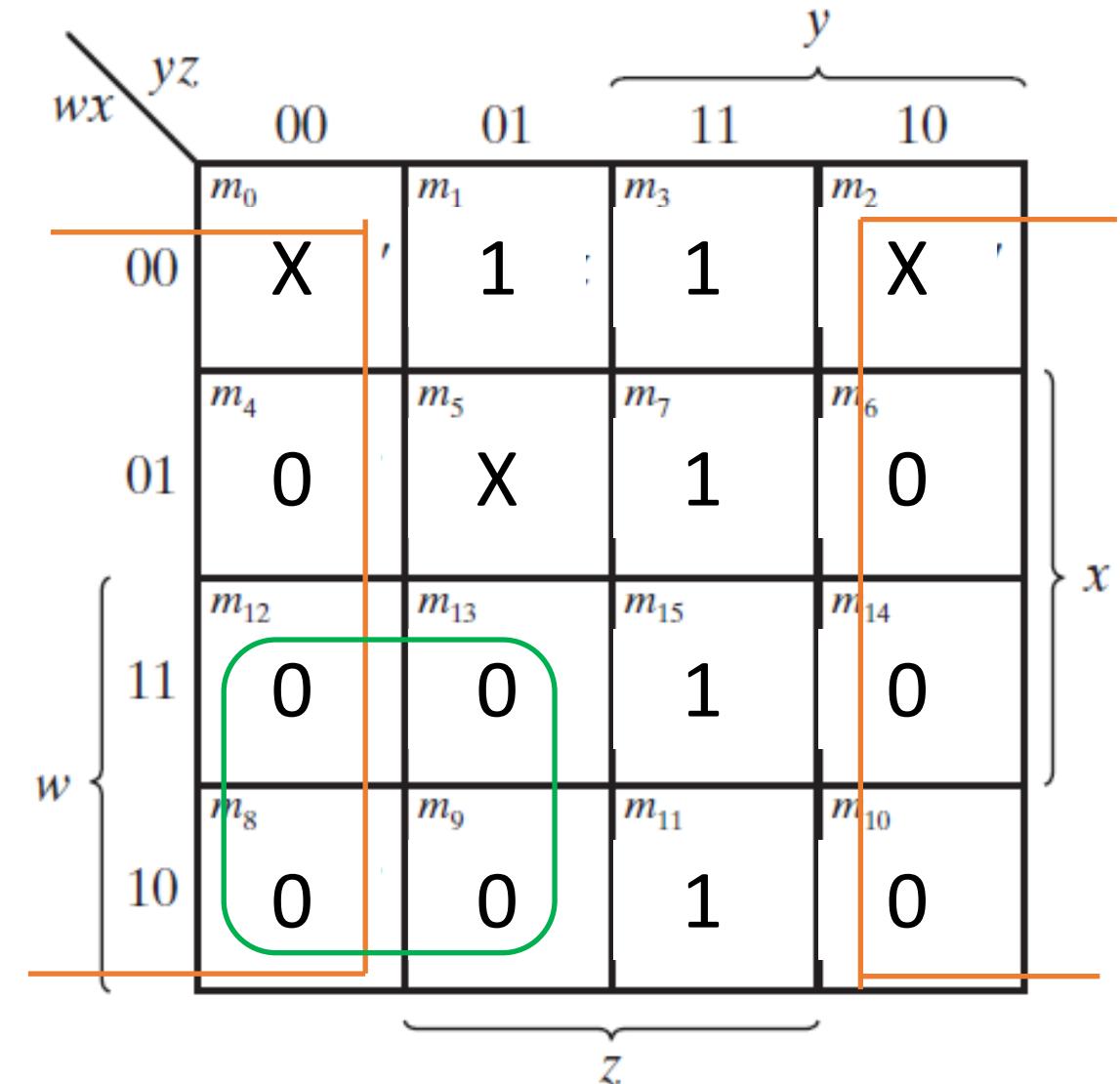
- Functions  $F = yz + w'z$  and  $F = yz + w'x'$  are different functions
- However, both expressions include minterms 1, 3, 7, 11, and 15 that make the function  $F$  equal to 1
- The don't-care minterms 0, 2, and 5 are treated differently in each expression
- The first expression includes minterms 0 and 2 with the 1's and leaves minterm 5 with the 0's
- The second expression includes minterm 5 with the 1's and leaves minterms 0 and 2 with the 0's
- The two expressions represent two functions that are not equal but follow the logic statement

		yz			
		wx		00	01
		00		$m_0$	$m_1$
w	00	X		1	1
	01	$m_4$	$m_5$	X	1
	11	0		0	0
	10	$m_{12}$	$m_{13}$	$m_{15}$	$m_{14}$
		00		1	0
		01	$m_8$	$m_9$	$m_{11}$
		11	0	0	1
		10			0

$y$   
 $x$   
 $z$

# The Other states

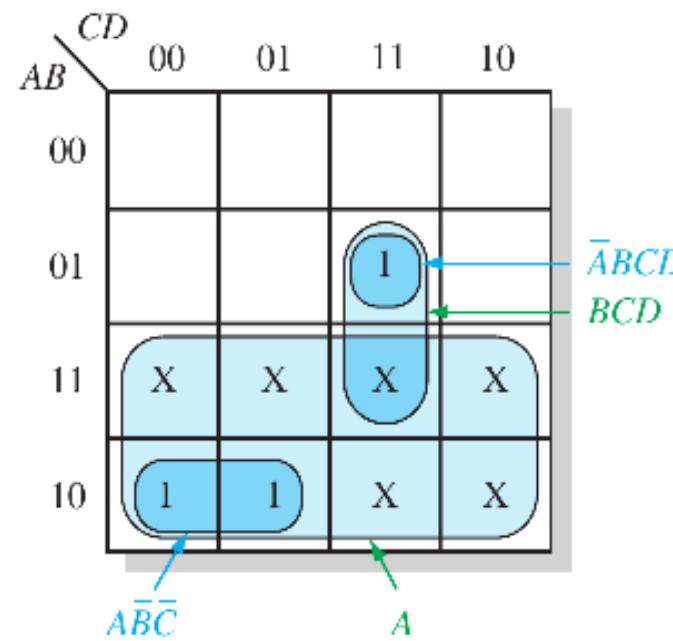
- The key question is: Are these the simplest possible representations for the given function?  $F = yz + w'z$  and  $F = yz + w'x'$
- What if we look at the product of sum simplification?
- We can get a cluster of 8 squares:  $z'$
- And a cluster of four squares:  $wy'$
- Thus, the function can be represented as  
$$F = (z' + wy')' = z(w' + y)$$



# The Other states

Inputs				Output
A	B	C	D	Y
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	X
1	0	1	1	X
1	1	0	0	X
1	1	0	1	X
1	1	1	0	X
1	1	1	1	X

Don't cares



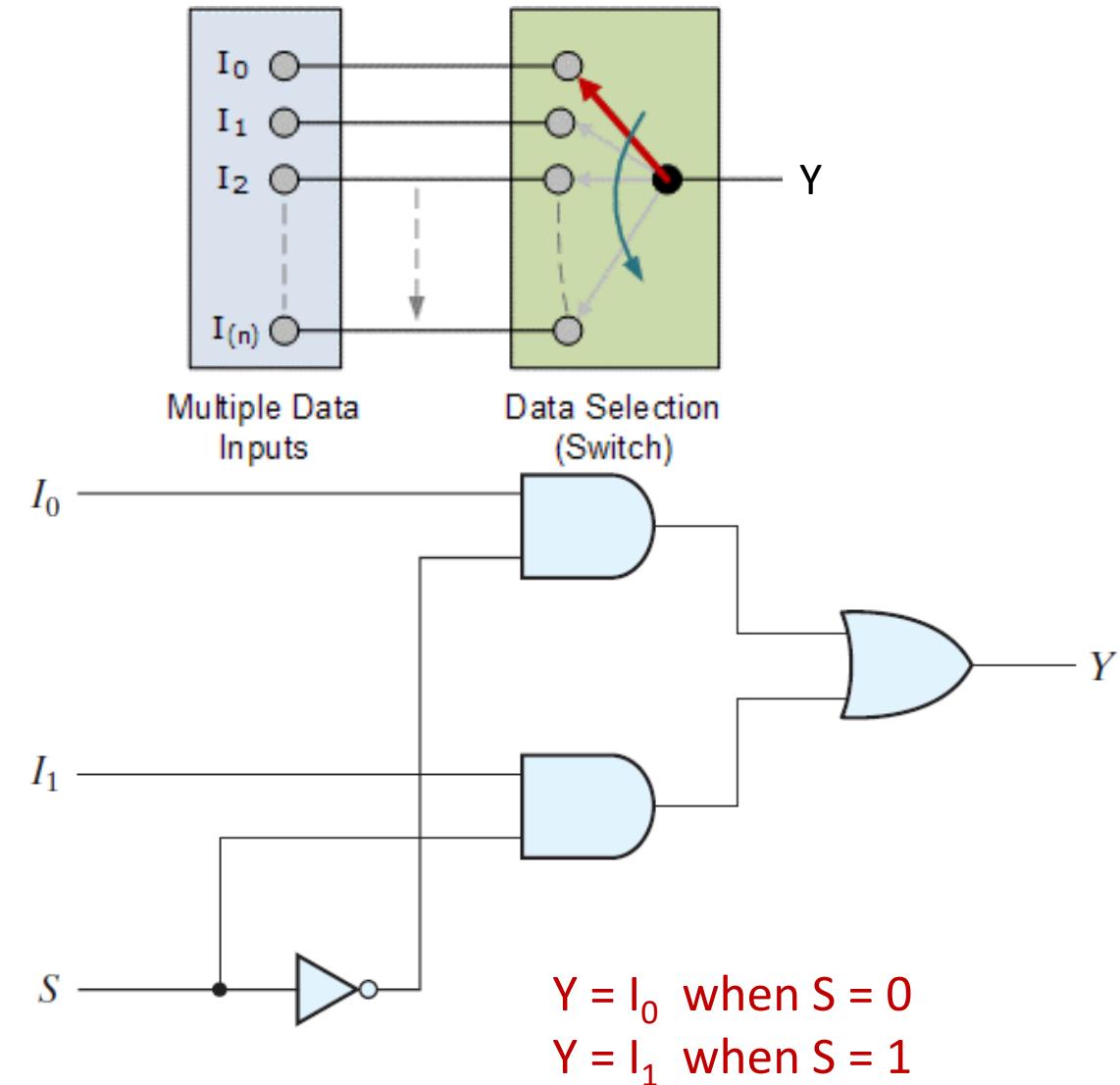
Without "don't cares"  $Y = A\bar{B}\bar{C} + \bar{A}B\bar{C}D$

With "don't cares"  $Y = A + BCD$

# Lecture 12 – Combinational logic circuits

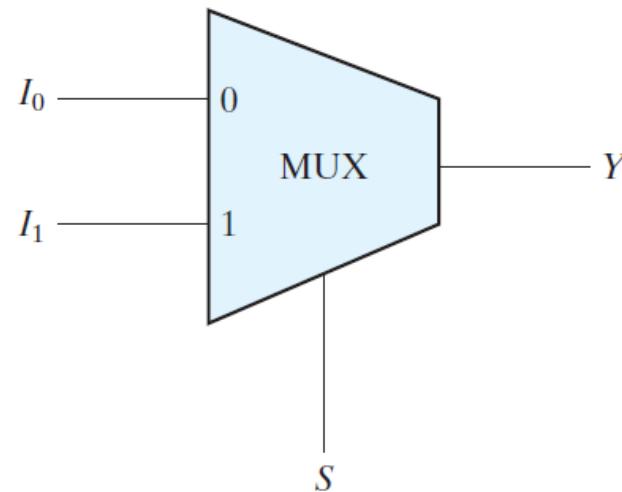
# Multiplexer

- A multiplexer is a combinational circuit that selects binary information from one of many input lines and directs it to a single output line
- The selection of a particular input line is controlled by a set of *selection lines*
- Normally, there are  $2^n$  input lines and  $n$  selection lines whose bit combinations determine which input is selected
  - Eg: A two-to-one-line multiplexer connects one of two 1-bit sources to a common destination
- The multiplexer acts like an electronic switch that selects one of the sources



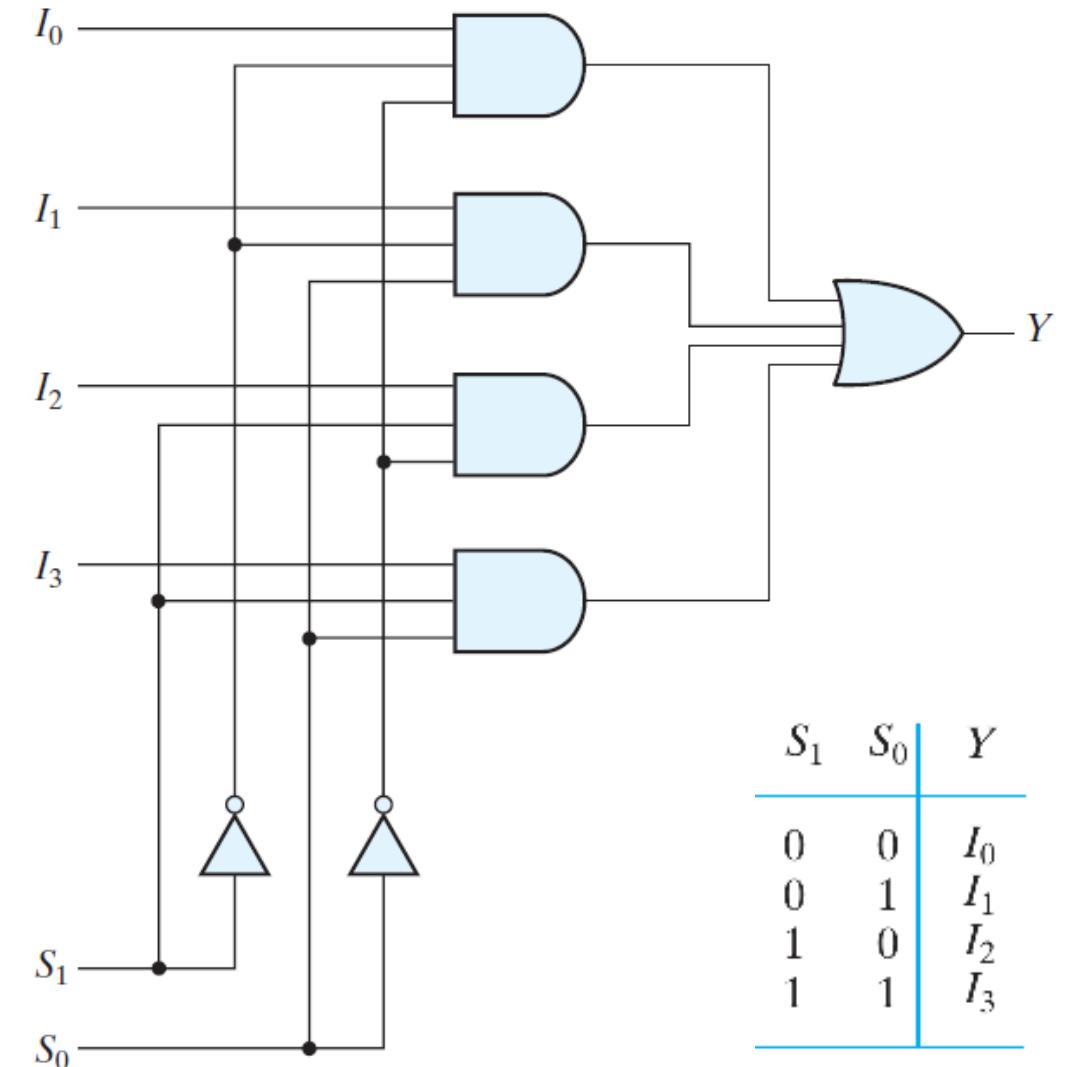
# Multiplexer

The block diagram of a multiplexer (also called **MUX**) is sometimes depicted by a wedge-shaped symbol



# Multiplexer

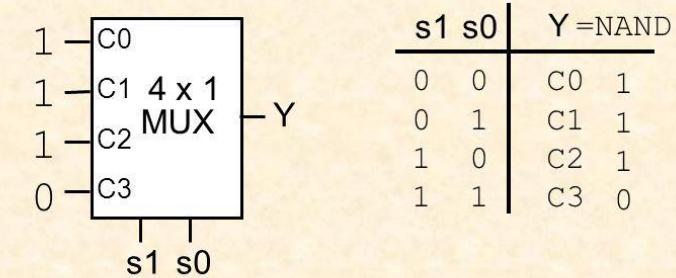
- Similarly, we can make a **four-to-one-line multiplexer**
- Each of the four inputs,  $I_0$  through  $I_3$ , is applied to one input of an AND gate
- The outputs of the AND gates are applied to a single OR gate that provides the one-line output
- A multiplexer is also called a ***data selector***, since it selects one of many inputs and steers the binary information to the output line
- In general, a  $2^n$ -to-1-line multiplexer is constructed ANDing each input line with  $2^n$  minterms*
- The outputs of the AND gates are applied to a single OR gate



# Multiplexer

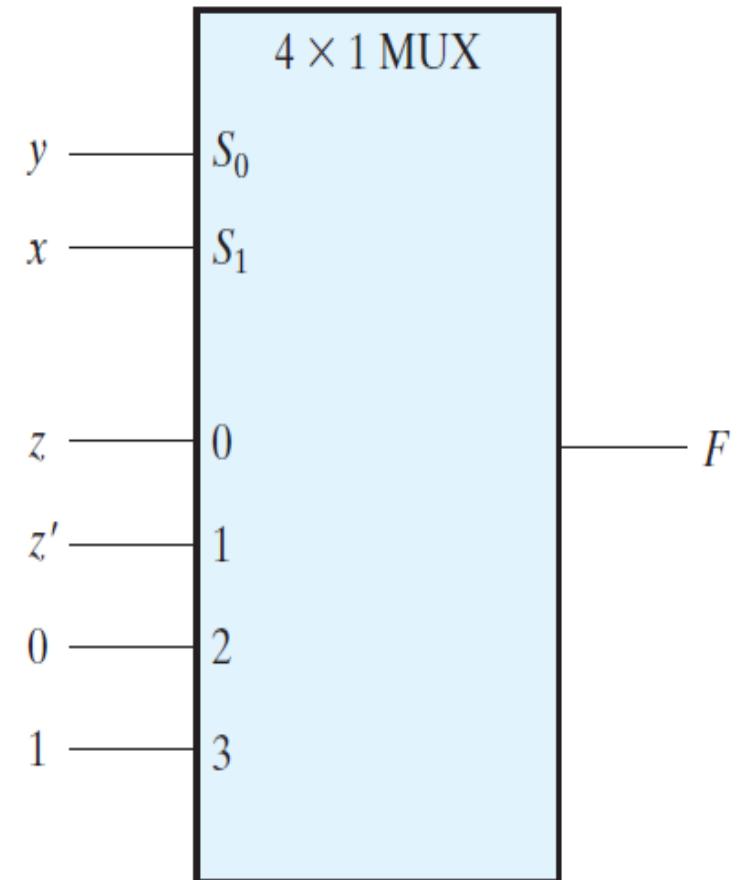
- *We can use MUXes to implement Boolean functions*
- The minterms of a function are generated in a multiplexer by the circuit associated with the selection inputs
- The individual minterms can be selected by the data inputs, thereby providing a method of implementing a Boolean function of  $n$  variables with a multiplexer that has  $n$  selection inputs and  $2^n$  data inputs, one for each minterm
- Example: implement NAND

NAND implementation using MUX



# Multiplexer

- There is a neat trick to obtain a more efficient method for implementing a Boolean function of  $n$  variables with a multiplexer that has  $n - 1$  selection inputs
- The first  $n - 1$  variables of the function are connected to the selection inputs of the multiplexer
- The remaining single variable of the function is used for the data inputs
- Say we have a three variable function  $F(x,y,z)$
- We take a 4to1 MUX (with two input select lines) and each data input of the multiplexer will be  $z$ ,  $z'$ , 1, or 0



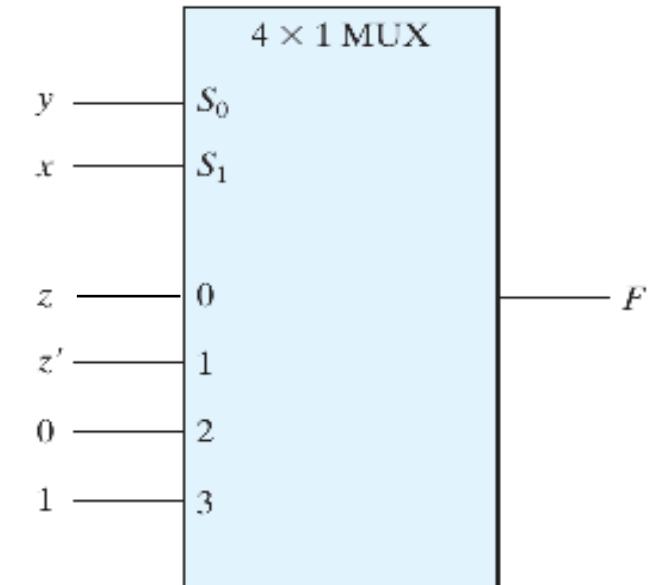
# Multiplexer

- Consider the function:

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

- This function of three variables can be implemented with a four-to-one-line multiplexer
- The two variables  $x$  and  $y$  are applied to the selection lines in that order;  $x$  is connected to the  $S_1$  input and  $y$  to the  $S_0$  input
- The values for the data input lines are determined from the truth table of the function
- When  $xy = 00$ , output  $F$  is equal to  $z$  because  $F = 0$  when  $z = 0$  and  $F = 1$  when  $z = 1$
- This requires that variable  $z$  be applied to data input 0
- In a similar fashion, we can determine the required input to data lines 1, 2, and 3 from the value of  $F$  when  $xy = 01, 10$ , and  $11$ , respectively

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



# Multiplexer

- The general procedure for implementing any Boolean function of  $n$  variables with a multiplexer with  $n - 1$  selection inputs and  $2^{n-1}$  data inputs follows from the previous example
  1. To begin with, Boolean function is listed in a truth table
  2. Then the most significant  $n - 1$  variables in the table are applied to the selection inputs of the multiplexer
  3. For each combination of the selection variables, we evaluate the output as a function of the last variable
  4. This function can be 0, 1, the variable, or the complement of the variable
  5. These values are then applied to the data inputs in the proper order

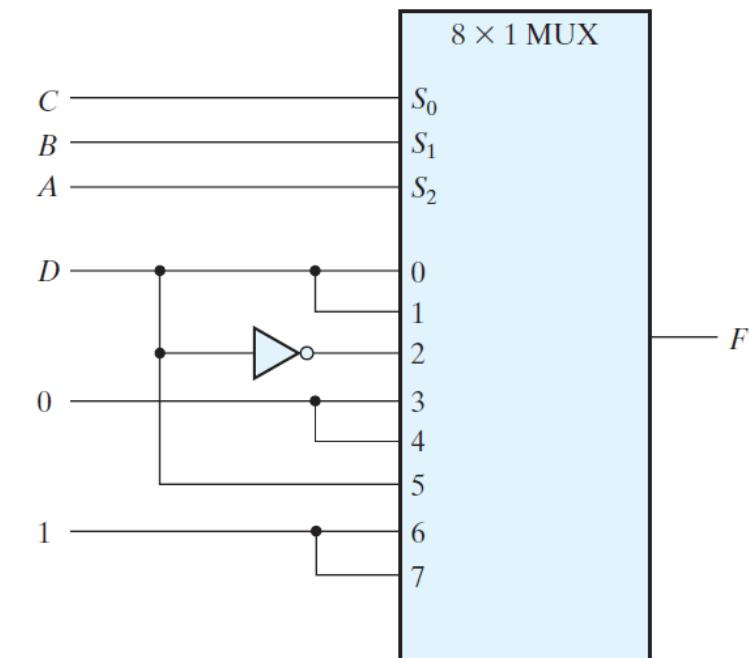
$$F(A, B, C, D) = \sum (1, 3, 4, 11, 12, 13, 14, 15)$$

# Multiplexer

- The general procedure for implementing any Boolean function of  $n$  variables with a multiplexer with  $n - 1$  selection inputs and  $2^{n-1}$  data inputs follows from the previous example
- To begin with, Boolean function is listed in a truth table
  - Then the most significant  $n - 1$  variables in the table are applied to the selection inputs of the multiplexer
  - For each combination of the selection variables, we evaluate the output as a function of the last variable
  - This function can be 0, 1, the variable, or the complement of the variable
  - These values are then applied to the data inputs in the proper order

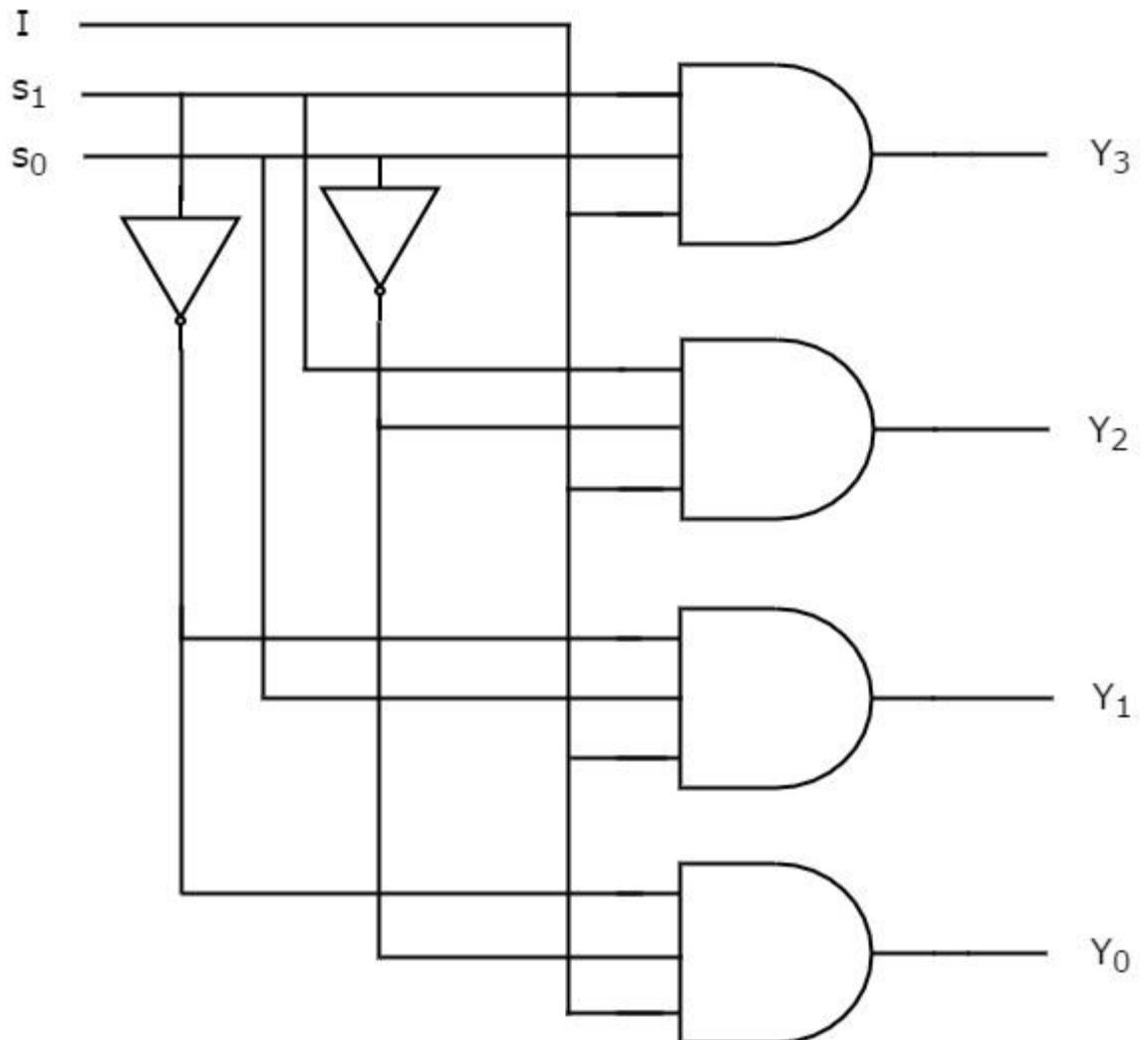
$$F(A, B, C, D) = \sum (1, 3, 4, 11, 12, 13, 14, 15)$$

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



# Demultiplexer

- Demultiplexers (Demux) do the exact opposite of MUX operation – take a single line input and direct it to an output line depending on the select line input
- 1-to- $2^n$  Demux will have n select lines
- We again make minterms from the available inputs and AND it with the single data line
- Based on the input signals the particular output is connected to the data line and all other outputs are zero



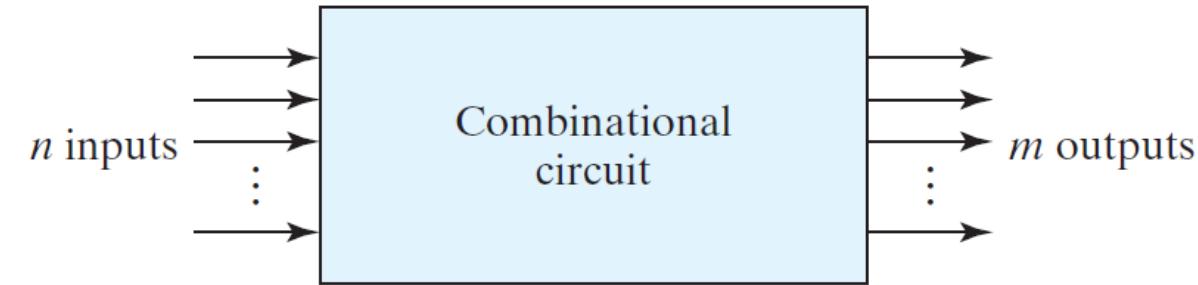
# Combinational circuits

---

- Logic circuits for digital systems may be *combinational or sequential*
- A combinational circuit consists of logic gates whose outputs at any time are determined from only the present combination of inputs
- A combinational circuit performs an operation that can be specified logically by a set of Boolean functions
- *In contrast, sequential circuits employ storage elements in addition to logic gates*
- Their outputs are a function of the inputs and the state of the storage elements
- Because the state of the storage elements is a function of previous inputs, *the outputs of a sequential circuit depend not only on present values of inputs, but also on past inputs*, and the circuit behavior must be specified by a time sequence of inputs and internal states

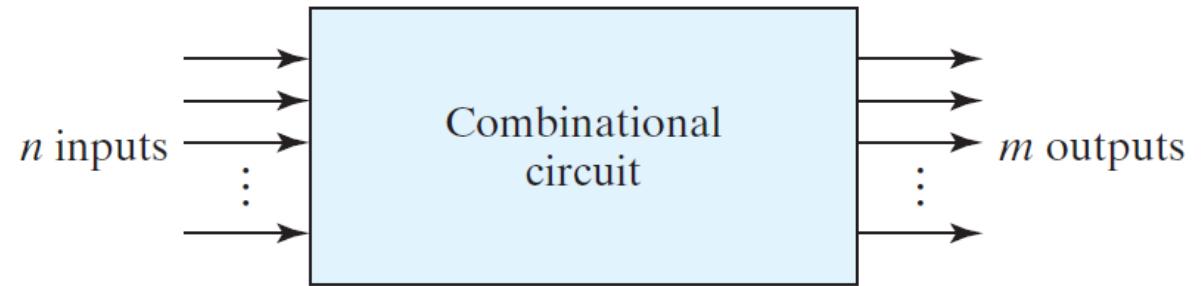
# Combinational circuits

- A combinational circuit consists of an interconnection of logic gates
- Combinational logic gates react to the values of the signals at their inputs and produce the value of the output signal, transforming binary information from the given input data to a required output data
- Each input and output variable exists physically as an analog signal whose values are interpreted to be a binary signal that represents logic 1 and logic 0



# Combinational circuits

- For  $n$  input variables, there are  $2^n$  possible combinations of the binary inputs
- For each possible input combination, there is one possible value for each output variable
- Thus, a combinational circuit can be specified with a truth table that lists the output values for each combination of input variables
- A combinational circuit also can be described by  $m$  Boolean functions, one for each output variable
- Each output function is expressed in terms of the  $n$  input variables



# Circuit design

- The design of combinational circuits starts from the specification of the design objective and culminates in a logic circuit diagram or a set of Boolean functions from which the logic diagram can be obtained
- The procedure involves the following steps:
  1. From the specifications of the circuit, determine the required number of inputs and outputs and assign a symbol to each
  2. Derive the truth table that defines the required relationship between inputs and outputs
  3. Make the K-map, if necessary
  4. Obtain the simplified Boolean functions for each output as a function of the input variables
  5. Draw the logic diagram and verify the correctness of the design (manually or by simulation)
- Here we go...

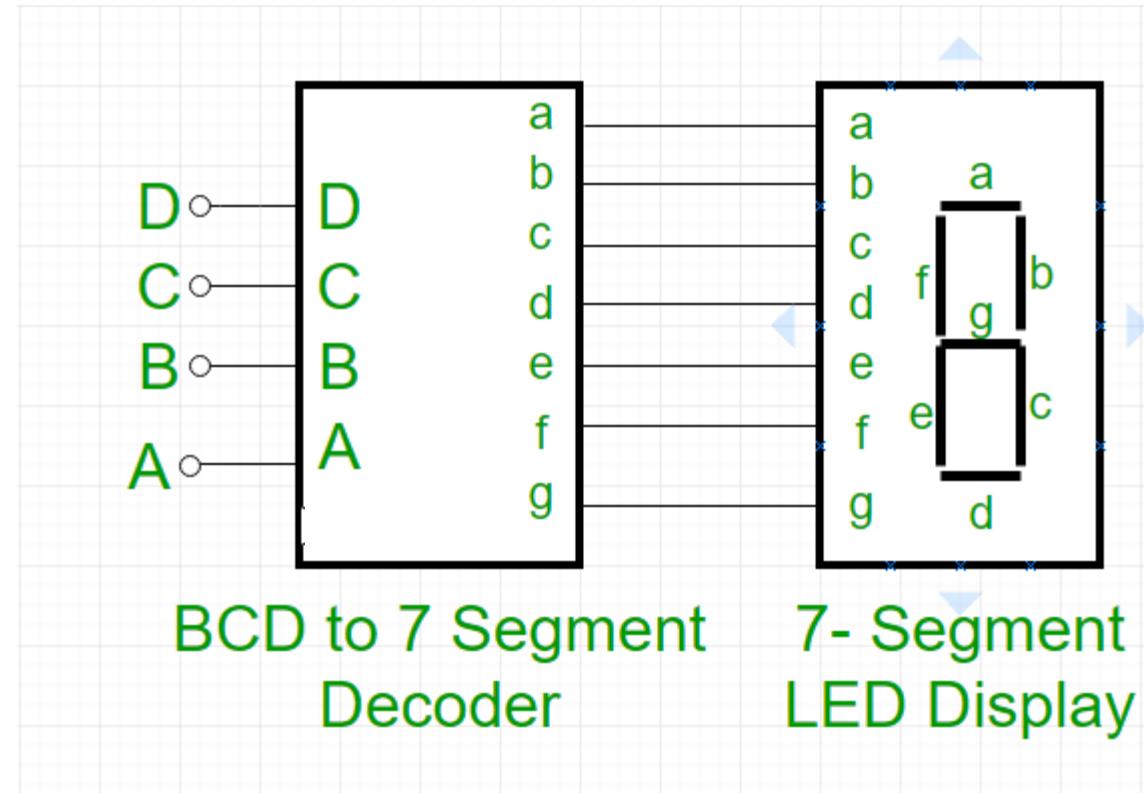


# The 7-segment decoder



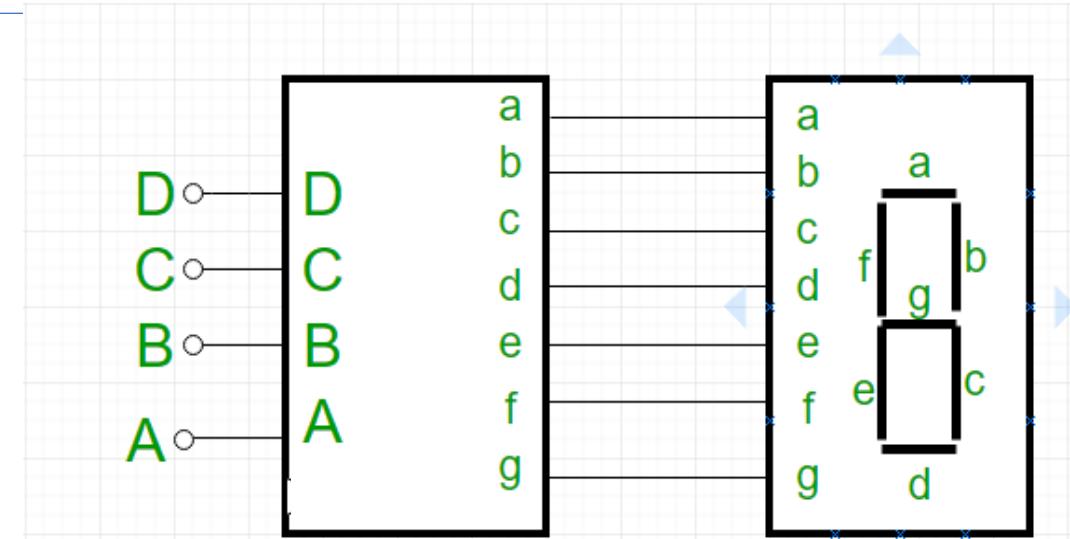
# 7-segment decoder

- We discussed many ways of representing symbols using binary variables – signed magnitude, 2's complement, BCD, ASCII etc.
- Going from one representation to the other may be considered as an encoding/decoding operation
- Consider the practical problem of displaying binary numbers using a 7-segment LED display
- For this, we need to “decode” the binary information into the display inputs



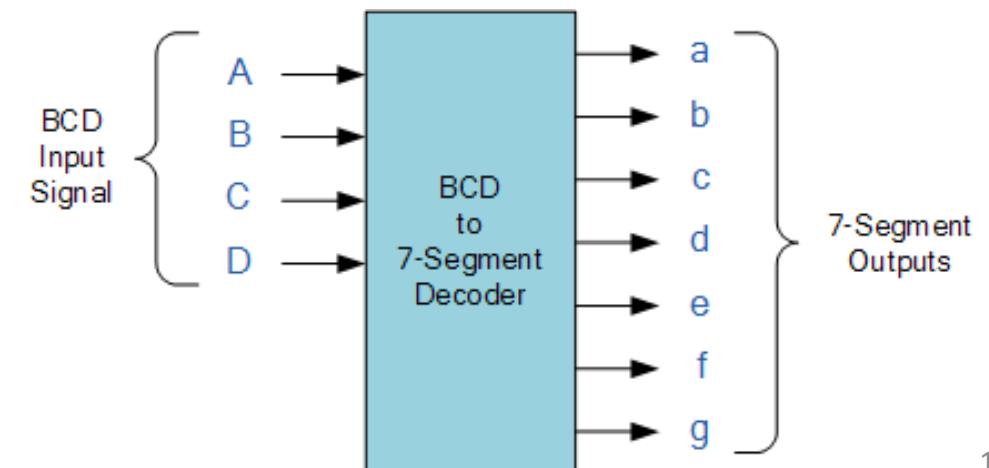
# 7-segment decoder

- First, we realize that there are 4 inputs and 7 outputs!
- Thus, the truth-table will be a 16 row table with 7 different columns for 7 outputs
- Hence, there will be 7 different functions we will be implementing to make this decoder
- The other thing we need to realize is whether a particular output should be **HIGH** or **LOW** for the LED to glow?



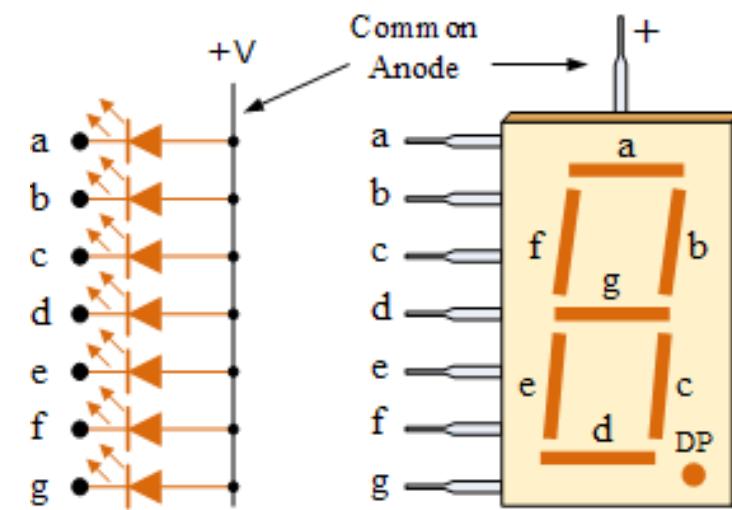
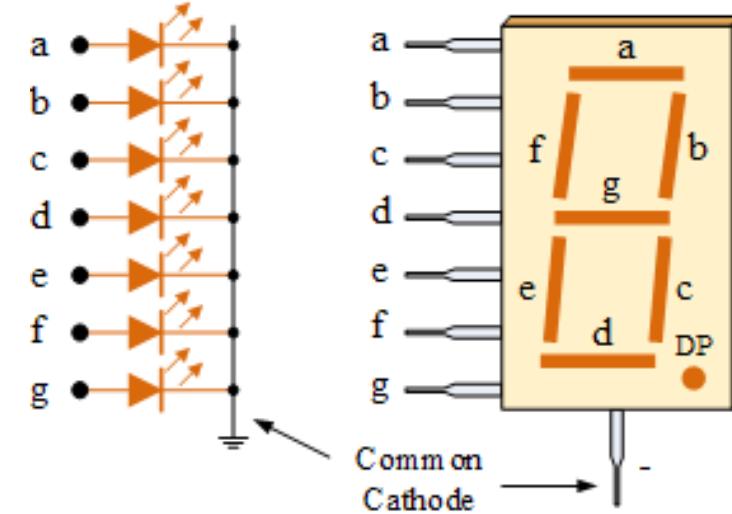
BCD to 7 Segment  
Decoder

7- Segment  
LED Display



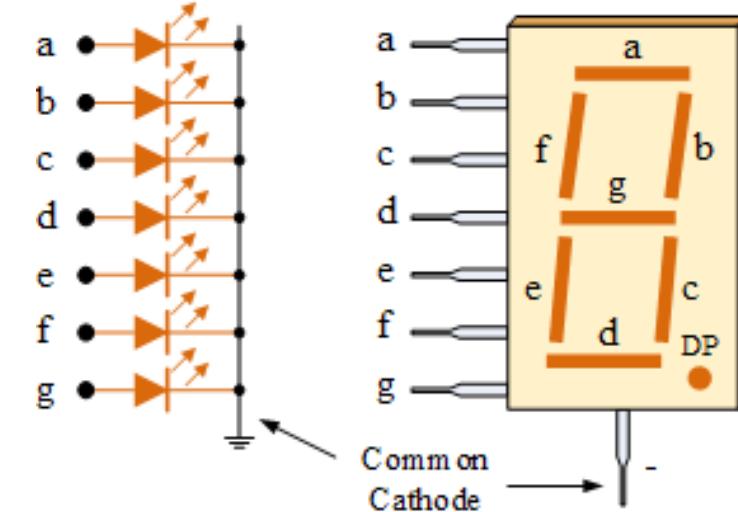
# 7-segment decoder

- The other thing we need to realize is whether a particular output should be HIGH or LOW for the LED to glow?
- To know the answer to this, we see that there are two types of 7-segment LED displays – **common anode** and **common cathode**
- The common cathode connects all LED cathodes to a common ground, thus, when input is high LED glows
- Conversely, the common anode connects all LED anodes to  $+V_{cc}$ , thus, when input is low, LED glows



# 7-segment decoder

- Let us choose common cathode LED display to make the function
- Thus, we can make the truth-table
- Obviously, the BCD system only goes from 0 to 9, while we have 16 rows for 4 inputs
- In the other rows, we fill all the outputs as *don't care*, because we are sure that these are not going to be input anyway (trust the engineer before you)
- Thus, we have six don't care conditions

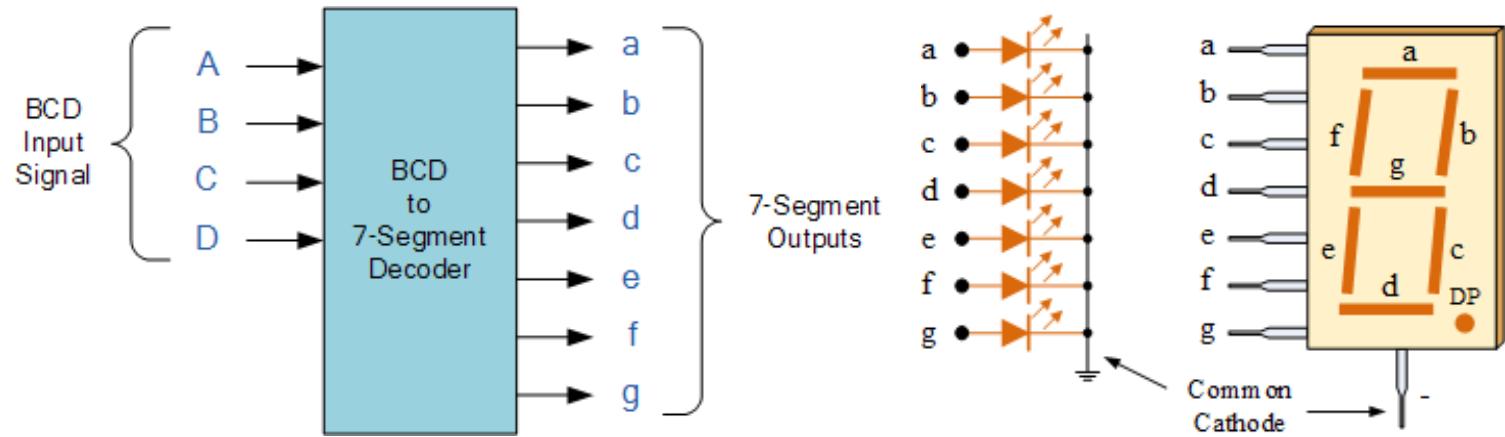


A	B	C	D	a	b	c	d	e	f	g
0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	0	1	1	0	0	0	0
0	0	1	0	1	1	0	1	1	0	1
0	0	1	1	1	1	1	1	0	0	1
0	1	0	1	0	1	1	0	0	1	1
0	1	0	1	1	0	1	1	0	1	1
0	1	1	0	1	0	1	1	1	1	1
0	1	1	1	1	1	1	0	0	0	0
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	1	0	1	1

# Lecture 13 – Combinational logic circuits 2

# 7-segment decoder

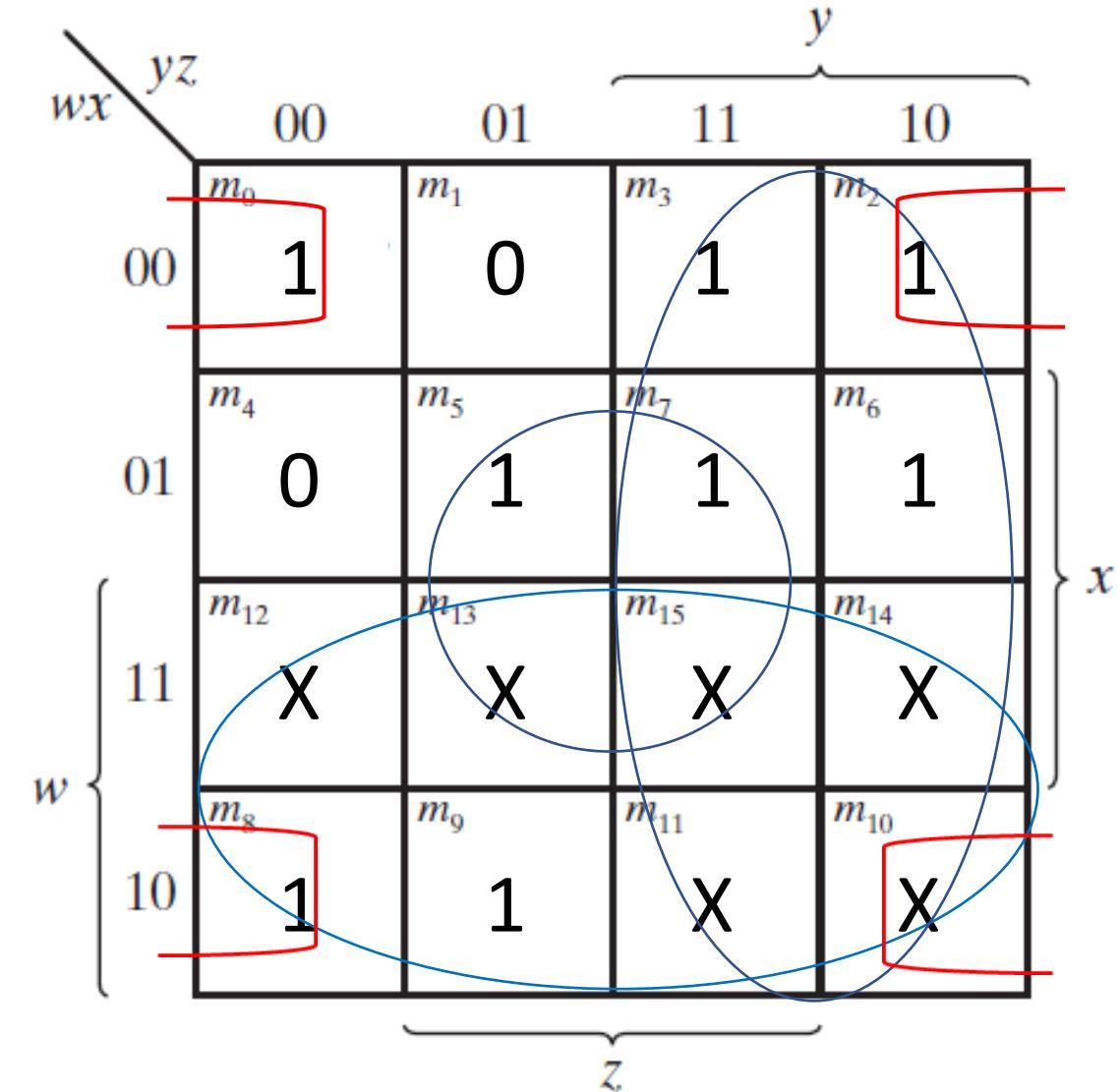
- Let us choose common cathode LED display to make the function
- The BCD system only goes from 0 to 9, while we have 16 rows for 4 inputs
- In the other rows, we fill all the outputs as **don't care**, because we are sure that these are not going to be input anyway
- Thus, we have six don't care conditions



A	B	C	D	a	b	c	d	e	f	g
0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	0	1	1	0	0	0	0
0	0	1	0	1	1	0	1	1	0	1
0	0	1	1	1	1	1	1	0	0	1
0	1	0	1	0	1	1	0	0	1	1
0	1	0	1	1	0	1	1	0	1	1
0	1	1	0	1	0	1	1	1	1	1
0	1	1	1	1	1	1	0	0	0	0
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	1	0	1	1

# 7-segment decoder

- Thus, the K-map for output  $a$  will look like this ( $A=w$ ,  $B=x$ ,  $C=y$ ,  $D=z$ )
  - We have two clusters of 8:  $y$  and  $w$
  - Two clusters of four:  $xz$  and  $x'z'$
- Thus, the logic function for  $a$  can be:  
$$F_a = y + w + xz + x'z'$$
- Or we can have PoS as:
  - One cluster of two:  $xy'z'$
  - One min-term:  $w'x'y'z$
- Thus,  
$$F_a = (x' + y + z)(w + x + y + z')$$
- This can be done for all the other outputs

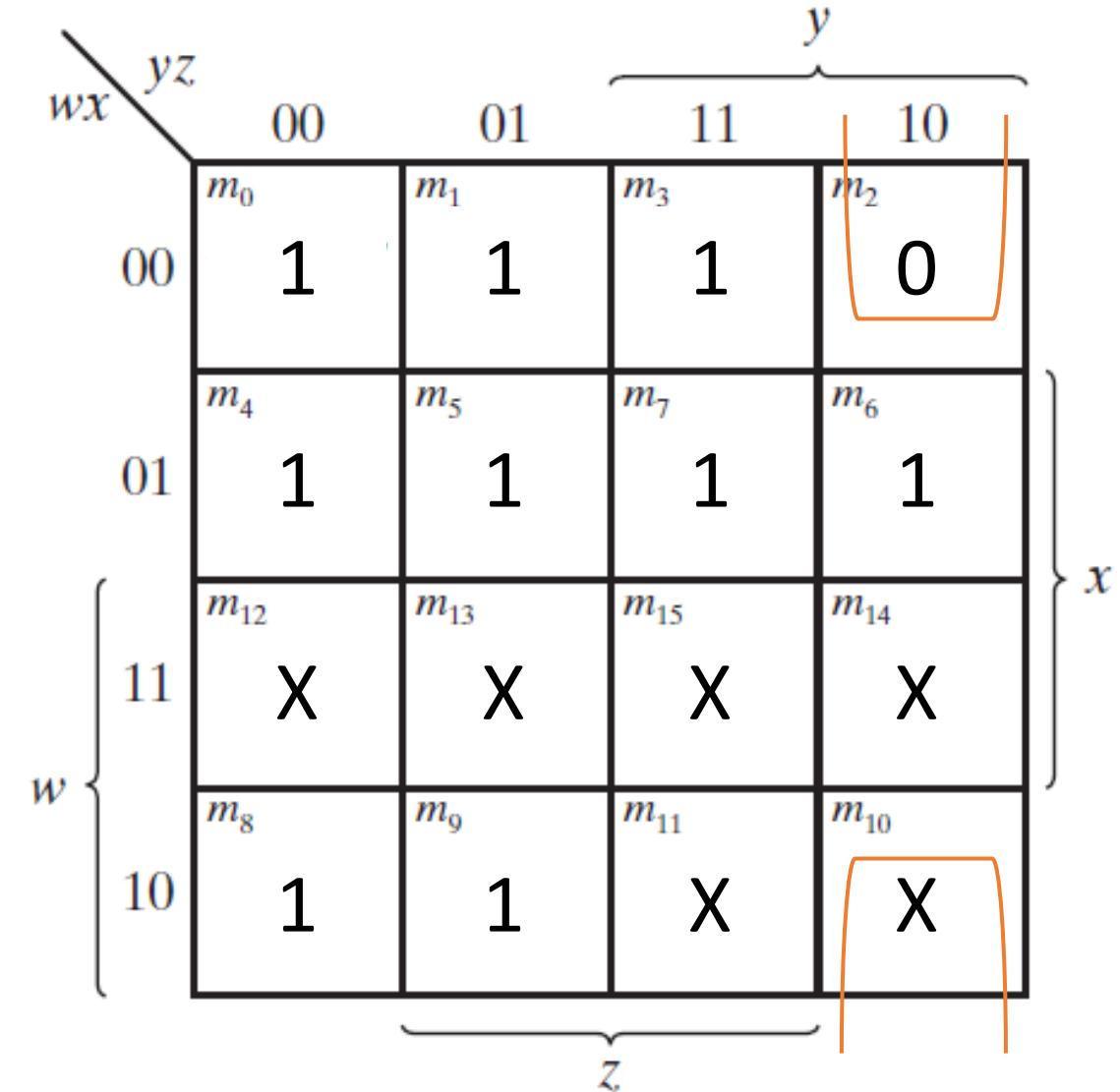


# 7-segment decoder

- Thus, the K-map for output  $c$  will look like this
- Consider PoS:
- The max term cluster of two is represented by  $yz'x'$
- In PoS form:

$$c = y' + z + x$$

This can be done for all the other outputs





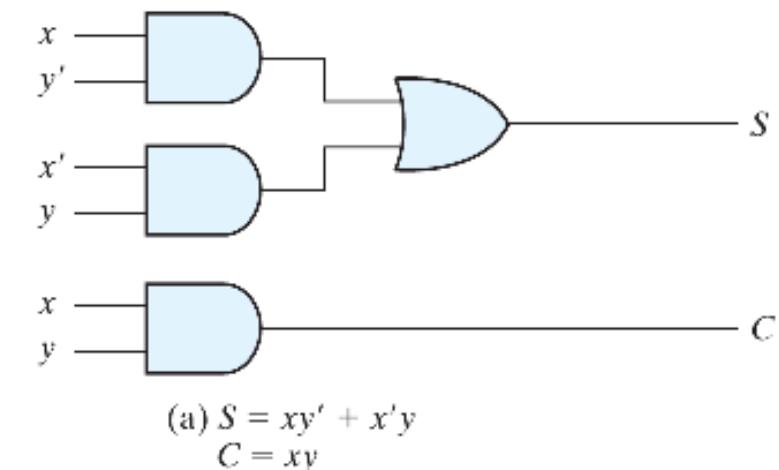
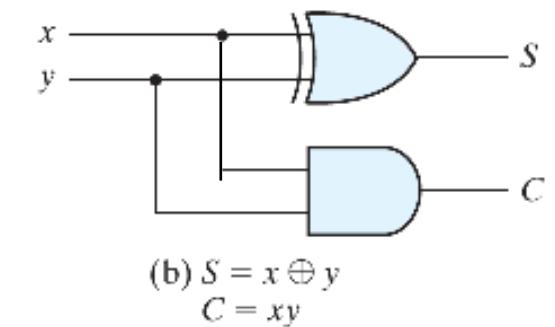
# The Binary adder

# Binary adder

- Digital computers perform a variety of information-processing tasks
- Among the functions encountered are the various arithmetic operations
- The most basic arithmetic operation is the addition of two binary digits:
  - $0+0=0, 1+0=1, 0+1=1, 1+1 = 10$
- A combinational circuit that performs the addition of two bits is called a *half adder*

Half Adder

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

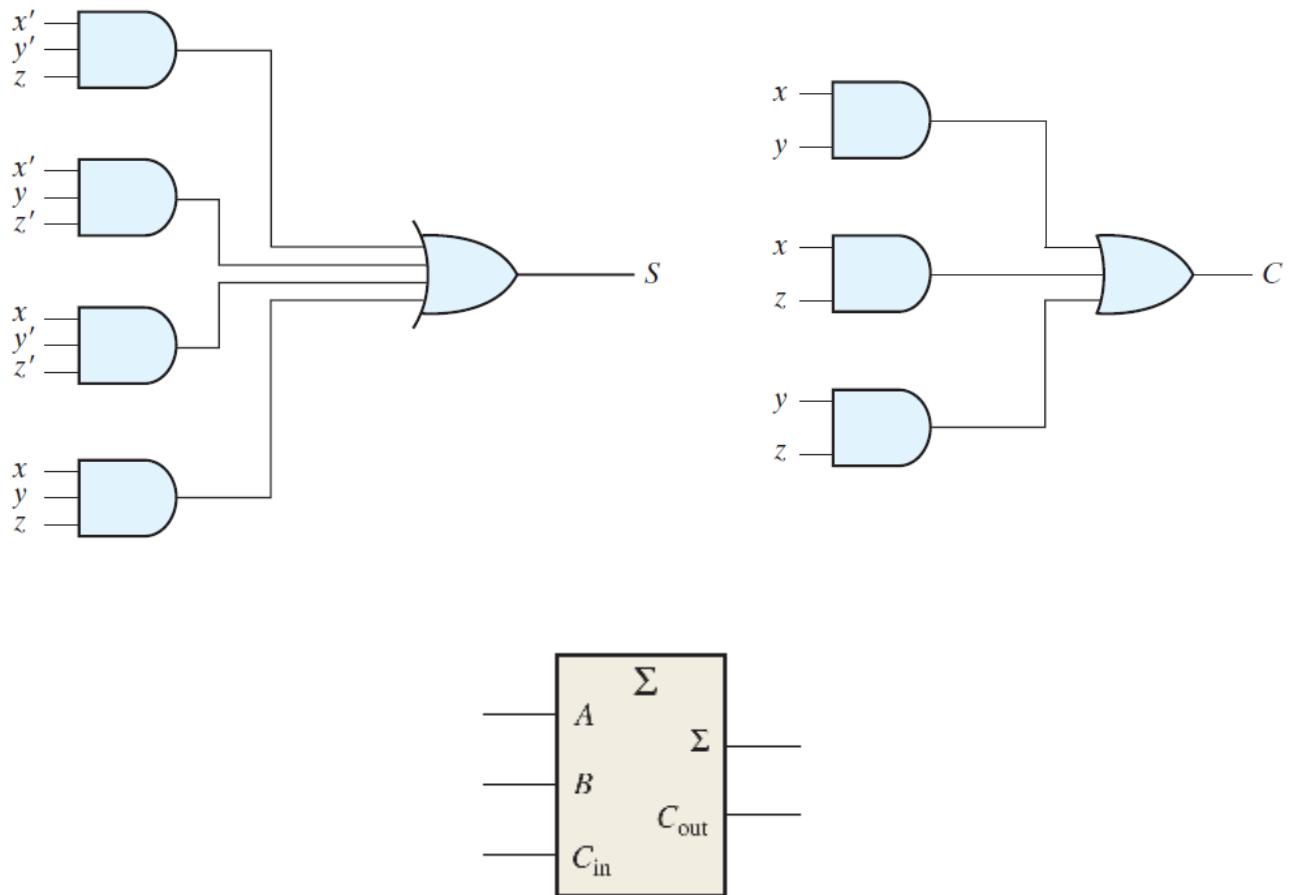


Sum S is also denoted by  $\Sigma$

# Binary adder

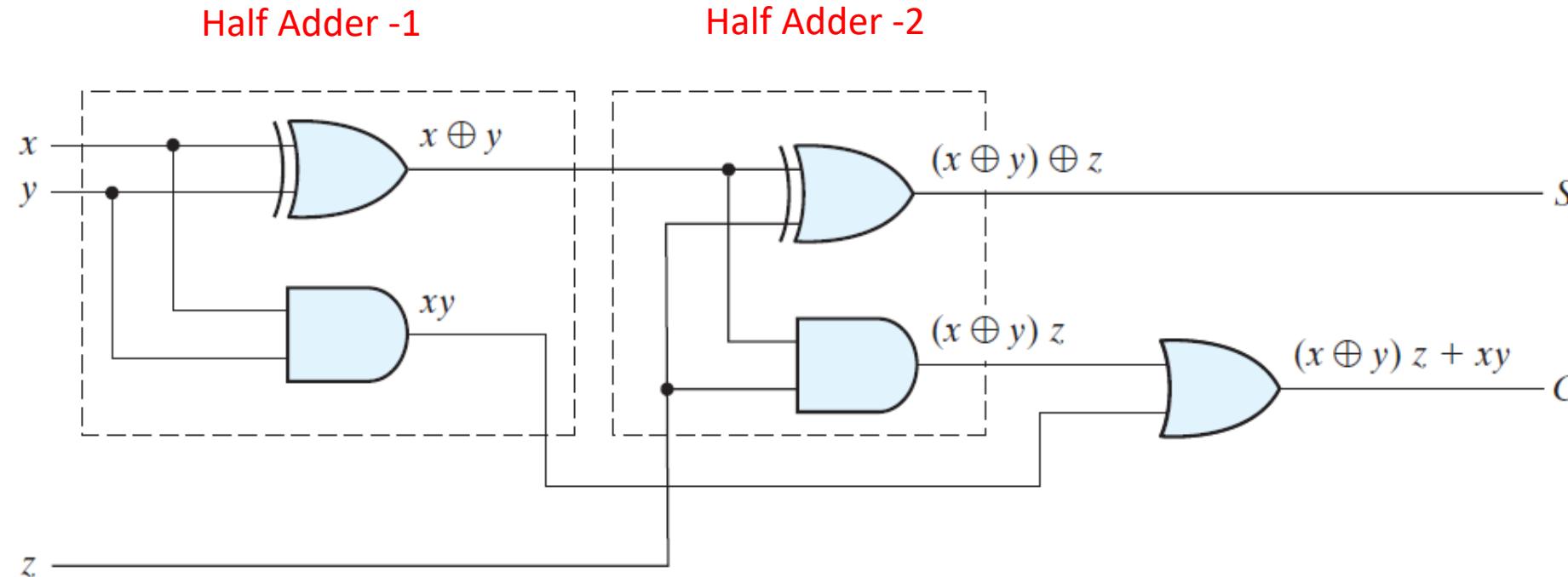
- Circuit that performs the addition of three bits (two significant bits and a previous carry) is a *full adder*

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



# Binary adder

- We can use two half adders to create a full adder



Full Adder

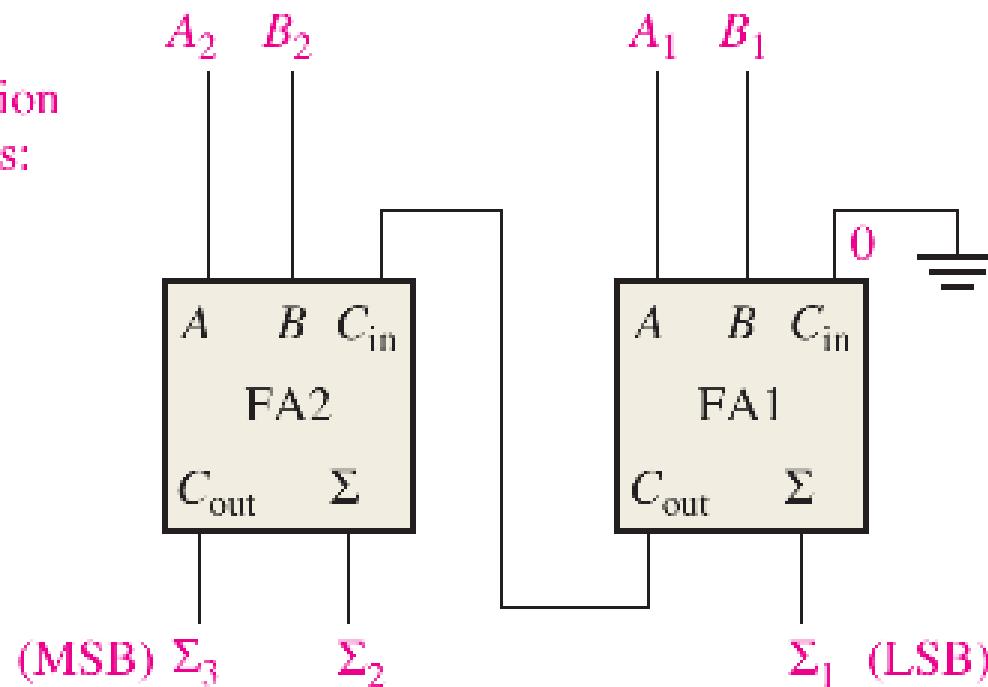
<b>x</b>	<b>y</b>	<b>z</b>	<b>C</b>	<b>S</b>
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

# n-bit binary adder

- Addition of  $n$ -bit numbers requires a chain of  $n$  full adders or a chain of one-half adder and  $n-1$  full adders
- Consider a 2-bit adder:

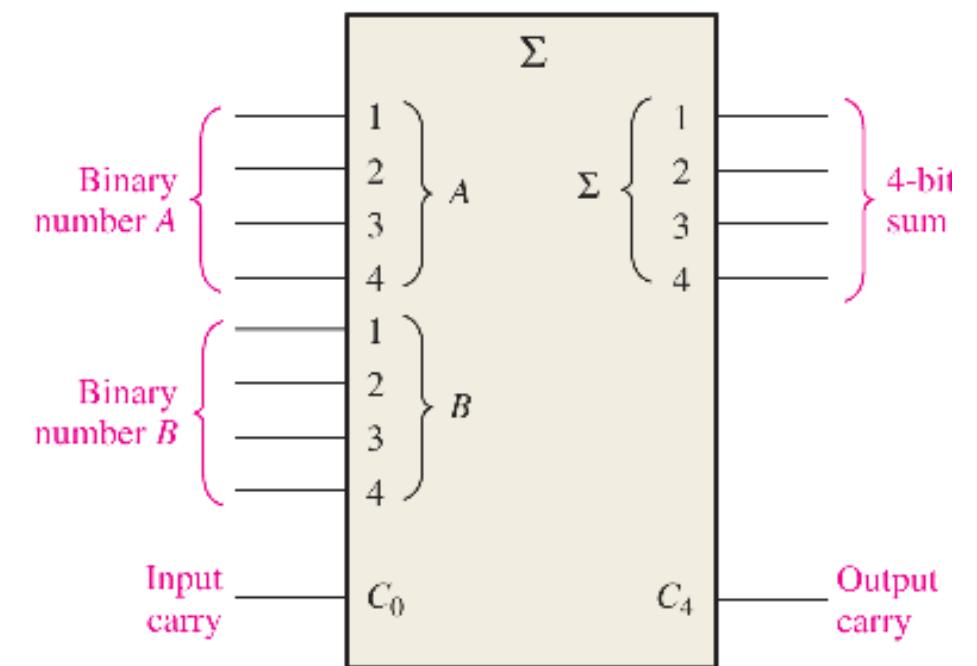
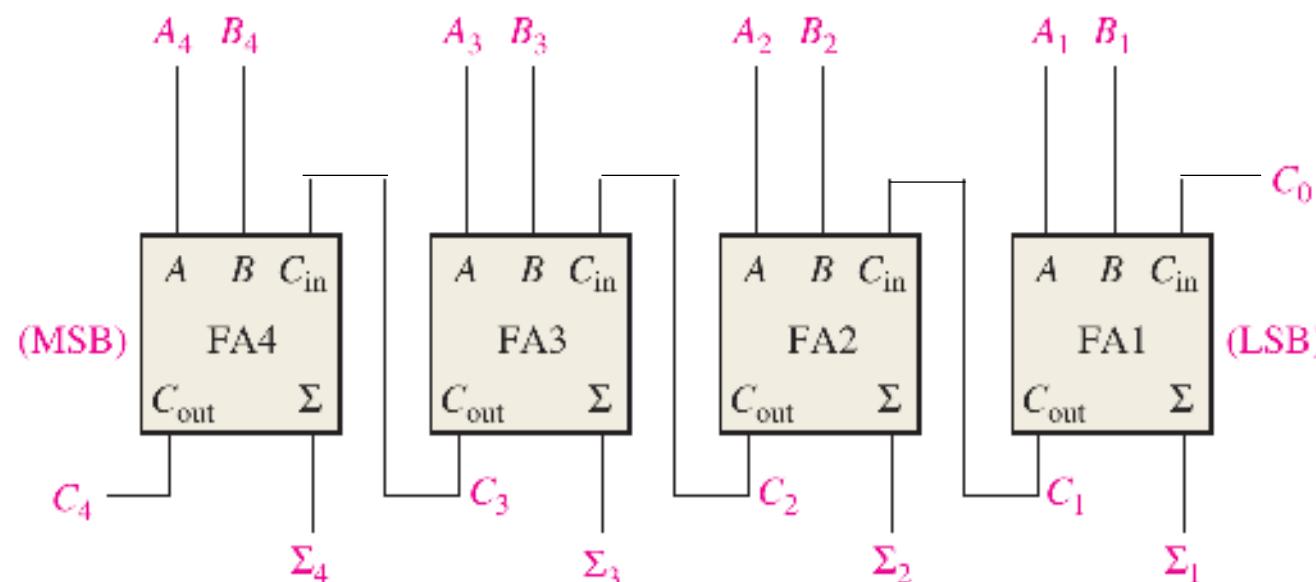
General format, addition  
of two 2-bit numbers:

$$\begin{array}{r} A_2 A_1 \\ + B_2 B_1 \\ \hline \Sigma_3 \Sigma_2 \Sigma_1 \end{array}$$



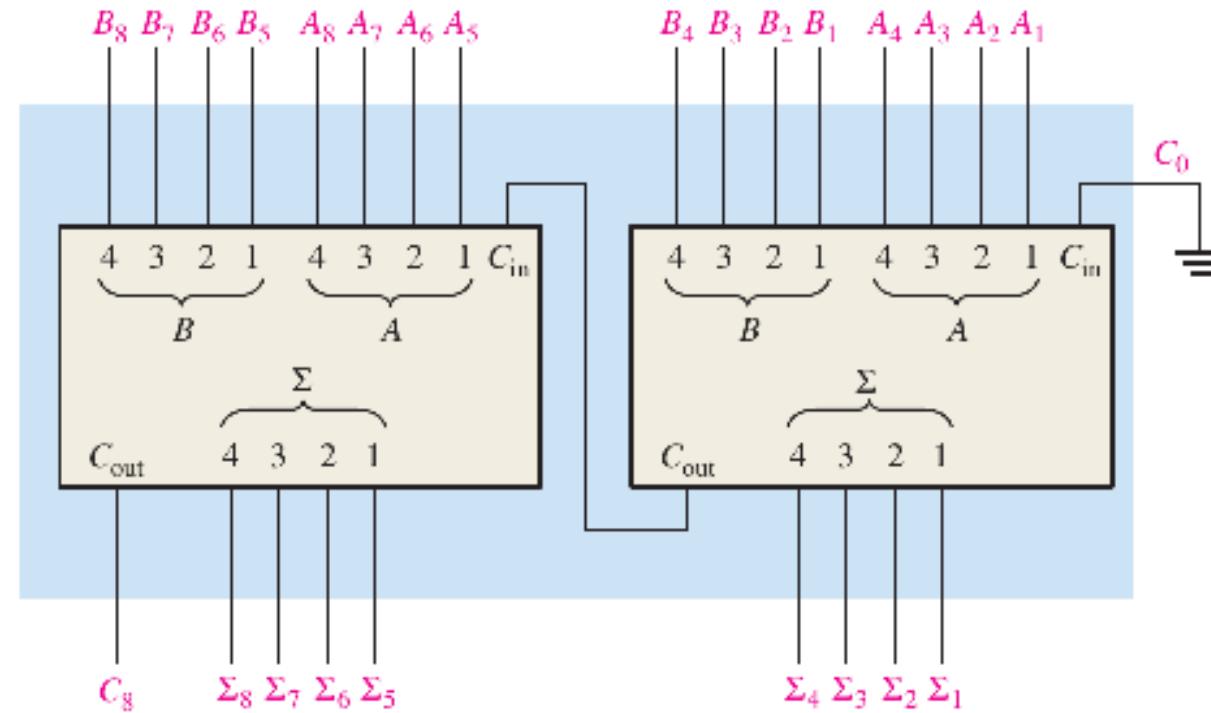
# n-bit binary adder

4-bit adder:



# n-bit binary adder

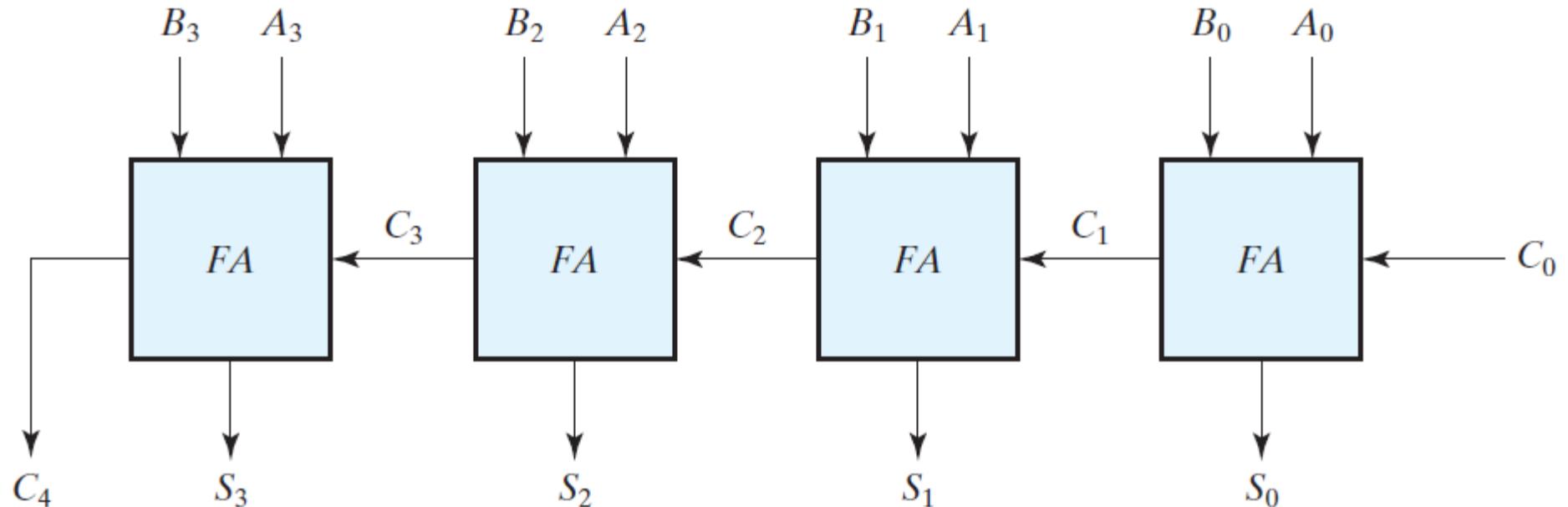
Adder expansion by cascading



Cascading of two 4-bit adders to form an 8-bit adder

# n-bit binary adder

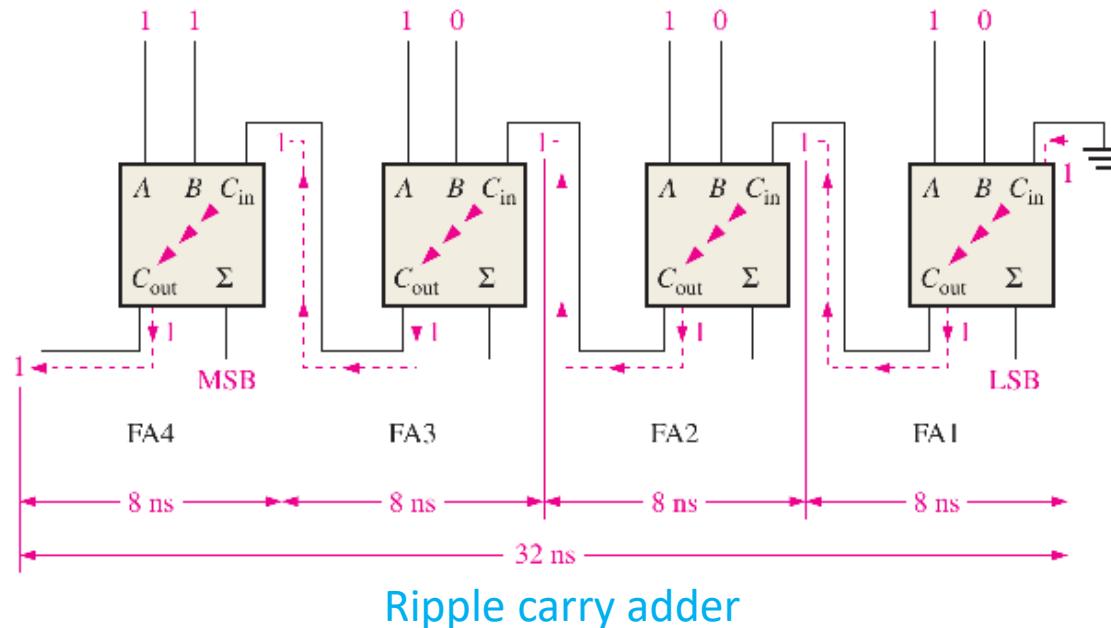
- Consider again the 4-bit adder: Can we make this circuit through the normal route?
- Note that the classical method would require a truth table (and K-map) with  $2^9 = 512$  entries, since there are nine inputs to the circuit
- By using an iterative method of cascading a standard function, it is possible to obtain a simple and straightforward implementation



Ripple carry adder

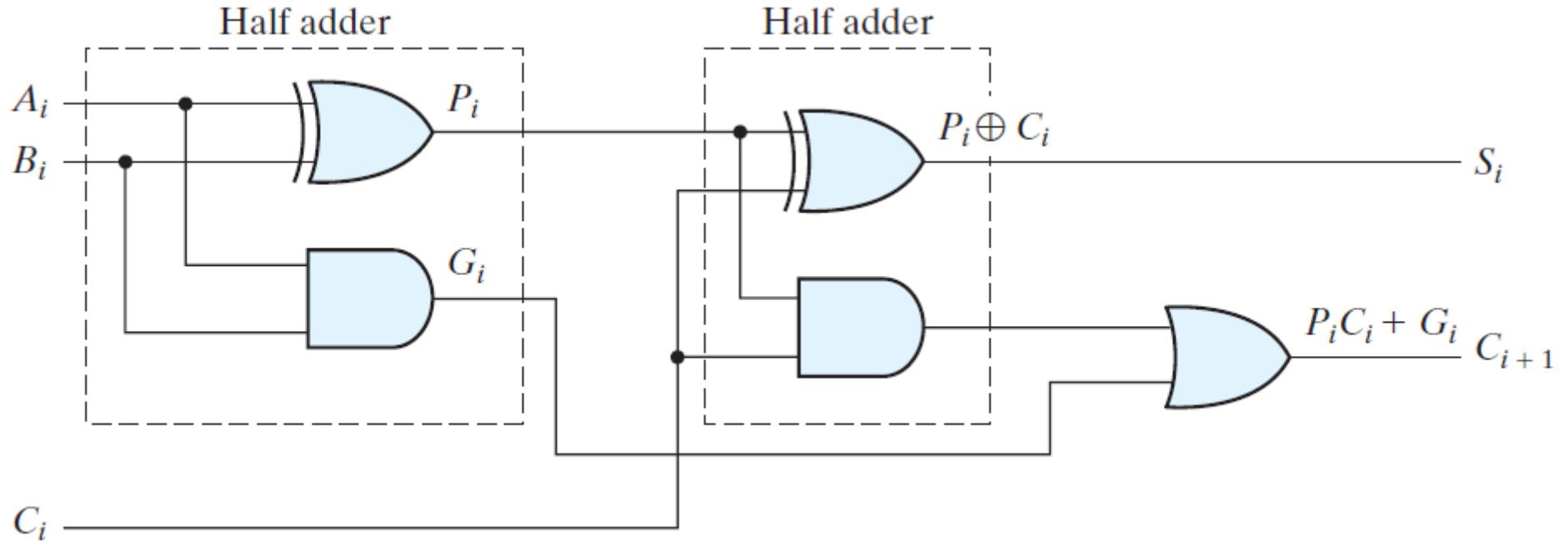
# Carry propagation problem

- The addition of two binary numbers in parallel implies that all the bits of the augend and addend are available for computation at the same time
- As in any combinational circuit, the signal must propagate through the gates before the correct output sum is available in the output terminals
- The total propagation time is equal to the propagation delay of a typical gate, times the number of gate levels in the circuit
- The longest propagation delay time in an adder is the time it takes the carry to propagate through the full adders
- Since each bit of the sum output depends on the value of the input carry, the value of  $S_i$  at any given stage in the adder will be in its steady-state final value only after the input carry to that stage has been propagated



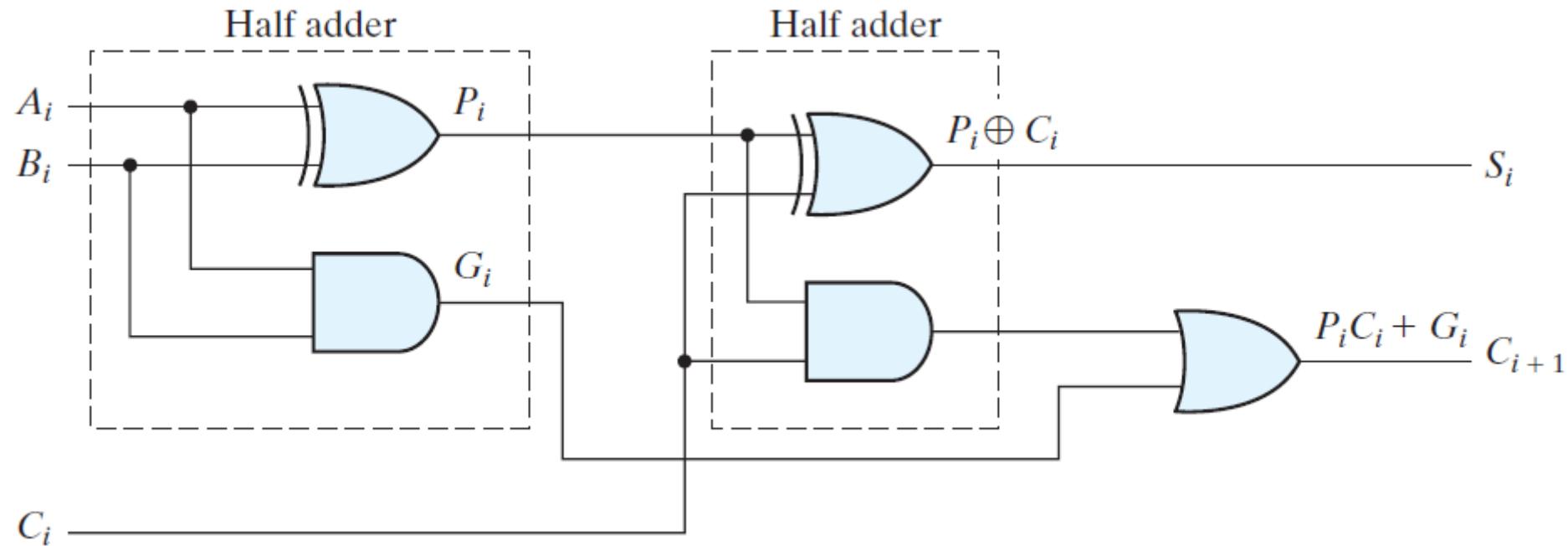
# Carry propagation problem

- The number of gate levels for the carry propagation can be found from the circuit of the full adder
- The signals at  $P_i$  and  $G_i$  settle to their steady-state values after they propagate through their respective gates
- These two signals are common to all half adders and depend on only the input augend and addend bits
- The signal from the input carry  $C_i$  to the output carry  $C_{i+1}$  propagates through an AND gate and an OR gate, which constitute two gate levels
- If there are four full adders in the adder, the output carry  $C_4$  would have  $2 * 4 = 8$  gate levels from  $C_0$  to  $C_4$
- For an  $n$ -bit adder, there are  $2n$  gate levels for the carry to propagate from input to output



# Carry propagation problem

- There are several techniques for reducing the carry propagation time in a parallel adder
- An obvious solution to this problem is to actually make the  $2^n$  truth-table, K-map and get a two level implementation (either SoP or PoS)
- The most widely used technique employs the principle of *carry lookahead logic*

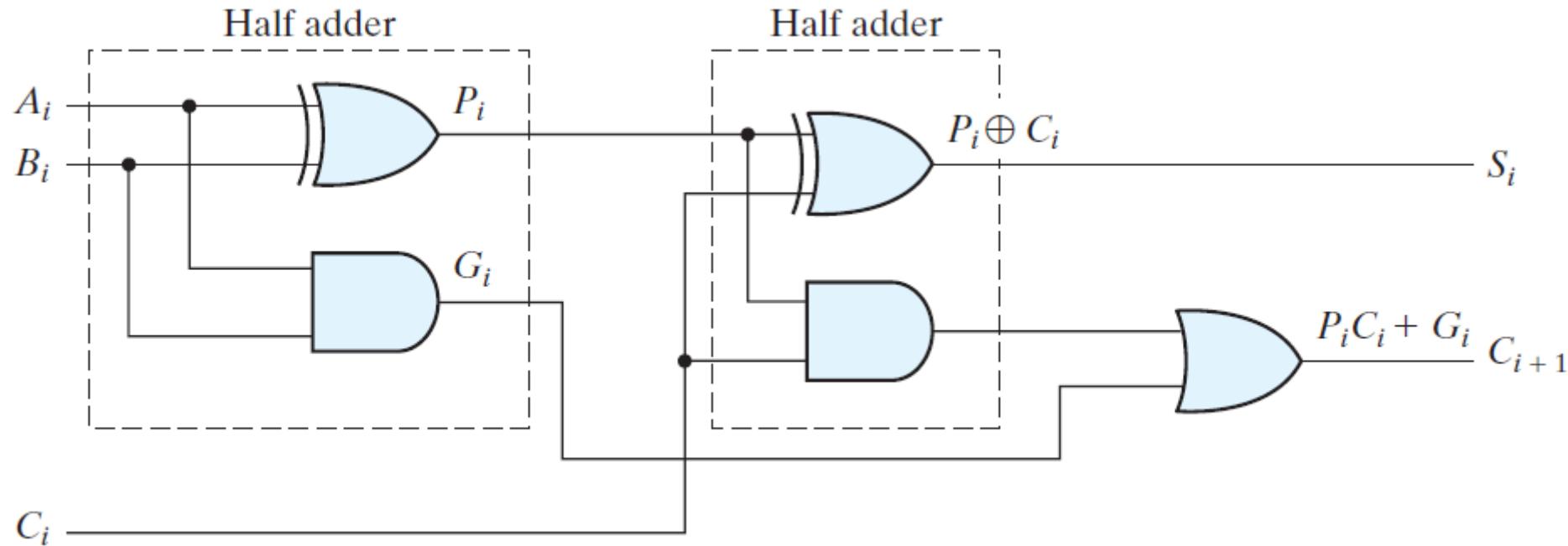


# Carry propagation problem

- With the definition of P and G, we can write:

$$S_i = P_i \oplus C_i \text{ and } C_{i+1} = G_i + P_i C_i$$

- $G_i$  is called a *carry generate*, and it produces a carry of 1 when both  $A_i$  and  $B_i$  are 1, regardless of the input carry  $C_i$
- $P_i$  is called a *carry propagate*, because it determines whether a carry into stage  $i$  will propagate into stage  $i + 1$



# Carry propagation problem

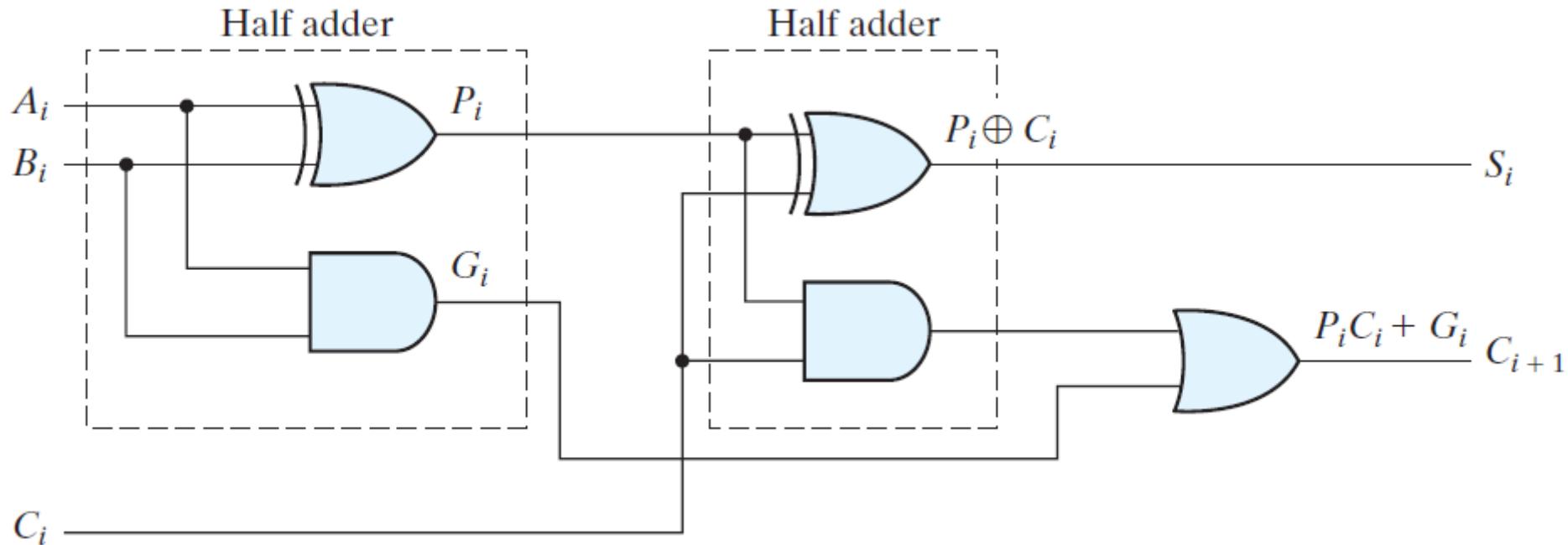
- We now write the Boolean functions for the carry outputs of each stage and substitute the value of each  $C_i$  from the previous equations ( $C_{i+1} = G_i + P_i C_i$ ):

$C_0 = \text{input carry}$

$$C_1 = G_0 + P_0 C_0$$

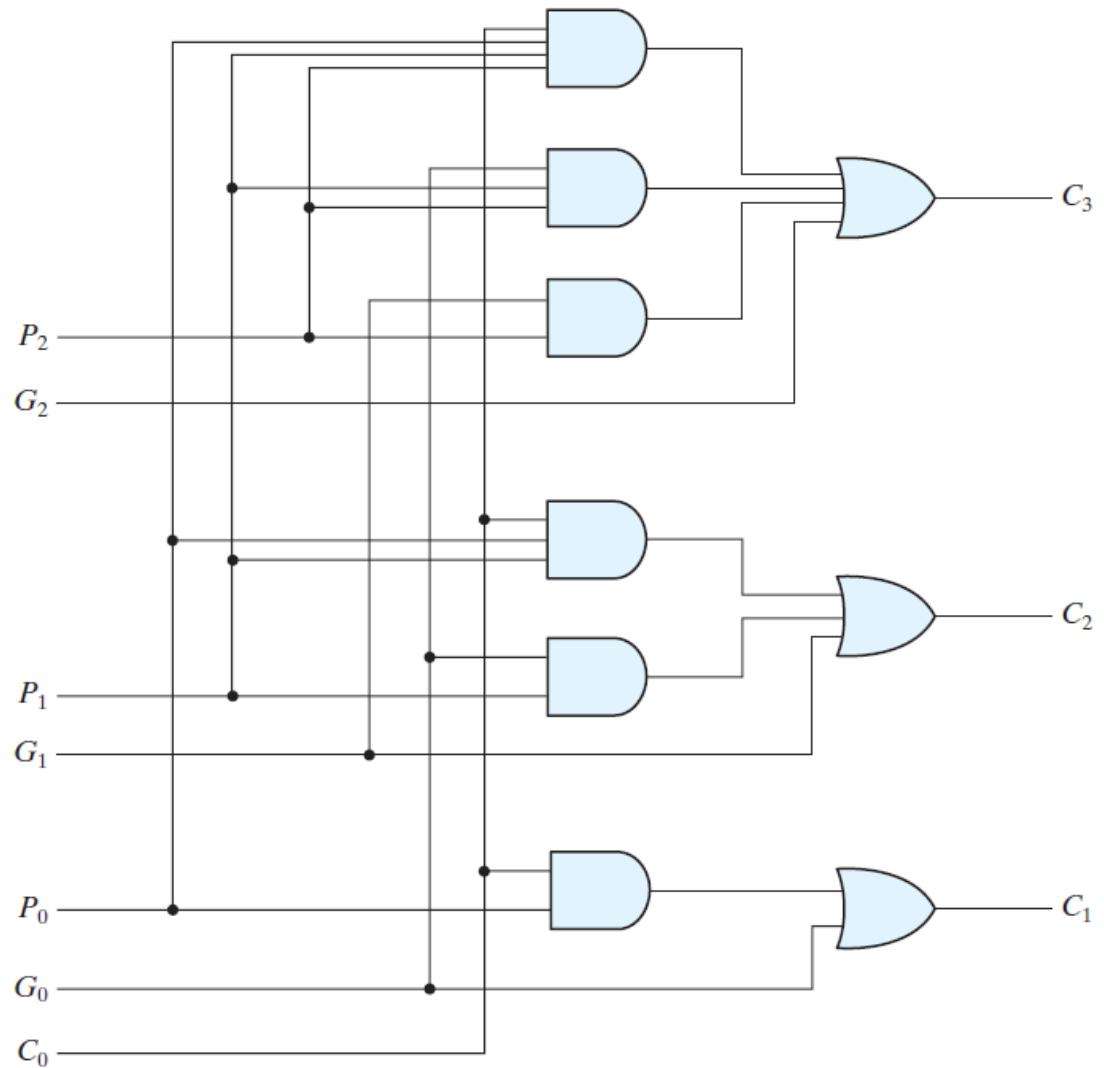
$$C_2 = G_1 + P_1 C_1 = G_1 + P_1 G_0 + P_1 P_0 C_0$$

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



# Carry propagation problem

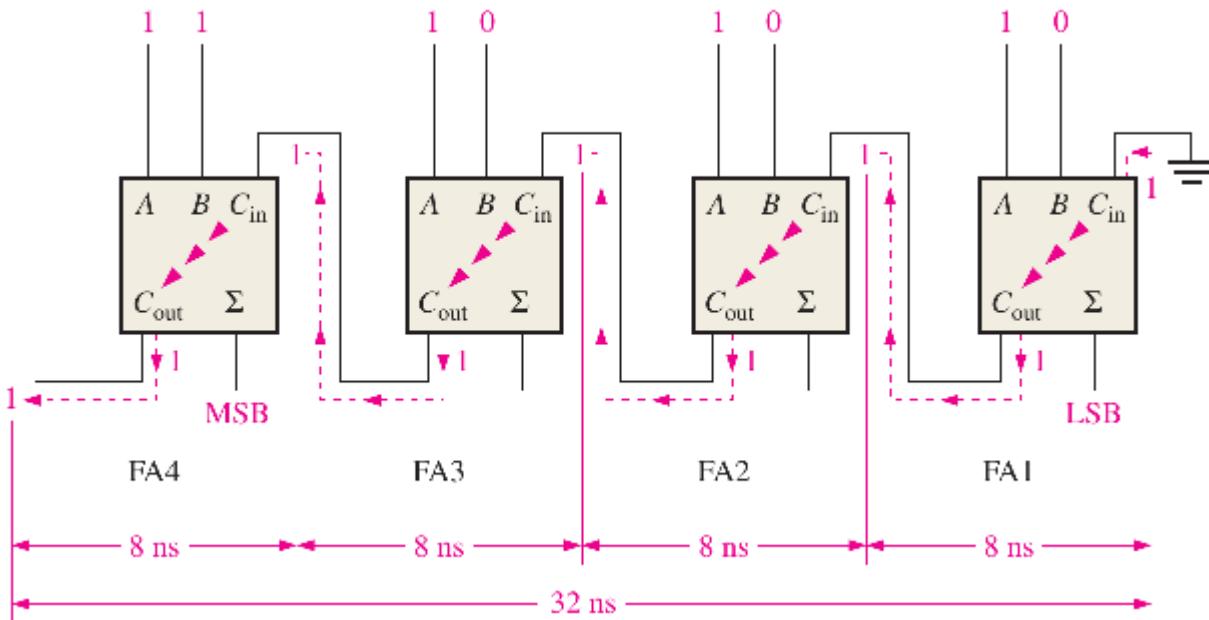
- Since the Boolean function for each output carry is expressed in sum-of-products form only dependent on  $P$  and  $G$ , each function can be implemented with one level of AND gates followed by an OR gate (or by a two-level NAND)
- Note that this circuit can add in less time because  $C_3$  does not have to wait for  $C_2$  and  $C_1$  to propagate; in fact,  $C_3$  is propagated at the same time as  $C_1$  and  $C_2$
- This gain in speed of operation is achieved at the expense of additional complexity (hardware)



# Lecture 14 – Combinational circuits 3

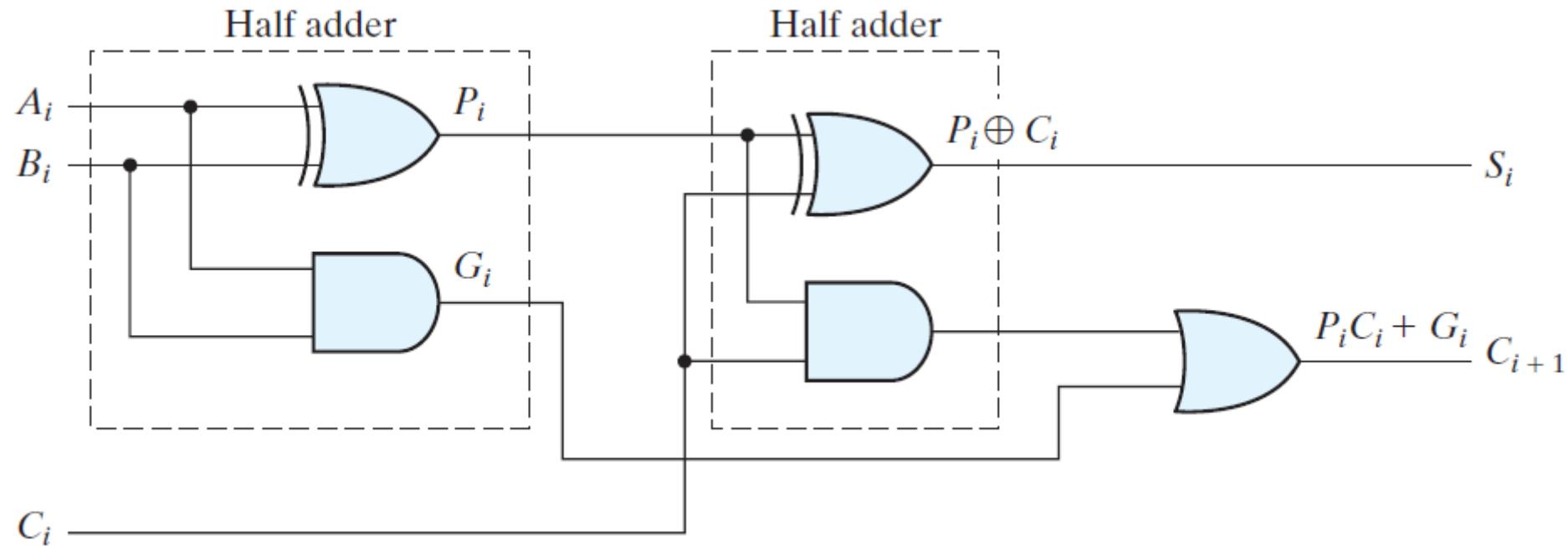
# Carry propagation problem

- The addition of two binary numbers in parallel implies that all the bits of the augend and addend are available for computation at the same time
- As in any combinational circuit, the signal must propagate through the gates before the correct output sum is available in the output terminals
- The total propagation time is equal to the propagation delay of a typical gate, times the number of gate levels in the circuit



# Carry propagation problem

- There are several techniques for reducing the carry propagation time in a parallel adder
- An obvious solution to this problem is to actually make the  $2^n$  truth-table, K-map and get a two level implementation (either SoP or PoS)
- The most widely used technique employs the principle of *carry lookahead logic*

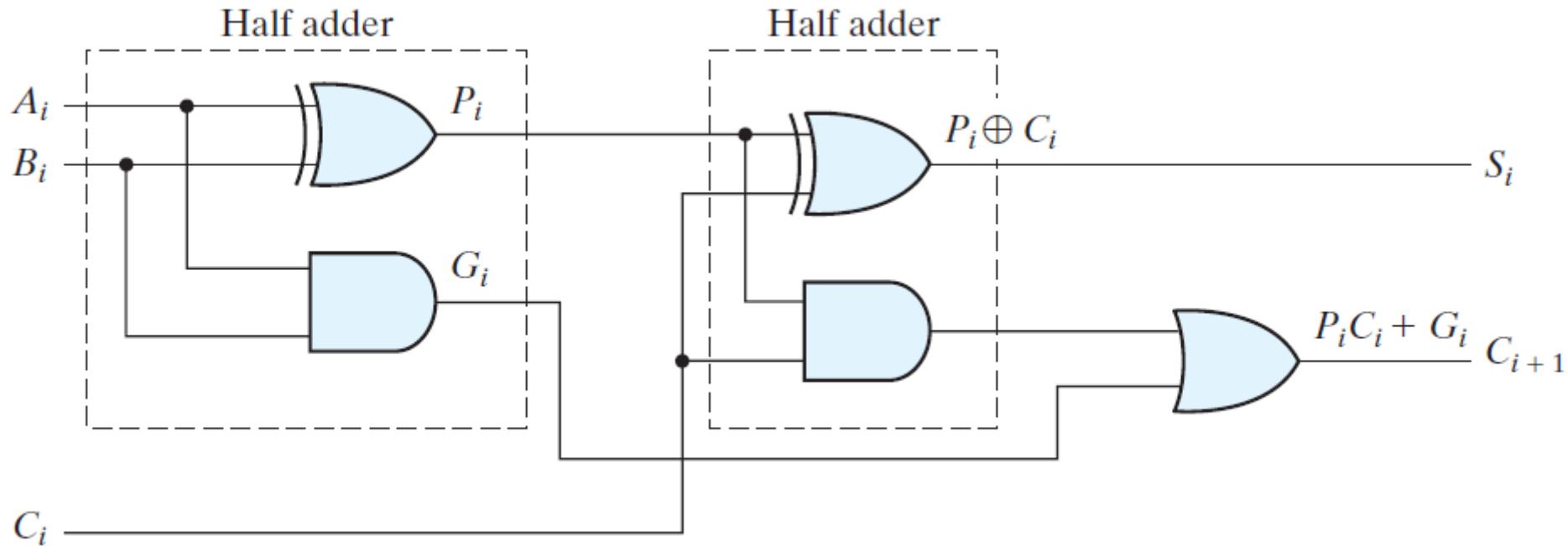


# Carry propagation problem

- With the definition of P and G, we can write:

$$S_i = P_i + C_i \text{ and } C_{i+1} = G_i + P_i C_i$$

- $G_i$  is called a *carry generate*, and it produces a carry of 1 when both  $A_i$  and  $B_i$  are 1, regardless of the input carry  $C_i$
- $P_i$  is called a *carry propagate*, because it determines whether a carry into stage  $i$  will propagate into stage  $i + 1$



# Carry propagation problem

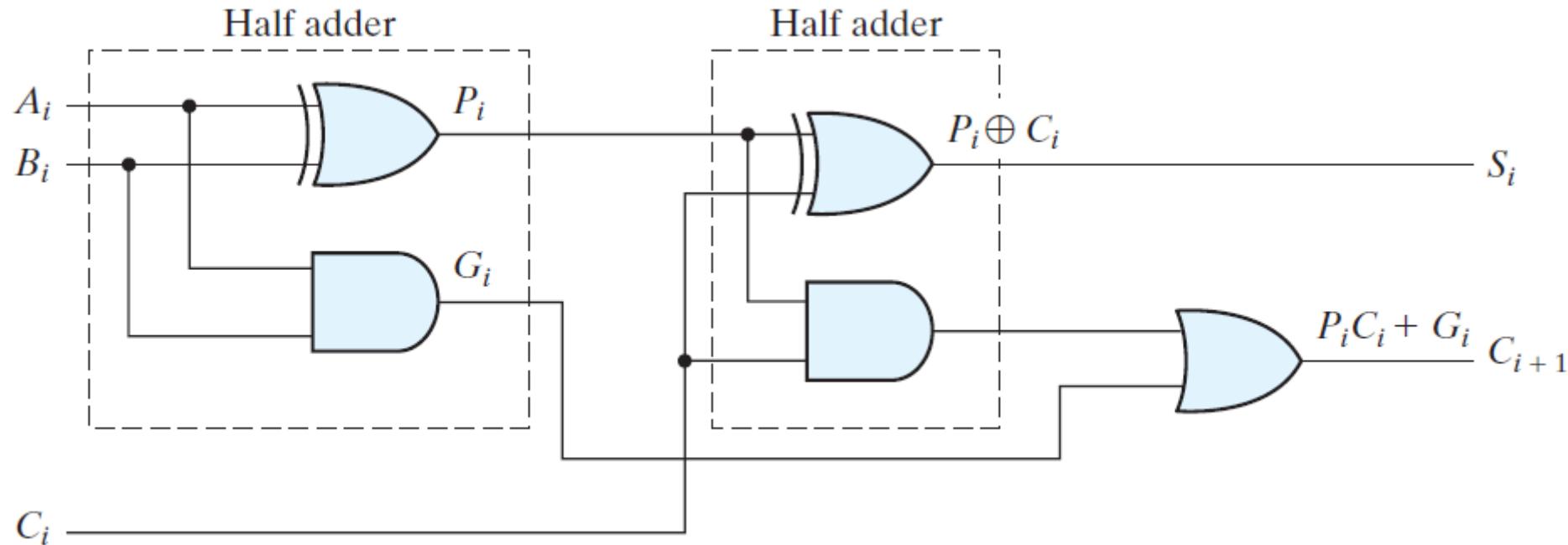
- We now write the Boolean functions for the carry outputs of each stage and substitute the value of each  $C_i$  from the previous equations:

$$C_0 = \text{input carry}$$

$$C_1 = G_0 + P_0 C_0$$

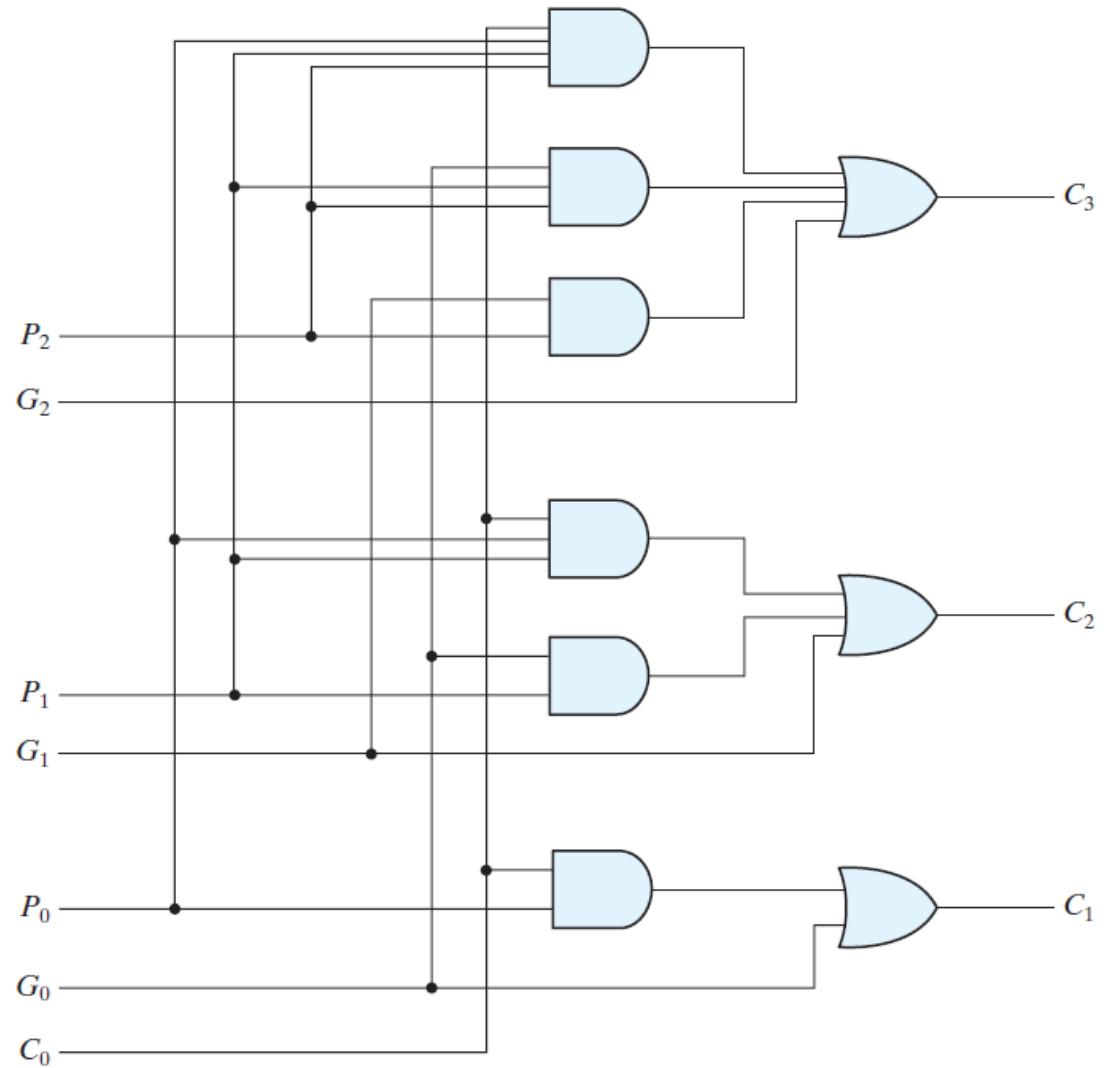
$$C_2 = G_1 + P_1 C_1 = G_1 + P_1 G_0 + P_1 P_0 C_0$$

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



# Carry propagation problem

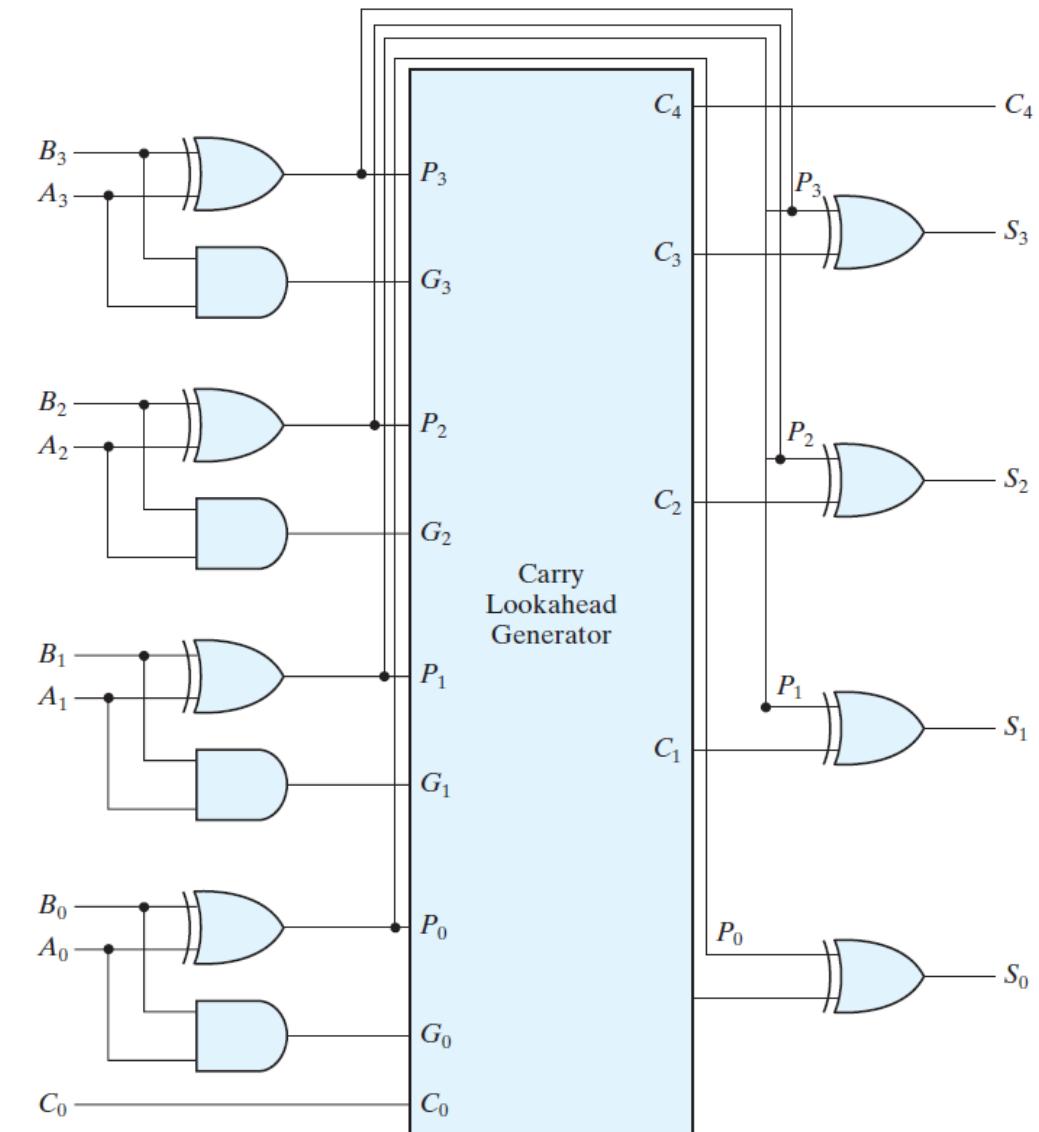
- Each output carry is expressed in sum-of-products form only dependent on P and G, each function can be implemented with one level of AND gates followed by an OR gate
- This circuit can add in less time because  $C_3$  does not have to wait for  $C_2$  and  $C_1$  to propagate; in fact,  $C_3$  is propagated at the same time as  $C_1$  and  $C_2$
- This gain in speed of operation is achieved at the expense of additional complexity (hardware)



Carry lookahead generator

# Carry propagation problem

- Each sum output requires two XOR gates
- The output of the first XOR gate generates the  $P_i$  variable, and the AND gate generates the  $G_i$  variable
- The carries are propagated through the carry lookahead generator and applied as inputs to the second XOR gate
- *All output carries are generated after a delay through only two levels of gates*
- Thus, outputs  $S_1$  through  $S_3$  have equal propagation delay times





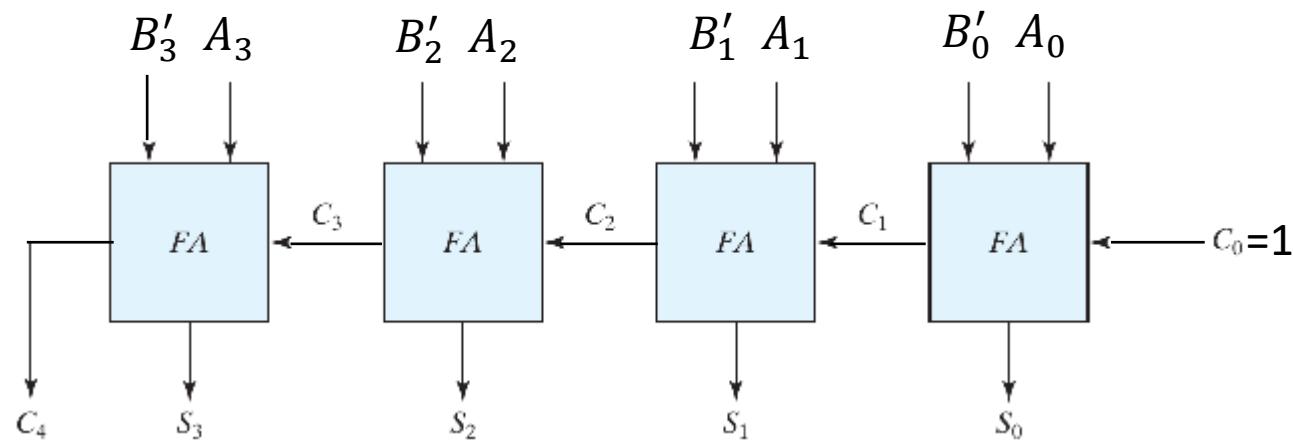
# The Binary subtractor

# Revisit : Subtraction with Radix complements

- Subtraction using method of borrowing is less efficient when implemented with digital hardware
- Consider subtraction  $M - N$  in base  $r$
- Here is the algorithm using Radix complement:
  1. Take radix complement of subtrahend  $N$ :  $r^n - N$
  2. Add this to  $M$ :  $(r^n - N) + M = r^n + (M - N) = r^n - (N - M)$
  3. If you get a carry in the  $(n+1)^{\text{th}}$  digit, then the result is positive, discard the carry and you are done
  4. If you **do not** get a carry in the  $(n+1)^{\text{th}}$  digit, then the result is **negative**. Take the radix complement of the number to get the answer, then put a negative sign

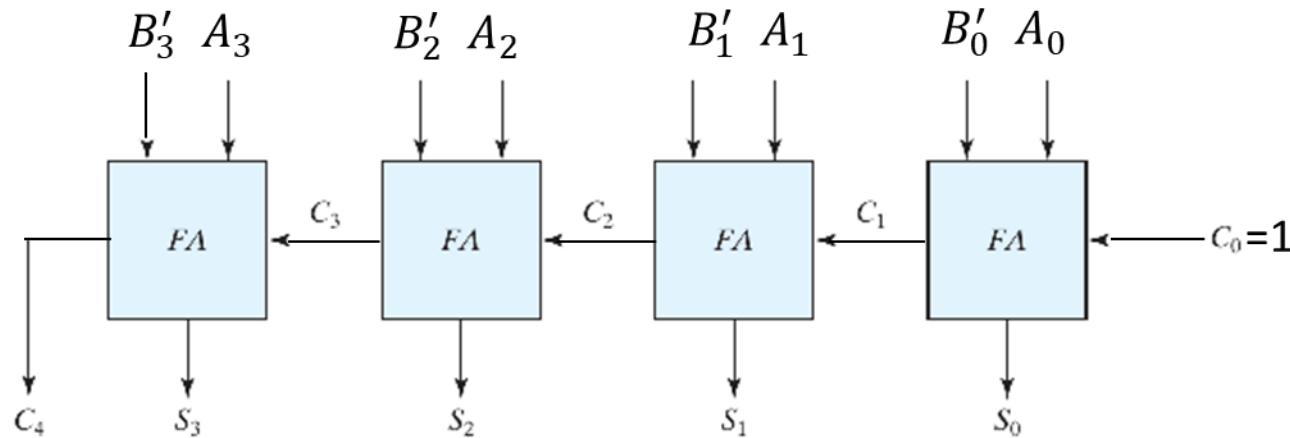
# Binary subtractor

- The subtraction of binary numbers can be done most conveniently by means of complements
- Remember that the subtraction  $A - B$  can be done by taking the 2's complement of  $B$  and **adding it to A**
- The 2's complement can be obtained by taking the 1's complement and adding 1 to the least significant pair of bits
- The 1's complement can be implemented with inverters, and a 1 can be added to the sum through the input carry



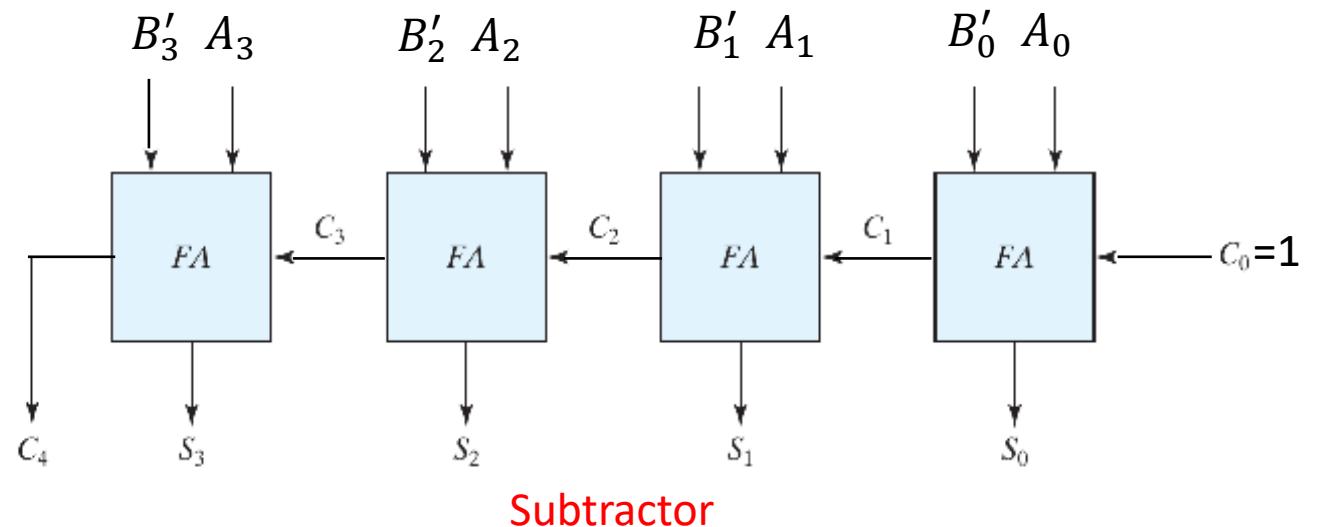
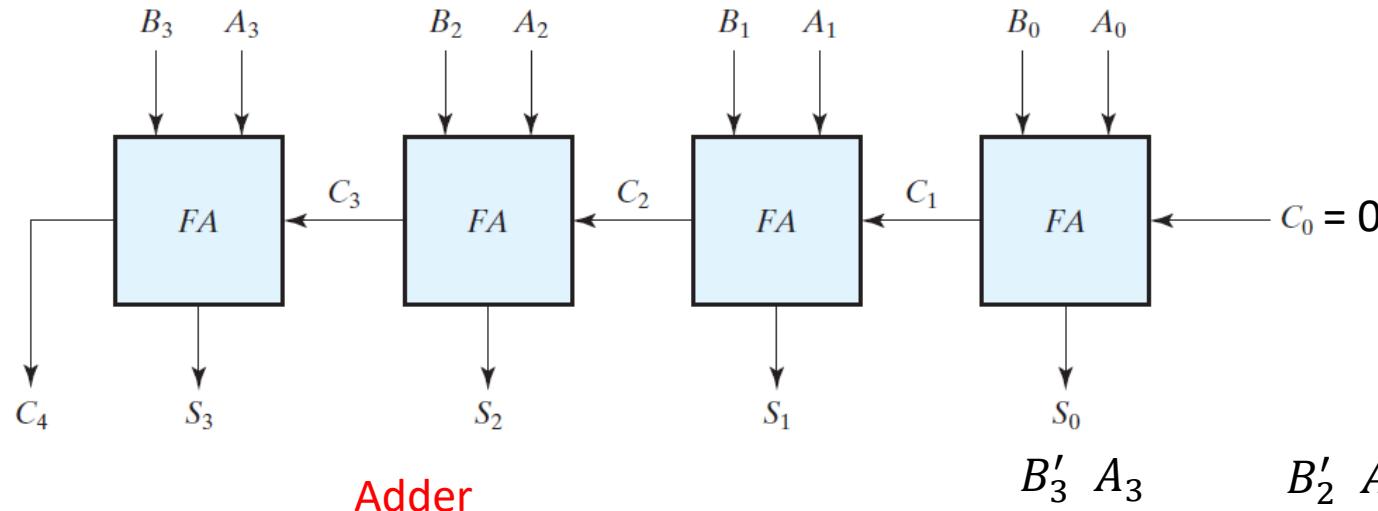
# Binary subtractor

- The circuit for subtracting  $A - B$  consists of an adder with inverters placed between each data input  $B$  and the corresponding input of the full adder
- **The input carry  $C_0$  must be equal to 1 when subtraction is performed**
- The operation thus performed becomes:  $A + (1\text{'s complement of } B) + 1$ .
  - This is equal to  $A + 2\text{'s complement of } B$
- That gives  $A - B$  if  $A \geq B$  or the 2's complement of  $B - A$  if  $A < B$



# Can we combine the binary adder & subtractor

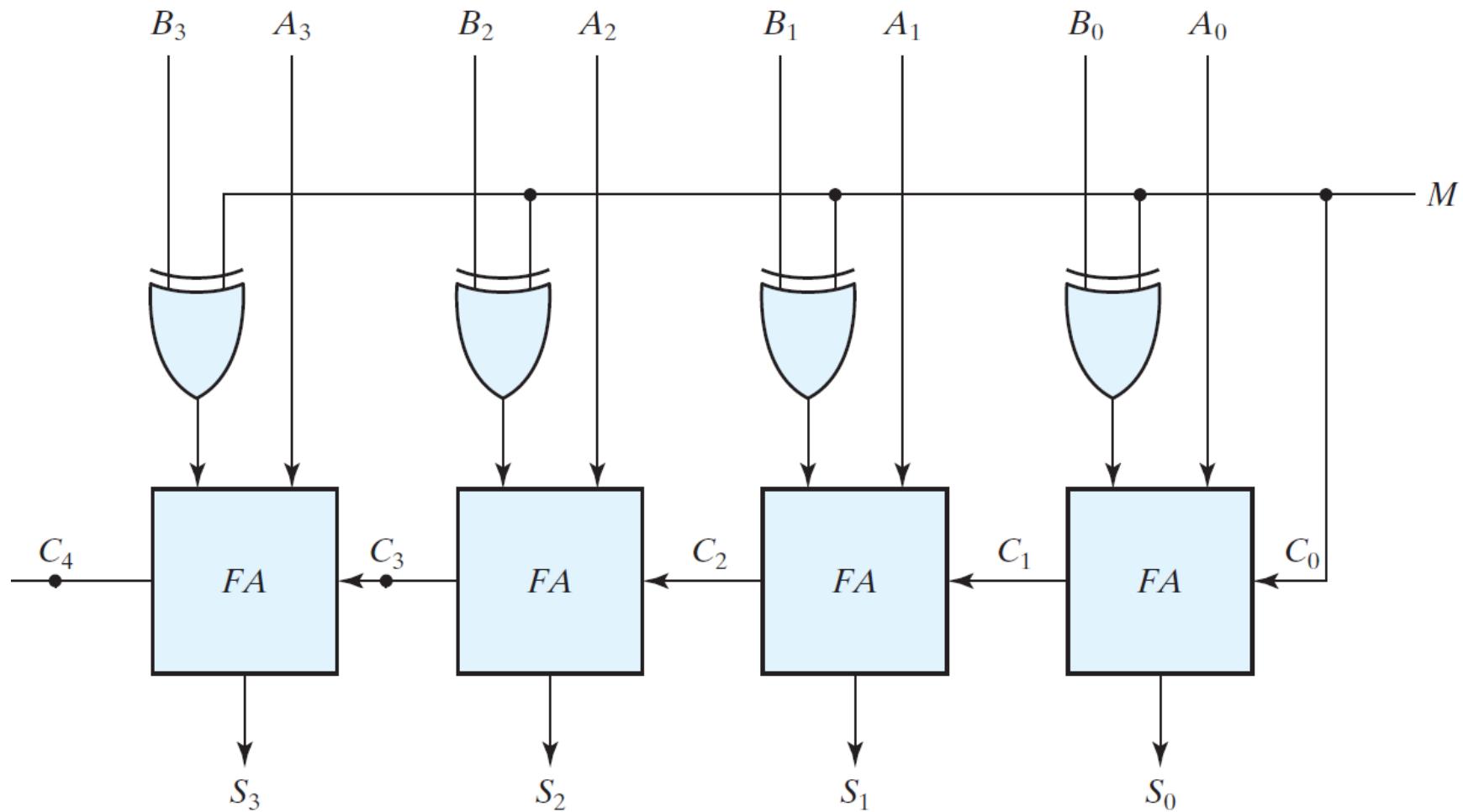
- Both use the 4-bit full adder. In one case, we use B and in another we use inverted B



# Binary adder-subtractor

- Here is some magic: The addition and subtraction operations can be combined into one circuit
- The mode input  $M$  controls the operation
- When  $M = 0$ , the circuit is an adder, and when  $M = 1$ , the circuit becomes a subtractor
- When  $M = 0$ , the full adders receive the value of  $B$ , the input carry is 0, and the circuit performs  $A + B$
- When  $M = 1$ , the full adders receive  $B'$  and  $C_0 = 1$
- Thus, the  $B$  inputs are all complemented and a 1 is added through the input carry
- The circuit performs the operation  $A$  plus the 2's complement of  $B$

# Binary adder-subtractor





# The BCD adder

# BCD adder

- Consider the arithmetic addition of two decimal digits in BCD, together with an input carry from a previous stage
- Since each input digit does not exceed 9, the output sum cannot be greater than  $9 + 9 + 1 = 19$ , the 1 in the sum being an input carry
- Suppose we apply two BCD digits to a four-bit binary adder
- The adder will form the sum in *binary* and produce a result that ranges from 0 through 19

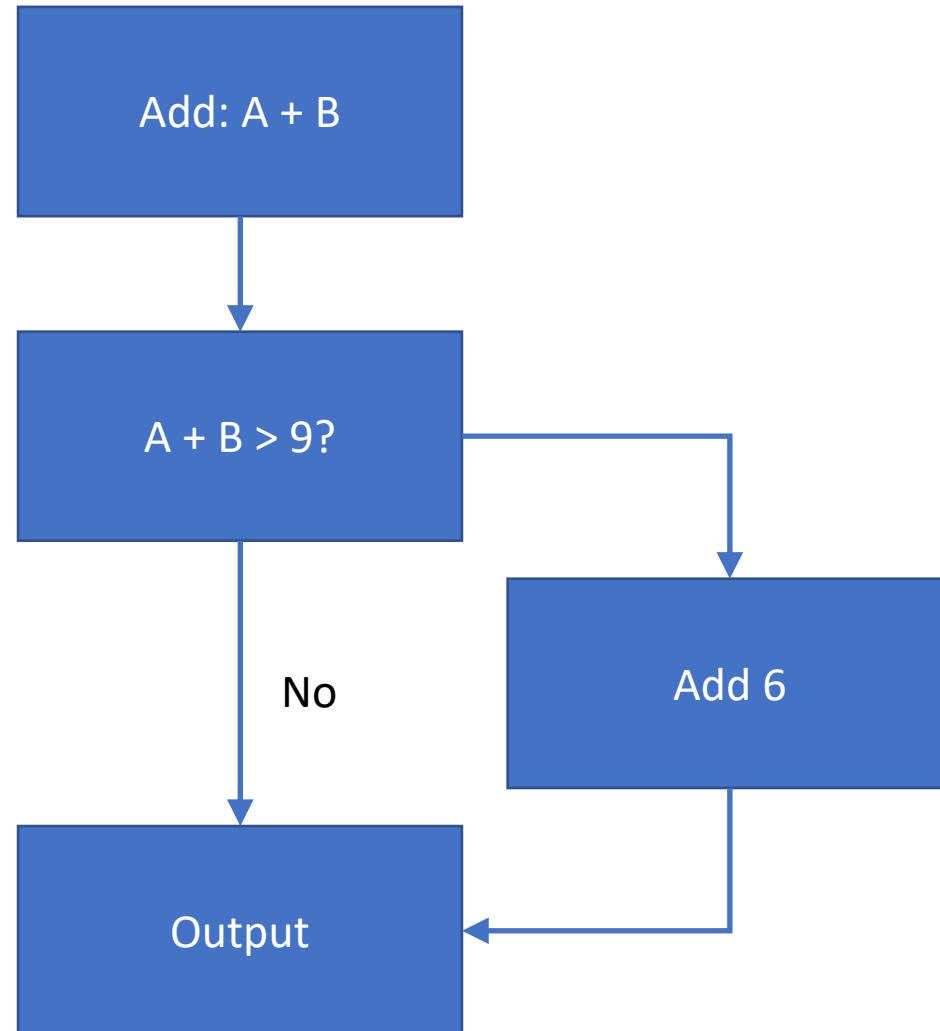
K	Binary Sum				c	BCD Sum				Decimal
	$z_8$	$z_4$	$z_2$	$z_1$		$s_8$	$s_4$	$s_2$	$s_1$	
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	2
0	0	0	1	1	0	0	0	1	1	3
0	0	1	0	0	0	0	1	0	0	4
0	0	1	0	1	0	0	1	0	1	5
0	0	1	1	0	0	0	1	1	0	6
0	0	1	1	1	0	0	1	1	1	7
0	1	0	0	0	0	1	0	0	0	8
0	1	0	0	1	0	1	0	0	1	9
0	1	0	1	0	1	0	0	0	0	10
0	1	0	1	1	1	0	0	0	1	11
0	1	1	0	0	1	0	0	1	0	12
0	1	1	0	1	1	0	0	1	1	13
0	1	1	1	0	1	0	1	0	0	14
0	1	1	1	1	1	0	1	0	1	15
1	0	0	0	0	1	0	1	1	0	16
1	0	0	0	1	1	0	1	1	1	17
1	0	0	1	0	1	1	0	0	0	18
1	0	0	1	1	1	1	0	0	1	19

# BCD adder

- When the binary sum is equal to or less than 1001, the corresponding BCD number is identical, and therefore no conversion is needed
- When the binary sum is greater than 1001, we obtain an invalid BCD representation
- *The addition of binary 6 (0110) to the binary sum converts it to the correct BCD representation and also produces an output carry as required*

K	Binary Sum				c	BCD Sum				Decimal
	$z_8$	$z_4$	$z_2$	$z_1$		$s_8$	$s_4$	$s_2$	$s_1$	
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	2
0	0	0	1	1	0	0	0	1	1	3
0	0	1	0	0	0	0	1	0	0	4
0	0	1	0	1	0	0	1	0	1	5
0	0	1	1	0	0	0	1	1	0	6
0	0	1	1	1	0	0	1	1	1	7
0	1	0	0	0	0	1	0	0	0	8
0	1	0	0	1	0	1	0	0	1	9
0	1	0	1	0	1	0	0	0	0	10
0	1	0	1	1	1	0	0	0	1	11
0	1	1	0	0	1	0	0	1	0	12
0	1	1	0	1	1	0	0	1	1	13
0	1	1	1	0	1	0	1	0	0	14
0	1	1	1	1	1	0	1	0	1	15
1	0	0	0	0	1	0	1	1	0	16
1	0	0	0	1	1	0	1	1	1	17
1	0	0	1	0	1	1	0	0	0	18
1	0	0	1	1	1	1	0	0	1	19

# BCD adder - algorithm



K	Binary Sum				BCD Sum				Decimal
	$Z_8$	$Z_4$	$Z_2$	$Z_1$	$C$	$S_8$	$S_4$	$S_2$	
0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	1
0	0	0	1	0	0	0	0	1	0
0	0	0	1	1	0	0	0	1	1
0	0	1	0	0	0	0	1	0	0
0	0	1	0	1	0	0	1	0	1
0	0	1	1	0	0	0	1	1	0
0	0	1	1	1	0	0	1	1	1
0	1	0	0	0	0	1	0	0	0
0	1	0	0	1	0	1	0	0	1
0	1	0	1	0	1	0	0	1	0
0	1	1	0	0	1	0	0	1	1
0	1	1	0	1	1	0	0	1	1
0	1	1	1	0	1	0	1	0	0
0	1	1	1	1	1	0	1	0	1
1	0	0	0	0	1	0	1	1	0
1	0	0	0	1	1	0	1	0	1
1	0	0	1	0	1	1	0	0	0
1	0	0	1	1	1	1	0	0	1
1	0	0	1	1	1	1	0	0	19

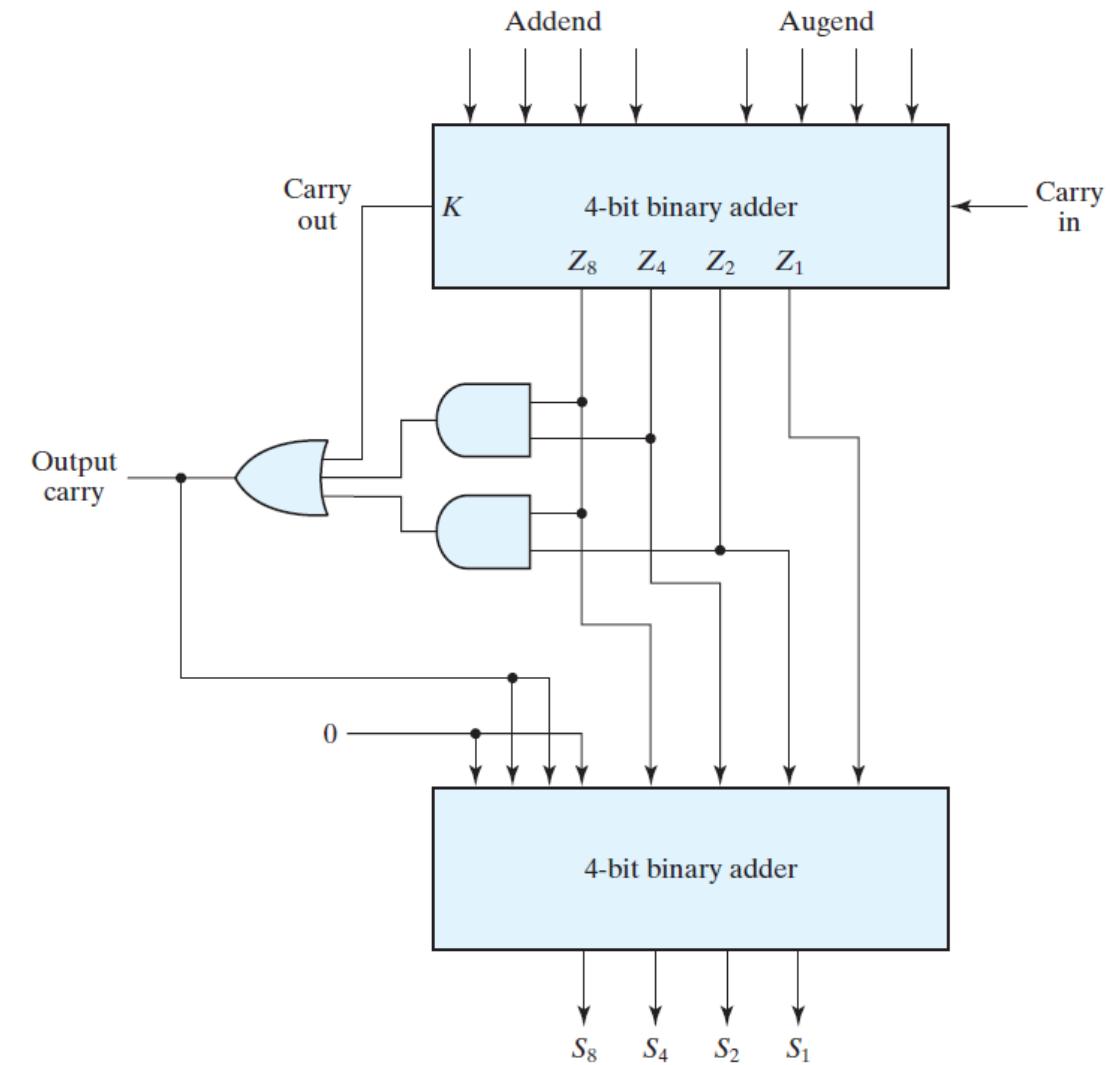
# BCD adder

- It is obvious that a correction is needed when the binary sum has an output carry  $K = 1$
- The other six combinations from 1010 through 1111 that need a correction have a 1 in position  $Z_8$
- To distinguish them from binary 1000 and 1001, which also have a 1 in position  $Z_8$ , we specify further that either  $Z_4$  or  $Z_2$  must have a 1
- Thus, the condition for a correction and an output carry can be expressed by the Boolean function:  $Cor = K + Z_8Z_4 + Z_8Z_2$
- When  $Cor = 1$ , it is necessary to add 0110 to the binary sum and provide an output carry for the next stage

K	Binary Sum					C	BCD Sum				Decimal
	$Z_8$	$Z_4$	$Z_2$	$Z_1$			$S_8$	$S_4$	$S_2$	$S_1$	
0	0	0	0	0		0	0	0	0	0	0
0	0	0	0	1		0	0	0	0	1	1
0	0	0	1	0		0	0	0	1	0	2
0	0	0	1	1		0	0	0	1	1	3
0	0	1	0	0		0	0	1	0	0	4
0	0	1	0	1		0	0	1	0	1	5
0	0	1	1	0		0	0	1	1	0	6
0	0	1	1	1		0	0	1	1	1	7
0	1	0	0	0		0	1	0	0	0	8
0	1	0	0	1		0	1	0	0	1	9
0	1	0	1	0		1	0	0	0	0	10
0	1	0	1	1		1	0	0	0	1	11
0	1	1	0	0		1	0	0	1	0	12
0	1	1	0	1		1	0	0	1	1	13
0	1	1	1	0		1	0	1	0	0	14
0	1	1	1	1		1	0	1	0	1	15
1	0	0	0	0		1	0	1	1	0	16
1	0	0	0	1		1	0	1	1	1	17
1	0	0	1	0		1	1	0	0	0	18
1	0	0	1	1		1	1	0	0	1	19

# BCD adder

- The logic circuit that detects the necessary correction can be derived from the entries in the table
- It is obvious that a correction is needed when the binary sum has an output carry  $K = 1$
- The other six combinations from 1010 through 1111 that need a correction have a 1 in position  $Z_8$
- To distinguish them from binary 1000 and 1001, which also have a 1 in position  $Z_8$ , we specify further that either  $Z_4$  or  $Z_2$  must have a 1
- Thus, the condition for a correction and an output carry can be expressed by the Boolean function:  $Cor = K + Z_8Z_4 + Z_8Z_2$
- When  $Cor = 1$ , it is necessary to add 0110 to the binary sum and provide an output carry for the next stage





# The Binary multiplier

# Binary multiplier

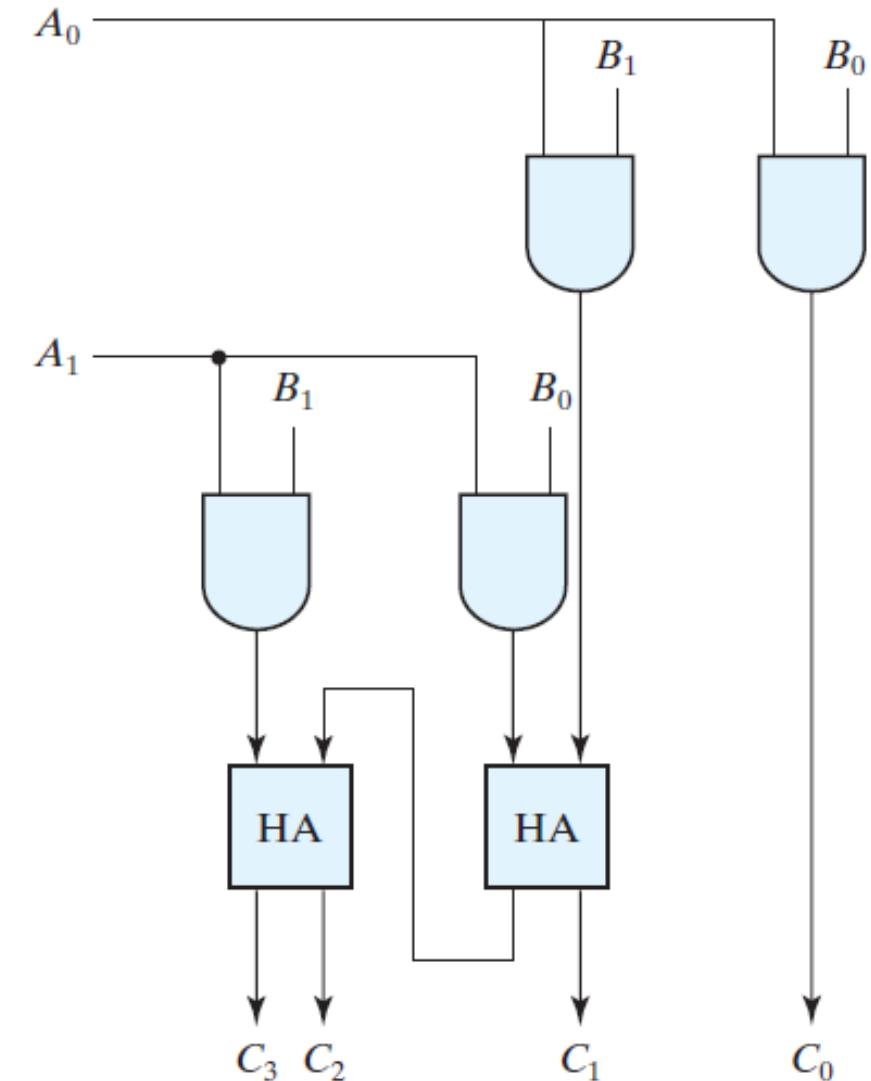
- Multiplication of binary numbers is performed in the same way as multiplication of decimal numbers
- The multiplicand is multiplied by each bit of the multiplier, starting from the least significant bit
- Each such multiplication forms a partial product
- Successive partial products are shifted one position to the left
- The final product is obtained from the sum of the partial products
- Consider a 2-bit  $\times$  2-bit multiplier.  
Number of inputs/outputs?

$$\begin{array}{r} B_1 \quad B_0 \\ A_1 \quad A_0 \\ \hline A_0B_1 \quad A_0B_0 \\ \hline A_1B_1 \quad A_1B_0 \\ \hline C_3 \quad C_2 \quad C_1 \quad C_0 \end{array}$$

# Binary multiplier

- The multiplicand bits are  $B_1$  and  $B_0$ , the multiplier bits are  $A_1$  and  $A_0$ , and the product is  $C_3C_2C_1C_0$

$$\begin{array}{r} B_1 \quad B_0 \\ A_1 \quad A_0 \\ \hline A_0B_1 \quad A_0B_0 \\ \hline A_1B_1 \quad A_1B_0 \\ \hline C_3 \quad C_2 \quad C_1 \quad C_0 \end{array}$$



# Lecture 15 – Combinational circuits 4

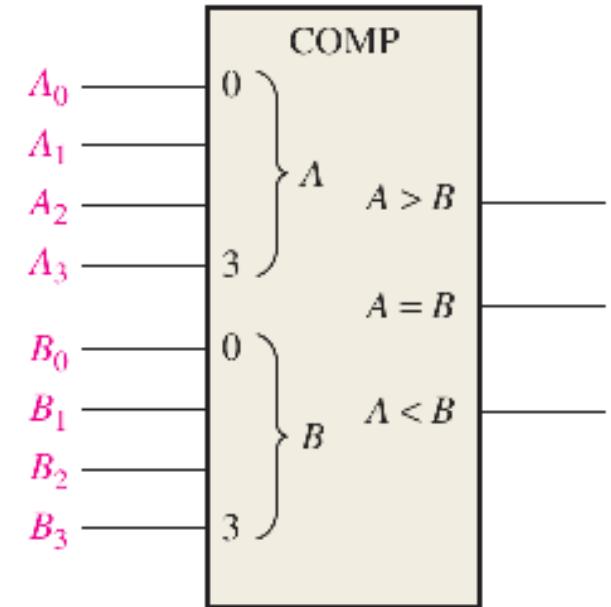
Chapter 5



# The Binary comparator

# Binary comparator

- The comparison of two numbers is an operation that determines whether one number is greater than, less than, or equal to the other number
- A *magnitude comparator* is a combinational circuit that compares two numbers  $A$  and  $B$  and determines their relative magnitudes
- The outcome of the comparison is specified by three binary variables that indicate whether  $A > B$ ,  $A = B$ , or  $A < B$
- On the one hand, the circuit for comparing two  $n$ -bit numbers has  $2^{2n}$  entries in the truth table and becomes too cumbersome, even with  $n = 3$
- On the other hand, as one may suspect, a comparator circuit possesses a certain amount of regularity



# Binary comparator

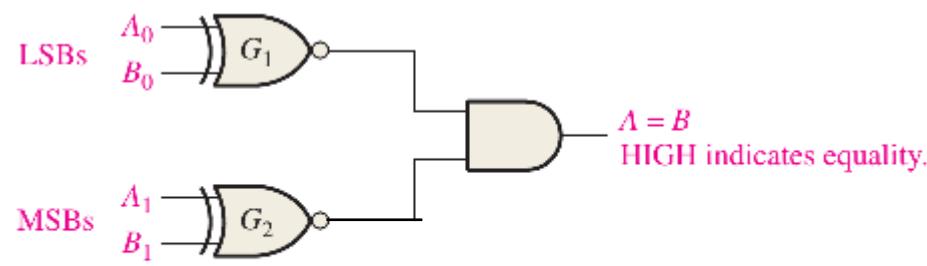
- Consider two numbers,  $A$  and  $B$ , with four digits each  $A_3A_2A_1A_0$  and  $B_3B_2B_1B_0$
- The two numbers are equal if all pairs of significant digits are equal:  $A_3 = B_3$ ,  $A_2 = B_2$ ,  $A_1 = B_1$ , **and**  $A_0 = B_0$
- To check bit-wise equality, we can use the XNOR gate

$$x_i = A_i B_i + A'_i B'_i \text{ for } i = 0, 1, 2, 3$$

- For equality to exist, all  $x_i$  variables must be equal to 1, a condition that dictates an AND operation of all variables:

$$(A = B) = x_3 x_2 x_1 x_0$$

Eg: Comparison of 2-bit numbers



General format: Binary number  $A \rightarrow A_1A_0$   
Binary number  $B \rightarrow B_1B_0$

# Binary comparator

- To determine whether  $A$  is greater or less than  $B$ , we inspect the relative magnitudes of pairs of significant digits, starting from the most significant position
- If the two digits of a pair are equal, we compare the next lower significant pair of digits
- The comparison continues until a pair of unequal digits is reached
- If the corresponding digit of  $A$  is 1 and that of  $B$  is 0, we conclude that  $A > B$ . Else, we have  $A < B$

Eg:  $A_3 A_2 A_1 A_0 = 1001$  and  $B_3 B_2 B_1 B_0 = 1010$

# Binary comparator

- The comparison can be expressed logically by the two Boolean functions:

$$(A > B) = A_3B'_3 + x_3A_2B'_2 + x_3x_2A_1B'_1 + x_3x_2x_1A_0B'_0$$

$$(A < B) = A'_3B_3 + x_3A'_2B_2 + x_3x_2A'_1B_1 + x_3x_2x_1A'_0B_0$$

# Binary comparator

$$x_i = A_i B_i + A'_i B'_i \text{ for } i = 0, 1, 2, 3$$

$$(A = B) = x_3 x_2 x_1 x_0$$

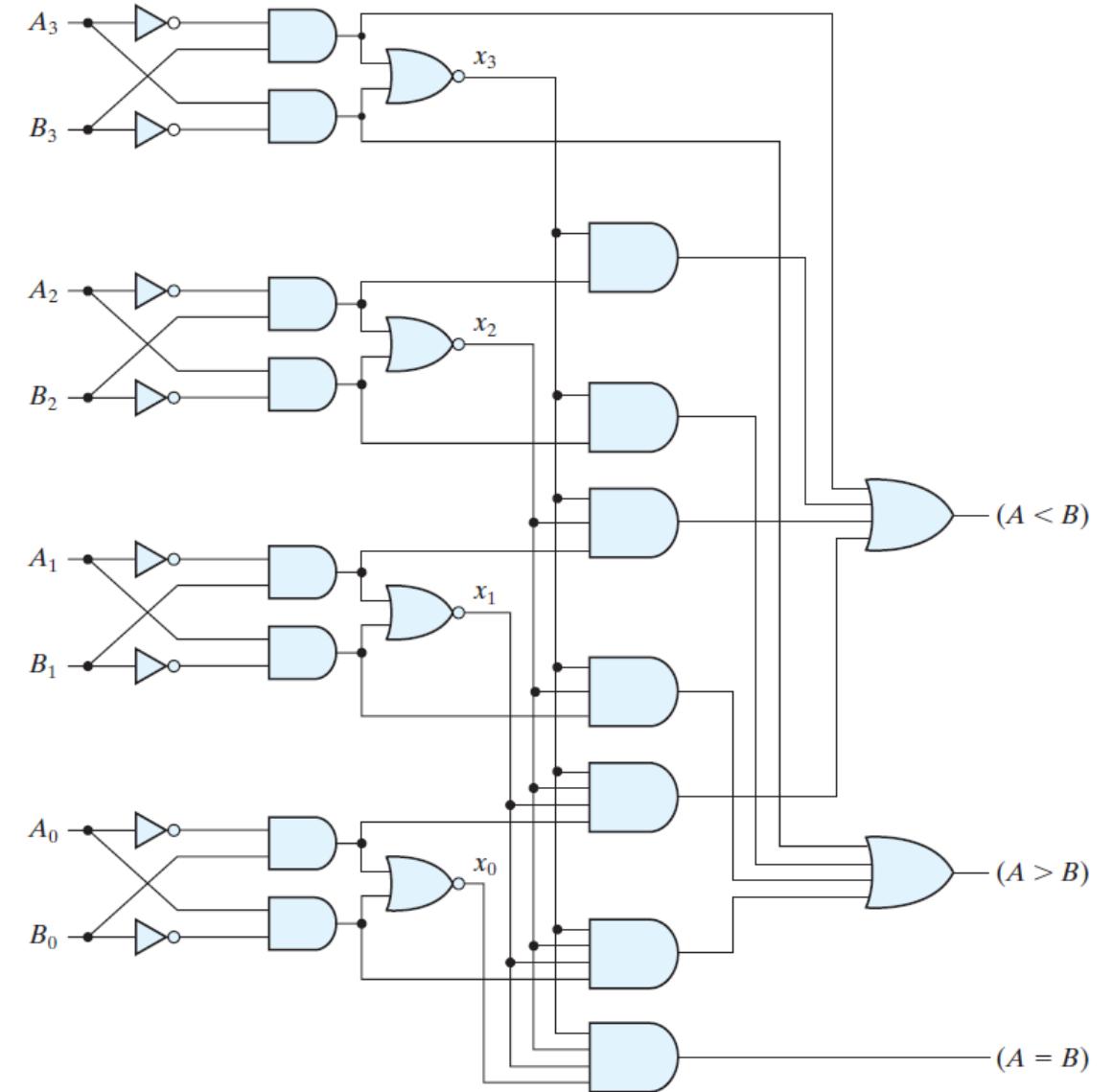
$$(A > B)$$

$$\begin{aligned} &= A_3 B'_3 + x_3 A_2 B'_2 + x_3 x_2 A_1 B'_1 \\ &+ x_3 x_2 x_1 A_0 B'_0 \end{aligned}$$

$$(A < B)$$

$$\begin{aligned} &= A'_3 B_3 + x_3 A'_2 B_2 + x_3 x_2 A'_1 B_1 \\ &+ x_3 x_2 x_1 A'_0 B_0 \end{aligned}$$

Interesting: Can we prove that only one of  $(A=B)$ ,  $(A>B)$  and  $(A<B)$  will be “1” at any given time?

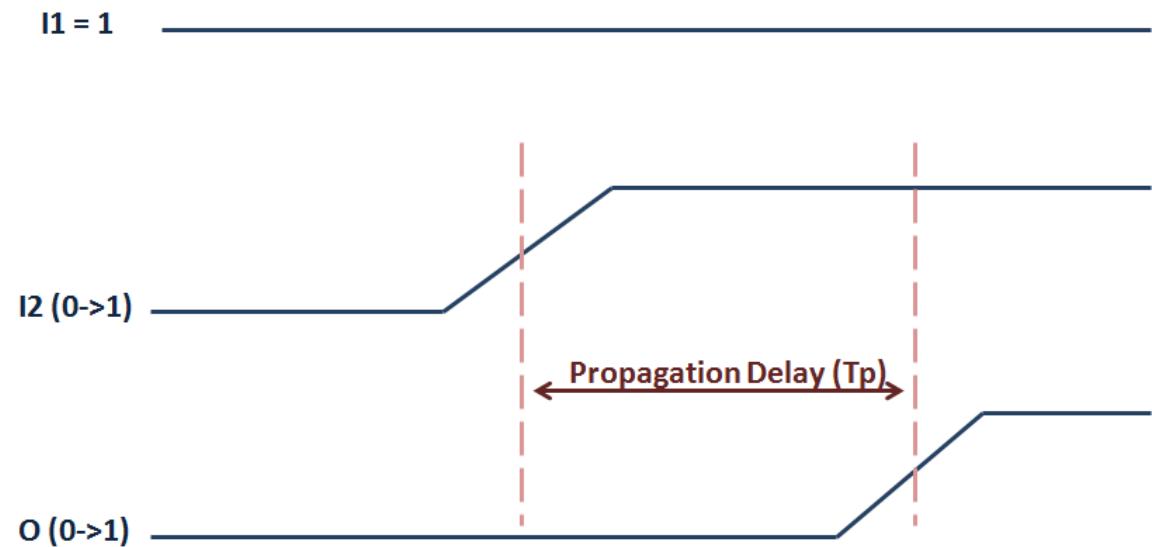
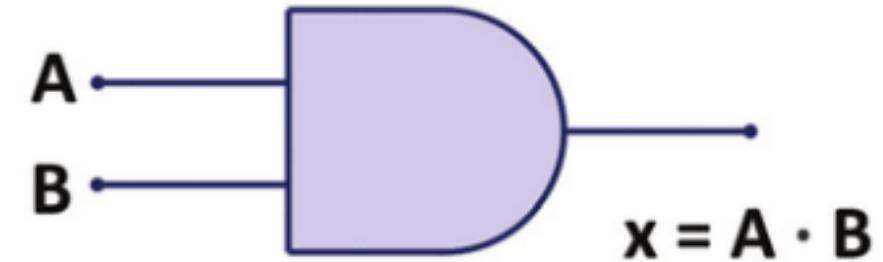


# Lecture 16 – Sequential circuits

Chapter 5

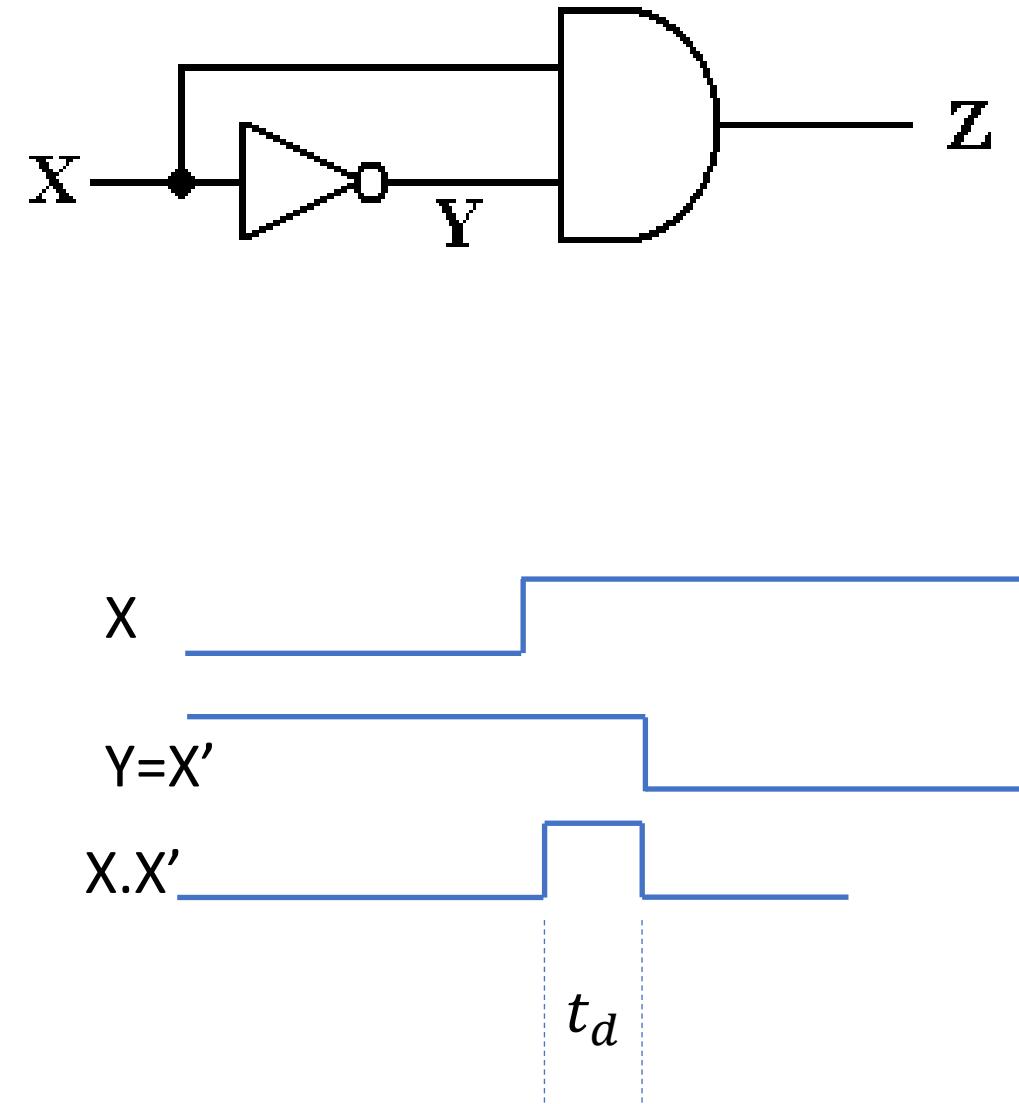
# Gate delays

- When we apply an input to a gate, the output does not change immediately – there is a delay between the application of the input and appearance of the correct output
- This delay is generally of the order of nanoseconds, but can add up as signal goes through multiple gates
- *While making a digital IC design, gate delay (and by extension timing) is the single biggest consideration of all time!*
- Other things to worry about are silicon real estate, power consumption, reliability, testability, etc.



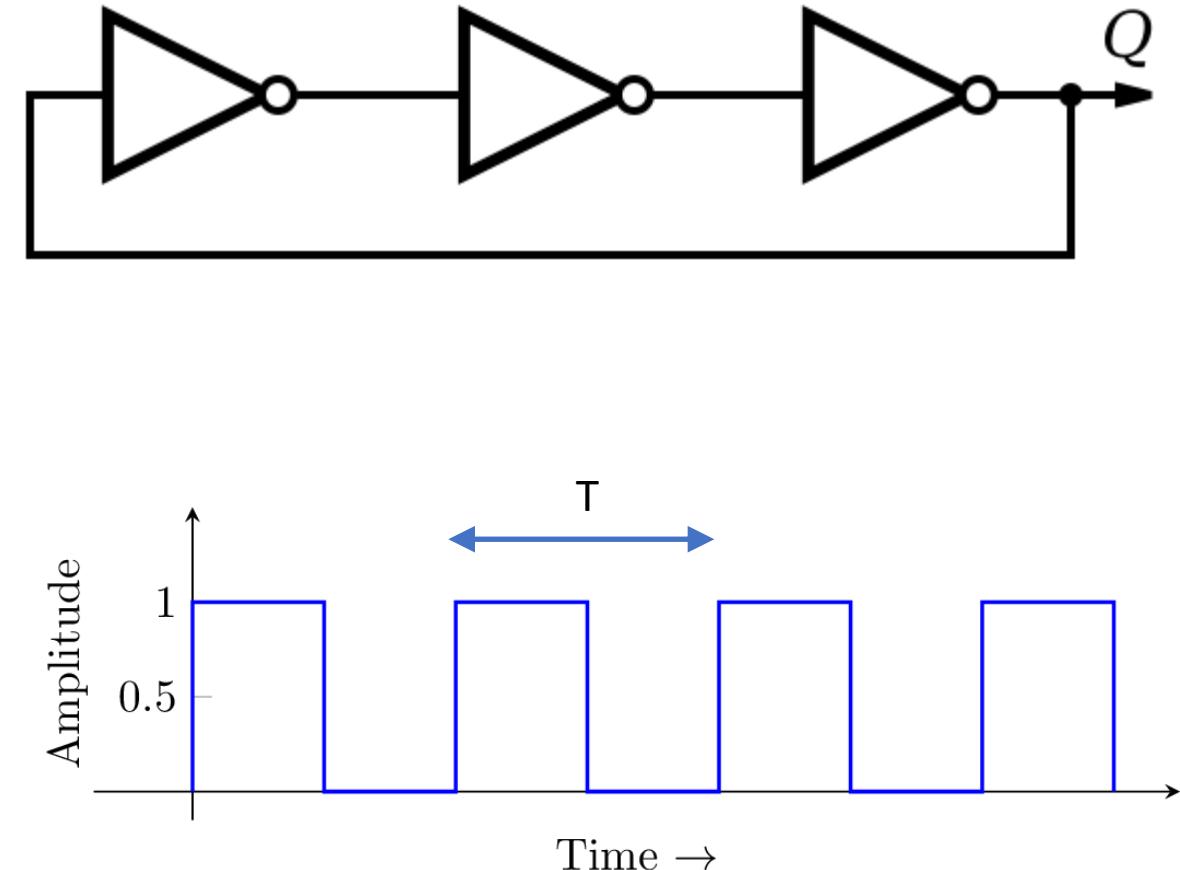
# Glitches

- *Improperly handled timing can lead to glitches and cause unexpected results*
- This is particularly true for multi-level logic implementation wherein different literals are bypassing some levels
- Consider an AND gate connected to X and  $X'$
- Such a connection will **ALWAYS** produce a glitch for X ( $0 \rightarrow 1$ )
- You can be sad about this glitch, or you can call this circuit a “pulse generator”!  
😊



# Ring oscillators

- Another clever use of gate delays in is in the construction of **ring oscillators**
- It is a series of odd number of NOT gates with the output connected back to the input gate of the first gate
- With this system, we can generate a continuous square wave at Q
- What is the frequency of the wave?
- In this case, it is  $f = \frac{1}{T} = \frac{1}{3 \times 2t_d}$

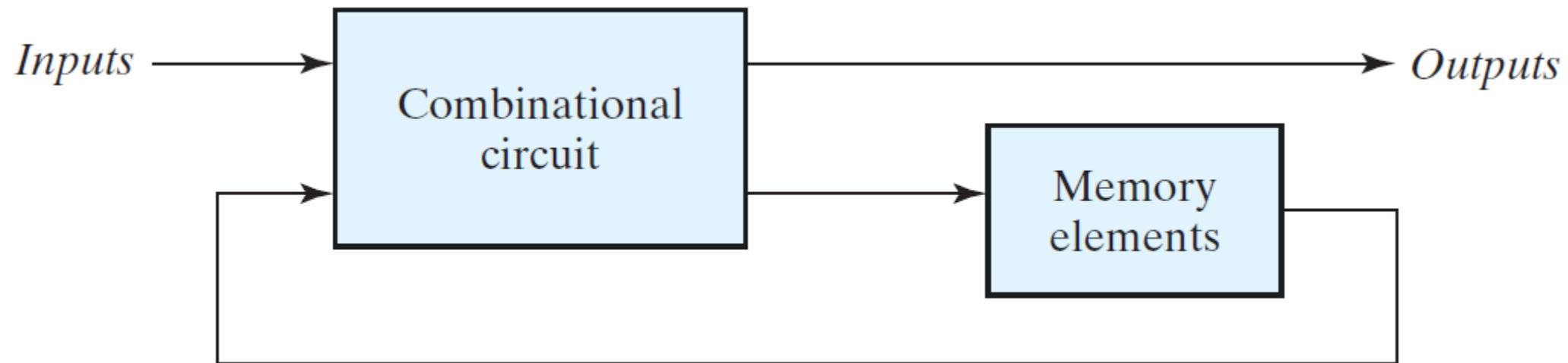


# Sequential circuits

- The technology enabling and supporting modern digital devices is critically dependent on electronic components that can store information, i.e., have memory
- We will examine the operation and control of these devices and their use in circuits and enables you to better understand what is happening in these devices when you interact with them
- The digital circuits considered thus far have been **combinational**—their output depends only and immediately on their inputs—they have no memory, i.e., dependence on past values of their inputs
- **Sequential circuits**, however, act as storage elements and have memory, i.e., their output depends on past inputs as well

# Sequential circuits

- The main way sequential circuits remember things is through feedback paths and memory elements
- We know how combinational circuits are created
- The simplest storage elements (memory) used in sequential circuits are called *latches*



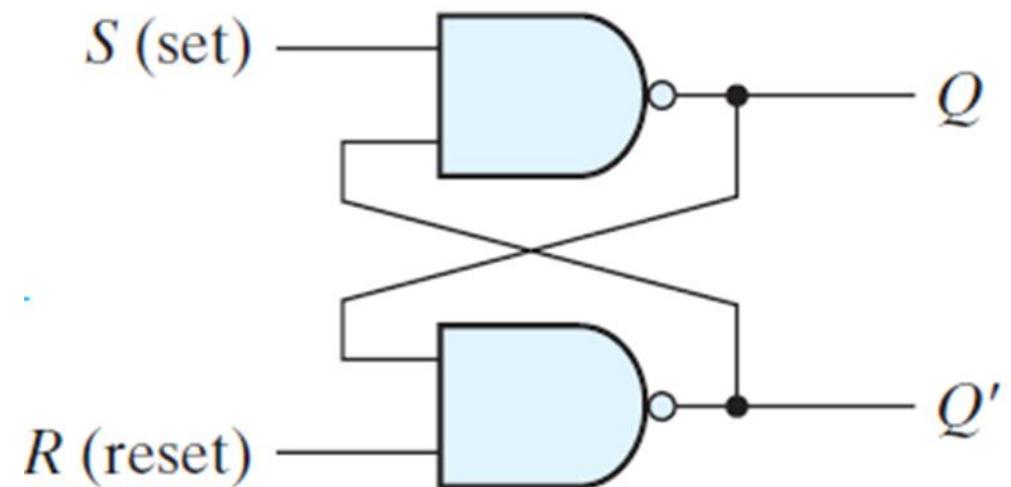
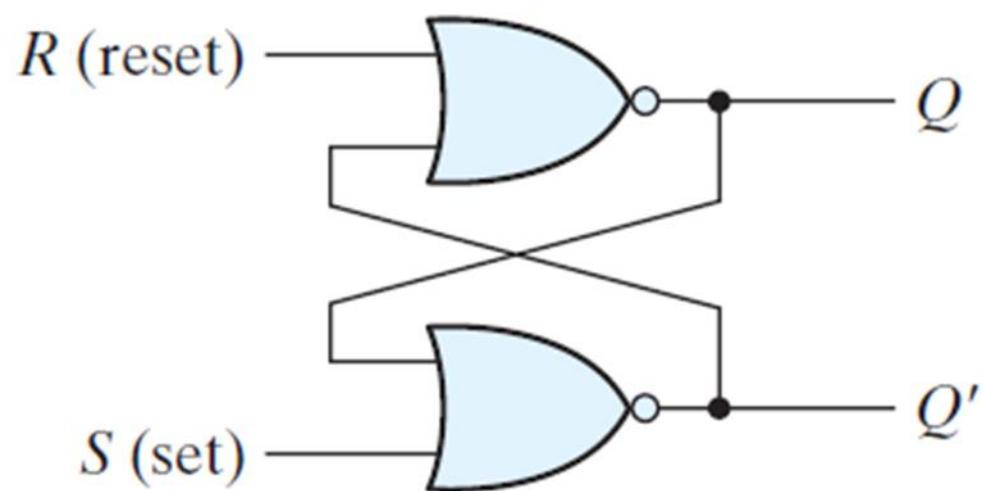
# Latches

---

- A storage element in a digital circuit can maintain a binary state indefinitely (as long as power is delivered to the circuit), until directed by an input signal to switch states
- Such a storage element is called a *latch*
- The major differences among various types of storage elements are in the number of inputs they possess and in the manner in which the inputs affect the binary state

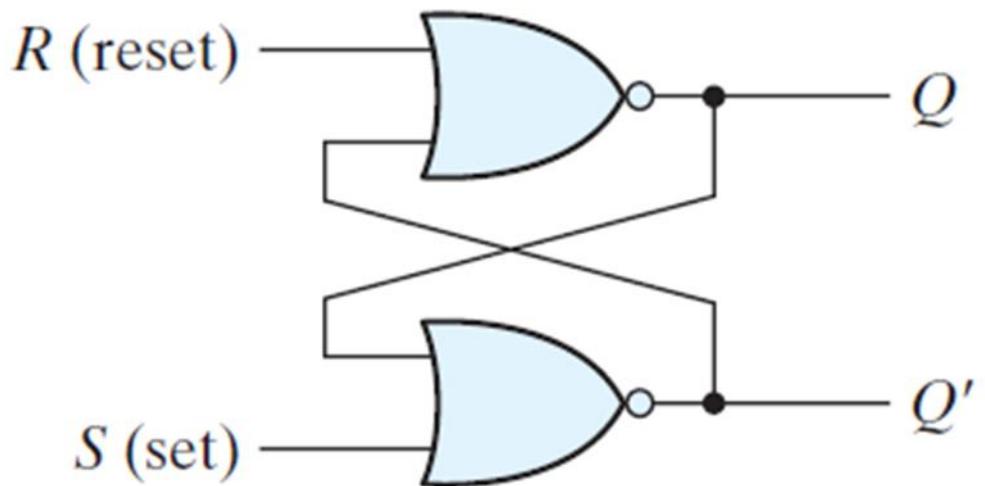
# SR Latch

- The ( Set – Reset) **SR latch** is a circuit with two cross-coupled NOR gates or two cross-coupled NAND gates, and two inputs labeled *S* and *R*



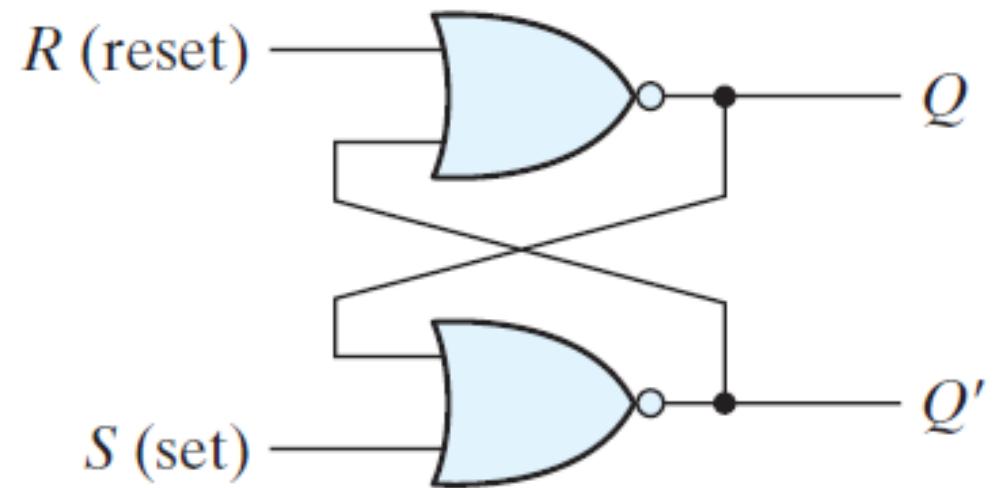
# SR Latch

- We realize that the outputs  $Q$  and  $Q'$  are normally the complement of each other
- The latch has two stable states
- When output  $Q = 1$  and  $Q' = 0$ , the latch is said to be in the *set state*
- When  $Q = 0$  and  $Q' = 1$ , it is in the *reset state*



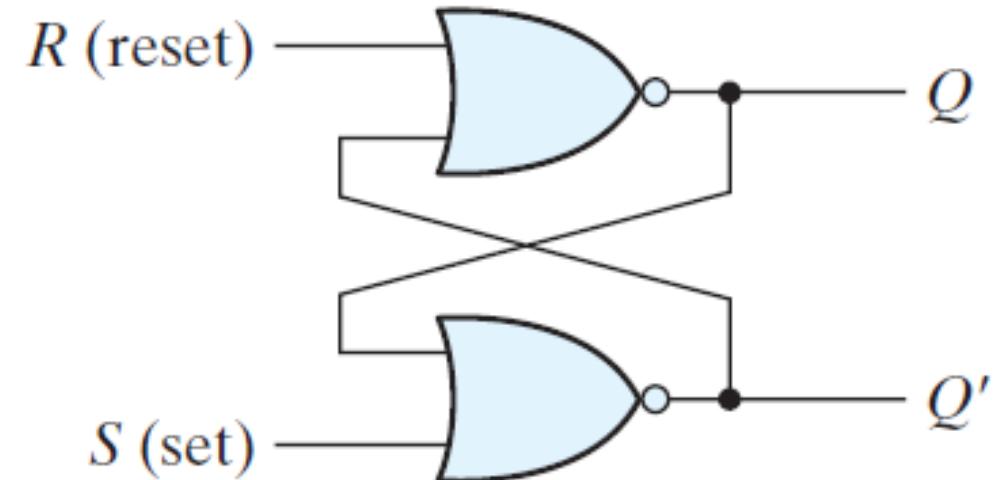
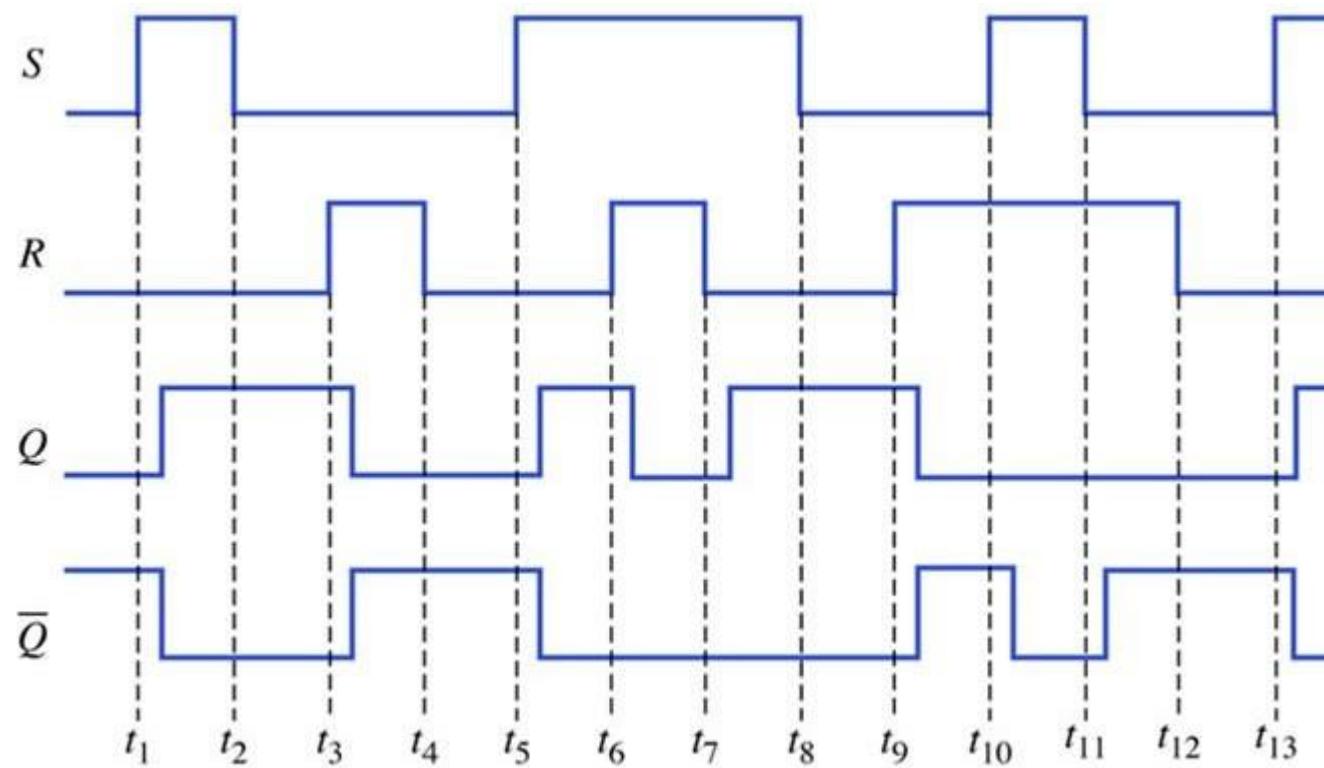
# SR Latch

- Setting and resetting can be done through the two inputs ( $S, R$ )
- After setting or resetting, applying 00 at the input retains the previous state
- However, when both inputs are equal to 1 at the same time, a condition in which both outputs are equal to 0 (rather than be mutually complementary) occurs
- If both inputs are then switched to 0 simultaneously, the device will enter an unpredictable or undefined state or a metastable state

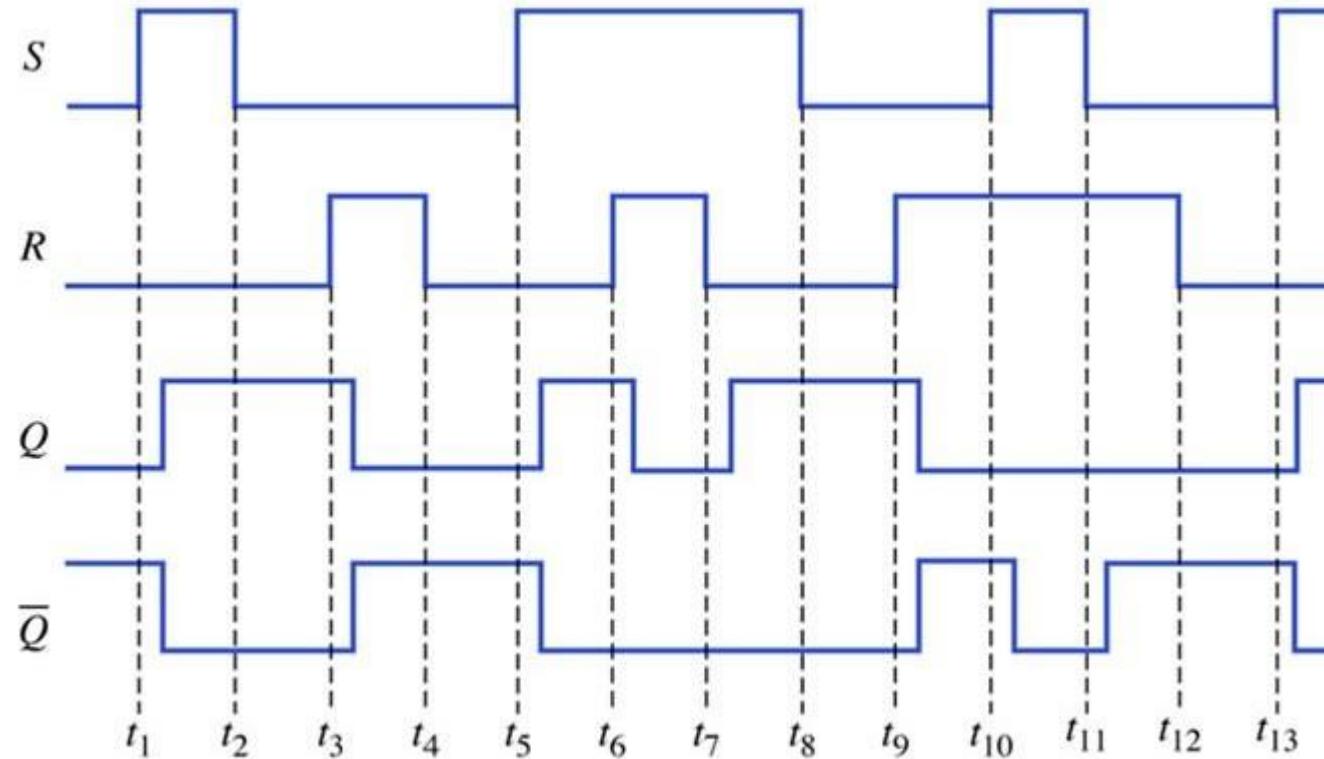


$S$	$R$	$Q$	$Q'$	
1	0	1	0	
0	0	1	0	(after $S = 1, R = 0$ )
0	1	0	1	
0	0	0	1	(after $S = 0, R = 1$ )
1	1	0	0	(forbidden)

# SR Latch

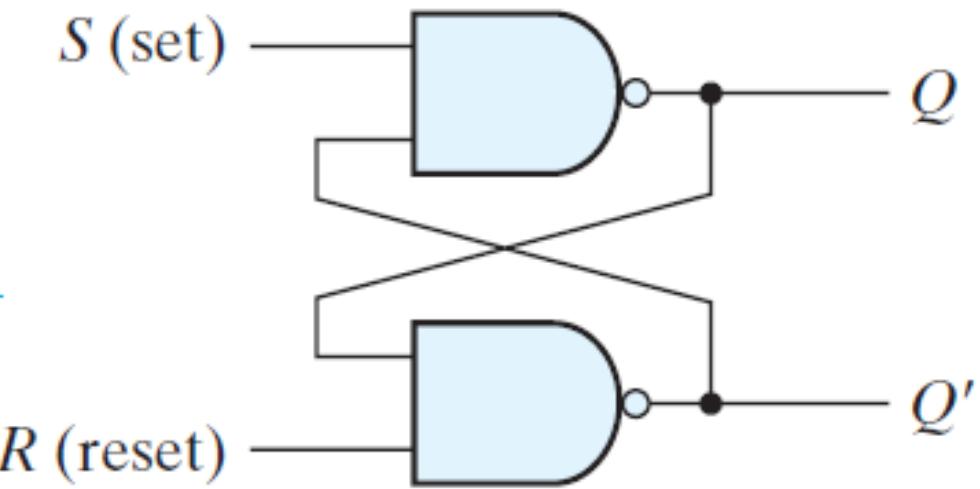


$S$	$R$	$Q$	$Q'$
1	0	1	0
0	0	1	0 (after $S = 1, R = 0$ )
0	1	0	1
0	0	0	1 (after $S = 0, R = 1$ )
1	1	0	0 (forbidden)



# SR Latch – NAND implementation

- The SR latch with two cross-coupled NAND gates behaves in a similar way to NOR implementation
- It operates with both inputs normally at 1, unless the state of the latch has to be changed
- The application of 0 to the S input causes output Q to go to 1, putting the latch in the set state
- When the S input goes back to 1, the circuit remains in the set state
- The condition that is forbidden for the NAND latch is both inputs being equal to 0 at the same time, an input combination that should be avoided



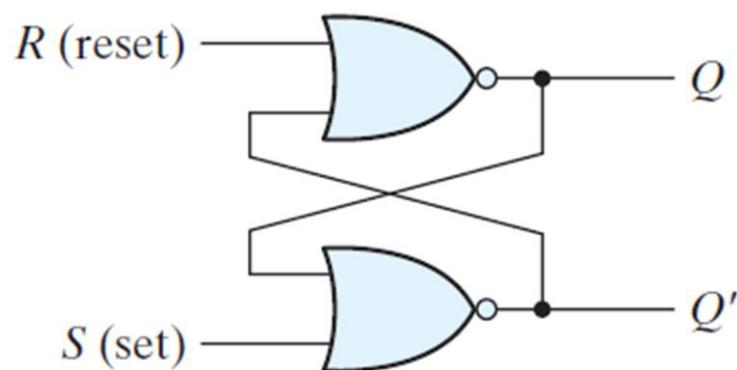
S	R	Q	Q'	
1	0	0	1	
1	1	0	1	(after $S = 1, R = 0$ )
0	1	1	0	
1	1	1	0	(after $S = 0, R = 1$ )
0	0	1	1	(forbidden)

# Lecture 17 – Sequential circuits 2

Chapter 5

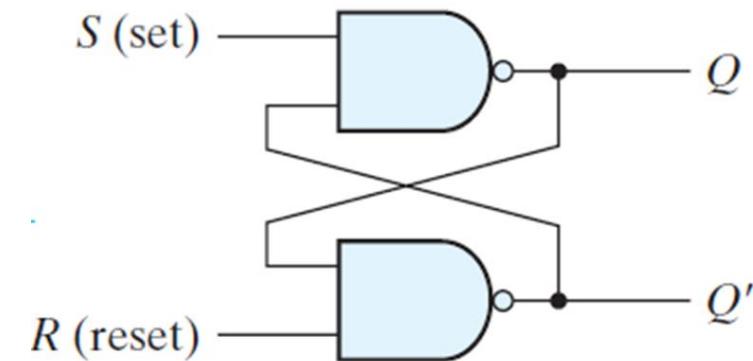
# SR Latch

- The ( Set – Reset) **SR latch** is a circuit with two cross-coupled NOR gates or two cross-coupled NAND gates, and two inputs labeled  $S$  and  $R$



SR NOR Latch

$S$	$R$	$Q$	$Q'$
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 (forbidden)

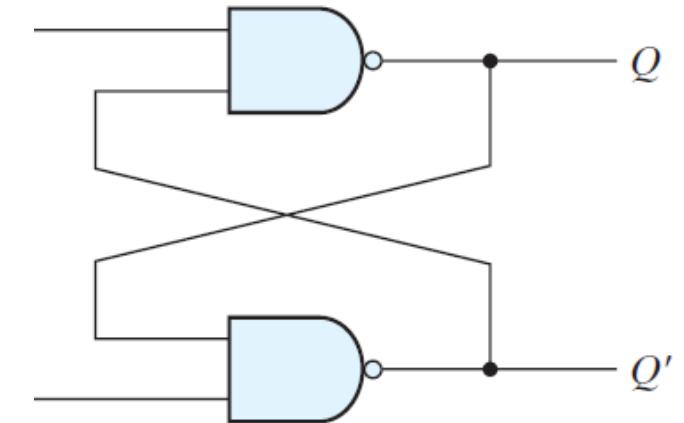


SR NAND Latch ( $S'R'$  latch)

$S$	$R$	$Q$	$Q'$
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 (forbidden)

# Latch with enable

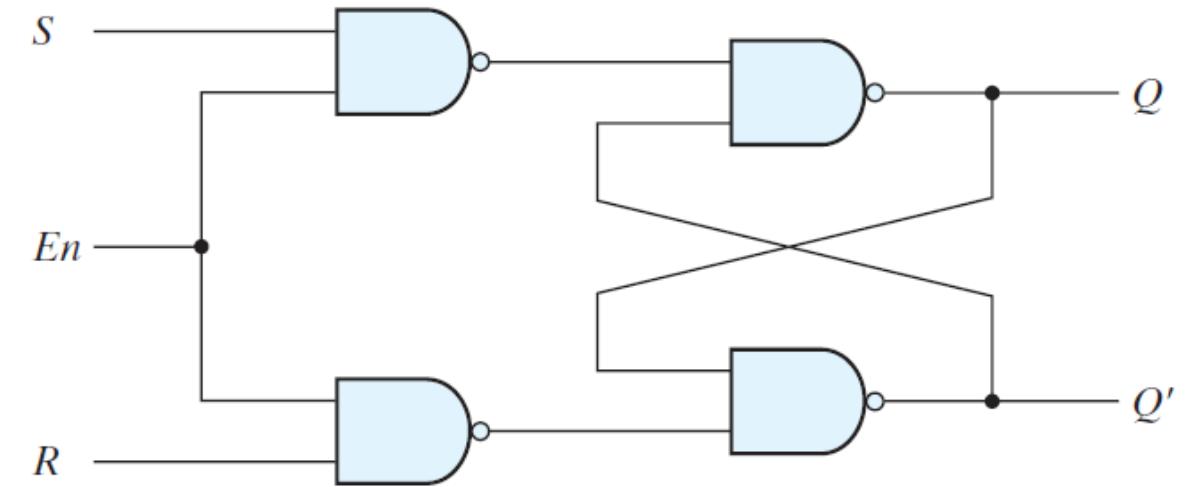
- The operation of the basic SR latch can be modified by providing an additional input signal that determines (controls) *when* the state of the latch can be changed by S and R ( $S'$  and  $R'$ )
- It consists of the basic SR latch and two additional NAND gates
- The control input *En* acts as an *enable* signal for the other two inputs
- The outputs of the NAND gates stay at the logic-1 level as long as the enable signal remains at 0
- This is the quiescent condition for the SR latch



<i>En</i>	<i>S</i>	<i>R</i>	Next state of <i>Q</i>
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

# Latch with enable

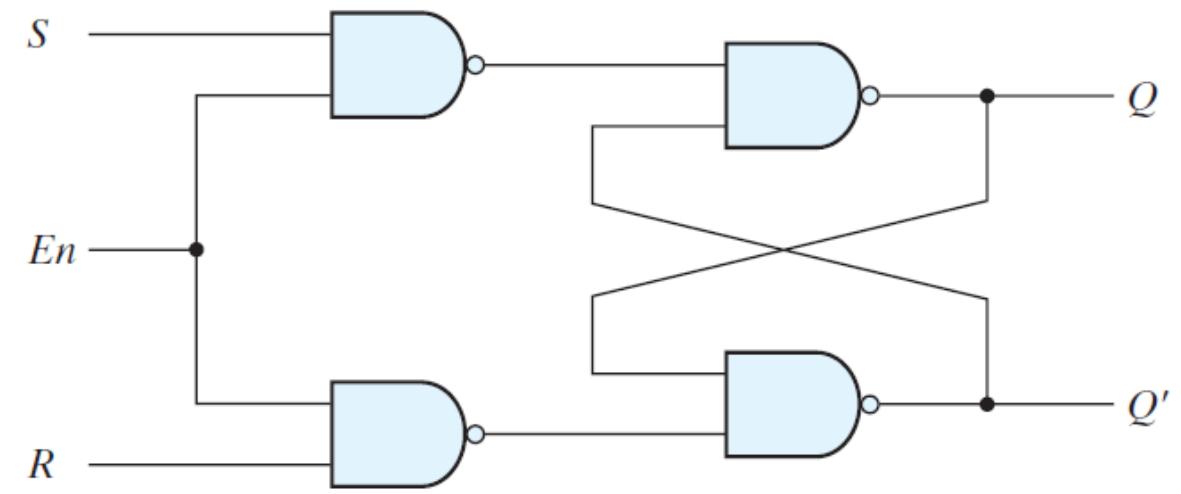
- When the enable input goes to 1, information from the  $S$  or  $R$  input is allowed to affect the latch
  - The set state is reached with  $S = 1, R = 0$ , and  $En = 1$  (active-high enabled) and reset is reached with  $S = 0, R = 1$ , and  $En = 1$
  - In either case, when  $En$  returns to 0, the circuit remains in its previous stable state
  - Further, when  $En = 1$  and both the  $S$  and  $R$  inputs are equal to 0, the state of the circuit does not change



$En$	$S$	$R$	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

# Latch with enable

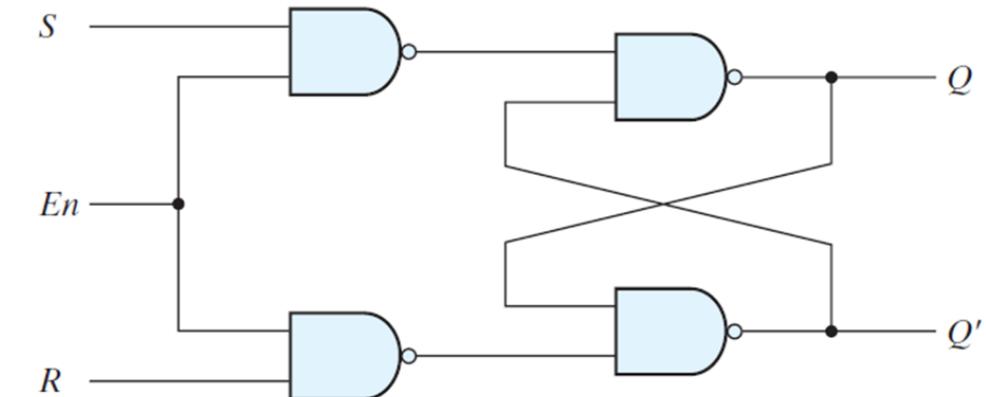
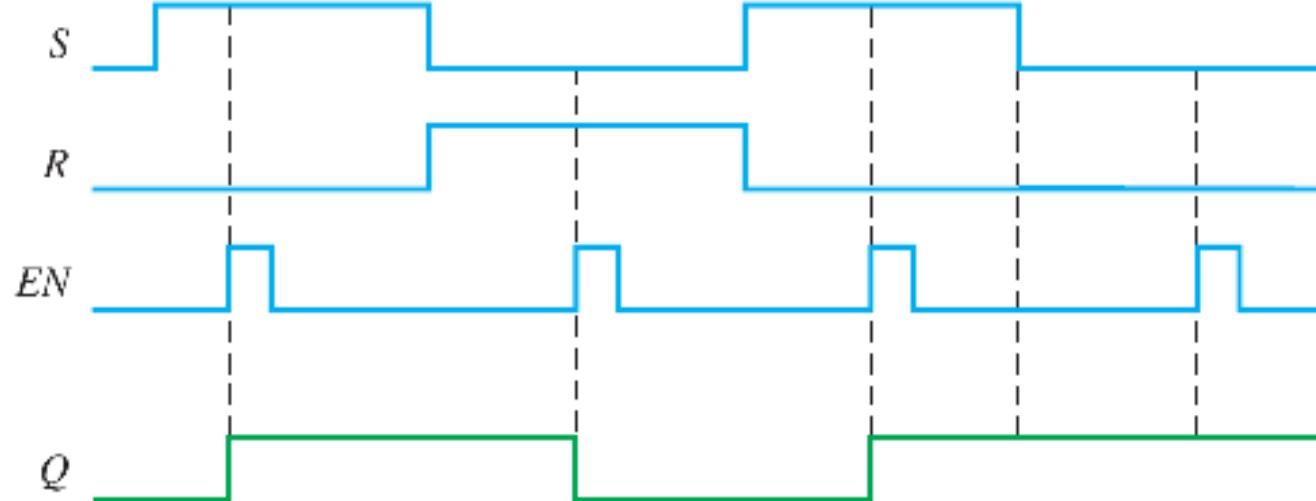
- An indeterminate condition occurs when all three inputs are equal to 1
  - This condition places 0's on both inputs of the basic SR latch, which puts it in the undefined state
  - When the enable input goes back to 0, one cannot conclusively determine the next state, because it depends on whether the  $S$  or  $R$  input goes to 0 first
  - This indeterminate condition makes this circuit difficult to manage, and it is seldom used in practice
- Nevertheless, the SR latch is an important circuit because flip-flops are constructed from it



$En$	$S$	$R$	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

# Latch with enable

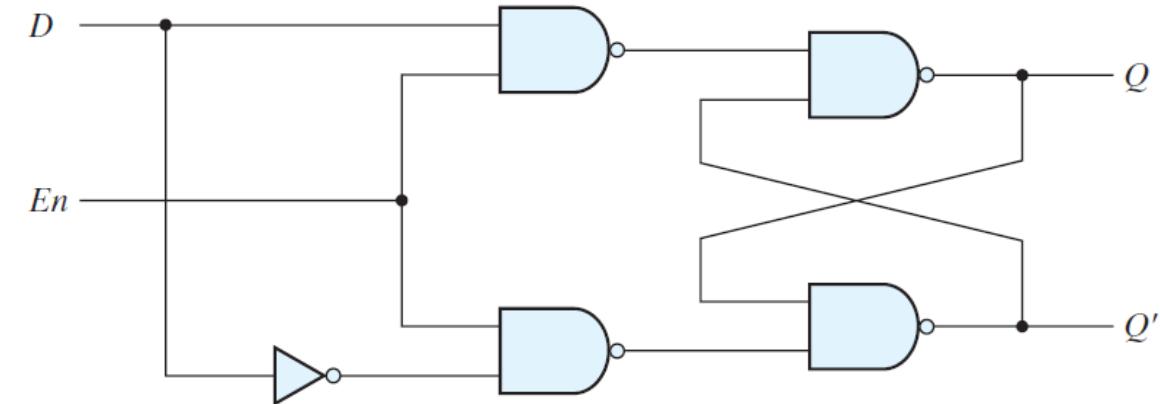
Eg: Determine the Q output waveform if the inputs shown are applied to a gated S-R latch with enable. Assume the latch is initially RESET.



En	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

# D Latch

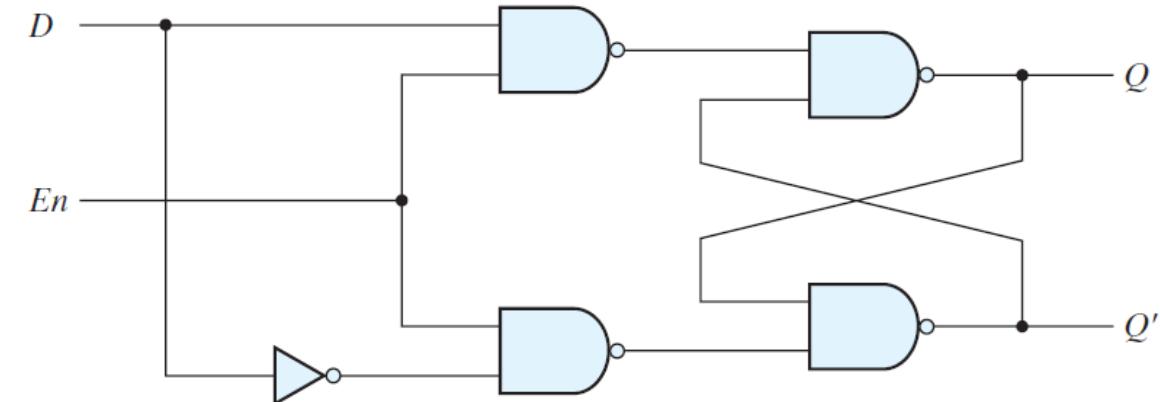
- One way to eliminate the undesirable condition of the indeterminate state in the SR latch is to ensure that inputs S and R are never equal to 1 at the same time
- This is done in the **D** latch
  - The *D* input goes directly to the *S* input, and its complement is applied to the *R* input
  - As long as the enable input is at 0, the cross-coupled SR latch has both inputs at the 1 level and the circuit cannot change state regardless of the value of *D*
  - The *D* input is sampled when *En* = 1. If *D* = 1, the *Q* output goes to 1, placing the circuit in the set state
  - If *D* = 0, output *Q* goes to 0, placing the circuit in the reset state



<i>En</i>	<i>D</i>	Next state of <i>Q</i>
0	X	No change
1	0	<i>Q</i> = 0; reset state
1	1	<i>Q</i> = 1; set state

# D Latch

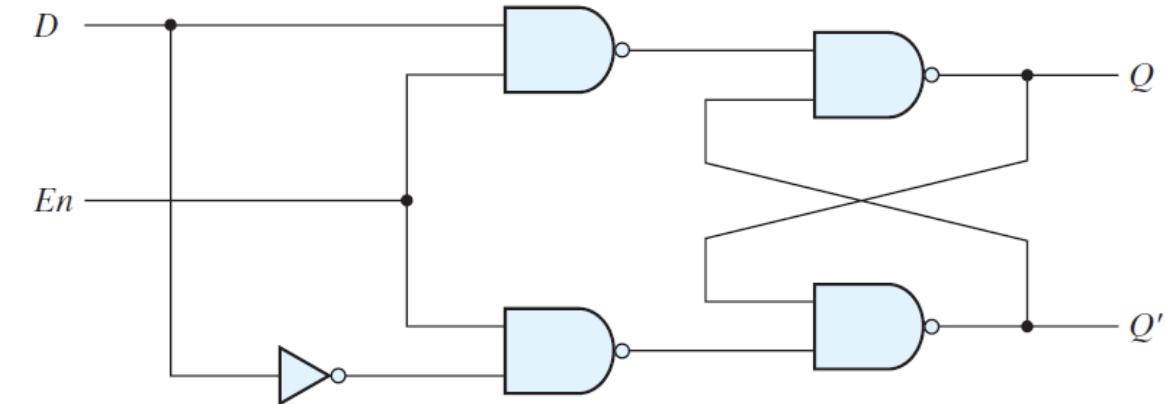
- The  $D$  latch receives that designation from its ability to hold *data* in its internal storage
- It is suited for use as a temporary storage for binary information between a unit and its environment
- The binary information present at the data input of the  $D$  latch is transferred to the  $Q$  output when the enable input is asserted
- The output follows changes in the data input as long as the enable input is asserted
- This situation provides a path from input  $D$  to the output, and for this reason, the circuit is often called a *transparent* latch



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

# D Latch

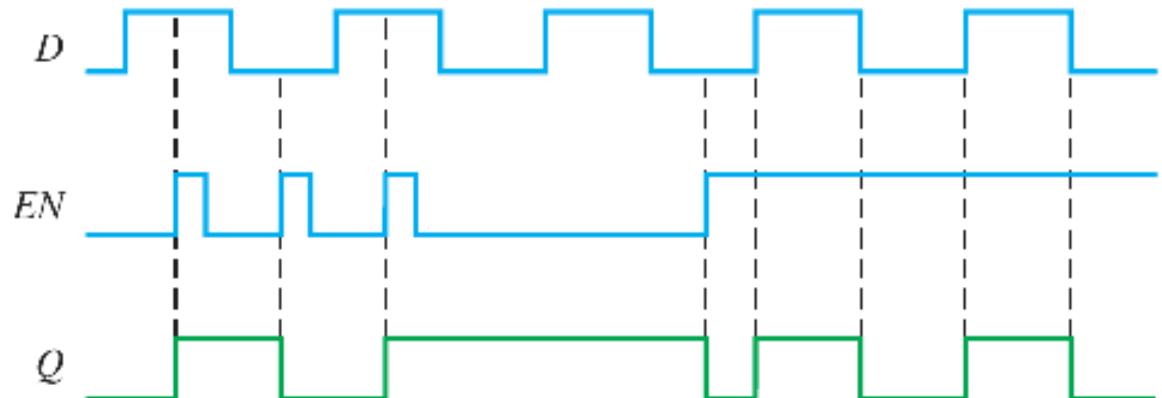
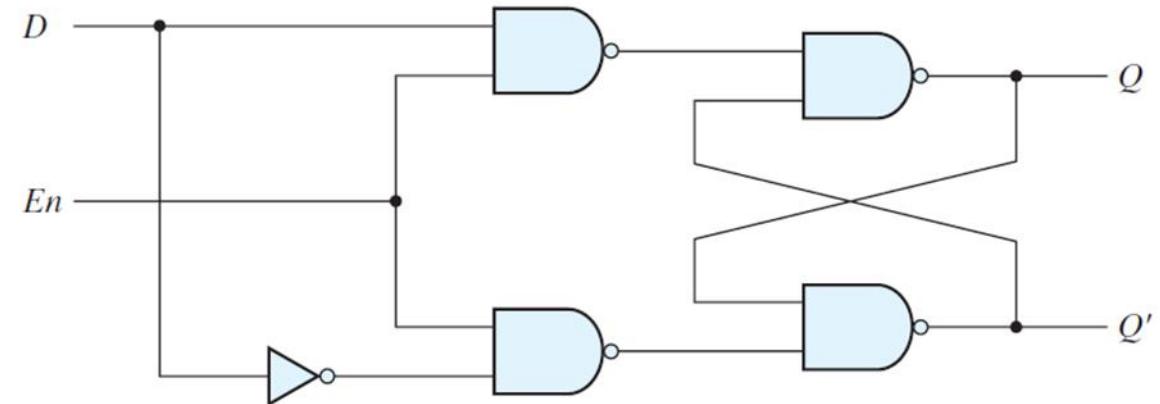
- When the enable input signal is de-asserted, the binary information that was present at the data input at the time the transition occurred is retained (i.e., stored) at the  $Q$  output until the enable input is asserted again
- Note that an inverter could be placed at the enable input
- Then, depending on the physical circuit, the external enabling signal will be a value of 0 (**active low**) or 1 (**active high**)



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

# D Latch

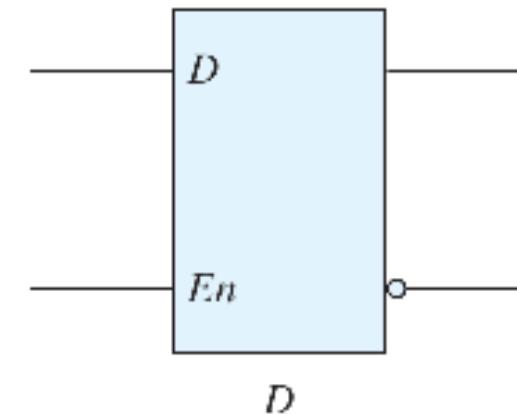
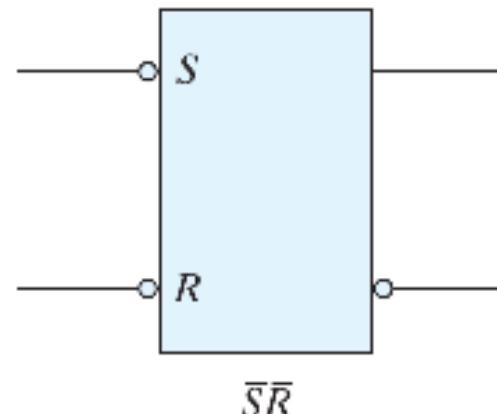
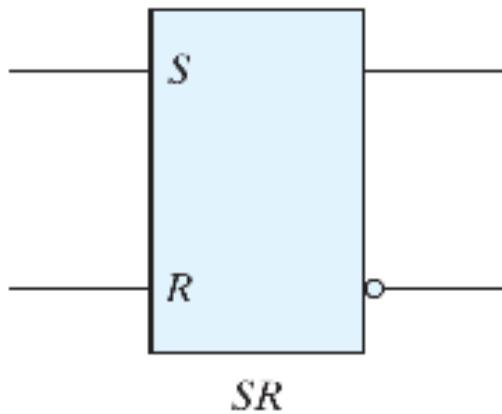
Eg: Determine the Q output waveform if the inputs shown in below are applied to a D latch, which is initially RESET.



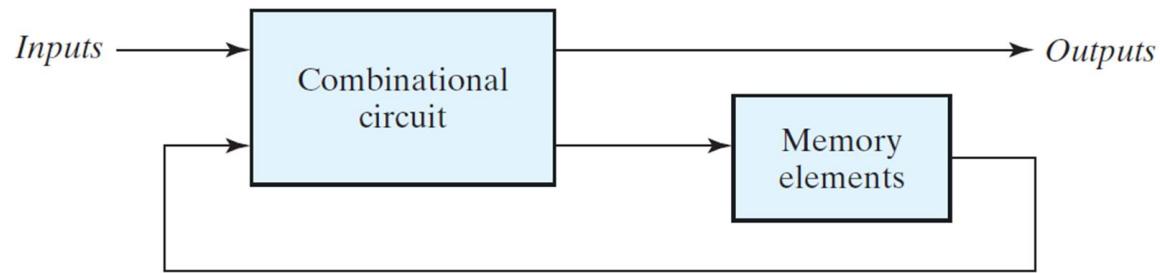
$En$	$D$	Next state of $Q$
0	X	No change
1	0	$Q = 0$ ; reset state
1	1	$Q = 1$ ; set state

# Latches

Graphic symbols for latches

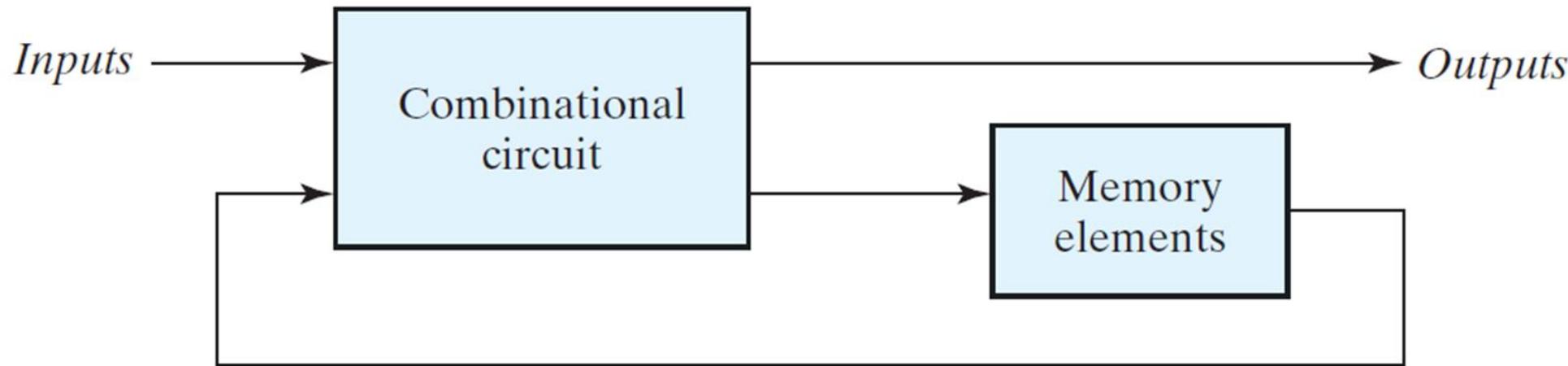


# Problem with latches



- A sequential circuit has a feedback path from the outputs of the flip-flops to the input of the combinational circuit
- Consequently, the inputs of the latches are derived in part from the outputs of the same and other latches
- The state transitions of the latches start as soon as the enable/data pulse changes to the logic-1 level
- The new state of a latch appears at the output while the pulse is still active
- This output is connected to the inputs of the latches through the combinational circuit
- If the inputs applied to the latches change again while the enable pulse is still at the logic-1 level, the latches will respond to new values and a new output state may occur

# Problem with latches



- If the inputs applied to the latches change again while the enable pulse is still at the logic-1 level, the latches will respond to new values and a new output state may occur
- The result is an unpredictable situation, since the state of the latches may keep changing for as long as the enable pulse stays at the active level
- Because of this unreliable operation, the output of a latch cannot be applied directly or through combinational logic to the input of the same or another latch when all the latches are triggered by a enable signal

# Flip flops

- **Flip-flops** are constructed in such a way as to make them operate properly when they are part of a sequential circuit that employs a **common clock**
- The problem with the latch is that it responds to a change in the **level** of a clock pulse
- A positive level response in the enable input allows changes in the output when the **D** input changes while the clock pulse stays at logic 1
- The key to the proper operation of a flip-flop is to trigger it only during a **signal transition**
- A clock pulse goes through two transitions: **positive transition (0 -> 1)** and **negative transition (1 -> 0)**



(a) Response to positive level



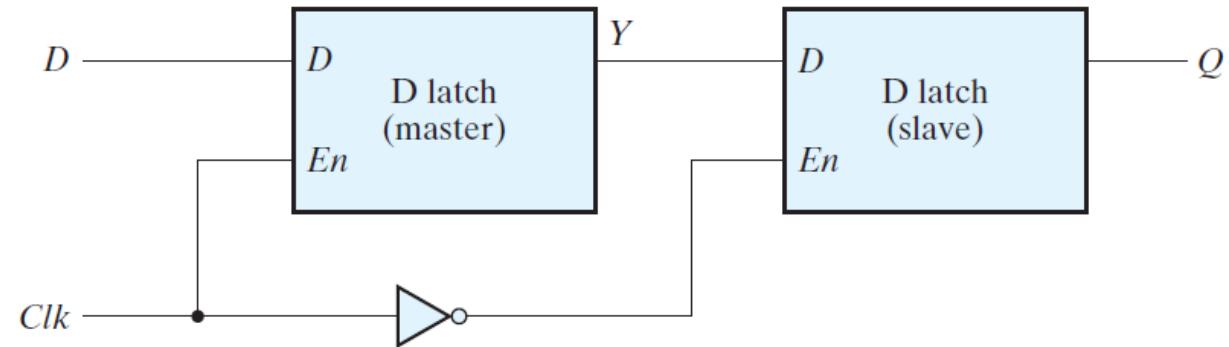
(b) Positive-edge response



(c) Negative-edge response

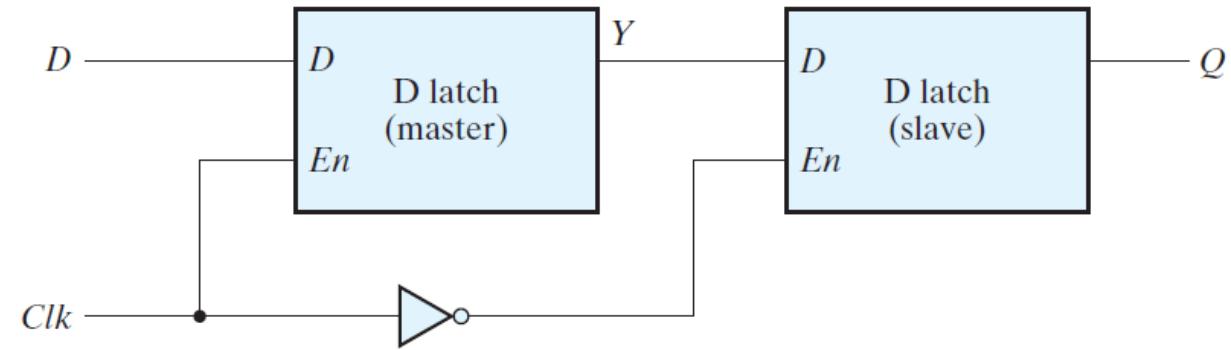
# Flip flops

- We can implement flip flops using two latches
- The first latch is called the *master* and the second the *slave*
- The circuit samples the  $D$  input and changes its output  $Q$  only at the negative edge of the synchronizing or controlling clock (designated as  $\text{Clk}$ )
  - When the  $\text{Clk} = 0$ , the slave latch is enabled, and its output  $Q$  is equal to the master output  $Y$
  - When the input pulse changes to 1, the data from the external  $D$  input are transferred to the master

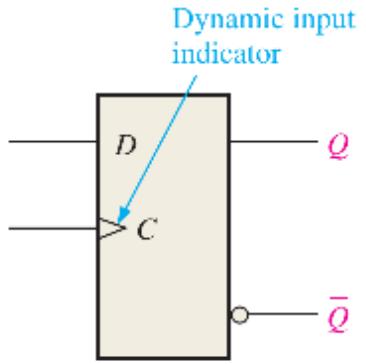


# Flip flops

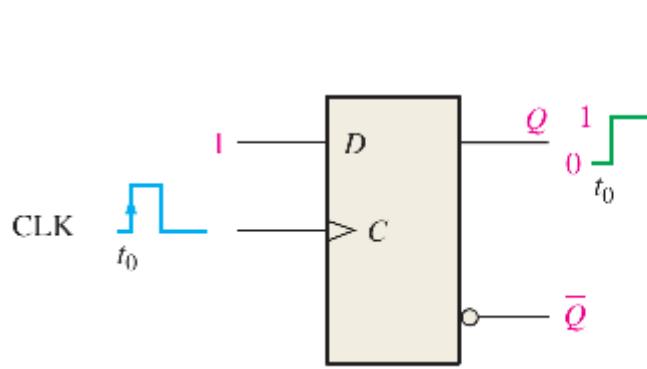
- The slave, however, is disabled as long as the clock remains at 1, because its *enable* input is equal to 0
- Any change in the input changes the master output at  $Y$ , but cannot affect the slave output
- When the clock pulse returns to 0, the master is disabled and is isolated from the  $D$  input
- At the same time, the slave is enabled and the value of  $Y$  is transferred to the output of the flip-flop at  $Q$
- Thus, a *change in the output of the flip-flop can be triggered only by and during the transition of the clock from 1 to 0 (-ve edge)*



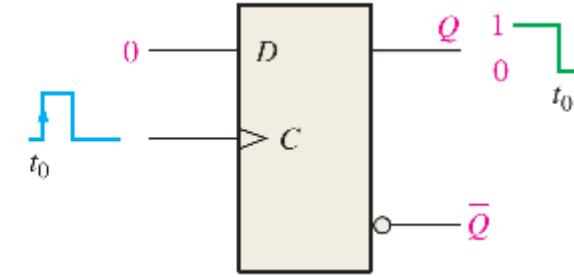
# Flip flops



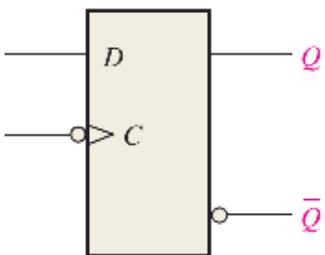
positive edge-triggered  
flip-flop



D = 1: flip-flop SETS on positive clock edge. (If already SET, it remains SET.)



D = 0: flip-flop RESETS on positive clock edge. (If already RESET, it remains RESET.)

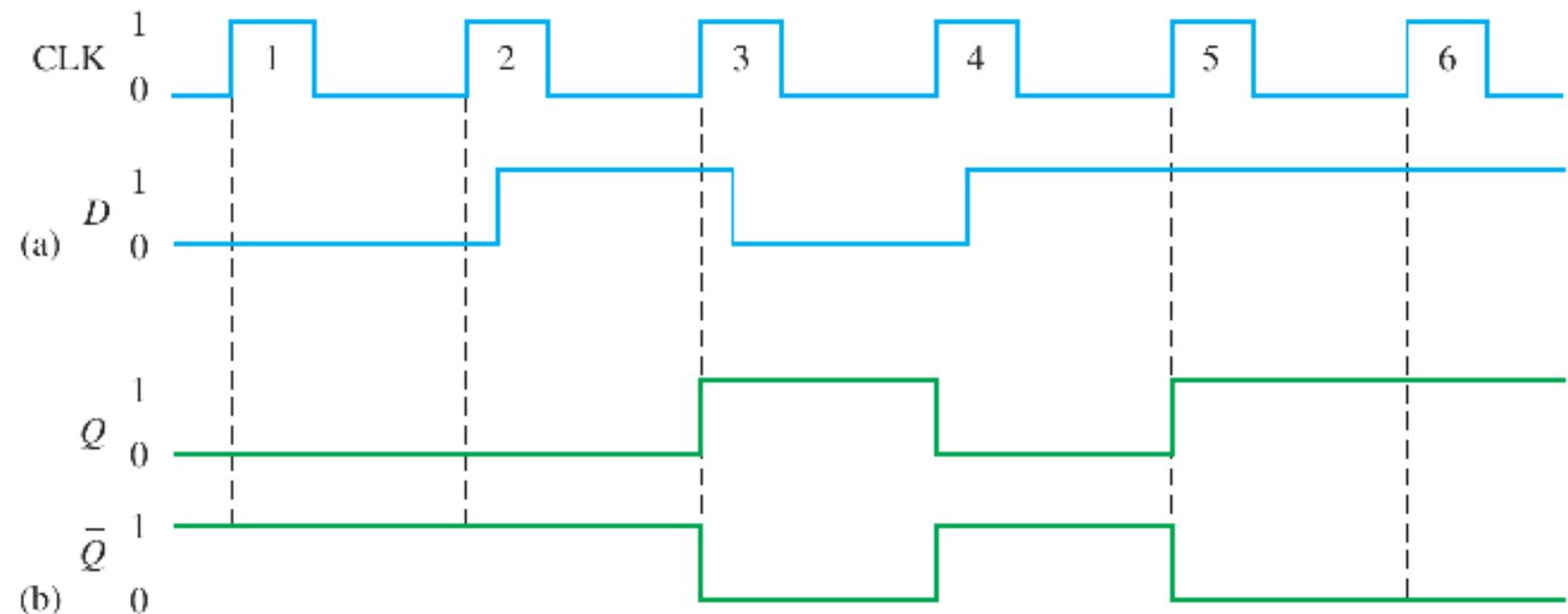
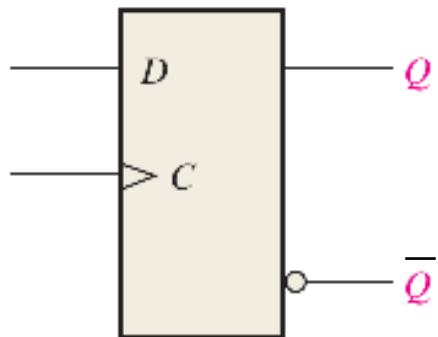


negative edge-triggered  
flip-flop

The dynamic input indicator means the flip-flop changes state only on the edge of a clock pulse.

# Flip flops

Determine the  $Q$  and  $\bar{Q}$  output waveforms of the positive edge-triggered D flip-flop. Assume the flip-flop is initially RESET.

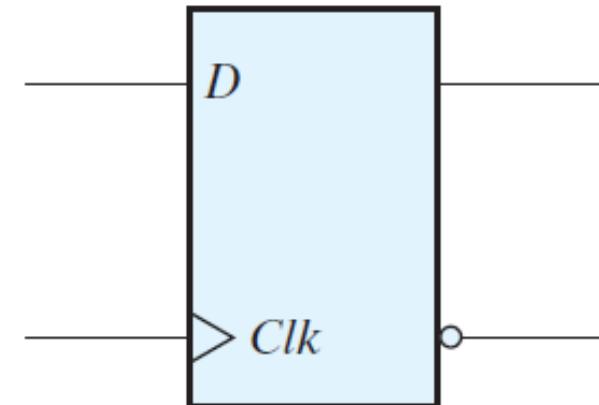


# Lecture 18 – Sequential circuits 3

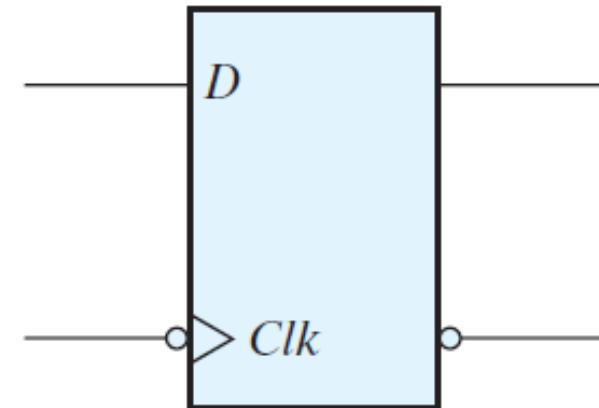
Chapter 5

# D Flip Flop

- D/Transparent flip-flop
- The bit at D is transferred to Q at the edge of the clock
- The information is retained upto the next edge of the clock



(a) Positive-edge

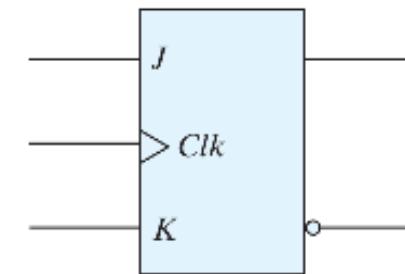
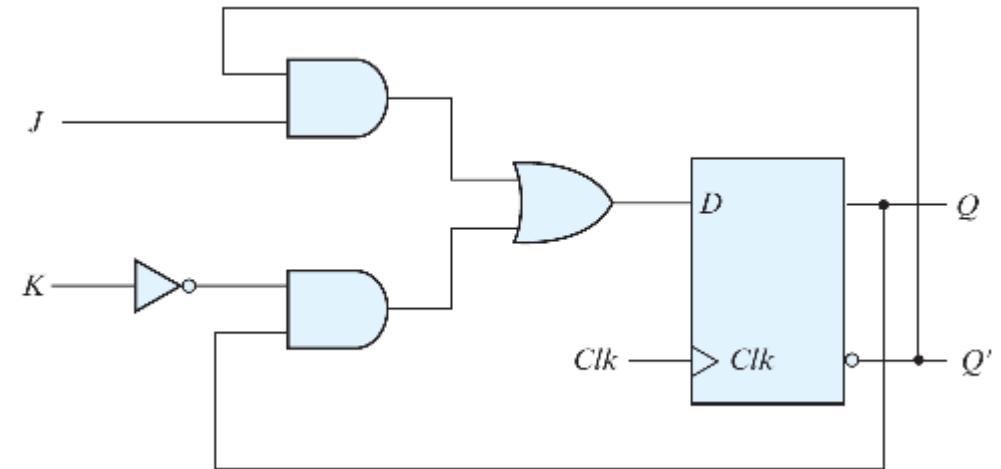


(a) Negative-edge

# JK Flip Flop

- There are four operations we are looking to perform in a flip-flop:
  - Set it to 1, reset it to 0, retain or complement its output
- With only a single input, the  $D$  flip-flop can set or reset the output, depending on the value of the  $D$  input immediately before the clock transition
- Synchronized by a clock signal, the  $JK$  flip-flop has two inputs and performs all four operations
- The  $J$  input sets the flip-flop to 1, the  $K$  input resets it to 0, and when both inputs are enabled, the output is complemented
- This can be obtained if the  $D$  input is:

$$D = JQ' + K'Q$$

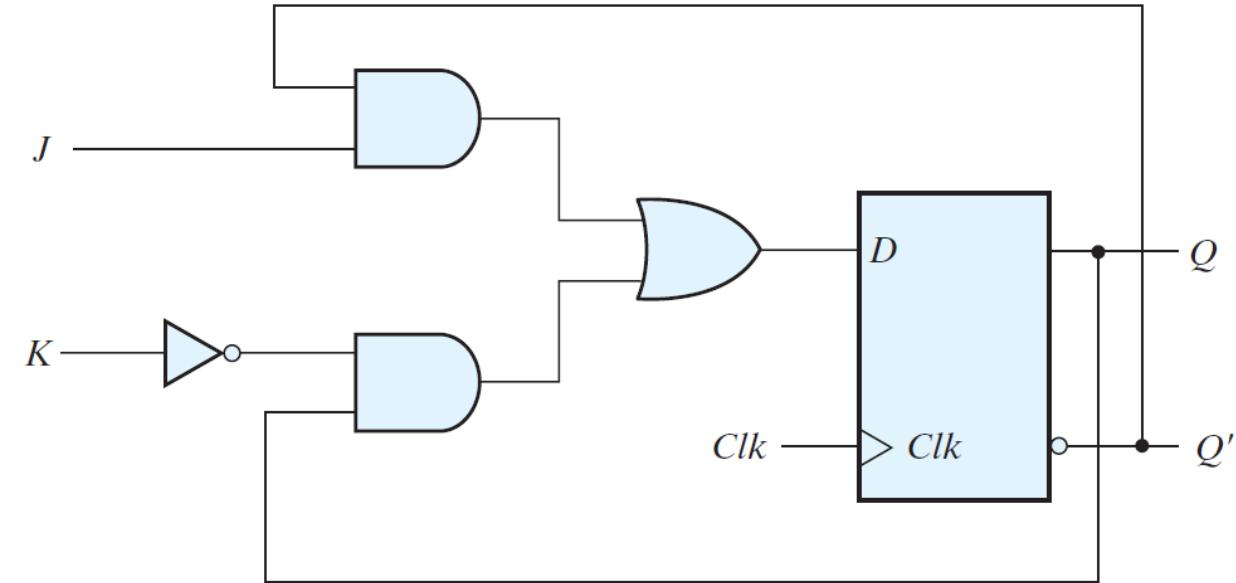


Graphic symbol of JK flip-flop

# JK Flip Flop

$$D = JQ' + K'Q$$

- When both  $J = K = 0$ ,  $D = Q$ , the clock edge leaves the output unchanged
- When  $J = 0$  and  $K = 1$ ,  $D = 0$ , so the next clock edge resets the output to 0
- When  $J = 1$  and  $K = 0$ ,  $D = Q' + Q = 1$ , so the next clock edge sets the output to 1
- When both  $J = K = 1$  and  $D = Q'$ , the next clock edge complements the output
- Because of their versatility, JK flip-flops are called *universal flip-flops*



(a) Circuit diagram

---

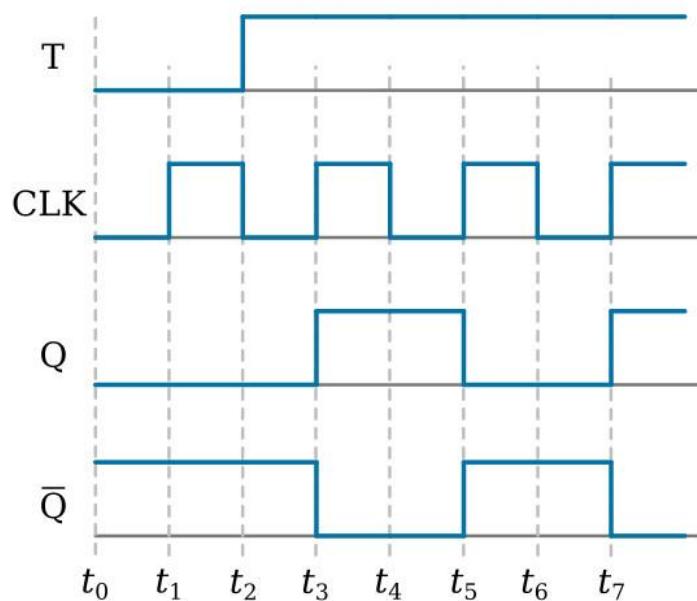
## JK Flip-Flop

---

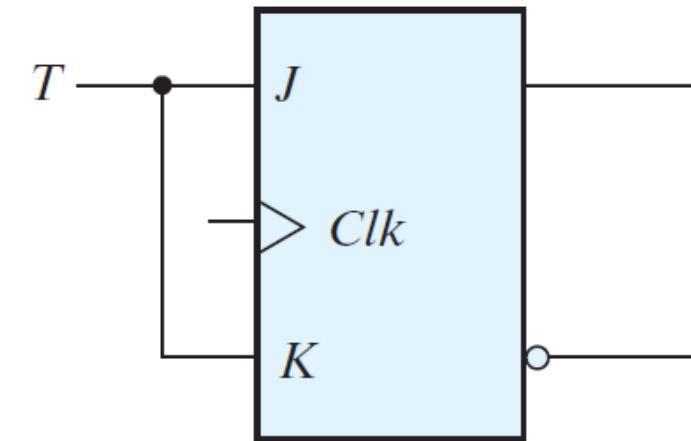
<b><i>J</i></b>	<b><i>K</i></b>	<b><i>Q(t + 1)</i></b>	
0	0	$Q(t)$	No change
0	1	0	Reset
1	0	1	Set
1	1	$Q'(t)$	Complement

# T Flip Flop

- The  $T$  (toggle) flip-flop is a complementing flip-flop and can be obtained from a  $JK$  flip-flop when inputs  $J$  and  $K$  are tied together
- When  $T = 0$  ( $J = K = 0$ ), a clock edge does not change the output
- When  $T = 1$  ( $J = K = 1$ ), a clock edge complements the output



The complementing flip-flop is useful for designing binary counters



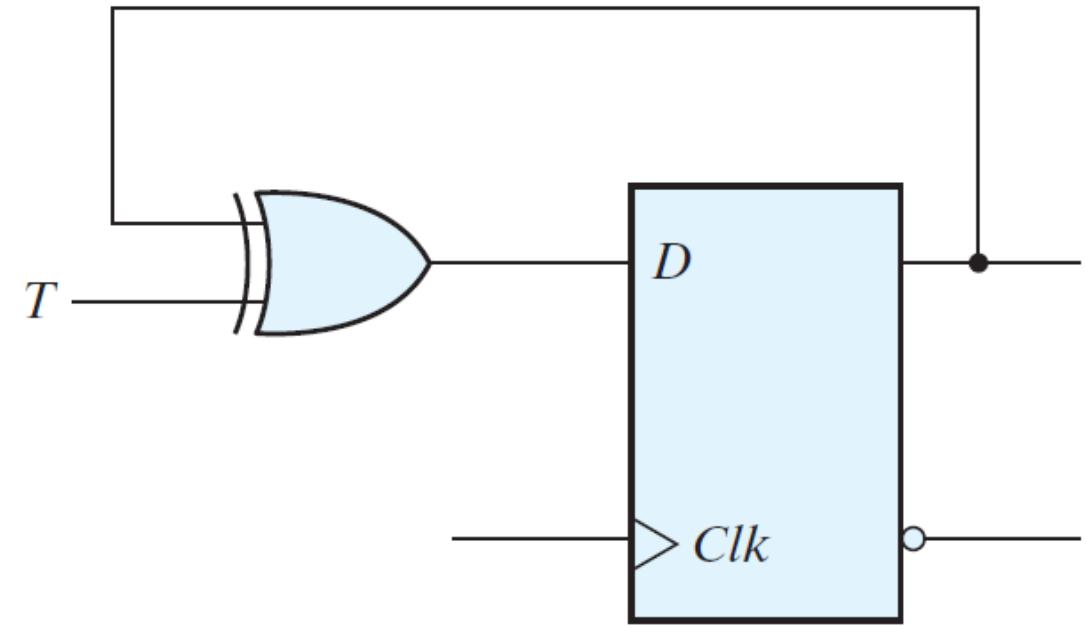
(a) From  $JK$  flip-flop

## **T Flip-Flop**

$T$	$Q(t + 1)$	
0	$Q(t)$	No change
1	$Q'(t)$	Complement

# T Flip Flop

- The  $T$  flip-flop can also be constructed using a  $D$  flip-flop
- The expression for the  $D$  input is:  
$$D = T'Q + TQ'$$
- When  $T = 0$ ,  $D = Q$  and there is no change in the output
- When  $T = 1$ ,  $D = Q'$  and the output complements
- The graphic symbol for this flip-flop has a  $T$  symbol in the input



**T Flip-Flop**

$T$	$Q(t + 1)$	
0	$Q(t)$	No change
1	$Q'(t)$	Complement

# Flip Flop characteristic tables

## JK Flip-Flop

J	K	$Q(t + 1)$	
0	0	$Q(t)$	No change
0	1	0	Reset
1	0	1	Set
1	1	$Q'(t)$	Complement

## D Flip-Flop

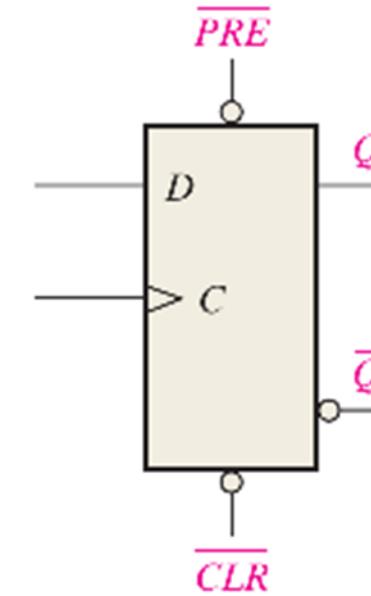
D	$Q(t + 1)$	
0	0	Reset
1	1	Set

## T Flip-Flop

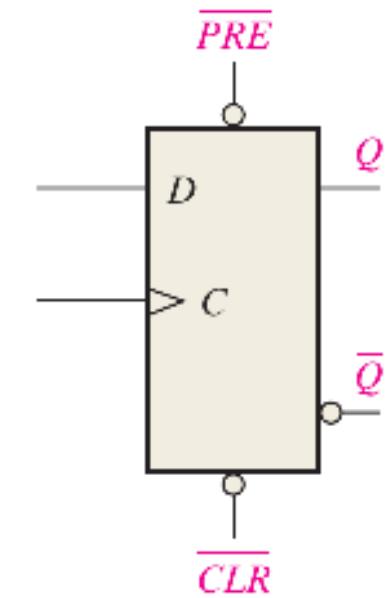
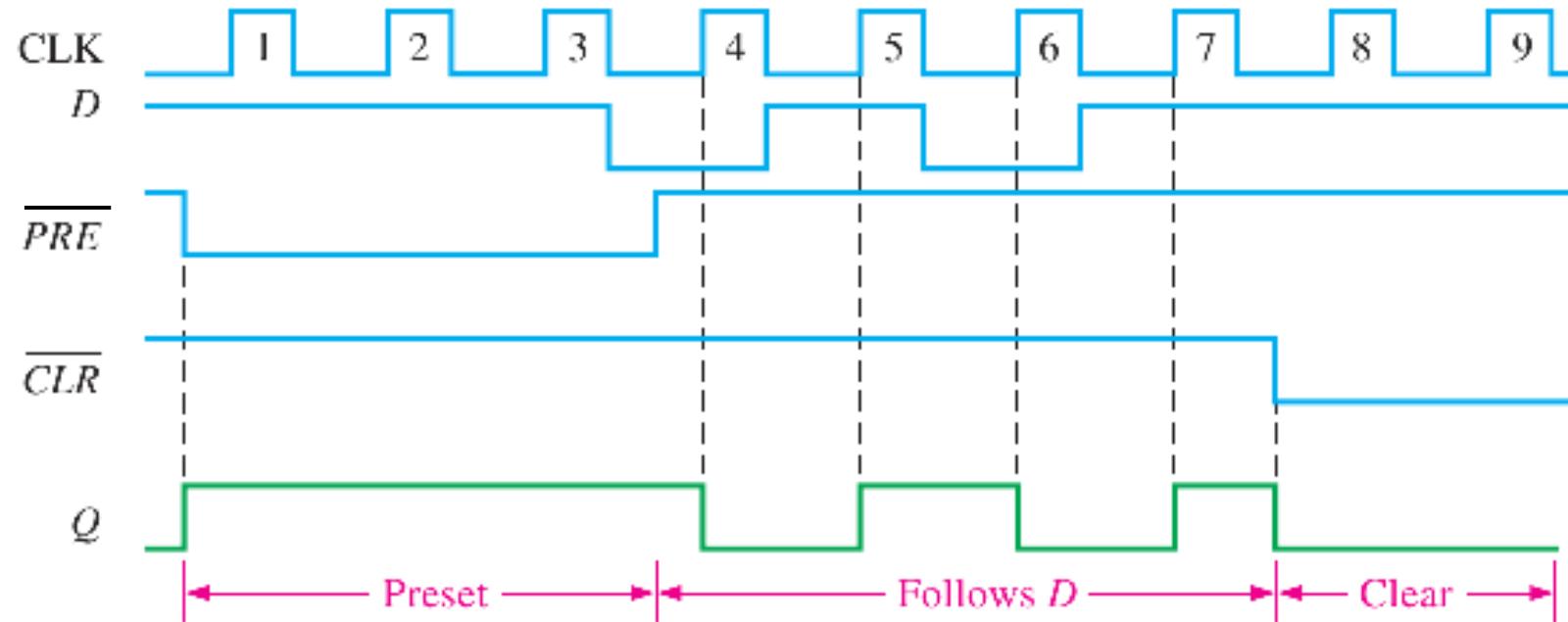
T	$Q(t + 1)$	
0	$Q(t)$	No change
1	$Q'(t)$	Complement

# Asynchronous inputs

- Some flip-flops have asynchronous inputs that are used to force the flip-flop to a particular state independently of the clock
- The input that sets the flip-flop to 1 is called *preset* or *direct set*
- The input that clears the flip-flop to 0 is called *clear* or *direct reset*
- When power is turned on in a digital system, the state of the flip-flops is unknown
- The direct inputs are useful for bringing all flip-flops in the system to a known starting state prior to the clocked operation.
- When the reset input is 0, it forces output  $Q'$  to stay at 1, which, in turn, clears output  $Q$  to 0, thus resetting the flip-flop



# Asynchronous inputs

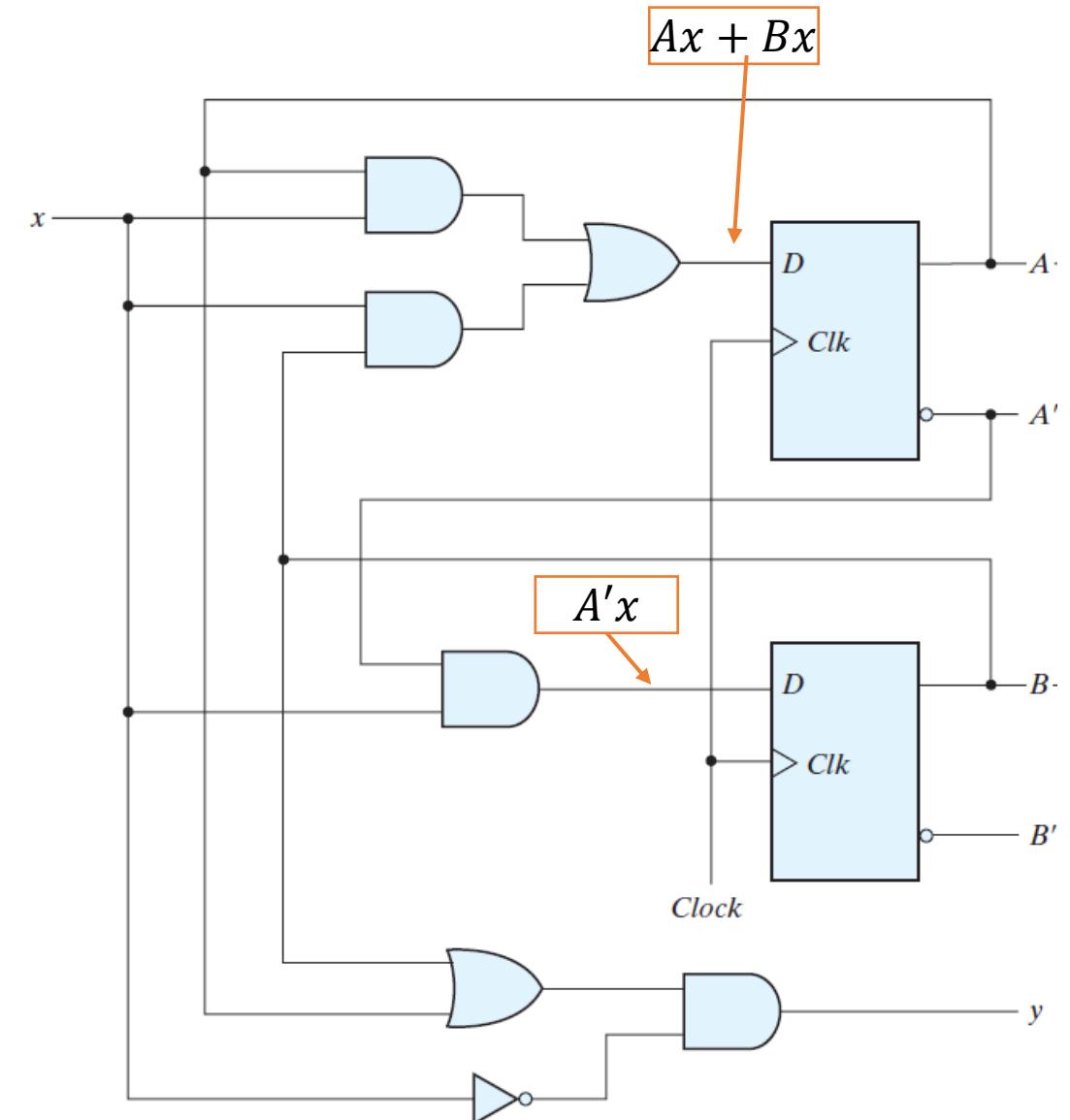


# Analysis of sequential circuits

- Analysis describes what a given circuit will do under certain operating conditions
- *The behavior of a clocked sequential circuit is determined from the inputs, the outputs, and the state of its flip-flops*
- *The outputs and the next state are both a function of the inputs and the present state*
- The analysis of a sequential circuit consists of obtaining a table or a diagram for the time sequence of inputs, outputs, and internal states
- It is also possible to write Boolean expressions that describe the behavior of the sequential circuit
- These expressions must include the necessary time sequence, either directly or indirectly

# Analysis of sequential circuits

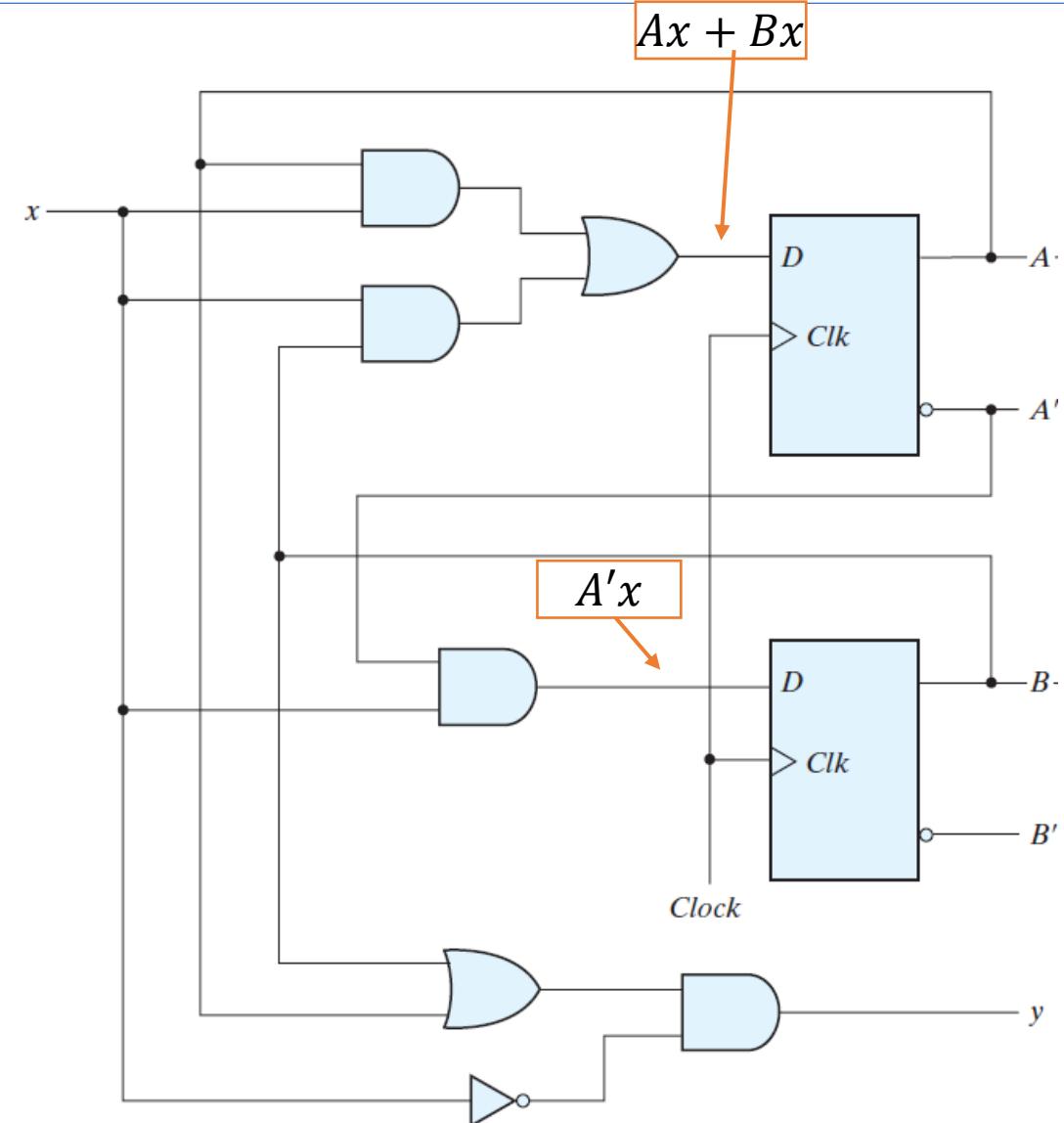
- Consider this sequential circuit
- It consists of two  $D$  flip-flops  $A$  and  $B$ , an input  $x$  and an output  $y$
- Since the  $D$  input of a flip-flop determines the value of the next state (i.e., the state reached after the clock transition), it is possible to write a set of state equations for the circuit as:  
$$A(t + 1) = A(t)x(t) + B(t)x(t)$$
  
$$B(t + 1) = A'(t)x(t)$$



# State equations

- A **state equation** is an algebraic expression that specifies the condition for a flip-flop state transition
  - The left side of the equation, with  $(t + 1)$ , denotes the next state of the flip-flop one clock edge later
  - The right side of the equation is a Boolean expression that specifies the present state and input conditions
- Since all the variables in the Boolean expressions are a function of the present state, we can omit the designation ( $t$ ) after each variable for convenience and can express the state equations in the more compact form

$$A(t + 1) = Ax + Bx$$
$$B(t + 1) = A'x$$



# State equations

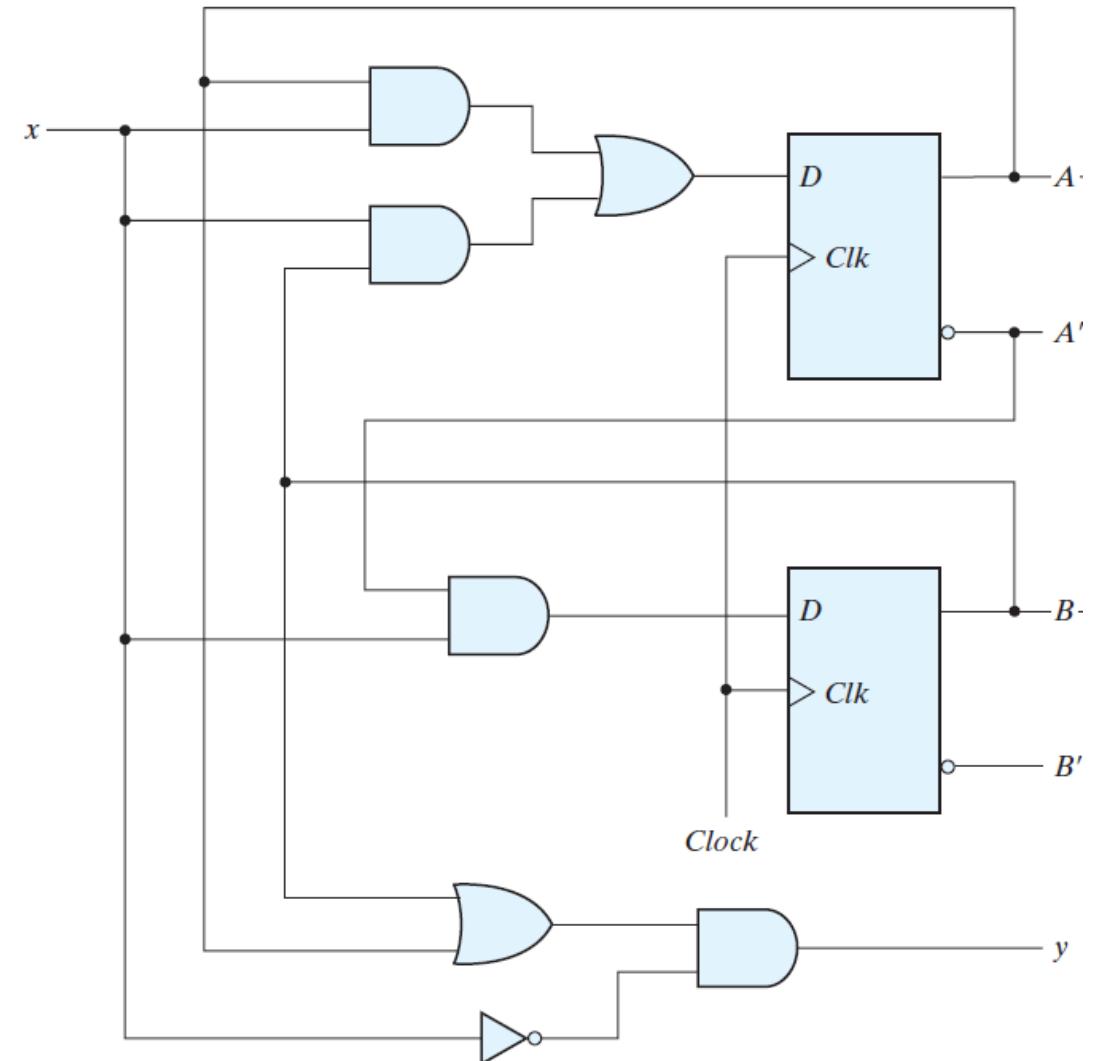
- The Boolean expressions for the state equations can be derived directly from the combinational circuit part of the sequential circuit, since the  $D$  values of the combinational circuit determine the next state

- Similarly, the present-state value of the output can be expressed algebraically as

$$y(t) = [A(t) + B(t)]x'(t)$$

- By removing the symbol  $(t)$  for the present state, we obtain the output Boolean equation:

$$y = (A + B)x'$$



# State tables

- Similar to truth tables, the derivation of a **state table** requires listing all possible binary combinations of present states and inputs
- In this case, we have eight binary combinations from 000 to 111
- The next-state values are then determined from the logic diagram or from the state equations
- The next state of flip-flops must satisfy the state equations:

$$A(t + 1) = Ax + Bx$$
$$B(t + 1) = A'x$$

Output is derived from:

$$y = (A + B)x'$$

Present State		Input
A	B	x
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

# State tables

- In general, a sequential circuit with  $m$  flipflops and  $n$  inputs needs  $2^{m+n}$  rows in the state table
- The binary numbers from 0 through  $2^{m+n} - 1$  are listed under the present-state and input columns
- The next-state section has  $m$  columns, one for each flip-flop
- The binary values for the next state are derived directly from the state equations
- The output section has as many columns as there are output variables
- Its binary value is derived from the circuit or from the Boolean function in the same manner as in a truth table

Present State		Input	Next State		Output
A	B	x	A	B	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

# State tables

- It is sometimes convenient to express the state table in a slightly different form having only three sections: present state, next state, and output
- The input conditions are enumerated under the next-state and output sections

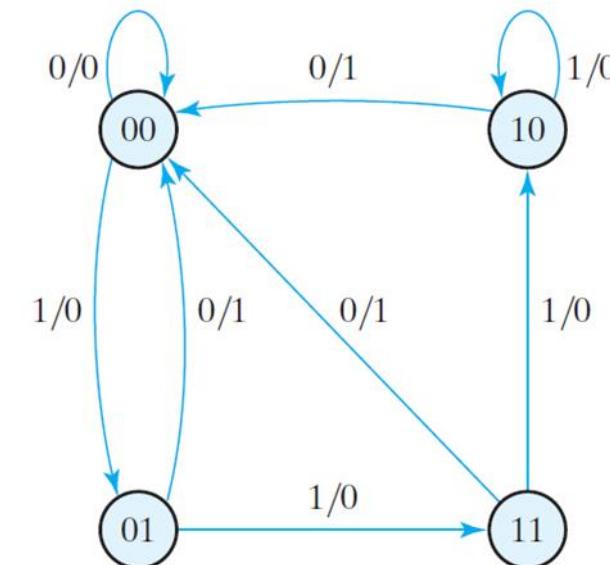
Present State	Next State				Output	
	x = 0		x = 1		x = 0	x = 1
A	B	A	B	y	y	
0	0	0	0	0	0	
0	1	0	0	1	0	
1	0	0	0	1	0	
1	1	0	0	1	0	

Present State		Input <i>x</i>	Next State		Output <i>y</i>
<i>A</i>	<i>B</i>		<i>A</i>	<i>B</i>	
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	1	0
1	1	0	0	0	1
1	1	1	1	0	0

# State diagram

- The information available in a state table can be represented graphically in the form of a state diagram
- In this type of diagram, a state is represented by a circle, and the (clock-triggered) transitions between states are indicated by directed lines connecting the circles
- The binary number inside each circle identifies the state of the flip-flops
- The directed lines are labeled with two binary numbers separated by a slash
- The input value during the present state is labeled first, and the number after the slash gives the output during the present state with the given input

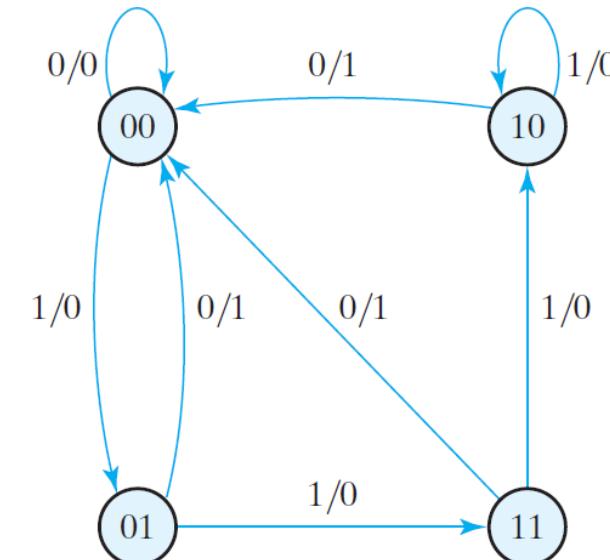
Present State		Next State				Output	
		$x = 0$	$x = 1$	$x = 0$	$x = 1$		
$A$	$B$	$A$	$B$	$A$	$B$	$y$	$y$
0	0	0	0	0	1	0	0
0	1	0	0	1	1	1	0
1	0	0	0	1	0	1	0
1	1	0	0	1	0	1	0



# State diagram

- For example, the directed line from state 00 to 01 is labeled 1/0, meaning that when the sequential circuit is in the present state 00 and the input is 1, the output is 0
- After the next clock cycle, the circuit goes to the next state, 01
- If the input changes to 0, then the output becomes 1, but if the input remains at 1, the output stays at 0
- This information is obtained from the state diagram along the two directed lines emanating from the circle with state 01
- A directed line connecting a circle with itself indicates that no change of state occurs

Present State		Next State				Output	
		$x = 0$	$x = 1$	$x = 0$	$x = 1$		
$A$	$B$	$A$	$B$	$A$	$B$	$y$	$y$
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



# Lecture 19 – Sequential circuits 4

Chapter 5

# Recap- Flip Flop characteristic tables

## JK Flip-Flop

J	K	$Q(t + 1)$	
0	0	$Q(t)$	No change
0	1	0	Reset
1	0	1	Set
1	1	$Q'(t)$	Complement

## D Flip-Flop

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

## T Flip-Flop

T	$Q(t + 1)$	
0	$Q(t)$	No change
1	$Q'(t)$	Complement

# Recap - Analysis of sequential circuits

- Analysis describes what a given circuit will do under certain operating conditions
  - *The behavior of a clocked sequential circuit is determined from the inputs, the outputs, and the state of its flip-flops*
  - *The outputs and the next state are both a function of the inputs and the present state*
  - The analysis of a sequential circuit consists of obtaining a table or a diagram for the time sequence of inputs, outputs, and internal states
  - It is also possible to write Boolean expressions that describe the behavior of the sequential circuit
  - These expressions must include the necessary time sequence, either directly or indirectly

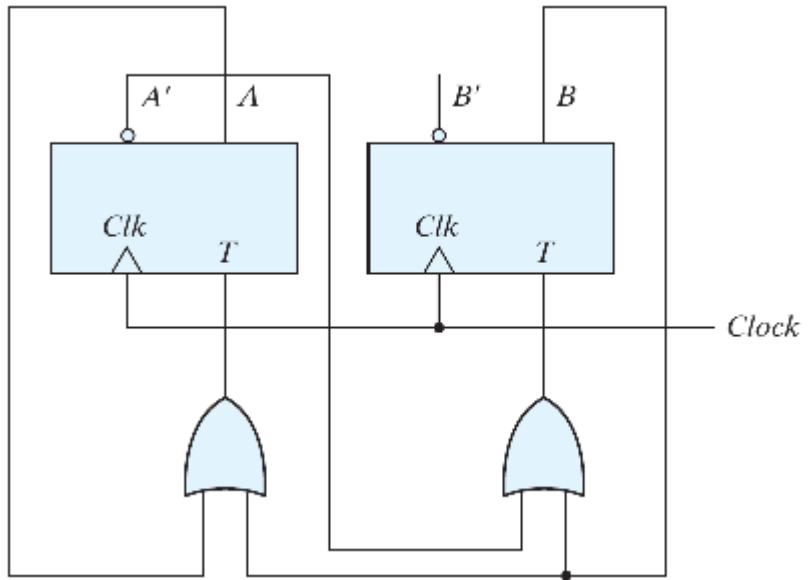
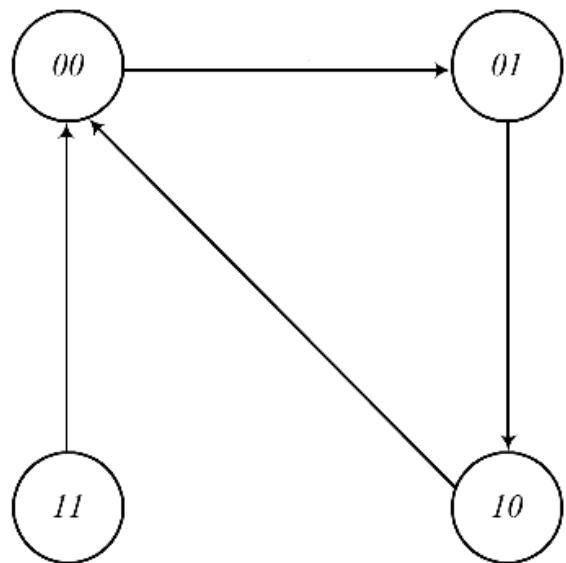
# Sequential circuits - Analysis

$$T_A = A + B$$

$$T_B = A' + B$$

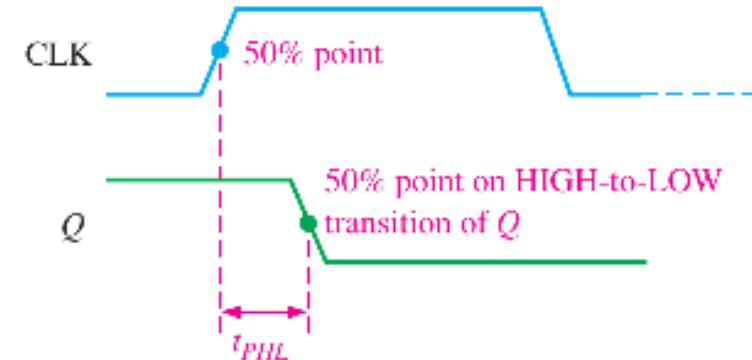
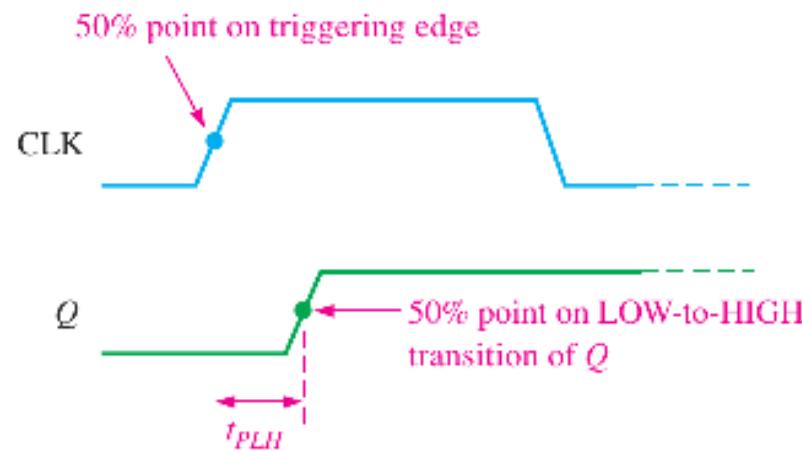
## State Table:

Present State		Next State		FF Inputs	
A	B	A	B	$T_A$	$T_B$
0	0	0	1	0	1
0	1	1	0	1	1
1	0	0	0	1	0
1	1	0	0	1	1



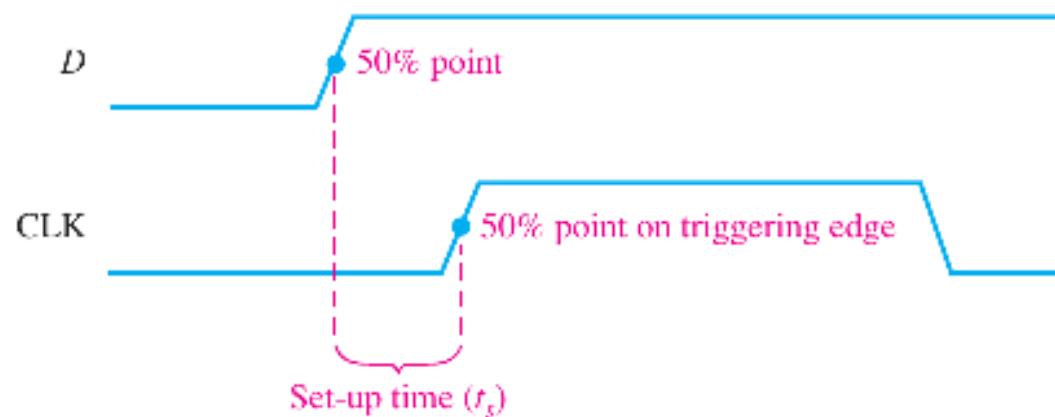
# Flip-Flop Operating Characteristics

Propagation delay is the interval of time required after an input signal has been applied for the resulting output change to occur



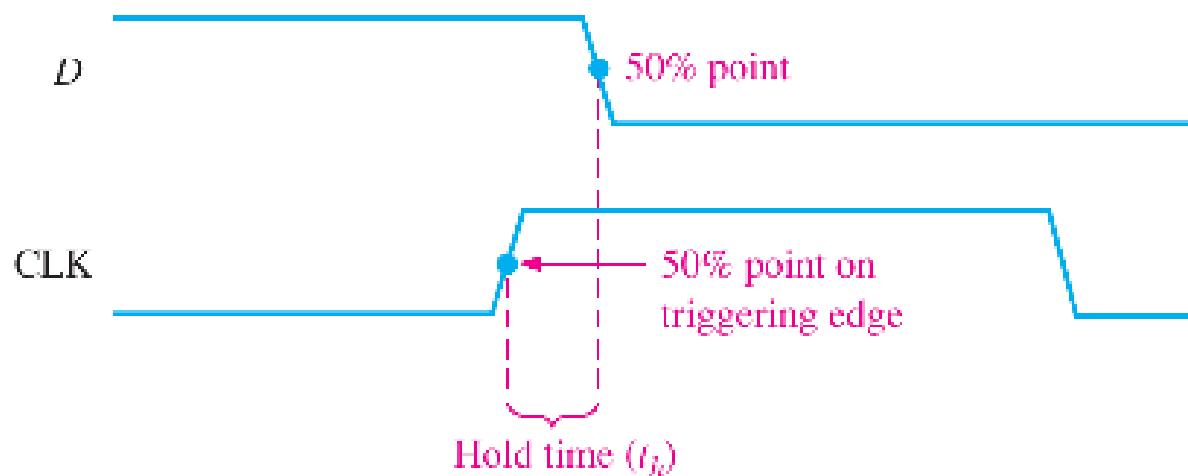
# Flip-Flop Operating Characteristics

**Set-up time ( $t_s$ ):** The logic level must be present on the input for a time equal to or greater than  $t_s$  before the triggering edge of the clock pulse for reliable data entry.



# Flip-Flop Operating Characteristics

The **hold time ( $t_h$ )**: is the minimum interval required for the logic levels to remain on the inputs after the triggering edge of the clock pulse in order for the levels to be reliably clocked into the flip-flop



# Sequential circuits - Design procedure

- The procedure for designing synchronous sequential circuits can be summarized by a list of recommended steps:
  1. Derive a state diagram for the circuit
  2. Assign binary values to the states
  3. Obtain the binary-coded state table
  4. Derive the simplified flip-flop input equations and output equations
  5. Draw the logic diagram

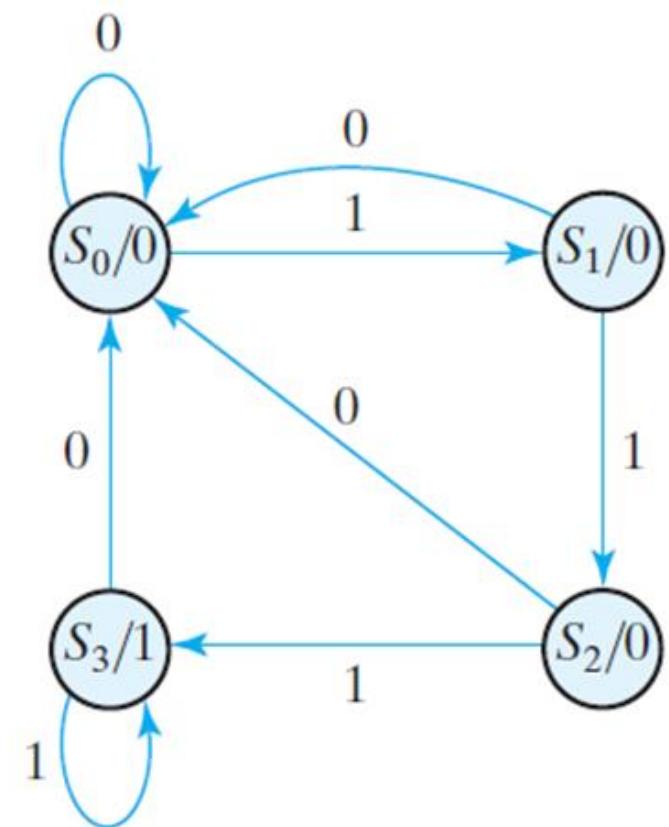


The sequence of three  
detector

# Sequence-of-three detector

- Suppose we wish to design a circuit that detects a sequence of three or more consecutive 1's in a string of bits coming through an input line (i.e., the input is a *serial bit stream* )
- We start with state  $S_0$ , the reset state
- If the input is 0, the circuit stays in  $S_0$ , but if the input is 1, it goes to state  $S_1$  to indicate that a 1 was detected
- If the next input is 1, the change is to state  $S_2$  to indicate the arrival of two consecutive 1's, but if the input is 0, the state goes back to  $S_0$

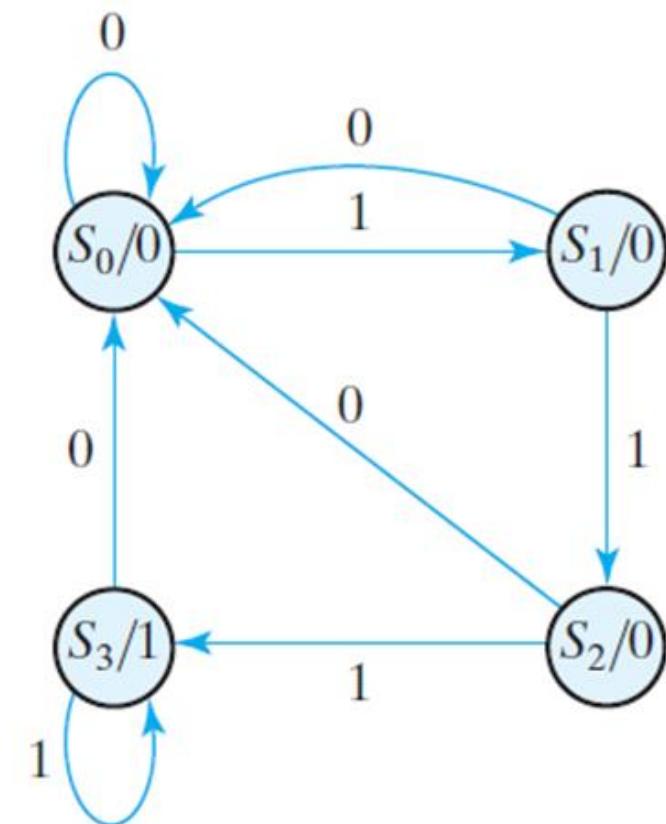
... 0111001111001 ...



# Sequence of three

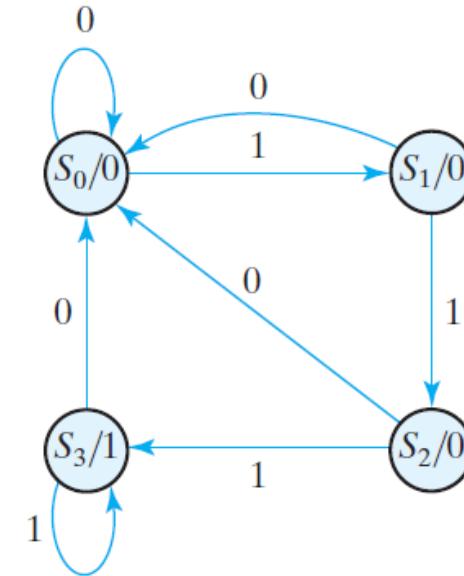
- The third consecutive 1 sends the circuit to state  $S_3$
- If more 1's are detected, the circuit stays in  $S_3$
- Thus, the circuit stays in  $S_3$  as long as there are three or more consecutive 1's received
- The output is 1 when the circuit is in state  $S_3$  and is 0 otherwise

... 0111001111001 ...



# Sequence of three

- To design the circuit, we need to assign binary codes to the states and list the **state table**
- The table is derived from the state diagram with a sequential binary assignment
- We choose two  $D$  flip-flops to represent the four states, and we label their outputs **A** and **B**
- There is one input **x** and one output **y**



Present State		Input <i>x</i>	Next State		Output <i>y</i>
<i>A</i>	<i>B</i>		<i>A</i>	<i>B</i>	
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	0	0
0	1	1	1	0	0
1	0	0	0	0	0
1	0	1	1	1	0
1	1	0	0	0	1
1	1	1	1	1	1

# Sequence of three

- The flip-flop input equations can be obtained directly from the next-state columns of  $A$  and  $B$  and expressed in sum-of-minterms form as:

$$A(t + 1) = D_A(A, B, x) = \sum(3, 5, 7)$$

$$B(t + 1) = D_B(A, B, x) = \sum(1, 5, 7)$$

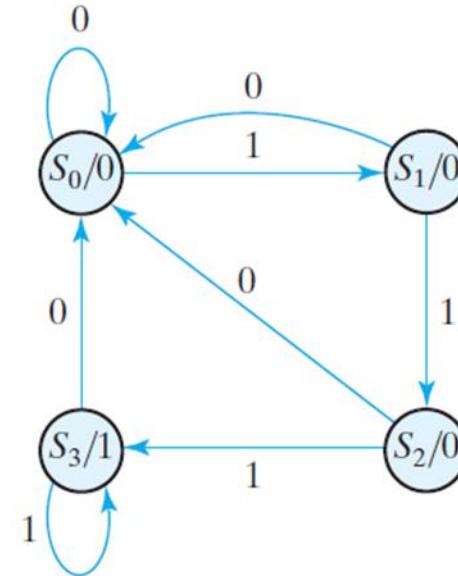
$$y(A, B, x) = \sum(6, 7)$$

- Using K-maps, we can find the expressions for  $D_A$ ,  $D_B$  and  $y$  as:

$$D_A = Ax + Bx$$

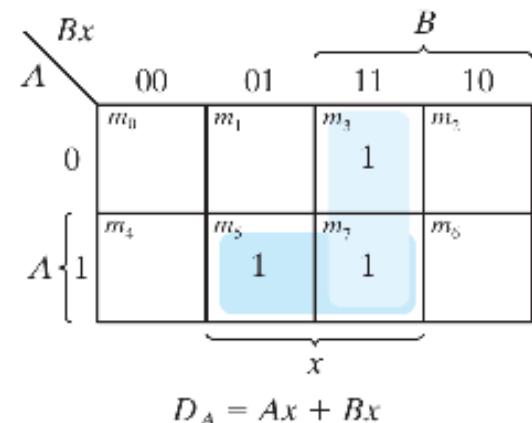
$$D_B = Ax + B'x$$

$$y = AB$$

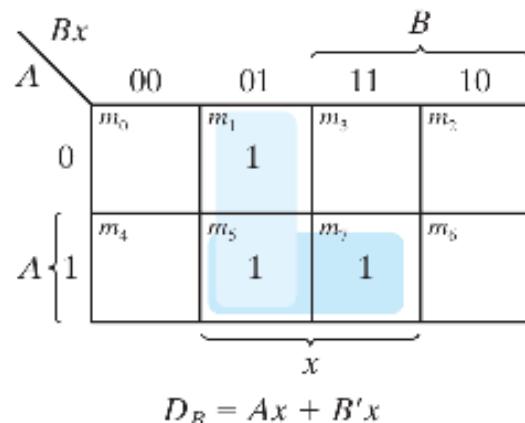


Present State		Input <i>x</i>	Next State		Output <i>y</i>
<i>A</i>	<i>B</i>		<i>A</i>	<i>B</i>	
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	0	0
0	1	1	1	0	0
1	0	0	0	0	0
1	0	1	1	1	0
1	1	0	0	0	1
1	1	1	1	1	1

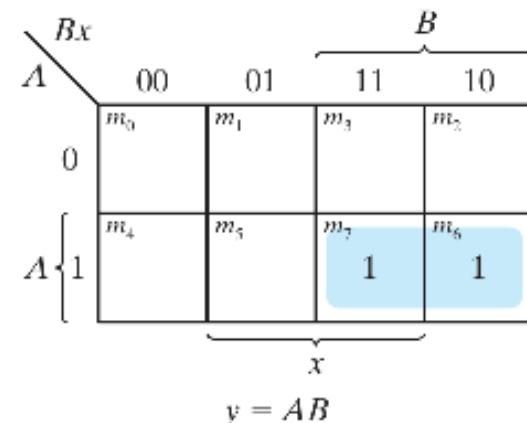
# Sequence of three



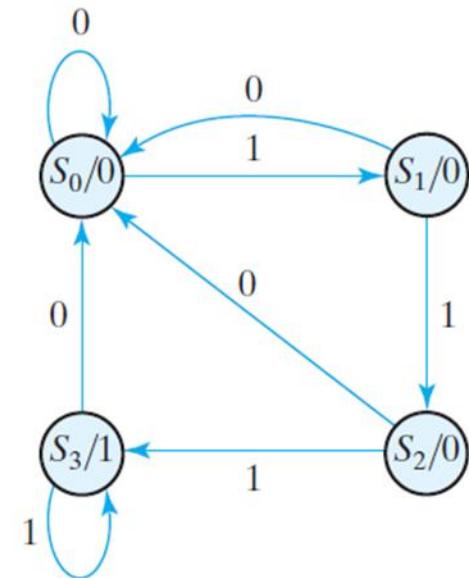
$$D_A = Ax + Bx$$



$$D_B = Ax + B'x$$



$$y = AB$$



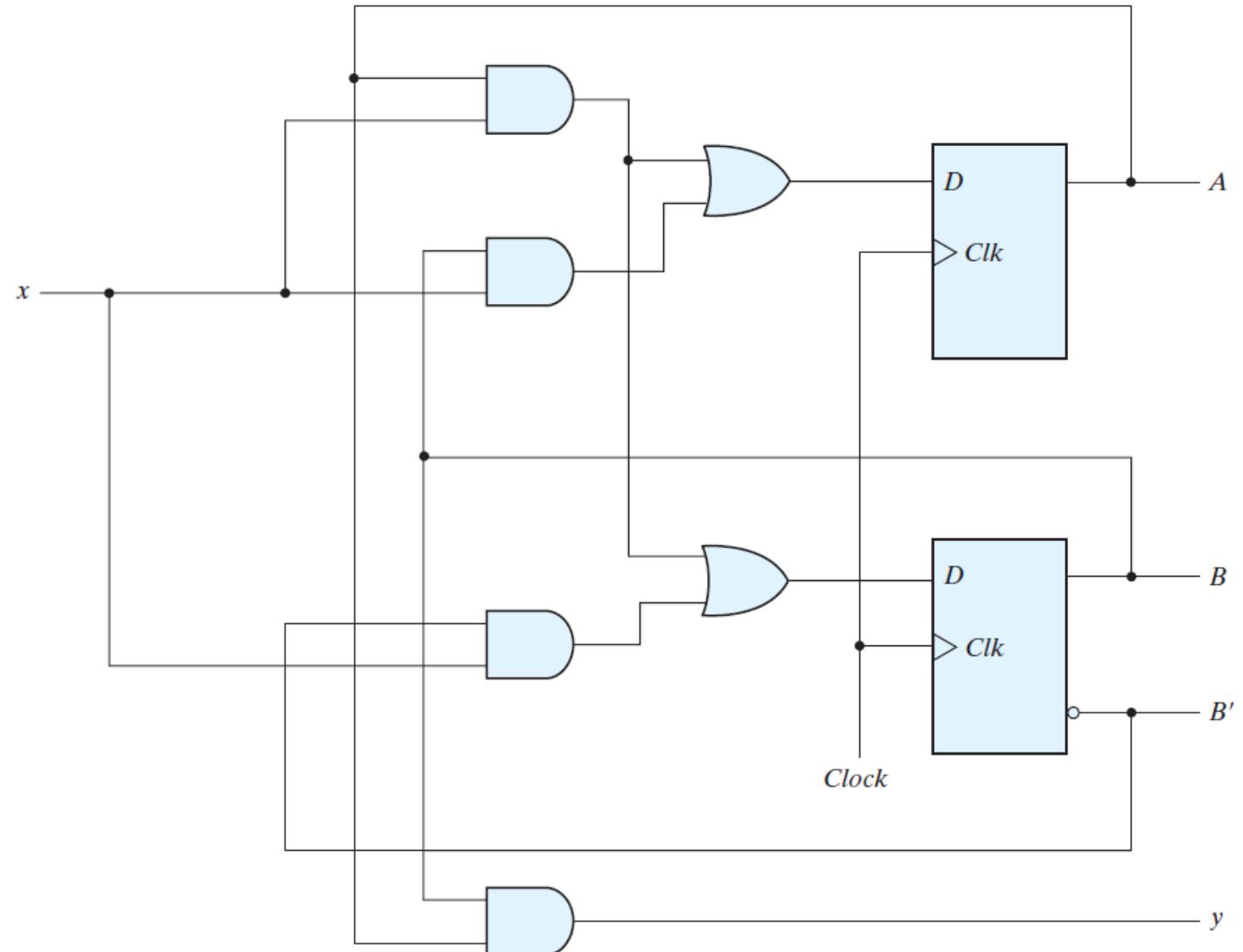
Present State		Input	Next State		Output
A	B		A	B	
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	0	0
0	1	1	1	0	0
1	0	0	0	0	0
1	0	1	1	1	0
1	1	0	0	0	1
1	1	1	1	1	1

# Sequence of three

$$D_A = Ax + Bx$$

$$D_B = Ax + B'x$$

$$y = AB$$



# Lecture 20 – Registers and Counters 1

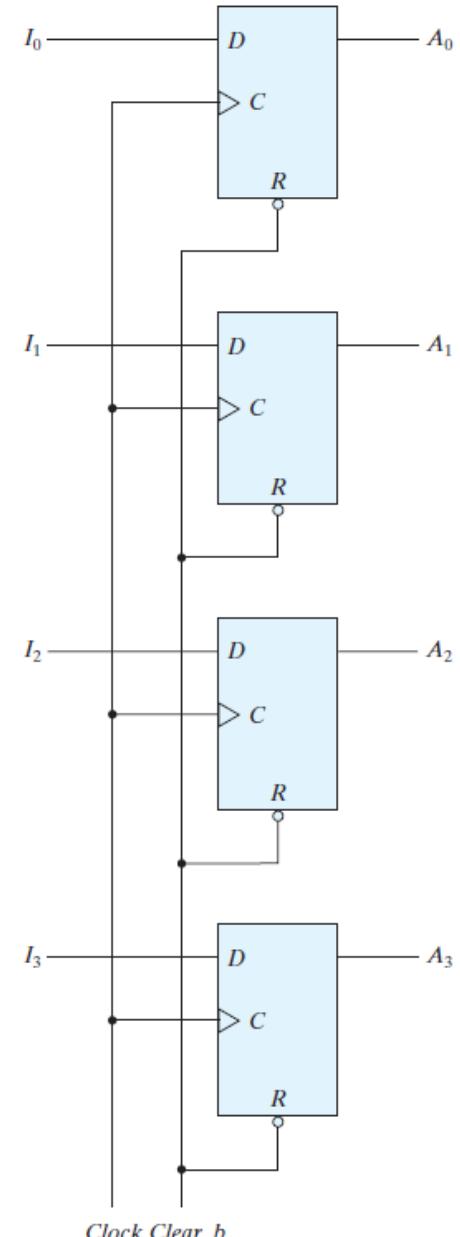
Chapter 6

# Registers and counters

- A *register* is a group of flip-flops, each one of which shares a common clock and is capable of storing one bit of information
- An  $n$ -bit register consists of a group of  $n$  flip-flops capable of storing  $n$  bits of binary information
- A *counter* is essentially a register that goes through a predetermined sequence of binary states
- The counter circuit is designed in such a way as to produce the prescribed sequence of states
- Although counters are a special type of register, it is common to differentiate them by giving them a different name

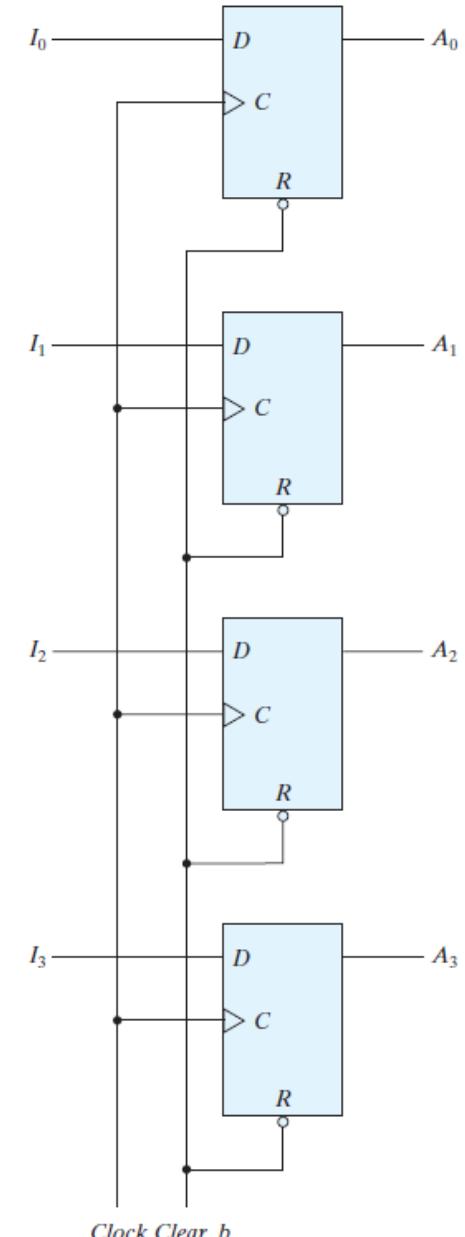
# Registers

- Consider a register constructed with four  $D$ -type flip-flops to form a four-bit data storage register
- The common clock input triggers all flip-flops on the positive edge of each pulse, and the binary data available at the four inputs are transferred into the register
- The value of  $(I_3, I_2, I_1, I_0)$  immediately before the clock edge determines the value of  $(A_3, A_2, A_1, A_0)$  after the clock edge



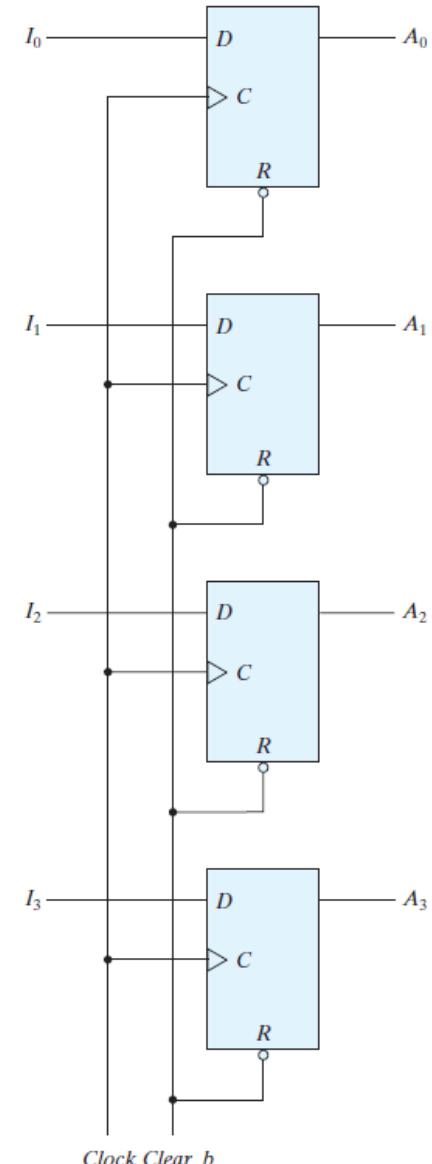
# Registers

- The four outputs can be sampled at any time to obtain the binary information stored in the register
- The input *Clear\_b* goes to the active-low *R* (reset) input of all four flip-flops
- When this input goes to 0, all flip-flops are reset asynchronously
- The *Clear\_b* input is useful for clearing the register to all 0's prior to its clocked operation
- The *R* inputs must be maintained at logic 1 (i.e., de-asserted) during normal clocked operation



# Registers with load input

- Synchronous digital systems have a master clock generator that supplies a continuous train of clock pulses
- The pulses are applied to all flip-flops and registers in the system
- The master clock acts like a drum that supplies a constant beat to all parts of the system (like the heart-beat of the processor)
- However, we might not be interested in changing data in the register every time, in some cases, we may want to keep the data unchanged
- *A separate control signal must be used to decide which register operation will execute at each clock pulse*
- *The transfer of new information into a register is referred to as loading or updating the register*

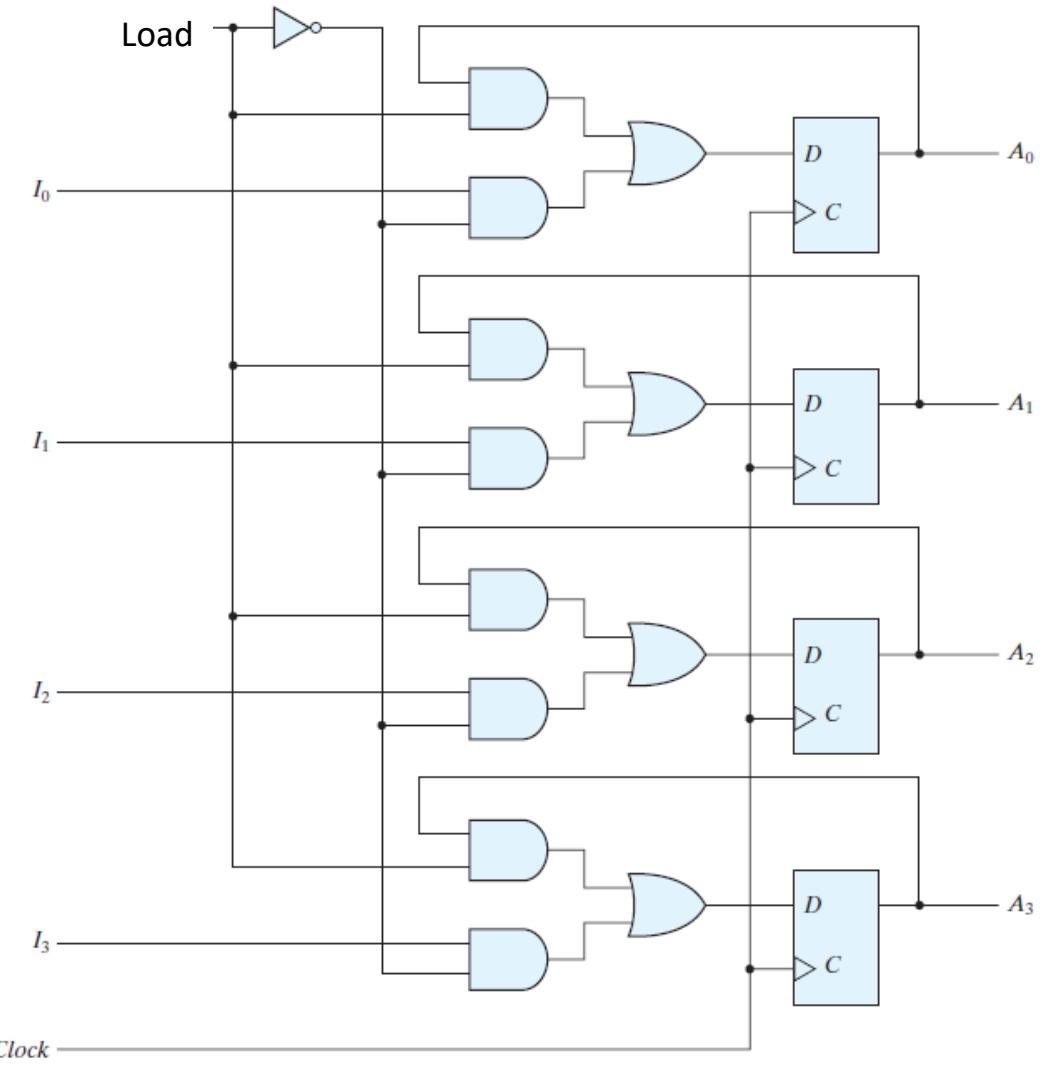


# Registers with load input

- In this configuration, if the contents of the register must be left unchanged:
  - the inputs must be held constant,
  - or the clock must be inhibited from the circuit
- However, inserting gates into the clock path is ill advised because it means that logic is performed with clock pulses
- The insertion of logic gates produces uneven propagation delays between the master clock and the inputs of flip-flops
- To fully synchronize the system, we must ensure that all clock pulses arrive at the same time anywhere in the system, so that all flip-flops trigger simultaneously
- For this reason, it is advisable to control the operation of the register with the  $D$  inputs, rather than controlling the clock in the  $C$  inputs of the flip-flops
- This creates the effect of a gated clock, but without affecting the clock path of the circuit

# Registers with load input

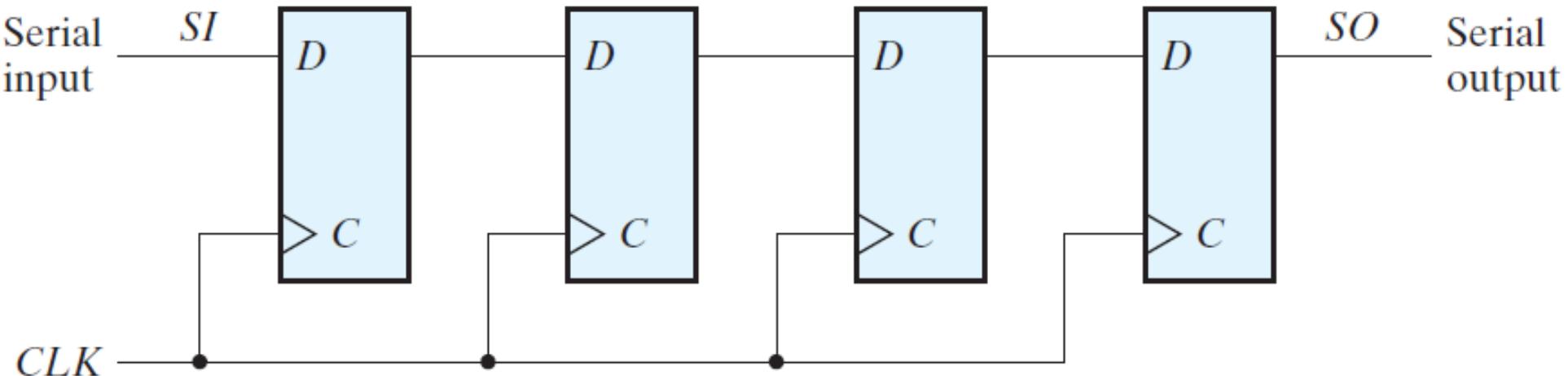
- The additional gates implement a two-channel mux whose output drives the input to the register with either the data bus or the output of the register
- When the load input is 0:** the data at the four external inputs are transferred into the register with the next positive edge of the clock
- When the load input is 1:** the outputs of the flip-flops are connected to their respective inputs
- Why is the feedback from flip-flop output to input necessary?
  - because a *D* flip-flop does not have a “no change” condition



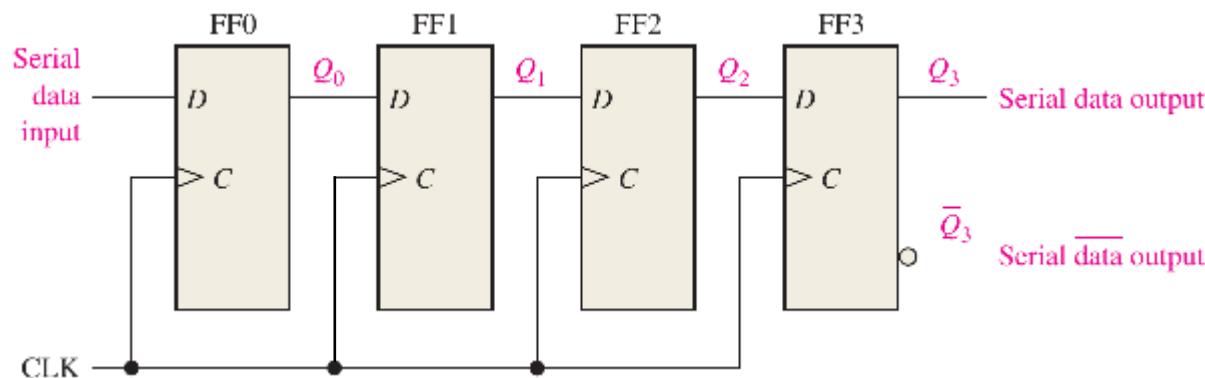
storage register with a load control input

# Shift register

- A register capable of shifting the binary information held in each cell to its neighboring cell, in a selected direction, is called a *shift register*
- The logical configuration of a shift register consists of a chain of flip-flops in cascade, with the output of one flip-flop connected to the input of the next flip-flop
- All flip-flops receive common clock pulses, which activate the shift of data from one stage to the next



# Shift register



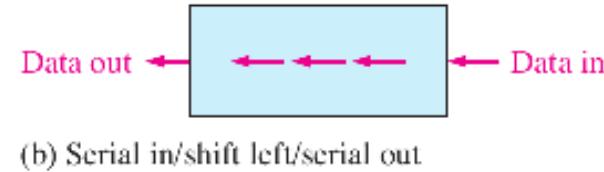
Shifting 1010 into the 4-bit register

CLK	FF0 ( $Q_0$ )	FF1 ( $Q_1$ )	FF2 ( $Q_2$ )	FF3 ( $Q_3$ )
Initial	0	0	0	0
1	0	0	0	0
2	1	0	0	0
3	0	1	0	0
4	1	0	1	0

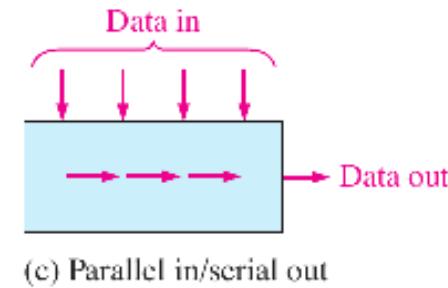
# Shift register



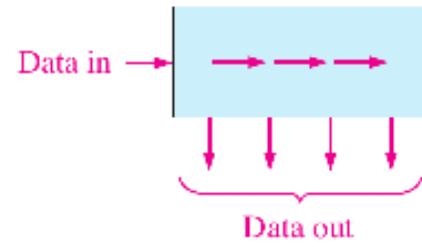
(a) Serial in/shift right/serial out



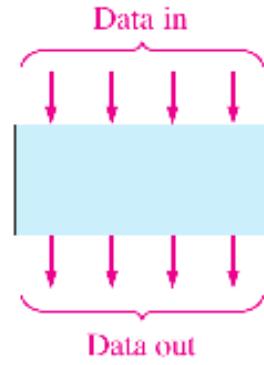
(b) Serial in/shift left/serial out



(c) Parallel in/serial out



(d) Serial in/parallel out



(e) Parallel in/parallel out



(f) Rotate right

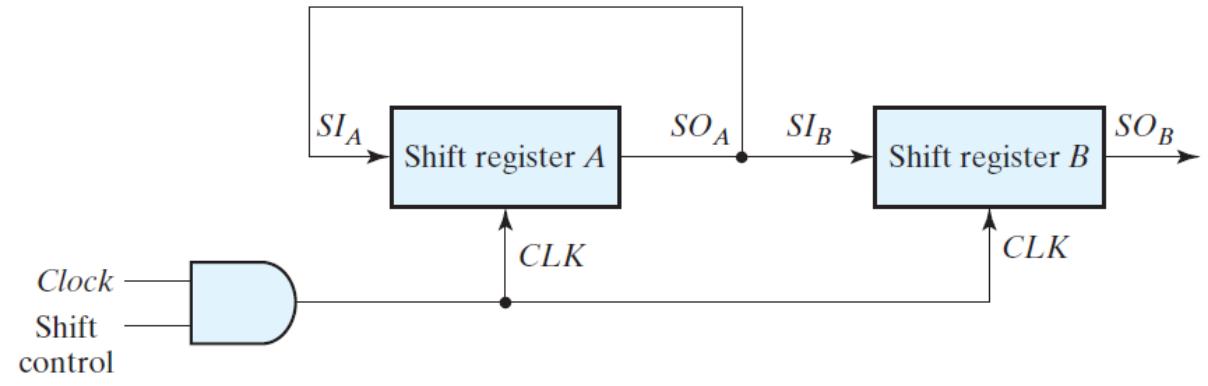


(g) Rotate left

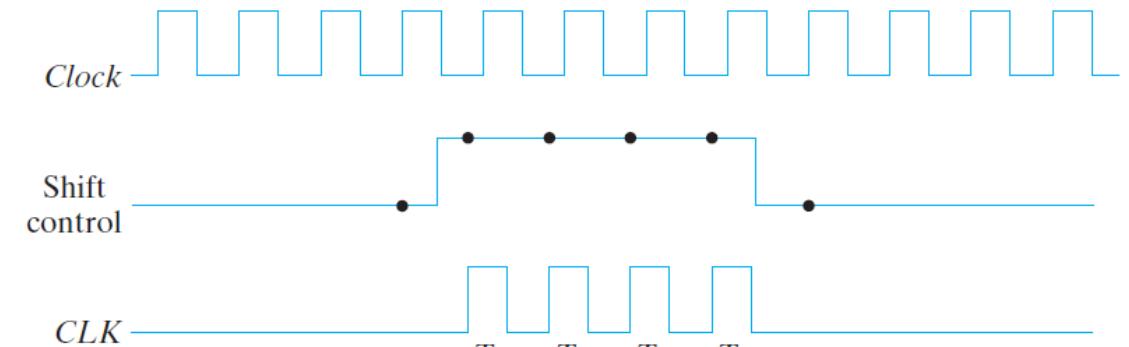
## Types of data movement in shift registers

# Serial transfer

- The datapath of a digital system is said to operate in *serial mode* when information is transferred and manipulated one bit at a time
- Information is transferred one bit at a time by shifting the bits out of the source register and into the destination register
- This type of transfer is in contrast to parallel transfer, whereby all the bits of the register are transferred at the same time
- The serial transfer of information from register A to register B is done with shift registers, as shown
- The serial output ( SO ) of register A is connected to the serial input ( SI ) of register B



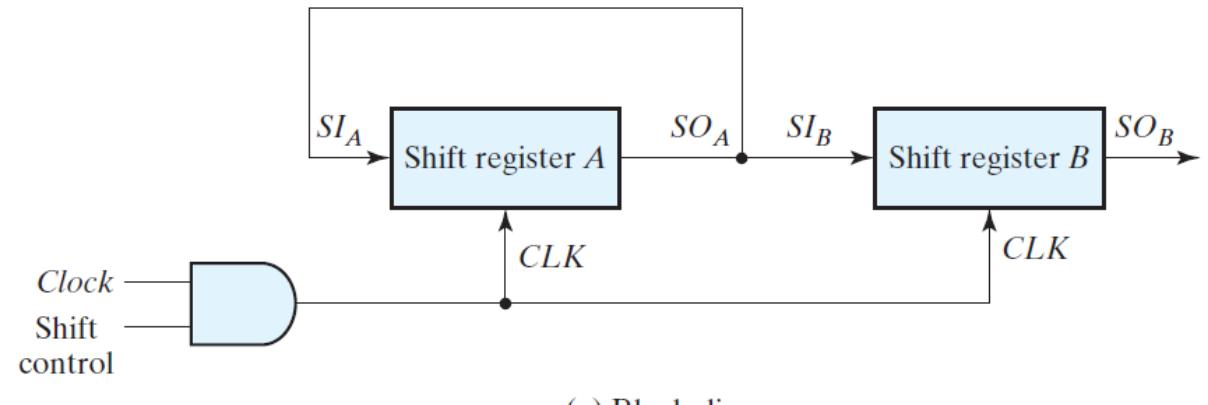
(a) Block diagram



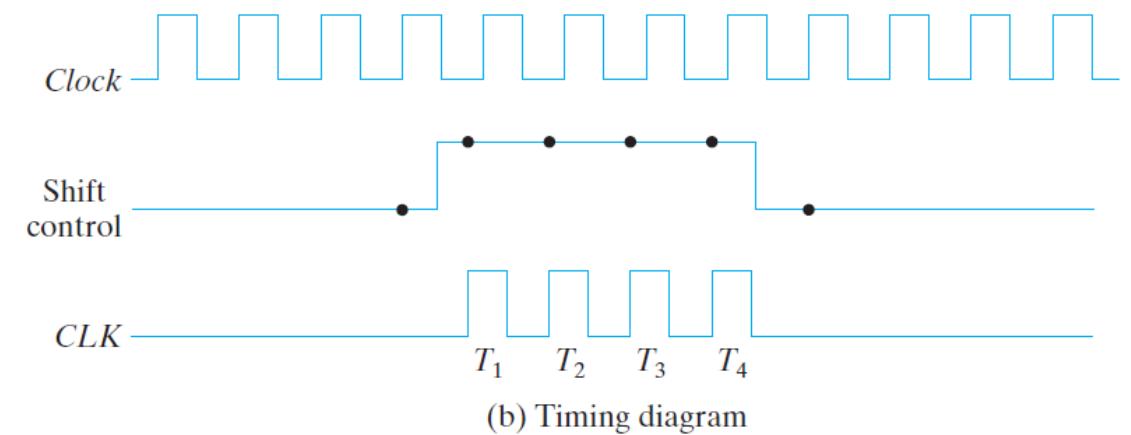
(b) Timing diagram

# Serial transfer

- To prevent the loss of information stored in the source register, the information in register A is made to circulate by connecting the serial output to its serial input
- The initial content of register B is shifted out through its serial output and is lost unless it is transferred to a third shift register
- The shift control input determines when and how many times the registers are shifted
- For simplicity here, this is done with an AND gate that allows clock pulses to pass into the  $CLK$  terminals only when the shift control is active



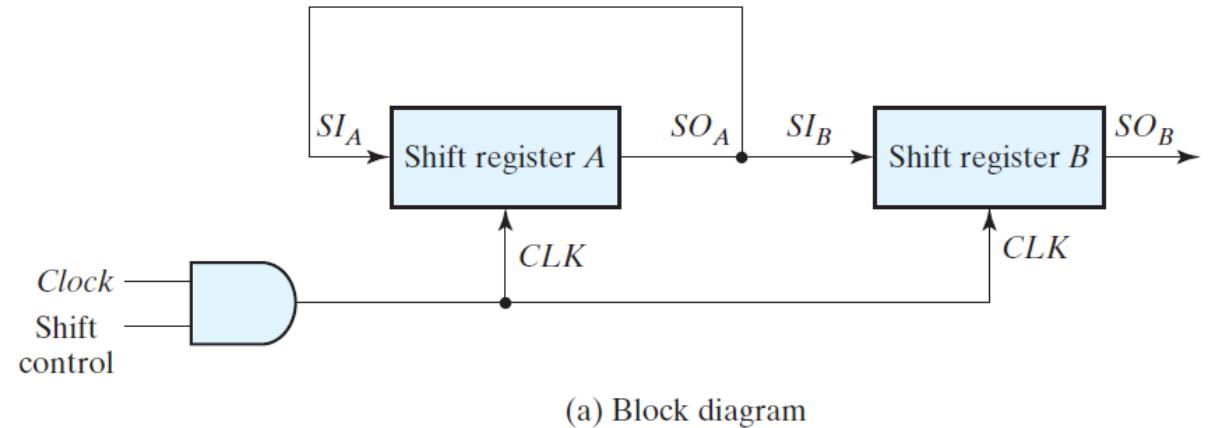
(a) Block diagram



(b) Timing diagram

# Serial transfer

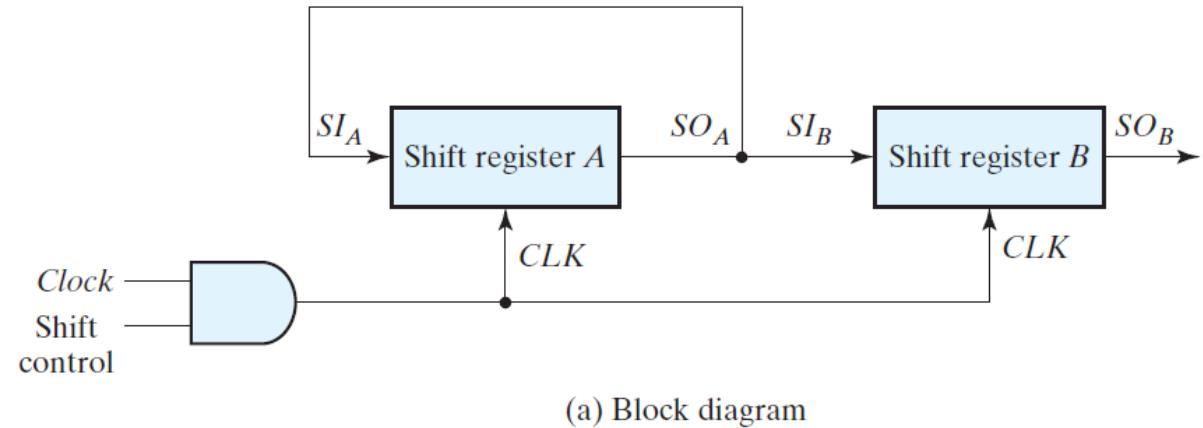
- Assume that the binary content of A before the shift is **1011** and that of B is **0010**
- The serial transfer from A to B occurs in four steps
- With the first pulse,  $T_1$ , the rightmost bit of A is shifted into the leftmost bit of B and is also circulated into the leftmost position of A
- At the same time, all bits of A and B are shifted one position to the right



(a) Block diagram

# Serial transfer

- The previous serial output from  $B$  in the rightmost position is lost, and its value changes from 0 to 1
- The next three pulses perform identical operations, shifting the bits of  $A$  into  $B$ , one at a time
- After the fourth shift, the shift control goes to 0, and registers  $A$  and  $B$  both have the value 1011
- Thus, the contents of  $A$  are copied into  $B$ , so that the contents of  $A$  remain unchanged i.e., the contents of  $A$  are restored to their original value

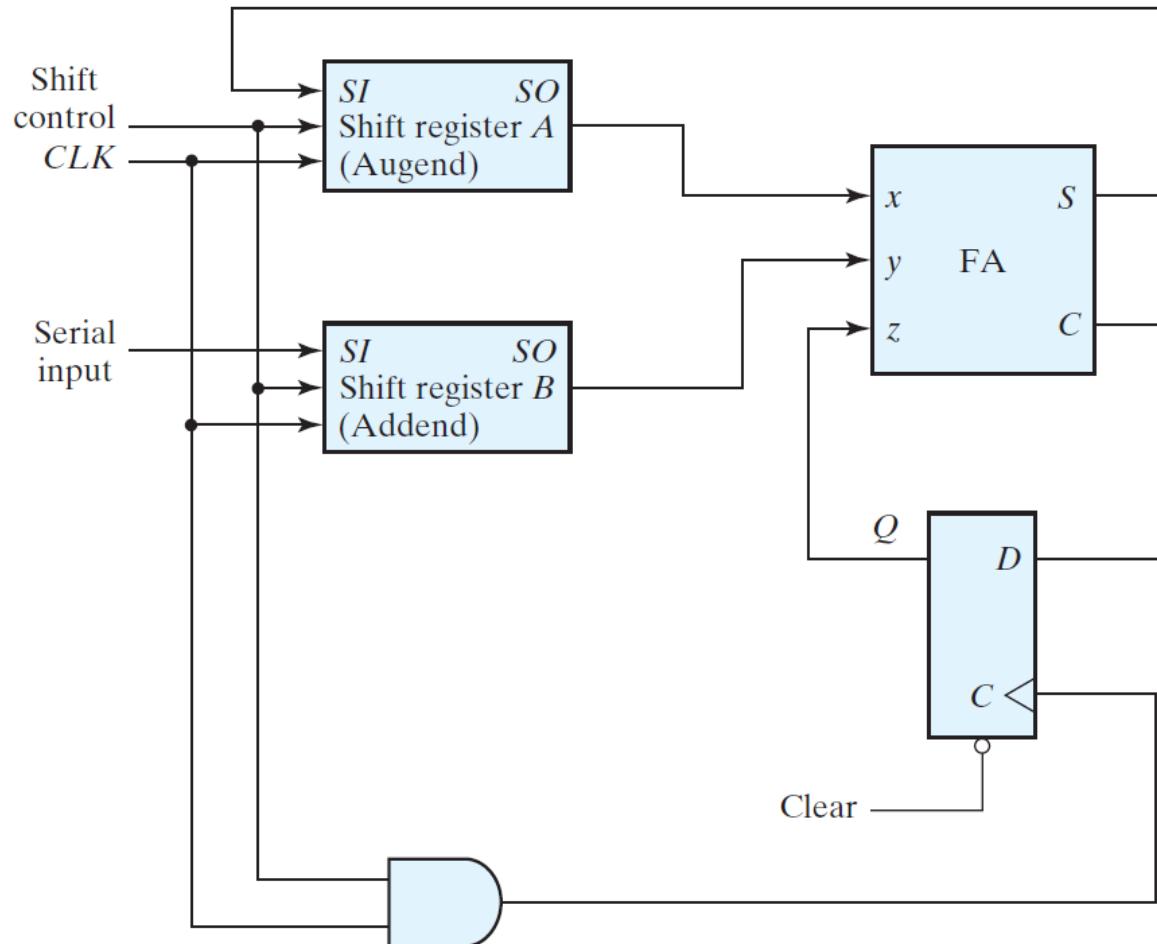


(a) Block diagram

Timing Pulse	Shift Register A	Shift Register B
Initial value	1 0 1 1	0 0 1 0
After $T_1$	1 1 0 1	1 0 0 1
After $T_2$	1 1 1 0	1 1 0 0
After $T_3$	0 1 1 1	0 1 1 0
After $T_4$	1 0 1 1	1 0 1 1

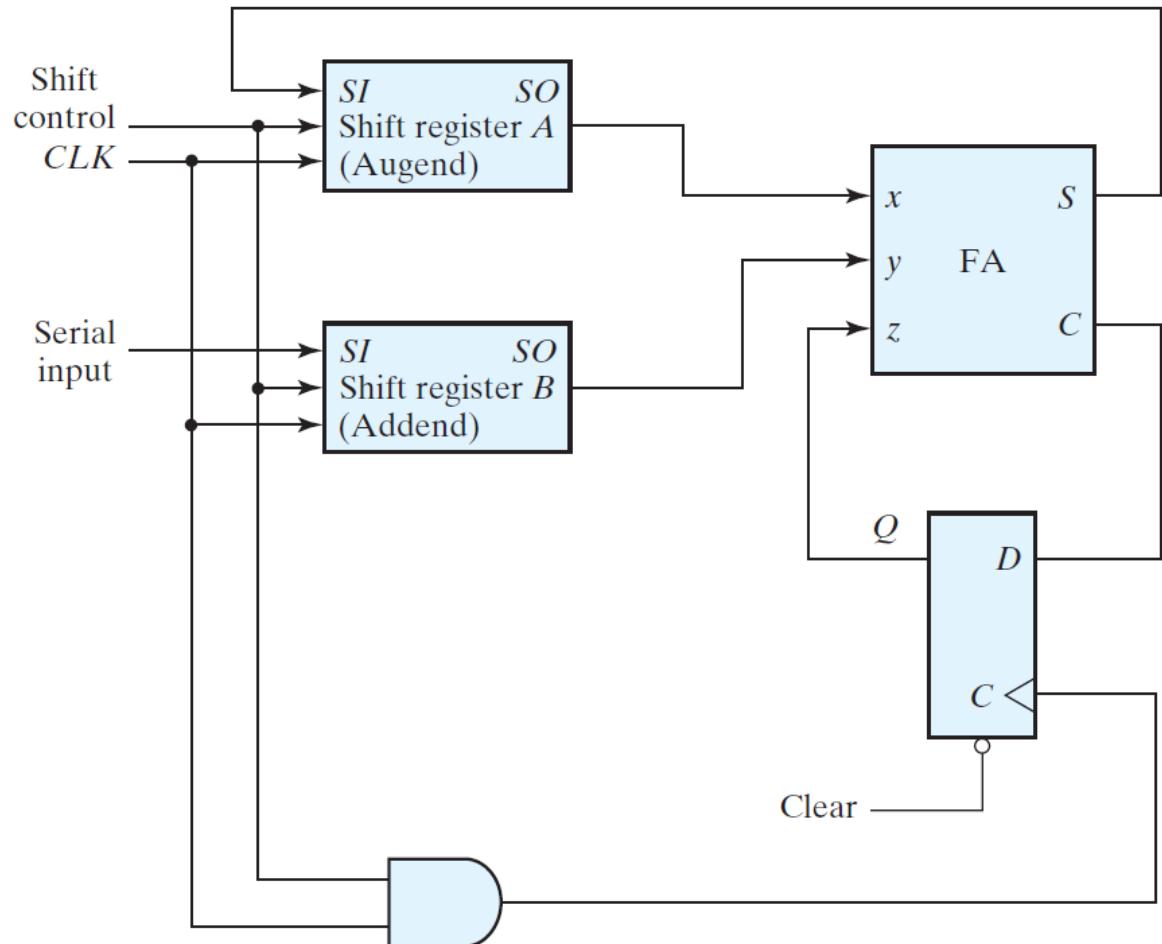
# Serial addition

- Can we design a serial addition circuit? The two binary numbers to be added serially are stored in two shift registers
- This is similar to the algorithm we use for adding manually
- Beginning with the least significant pair of bits, the circuit adds one pair at a time through a single full-adder (FA) circuit
- The carry out of the full adder is transferred to a  $D$  flip-flop, the output of which is then used as the carry input for the next pair of significant bits
- The sum bit from the  $S$  output of the full adder could be transferred into a third shift register
- By shifting the sum into  $A$  while the bits of  $A$  are shifted out, it is possible to use one register for storing both the augend and the sum bits
- The serial input of register  $B$  can be used to transfer a new binary number while the addend bits are shifted out during the addition



# Serial addition

- Comparing the serial adder with the parallel adder (4-bit adder), we note several differences
- The parallel adder uses registers with a parallel load, whereas the serial adder uses shift registers
- The number of full-adder circuits in the parallel adder is equal to the number of bits in the binary numbers, whereas the serial adder requires only one full-adder circuit and a carry flip-flop
- Excluding the registers, the parallel adder is a combinational circuit, whereas the serial adder is a sequential circuit which consists of a full adder and a flip-flop that stores the output carry

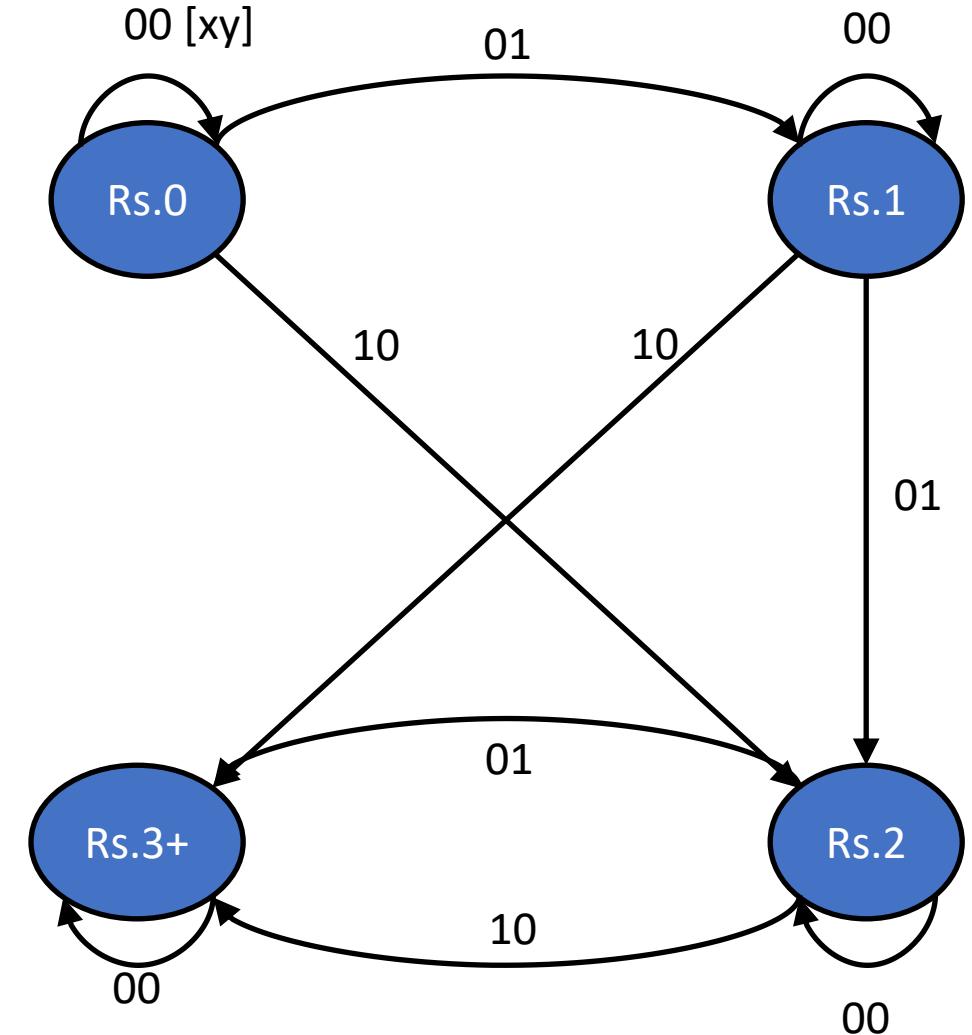


# Lecture 21 – Registers and Counters 2

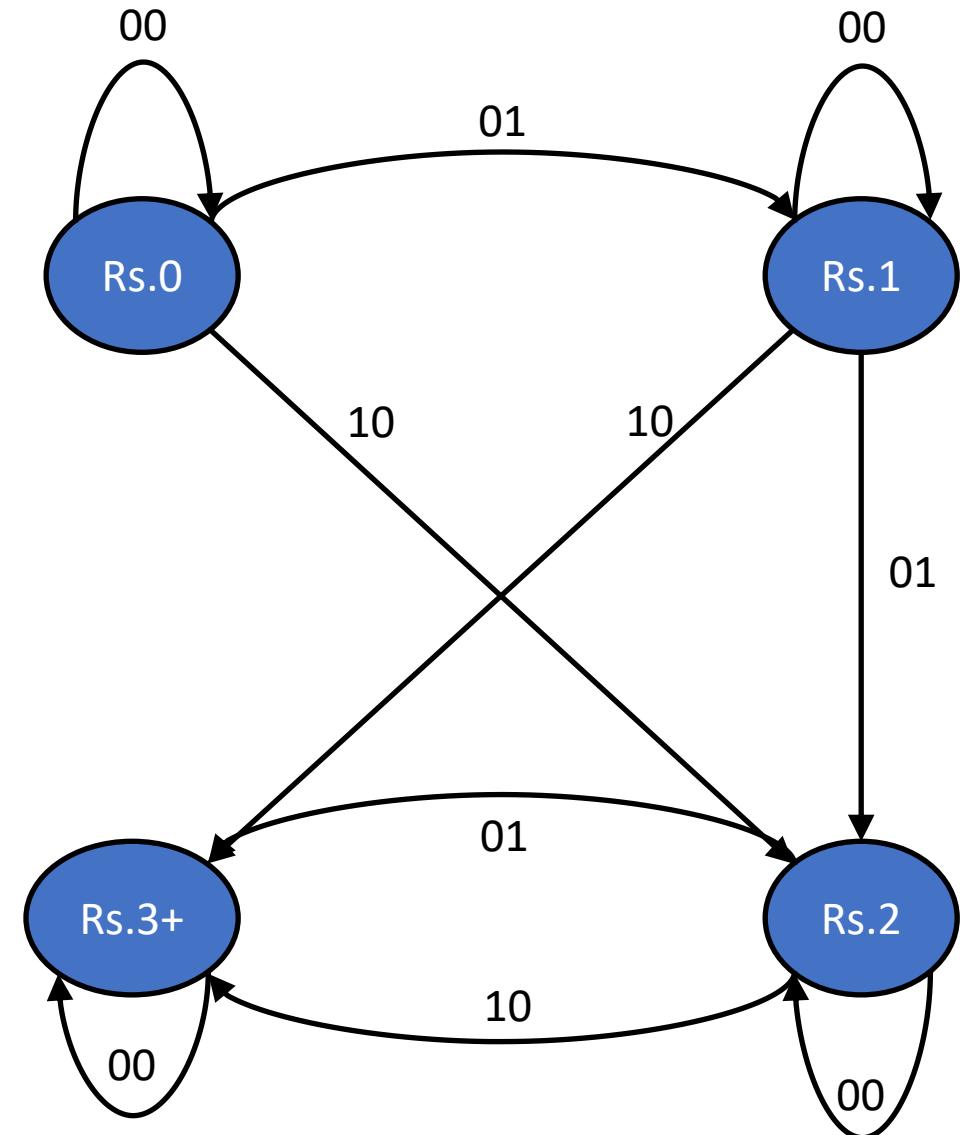
Chapter 6

# Design of sequential circuits - The vending machine

- Let's say we are asked to design a circuit for a vending machine that dispenses candy for Rs. 3
- The input consists of a coin slot that can accept Rs. 1 and Rs. 2 coins
- The deposit of these coins by the user is detected by a circuit that gives out two outputs  $x$  and  $y$ 
  - when Rs. 1 is inserted,  $y$  goes to one for one clock cycle
  - when Rs. 2 is inserted,  $x$  goes to one, for one clock cycle.
  - $x$  and  $y$  are at zero by default
- Only one coin can be entered at once
- We need to design a circuit that takes  $x$  and  $y$  as inputs and outputs 1 if the sum is  $\geq 3$ , so that the machine can dispense the candy

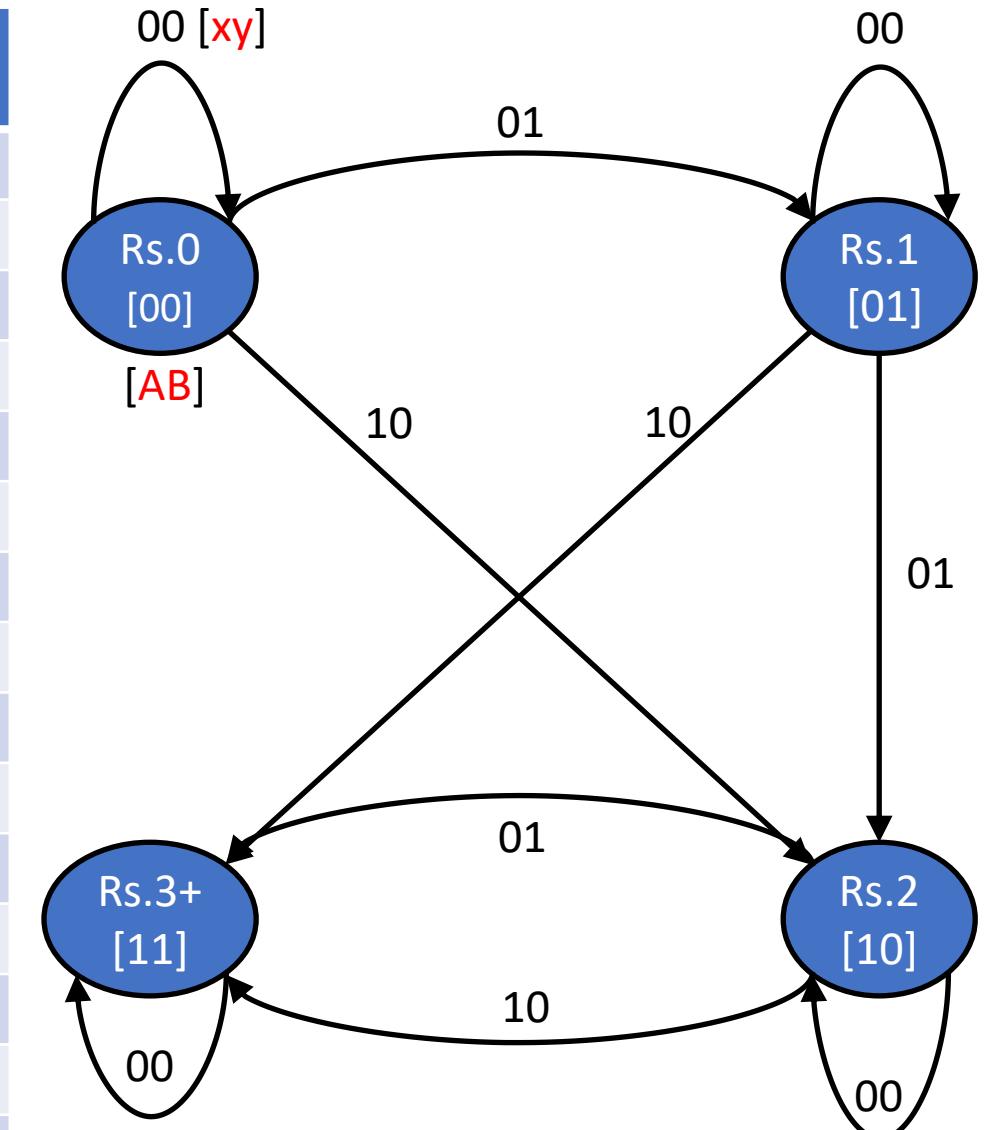


# The vending machine

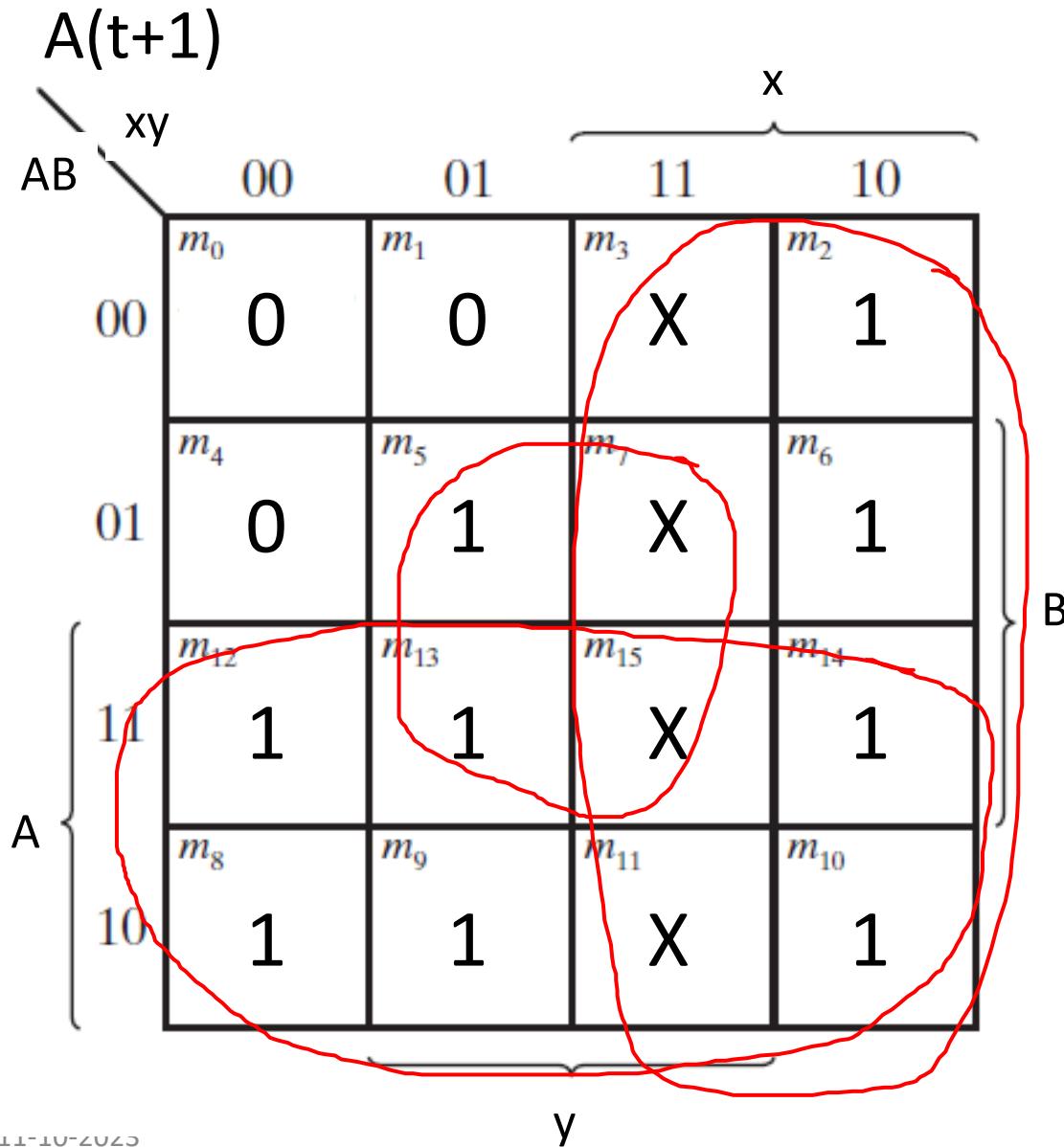


# The vending machine

A	B	x	y	A(t+1)	B(t+1)	z (output)
0	0	0	0	0	0	0
0	0	0	1	0	1	0
0	0	1	0	1	0	0
0	0	1	1	x	x	0
0	1	0	0	0	1	0
0	1	0	1	1	0	0
0	1	1	0	1	1	0
0	1	1	1	x	x	0
1	0	0	0	1	0	0
1	0	0	1	1	1	0
1	0	1	0	1	1	0
1	0	1	1	x	x	0
1	1	0	0	1	1	1
1	1	0	1	1	1	1
1	1	1	0	1	1	1
1	1	1	1	x	x	1

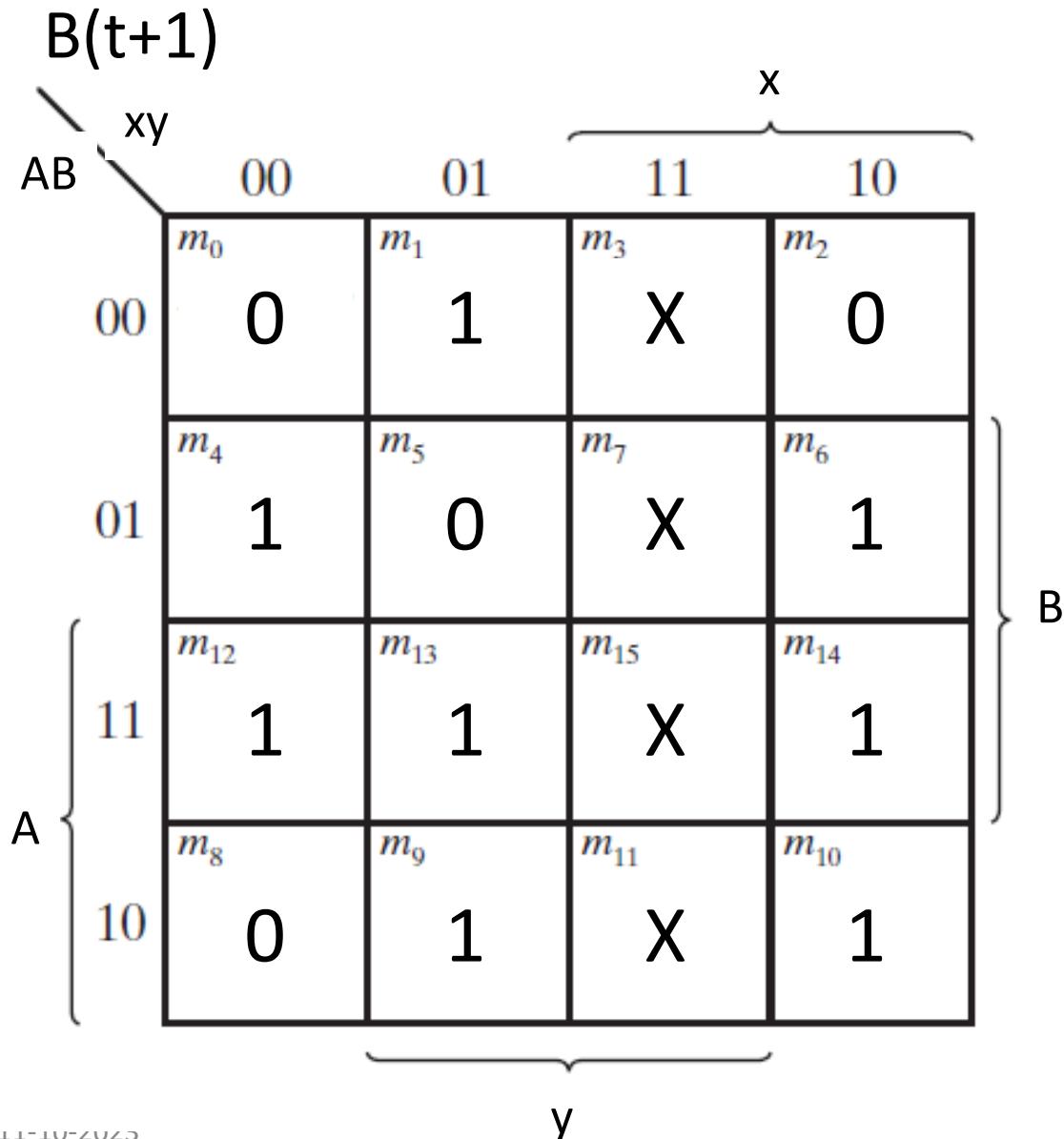


# The vending machine



$$A(t + 1) = A + x + By$$

# The vending machine



$$B(t+1) = (B + y + Ax)(B' + y' + A)$$

# The vending machine

---

$$A(t+1) = A + x + By$$

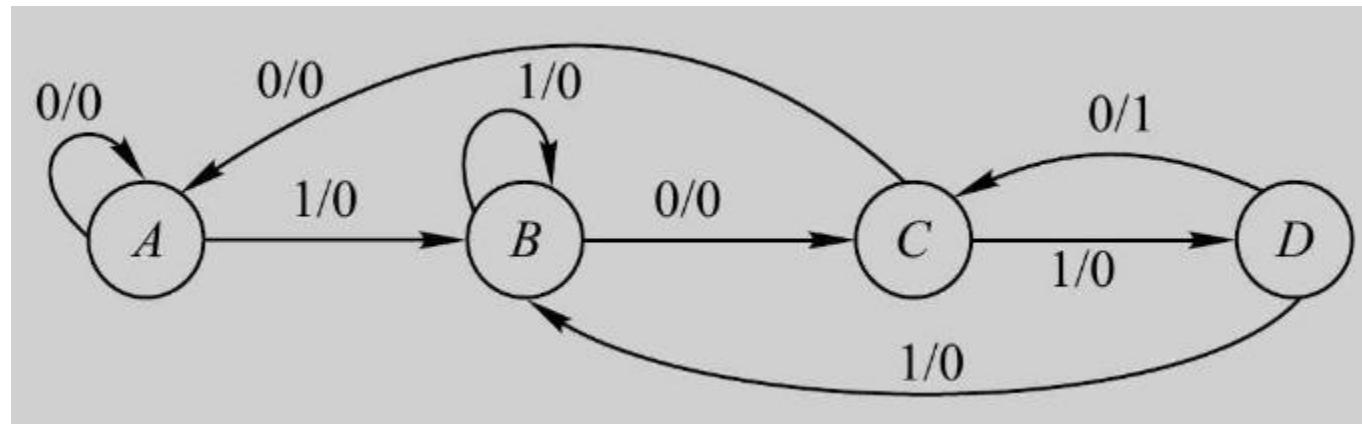
$$B(t+1) = (B + y + Ax)(B' + y' + A)$$

$$z = AB$$

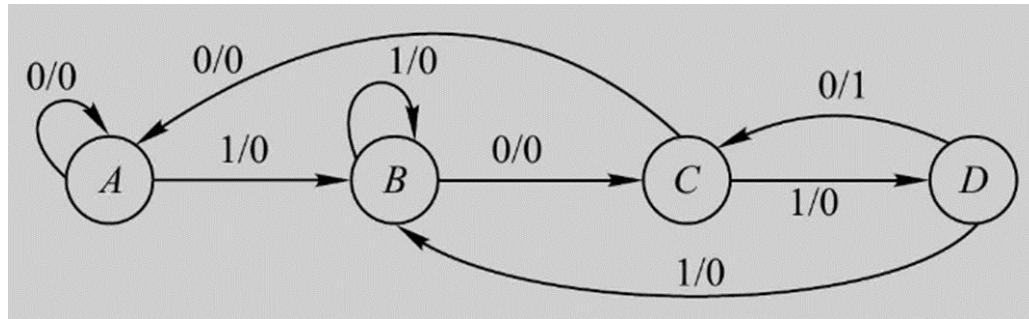
# Sequential circuit - pattern detector

- A sequence/pattern detector to detect a sequence of **1010**. Output should go to high when the sequence is detected.

- Eg: input  $x \dots 1 0 1 0 1 0 1 0 1 0 0 0 \dots$   
output  $y \dots 0 0 0 1 0 1 0 1 0 1 0 0 \dots$



# Sequential circuit - design

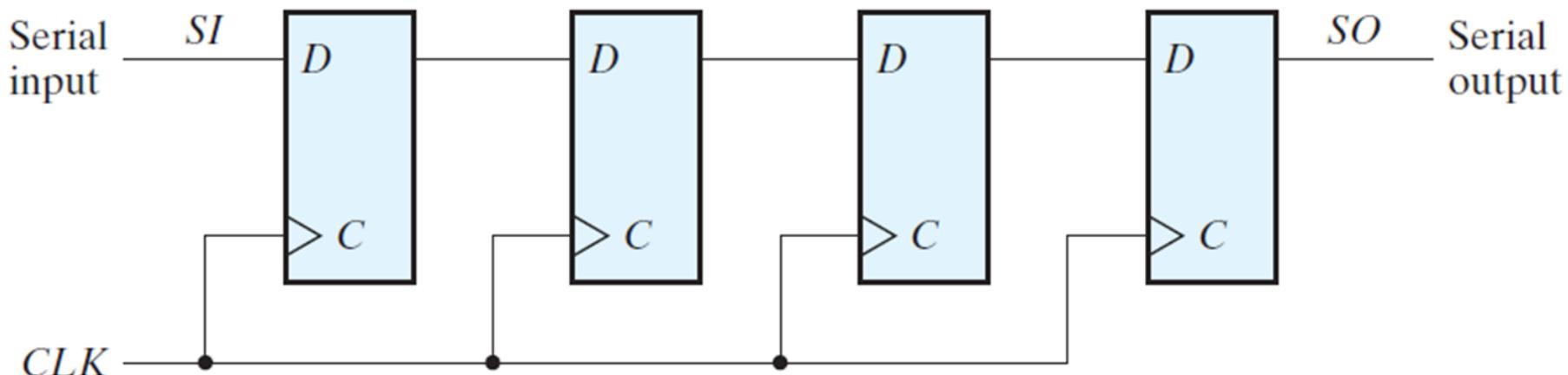


A → 00  
B → 01  
C → 10  
D → 11

Present state		Input $X$	Next state		Output $Y(t)$
$Q_1$	$Q_0$		$Q_1(t+1)$	$Q_0(t+1)$	
0	0	0	0	0	0
0	1	0	1	0	0
1	0	0	0	0	0
1	1	0	1	0	1
0	0	1	0	1	0
0	1	1	0	1	0
1	0	1	1	1	0
1	1	1	0	1	0

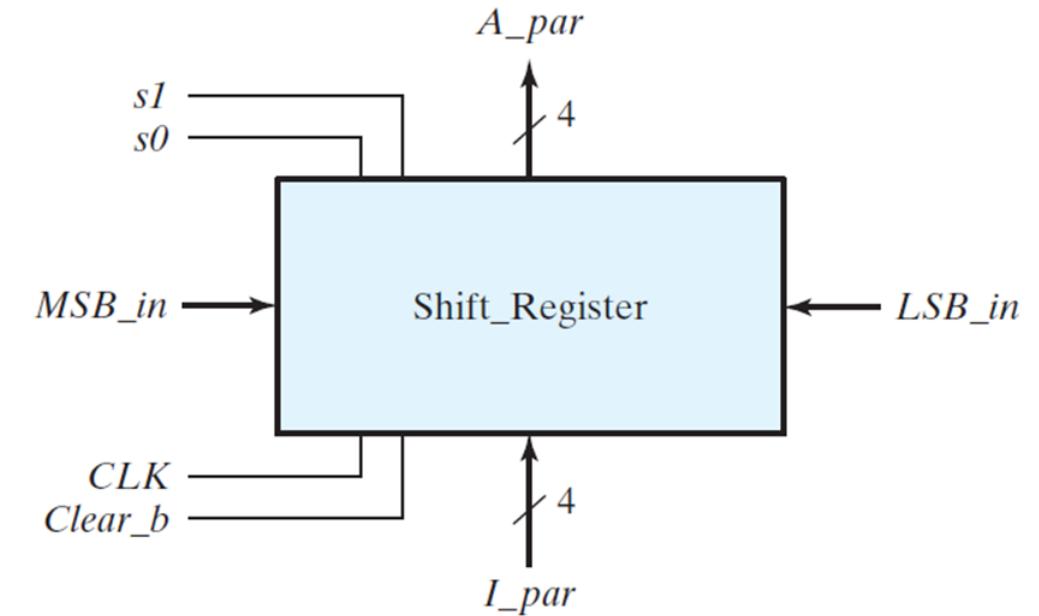
# Universal register

- If the flip-flop outputs of a shift register are accessible, then information entered serially by shifting can be taken out in parallel from the outputs of the flip-flops
- If a parallel load capability is added to a shift register, then data entered in parallel can be taken out in serial fashion by shifting the data stored in the register
- Some shift registers provide the necessary input and output terminals for parallel transfer
- They may also have both shift-right and shift-left capabilities



# Universal register

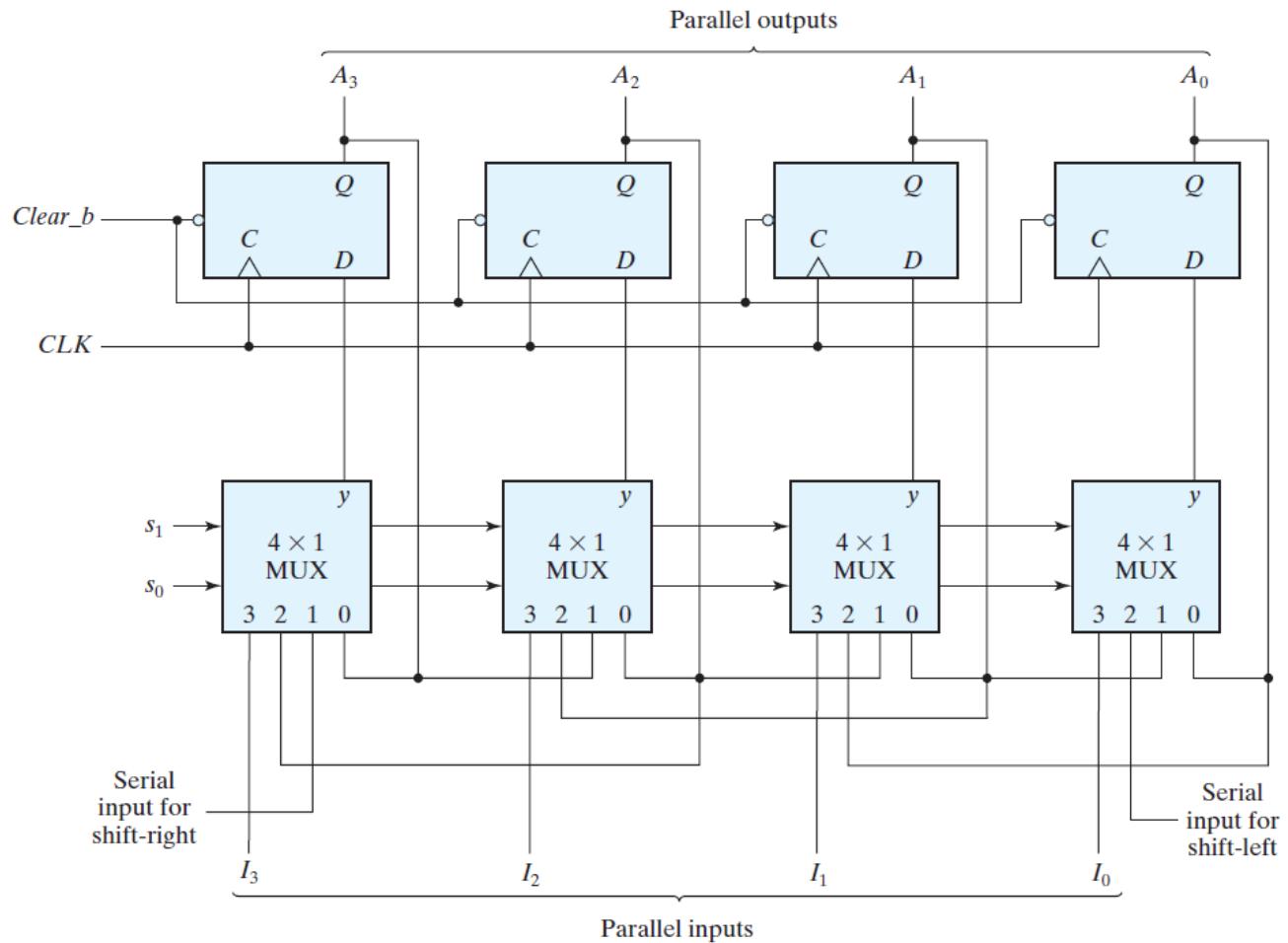
- The most general shift register has the following capabilities:
  - A *clear* control to clear the register to 0
  - A *clock* input to synchronize the operations
  - A *shift-right* control to enable the shift-right operation and the *serial input* and *output* lines associated with the shift right
  - A *shift-left* control to enable the shift-left operation and the *serial input* and *output* lines associated with the shift left
  - A *parallel-load* control to enable a parallel transfer and the  $n$  input lines associated with the parallel transfer
  - n parallel output lines*
  - A *control state* that leaves the information in the register unchanged in response to the clock



Mode Control		Register Operation
$s_1$	$s_0$	
0	0	No change
0	1	Shift right
1	0	Shift left
1	1	Parallel load

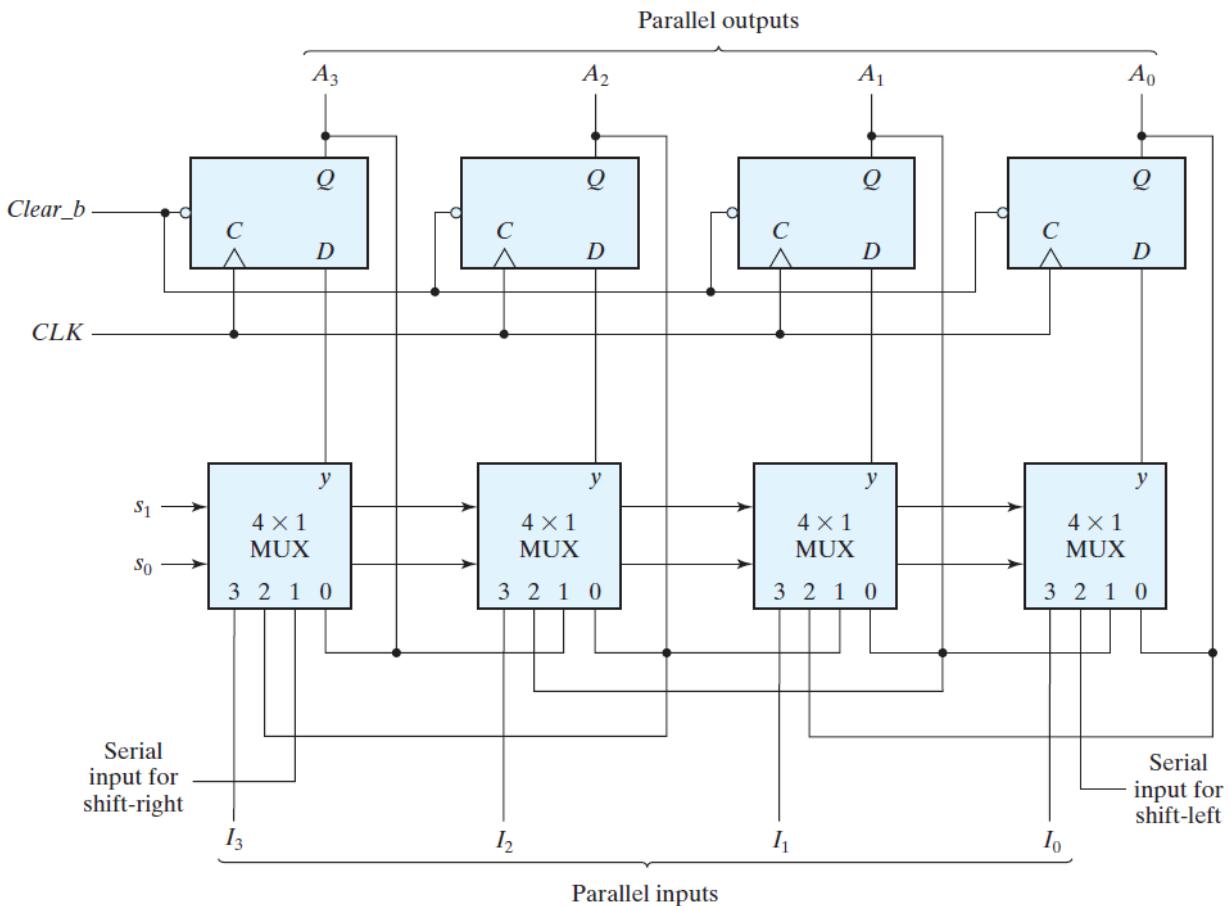
# Universal register

- The circuit consists of four  $D$  flip-flops and four multiplexers
- The four multiplexers have two common selection inputs  $s_1$  and  $s_0$
- The selection inputs control the mode of operation of the register according to the function required
- The output of the MUXes are applied to the inputs of the FFs which controls the next state of the FF



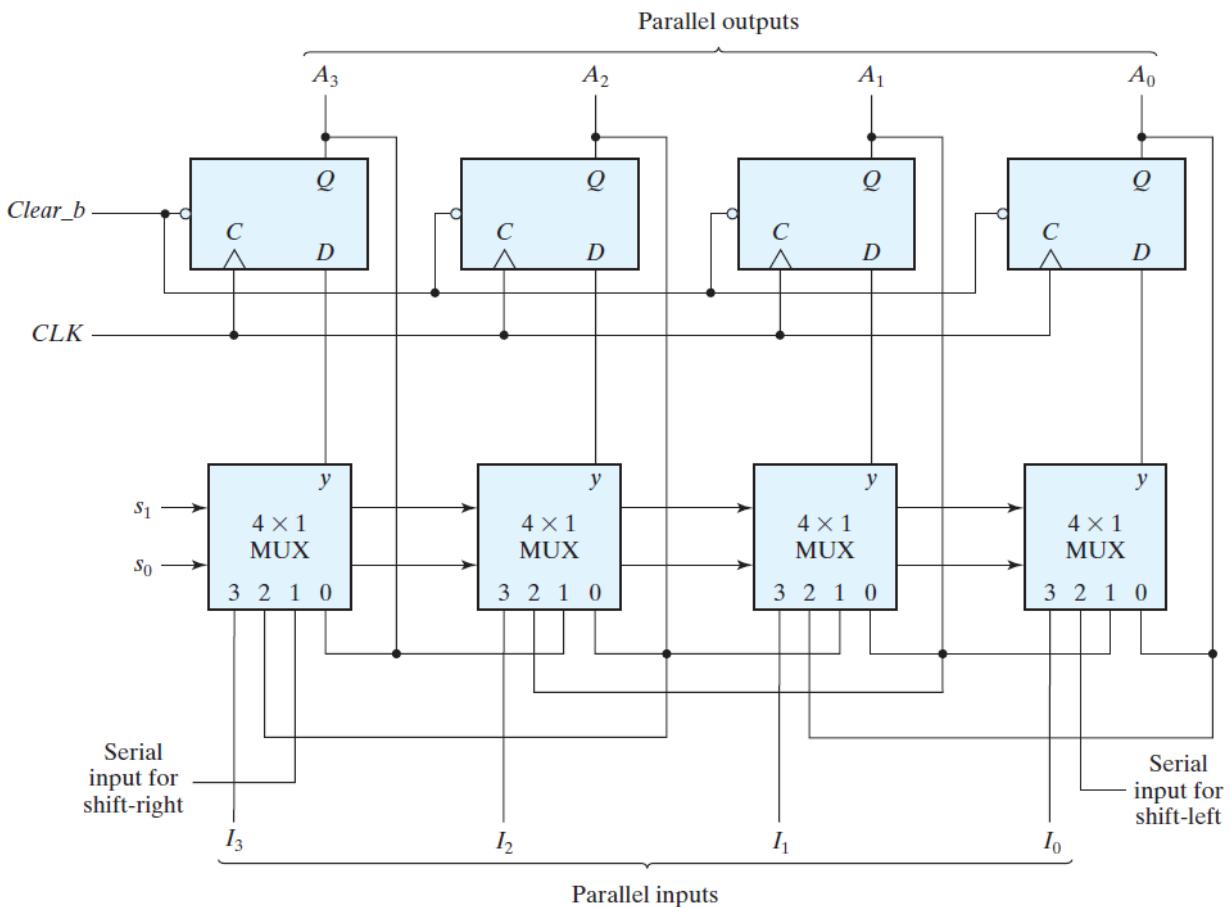
# Universal register

- When  $s_1s_0 = 00$ , the present value of the register is applied to the  $D$  inputs of the flip-flops
- This condition forms a path from the output of each flip-flop into the input of the same flip-flop, so that the output recirculates to the input in this mode of operation
- The next clock edge transfers into each flip-flop the binary value it held previously, and no change of state occurs



# Universal register

- When  $s_1s_0 = 01$ , terminal 1 of the multiplexer inputs has a path to the  $D$  inputs of the flip-flops
- This causes a shift-right operation, with the serial input transferred into flip-flop  $A_3$
- When  $s_1s_0 = 10$ , a shift-left operation results, with the other serial input going into flip-flop  $A_0$
- Finally, when  $s_1s_0 = 11$ , the binary information on the parallel input lines is transferred into the register simultaneously during the next clock edge



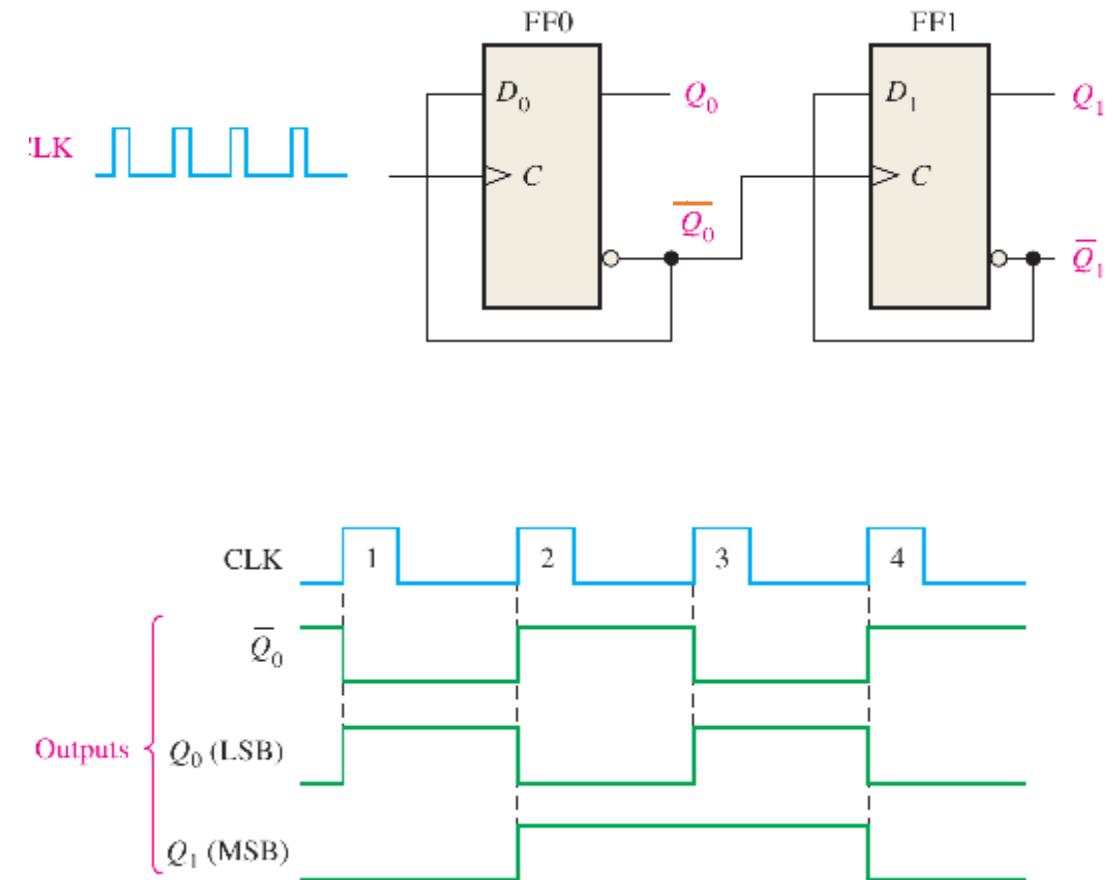
# Counters

---

- A register that goes through a prescribed sequence of states upon the application of input pulses is called a *counter*
- The input pulses may be clock pulses, or they may originate from some external source and may occur at a fixed interval of time or at random
- The sequence of states may follow the binary number sequence or any other sequence of states
- Counters are available in two categories: **ripple counters** and **synchronous counters**
- In a ripple counter, a flip-flop output transition serves as a source for triggering other flip-flops
- In a synchronous counter, the  $C$  inputs of all flip-flops receive the common clock

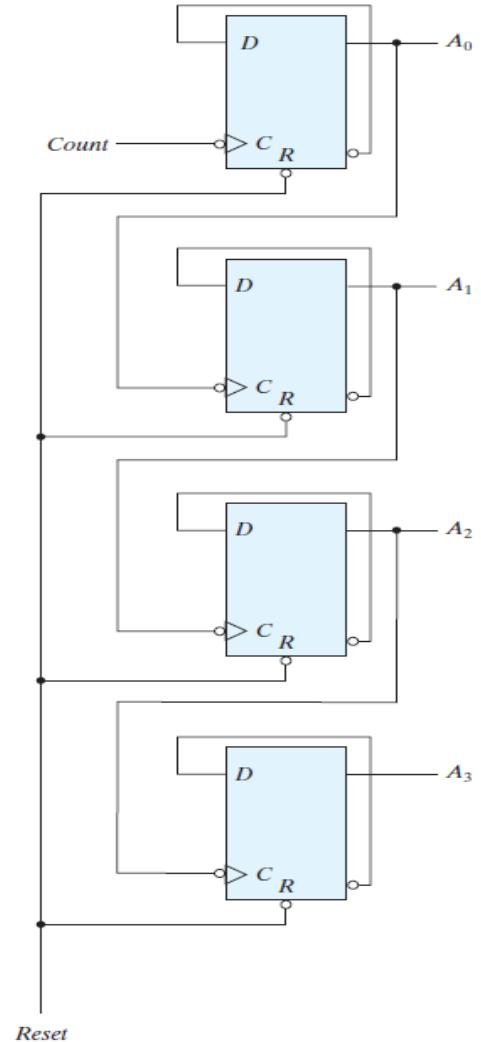
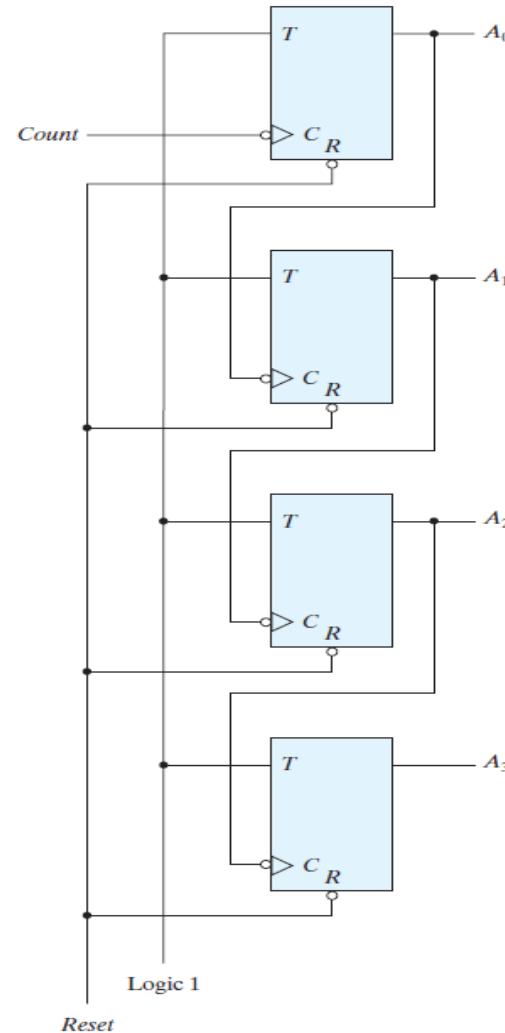
# Ripple counter - binary

- A binary ripple counter consists of a series connection of complementing flip-flops, with the output of each flip-flop connected to the  $C$  input of the next higher order flip-flop
- The flip-flop holding the least significant bit receives the incoming count pulses
- A complementing flip-flop can be obtained from a  $JK$  flip-flop with the  $J$  and  $K$  inputs tied together or from a  $T$  flip-flop
- Another possibility is to use a  $D$  flip-flop with the complement output connected to the  $D$  input
- In this way, the  $D$  input is always the complement of the present state, and the next clock pulse will cause the flip-flop to complement



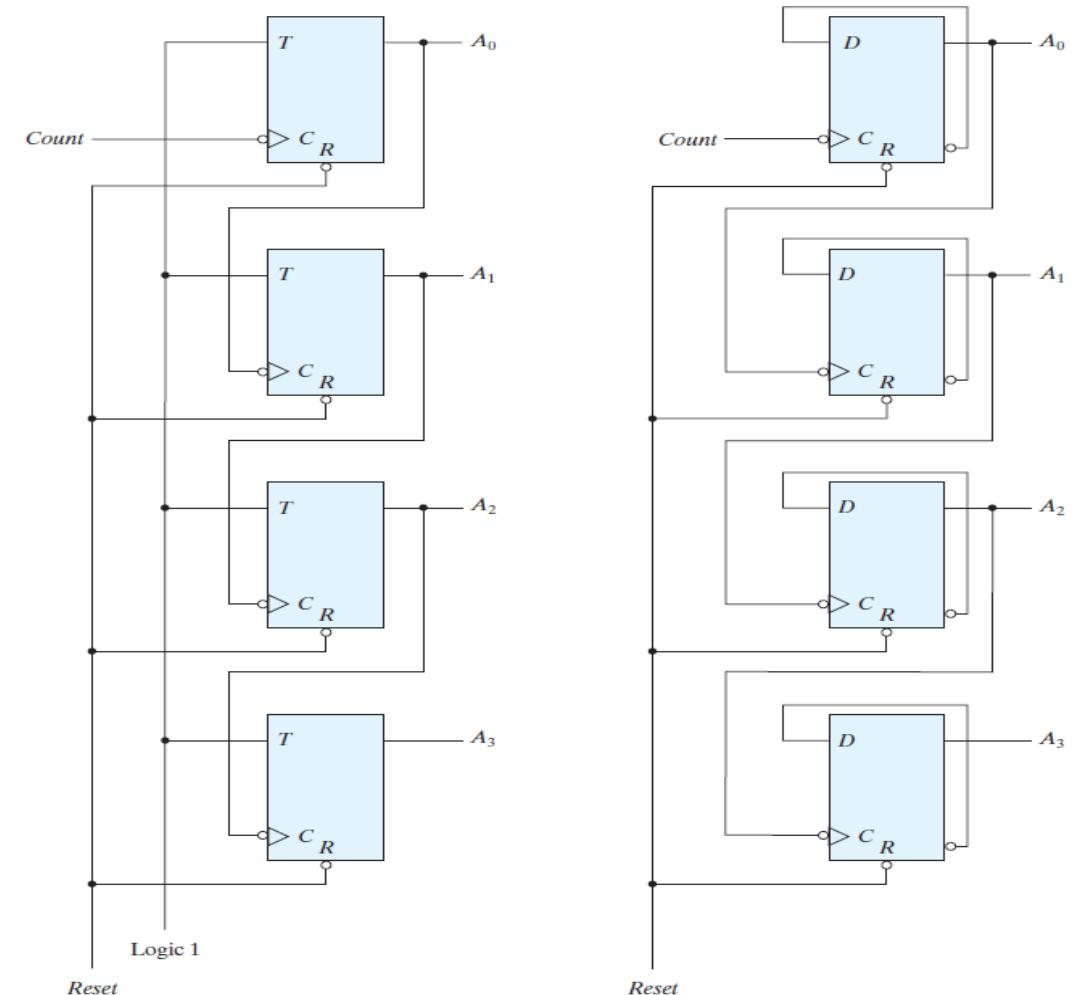
# Ripple counter - binary

- The count starts with binary 0 and increments by 1 with each count pulse input
- After the count of 15, the counter goes back to 0 to repeat the count
- The least significant bit,  $A_0$ , is complemented with each count pulse input
- Every time that  $A_0$  goes from 1 to 0, it complements  $A_1$  and so on...



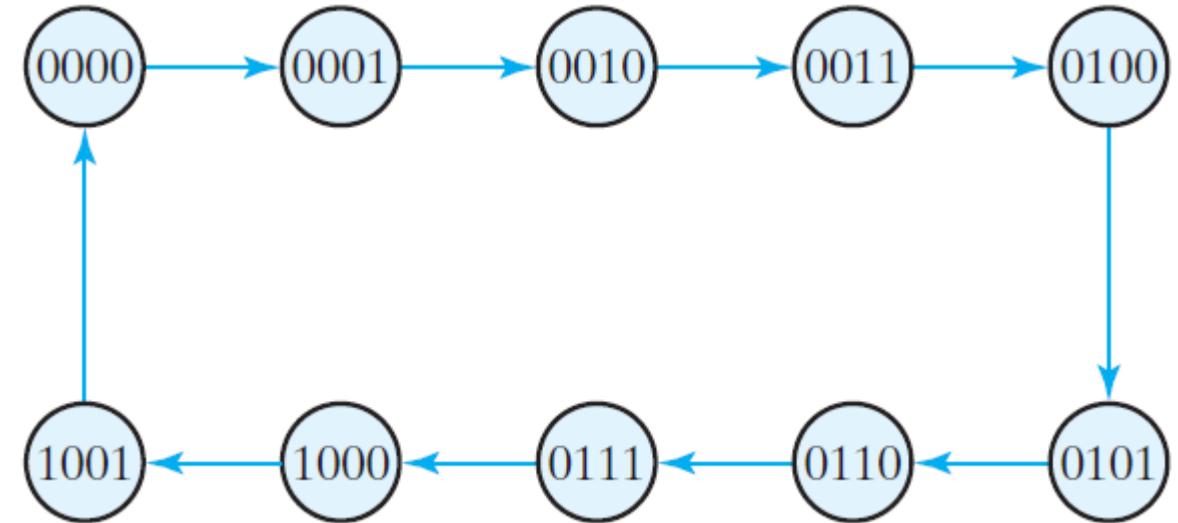
# Ripple counter - binary

- For example, consider the transition from count 0011 to 0100
- $A_0$  is complemented with the count pulse
- Since  $A_0$  goes from 1 to 0, it triggers  $A_1$  and complements it
- As a result,  $A_1$  goes from 1 to 0, which in turn complements  $A_2$ , changing it from 0 to 1
- $A_2$  does not trigger  $A_3$ , because  $A_2$  produces a positive transition, and the flip-flop responds only to negative transitions



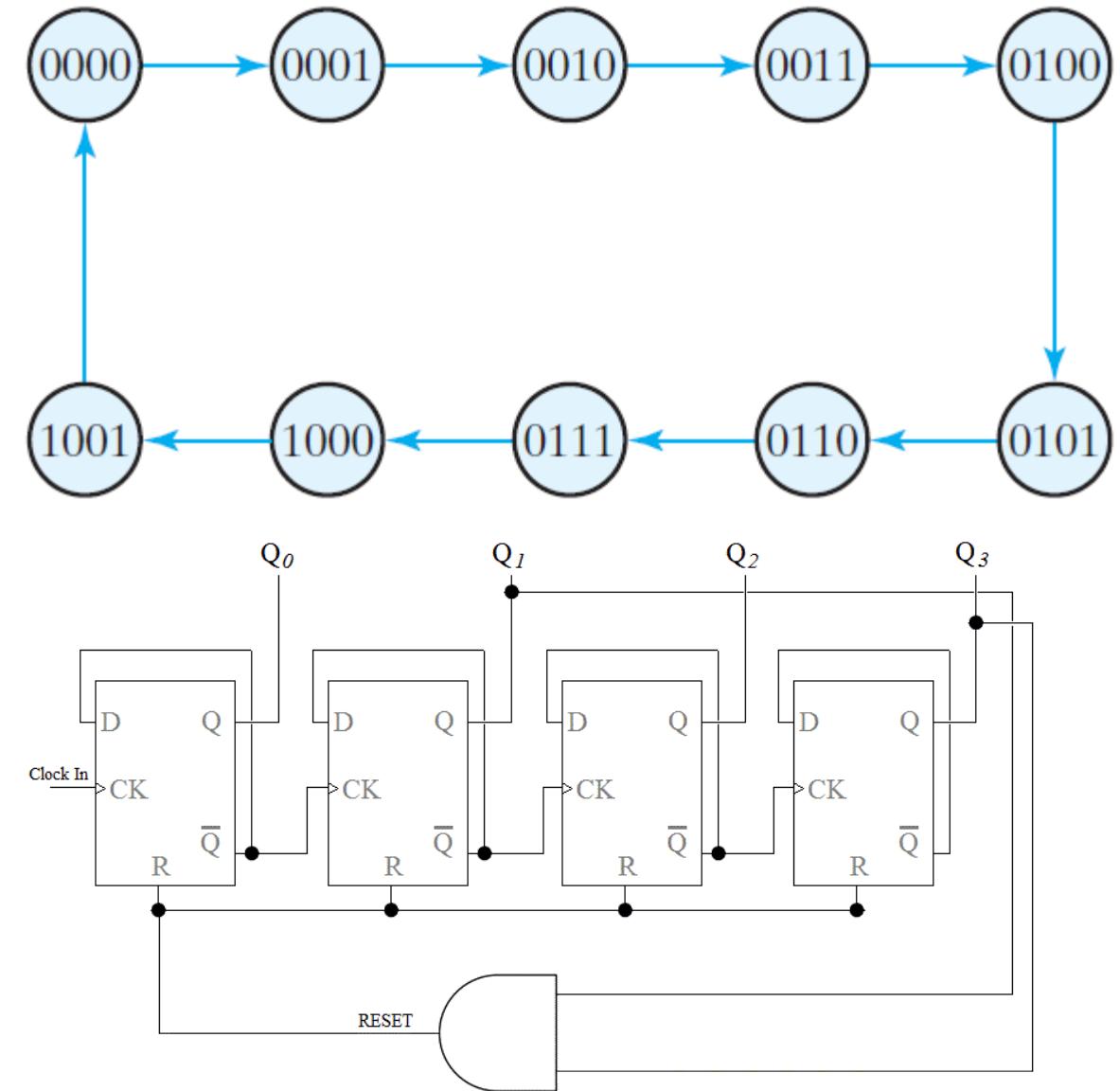
# Ripple counter - BCD

- A decimal counter follows a sequence of 10 states and returns to 0 after the count of 9
- Such a counter must have at least four flip-flops to represent each decimal digit, since a decimal digit is represented by a binary code with at least four bits
- The sequence of states in a decimal counter is dictated by the binary code used to represent a decimal digit
- A decimal counter is similar to a binary counter, except that the state after 1001 (the code for decimal digit 9) is 0000 (the code for decimal digit 0)



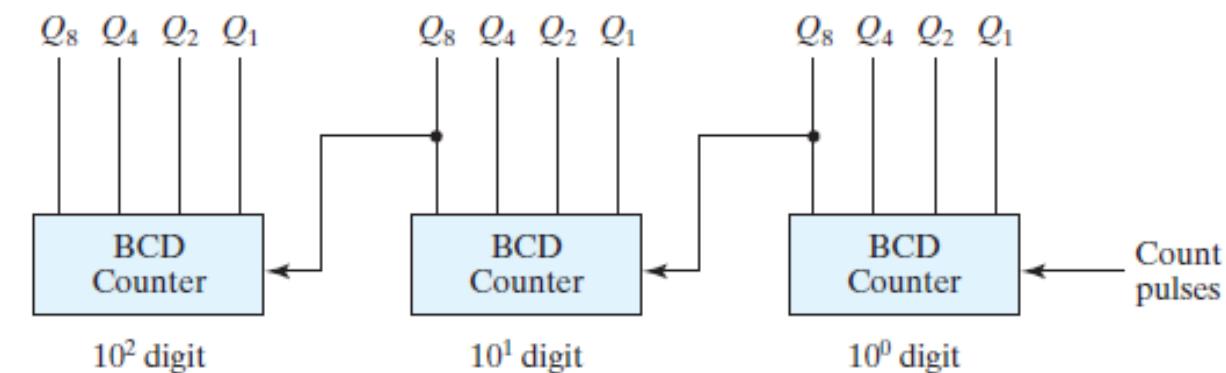
# Ripple counter - BCD

- We can obtain a decade counter by clearing all the flip-flops as soon as the state 1010 is obtained
- This can be done with the asynchronous input of CLR
- The condition for 1010 is checked by ANDing  $Q_3$  and  $Q_1$
- This is a very commonly used counter for making clocks/timer circuits



# Ripple counter - BCD

- A **decade counter** counts from 0 to 9
- To count in decimal from 0 to 99, we need a two-decade counter
- To count from 0 to 999, we need a three-decade counter
- Multiple decade counters can be constructed by connecting BCD counters in cascade, one for each decade
- The inputs to the second and third decades come from  $Q_8$  of the previous decade
- When  $Q_8$  in one decade goes from 1 to 0, it triggers the count for the next higher order decade while its own decade goes from 9 to 0

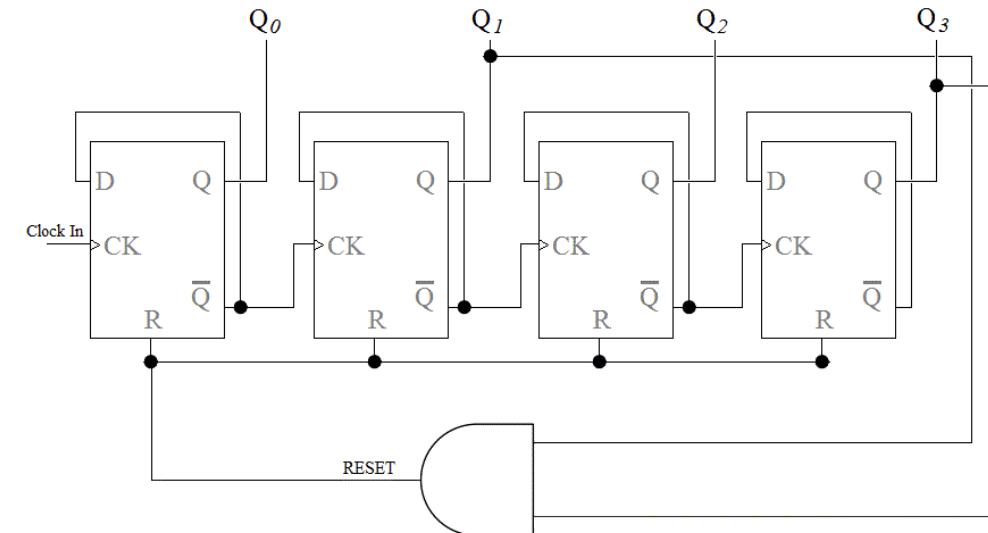
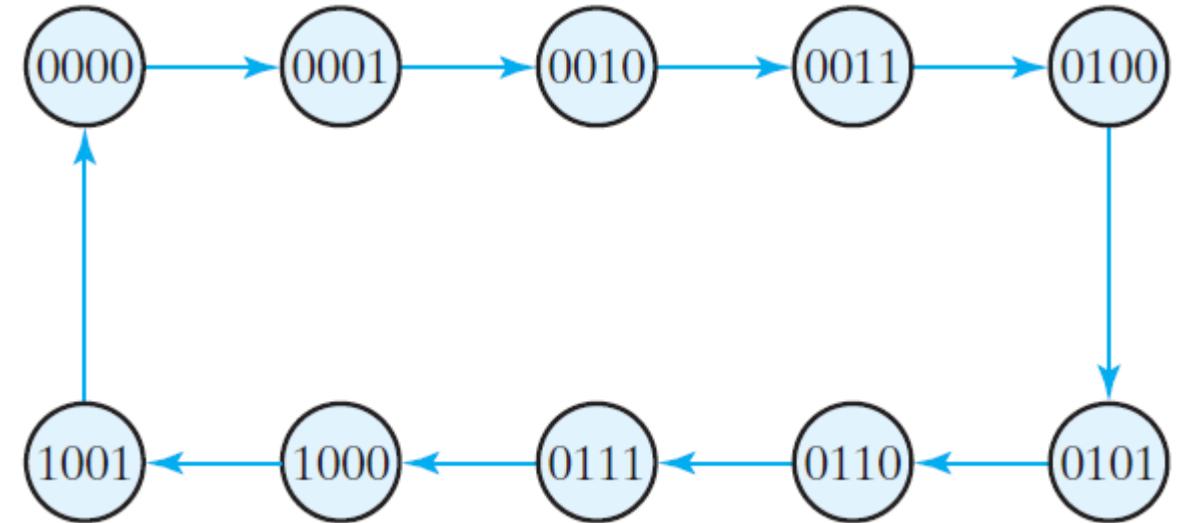


# Lecture 22 – Registers and Counters 3

Chapter 6

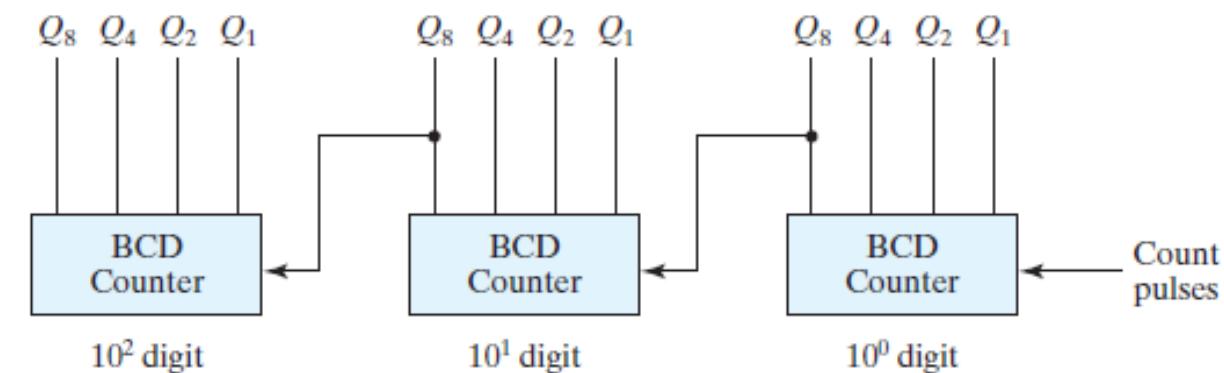
# Ripple counter - BCD

- A decimal counter follows a sequence of 10 states and returns to 0 after the count of 9
- Such a counter must have at least four flip-flops to represent each decimal digit, since a decimal digit is represented by a binary code with at least four bits
- The sequence of states in a decimal counter is dictated by the binary code used to represent a decimal digit
- A decimal counter is similar to a binary counter, except that the state after 1001 (the code for decimal digit 9) is 0000 (the code for decimal digit 0)



# Ripple counter - BCD

- A **decade counter** counts from 0 to 9
- To count in decimal from 0 to 99, we need a two-decade counter
- To count from 0 to 999, we need a three-decade counter
- Multiple decade counters can be constructed by connecting BCD counters in cascade, one for each decade
- The inputs to the second and third decades come from  $Q_8$  of the previous decade
- When  $Q_8$  in one decade goes from 1 to 0, it triggers the count for the next higher order decade while its own decade goes from 9 to 0

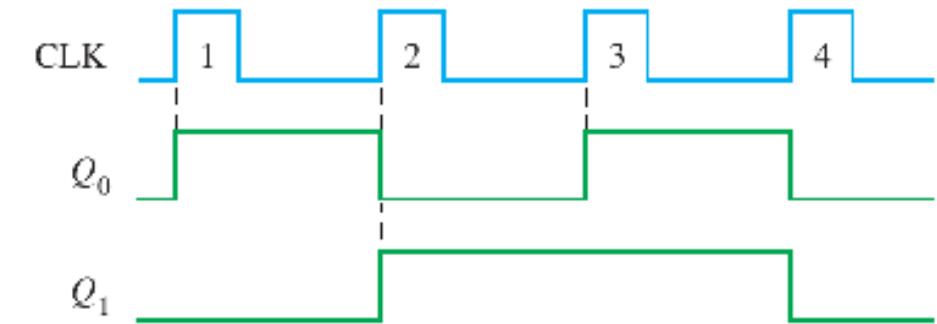
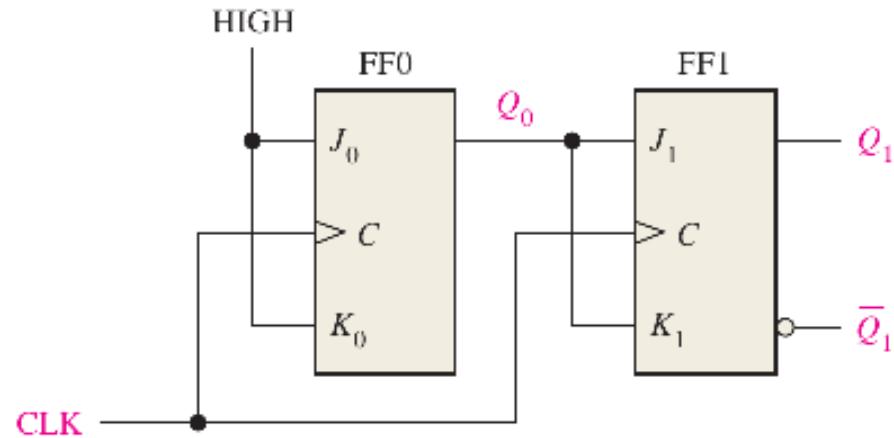


# Synchronous counter - binary

- Synchronous counters are different from ripple counters in that clock pulses are applied to the inputs of all flip-flops
- A common clock triggers all flip-flops simultaneously, rather than one at a time in succession as in a ripple counter
- The decision whether a flip-flop is to be complemented is determined from the values of the data inputs, such as  $T$  or  $J$  and  $K$  at the time of the clock edge
- If  $T = 0$  or  $J = K = 0$ , the flip-flop does not change state. If  $T = 1$  or  $J = K = 1$ , the flip-flop complements

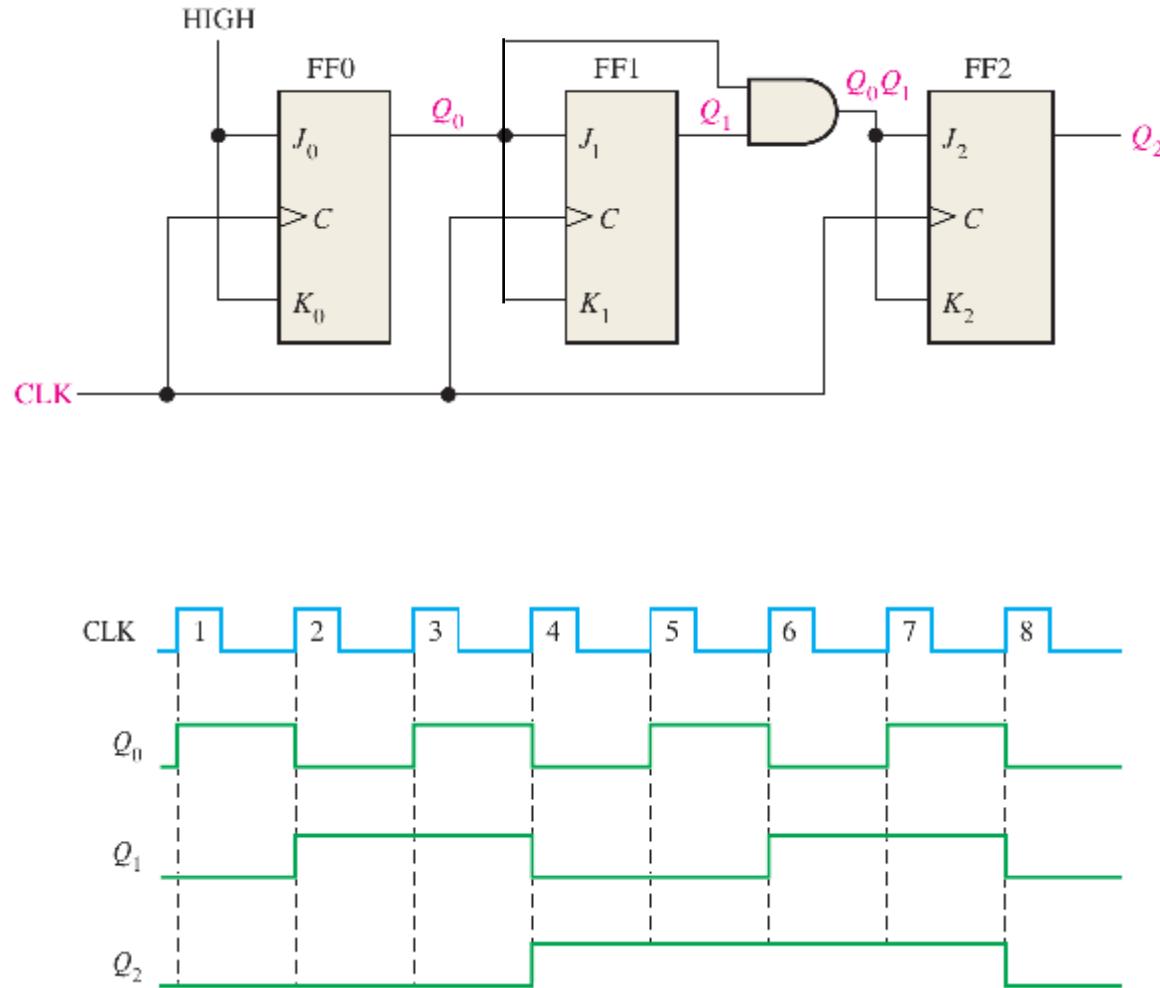
# Synchronous counter - binary

## 2-bit synchronous binary counter



# Synchronous counter - binary

## 3-bit synchronous binary counter



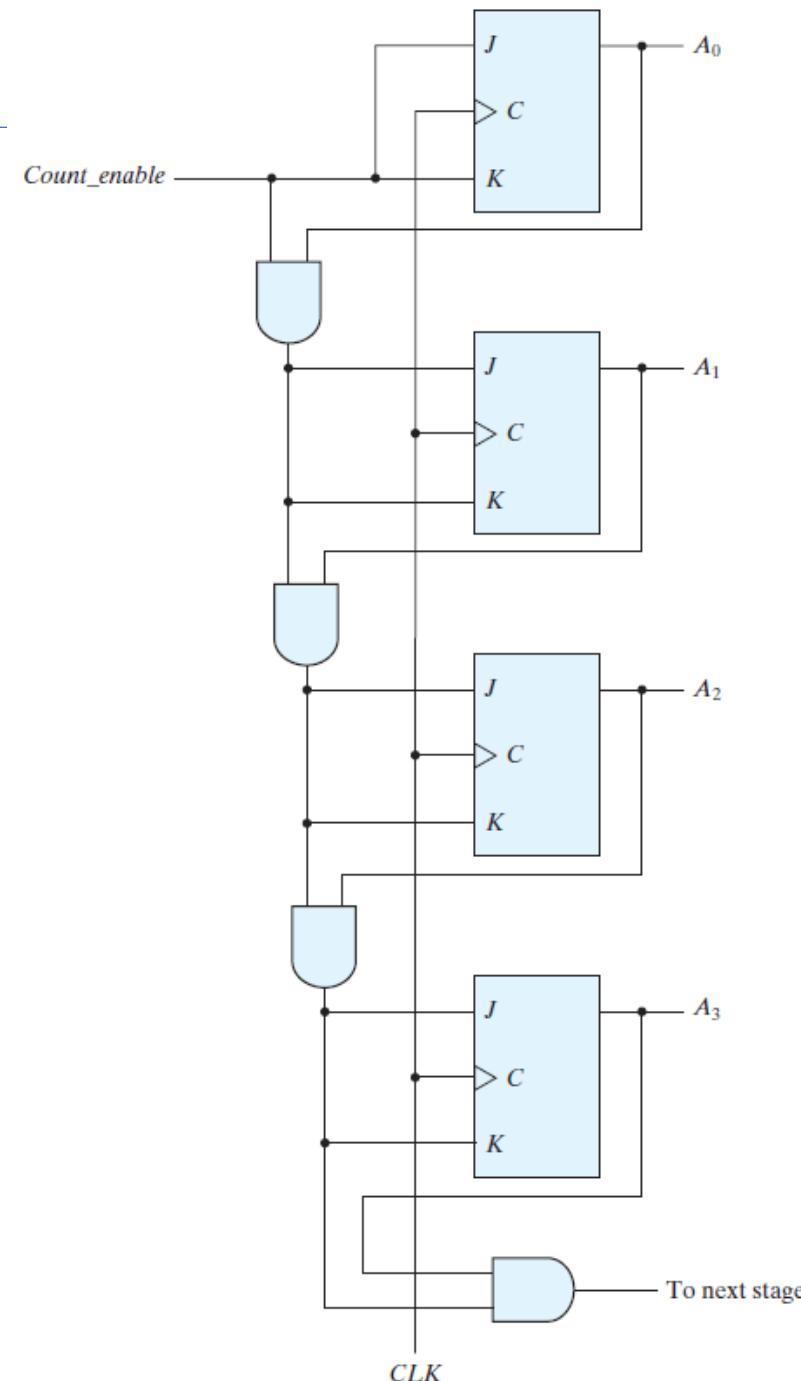
Clock Pulse	$Q_2$	$Q_1$	$Q_0$
Initially	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1
8 (recycles)	0	0	0

# Synchronous counter - binary

- The design of a synchronous binary counter is very simple
- In a synchronous binary counter, the flip-flop in the least significant position is complemented with every pulse
- *A flip-flop in any other position is complemented when all the bits in the lower significant positions are equal to 1*
- For example, if the present state of a four-bit counter is  $A_3A_2A_1A_0 = 0011$ , the next count is 0100
  - $A_0$  is always complemented
  - $A_1$  is complemented because the present state of  $A_0 = 1$
  - $A_2$  is complemented because the present state of  $A_1A_0 = 11$
  - However,  $A_3$  is not complemented, because the present state of  $A_2A_1A_0 = 011$ , which does not give an all-1's condition

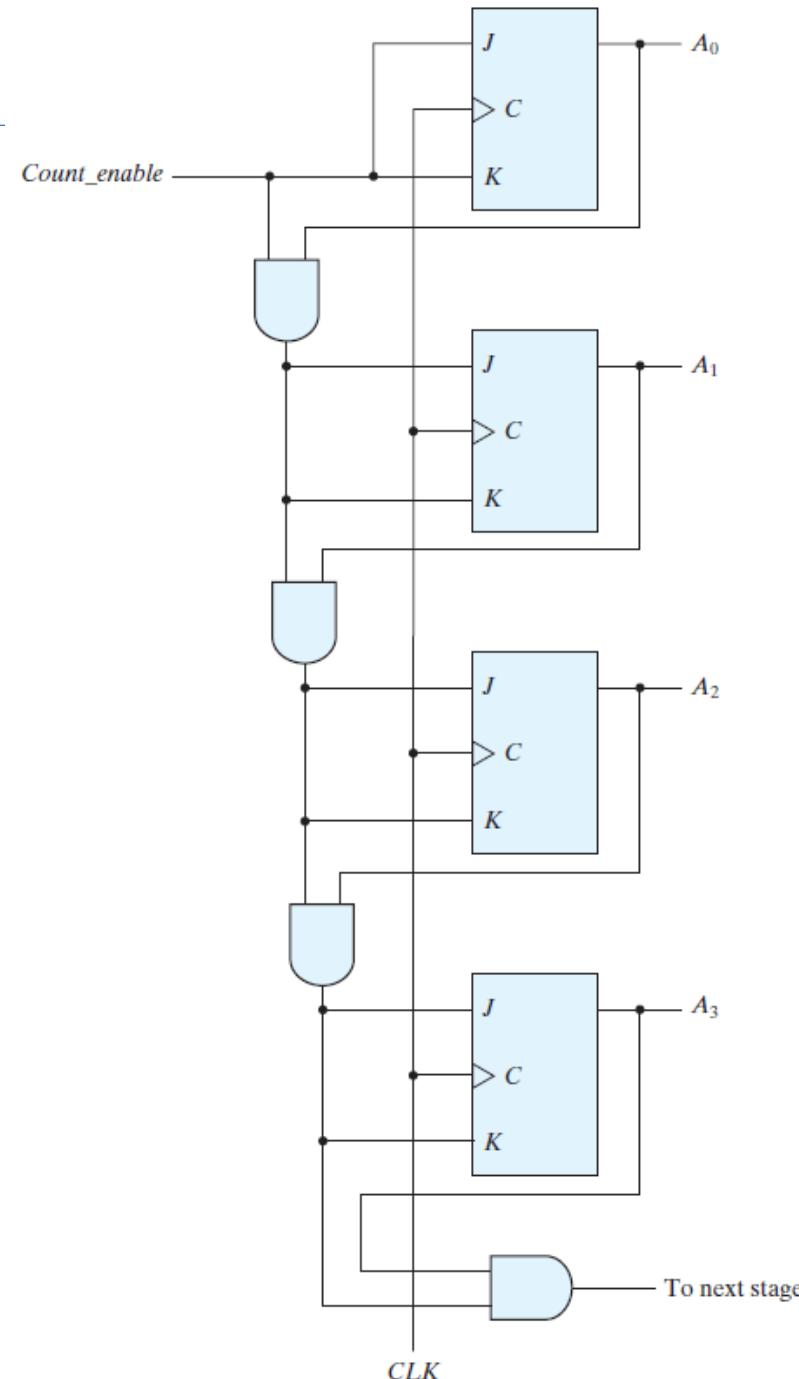
# Synchronous counter - binary

- Synchronous binary counters have a regular pattern and can be constructed with complementing flip-flops and gates
- The regular pattern can be seen from the four-bit counter
- The  $C$  inputs of all flip-flops are connected to a common clock
- The counter is enabled by  $Count\_enable$
- If the enable input is 0, all  $J$  and  $K$  inputs are equal to 0 and the clock does not change the state of the counter
- The first stage,  $A_0$ , has its  $J$  and  $K$  equal to 1 if the counter is enabled
- The other  $J$  and  $K$  inputs are equal to 1 if all previous least significant stages are equal to 1 and the count is enabled



# Synchronous counter - binary

- Note that the flip-flops trigger on the positive edge of the clock
- The synchronous counter can be triggered with either the positive or the negative clock edge
- The complementing flip-flops in a binary counter can be of either the *JK* type, the *T* type, or the *D* type with XOR gates



# Synchronous counter – down counter

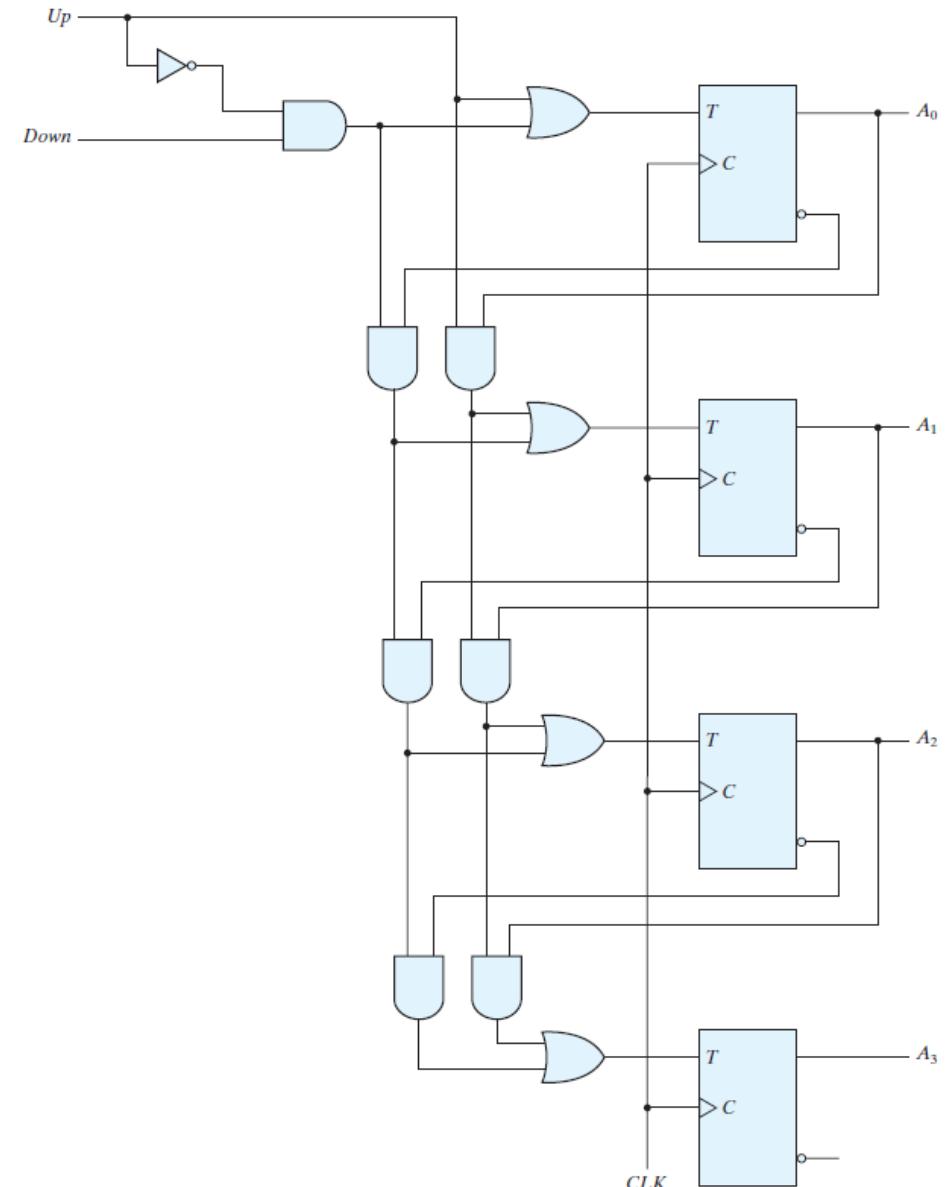
- A synchronous countdown binary counter goes through the binary states in reverse order, from 1111 down to 0000 and back to 1111 to repeat the count
- It is possible to design a countdown counter in the usual manner, but the result is predictable by inspection of the downward binary count
- The bit in the least significant position is complemented with each pulse
- *A bit in any other position is complemented if all lower significant bits are equal to 0*
- For example, the next state after the present state of 0100 is 0011
- The second significant bit is complemented because the first bit was 0
- The third significant bit is complemented because the first two bits were equal to 0
- But the fourth bit does not change, because not all lower significant bits are equal to 0
- A countdown binary counter can be constructed as a regular counter, except that the inputs to the AND gates must come from the complemented outputs, instead of the normal outputs, of the previous flip-flops

# Synchronous counter – up/down counter

- Let us see if we can design a counter that counts up/down based on a control input
- The counter should count up if  $U=1$ ,  $D=0$ ; down if  $U=0$ ,  $D=1$ ; no change if  $U=D=0$ ; and up if  $U=D=1$
- We can cascade this logic to make the synchronous up/down counter

# Synchronous counter – up/down counter

- Let us see if we can design a counter that counts up/down based on an input
- The counter should count up if  $U=1$ ,  $D=0$ ; down if  $U=0$ ,  $D=1$ ; no change if  $U=D=0$ ; and up if  $U=D=1$
- We can cascade this logic to make the synchronous up/down counter



# Lecture 23 – Registers and Counters 4

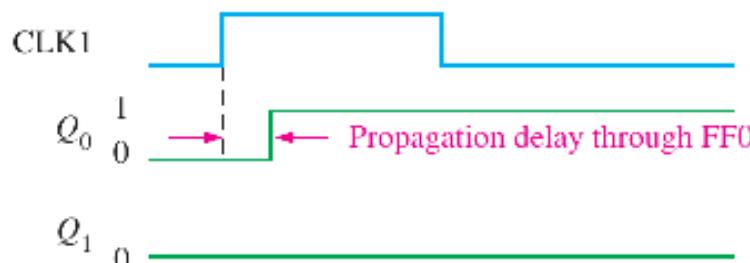
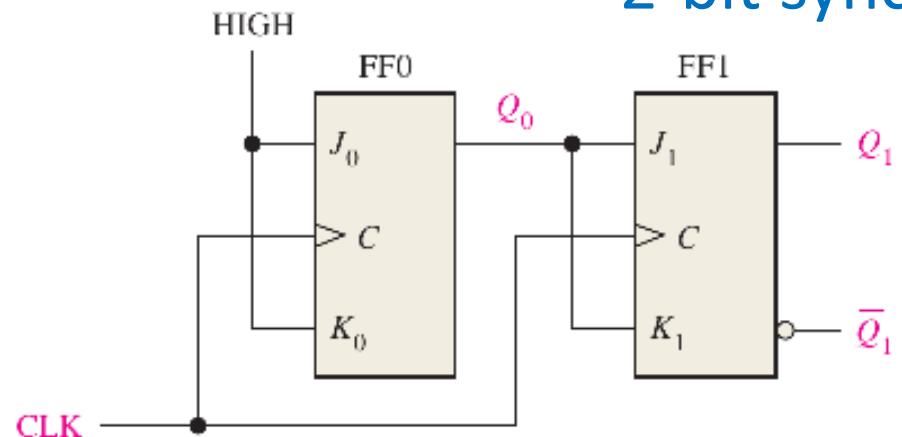
Chapter 6

# Synchronous counter - binary

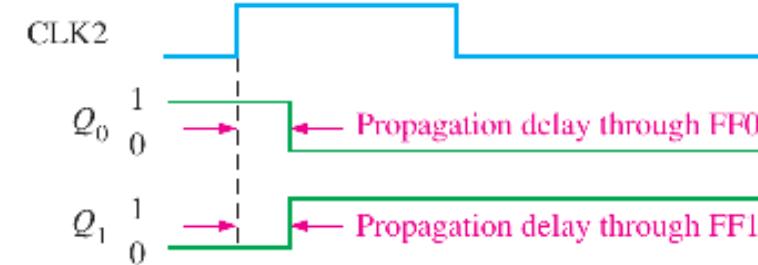
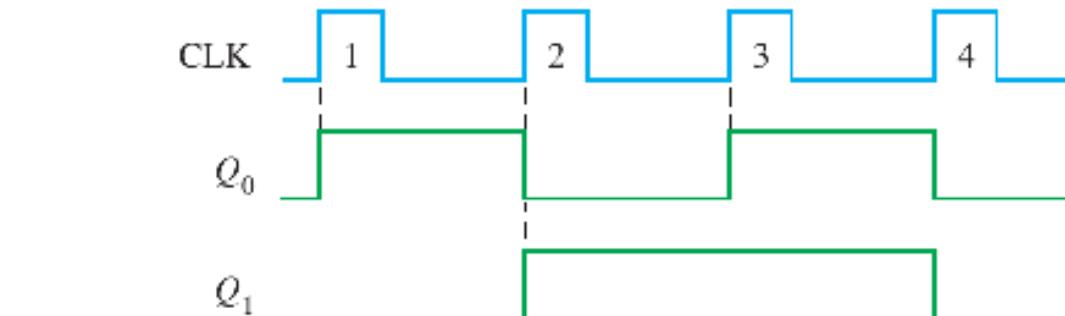
- Synchronous counters are different from ripple counters in that clock pulses are applied to the inputs of all flip-flops
- A common clock triggers all flip-flops simultaneously, rather than one at a time in succession as in a ripple counter
- The decision whether a flip-flop is to be complemented is determined from the values of the data inputs, such as  $T$  or  $J$  and  $K$  at the time of the clock edge
- If  $T = 0$  or  $J = K = 0$ , the flip-flop does not change state. If  $T = 1$  or  $J = K = 1$ , the flip-flop complements

# Synchronous counter - binary

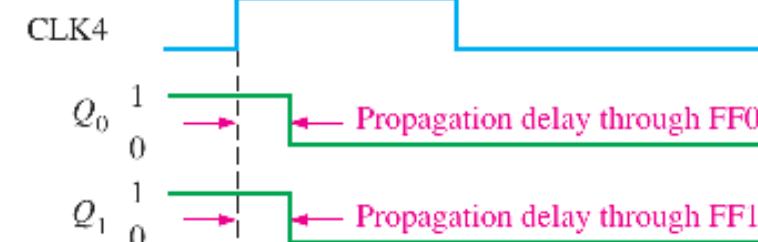
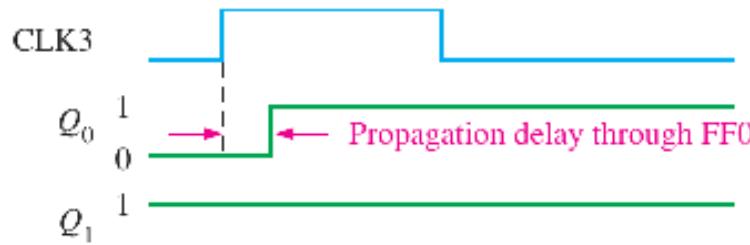
## 2-bit synchronous binary counter



(a)



(b)

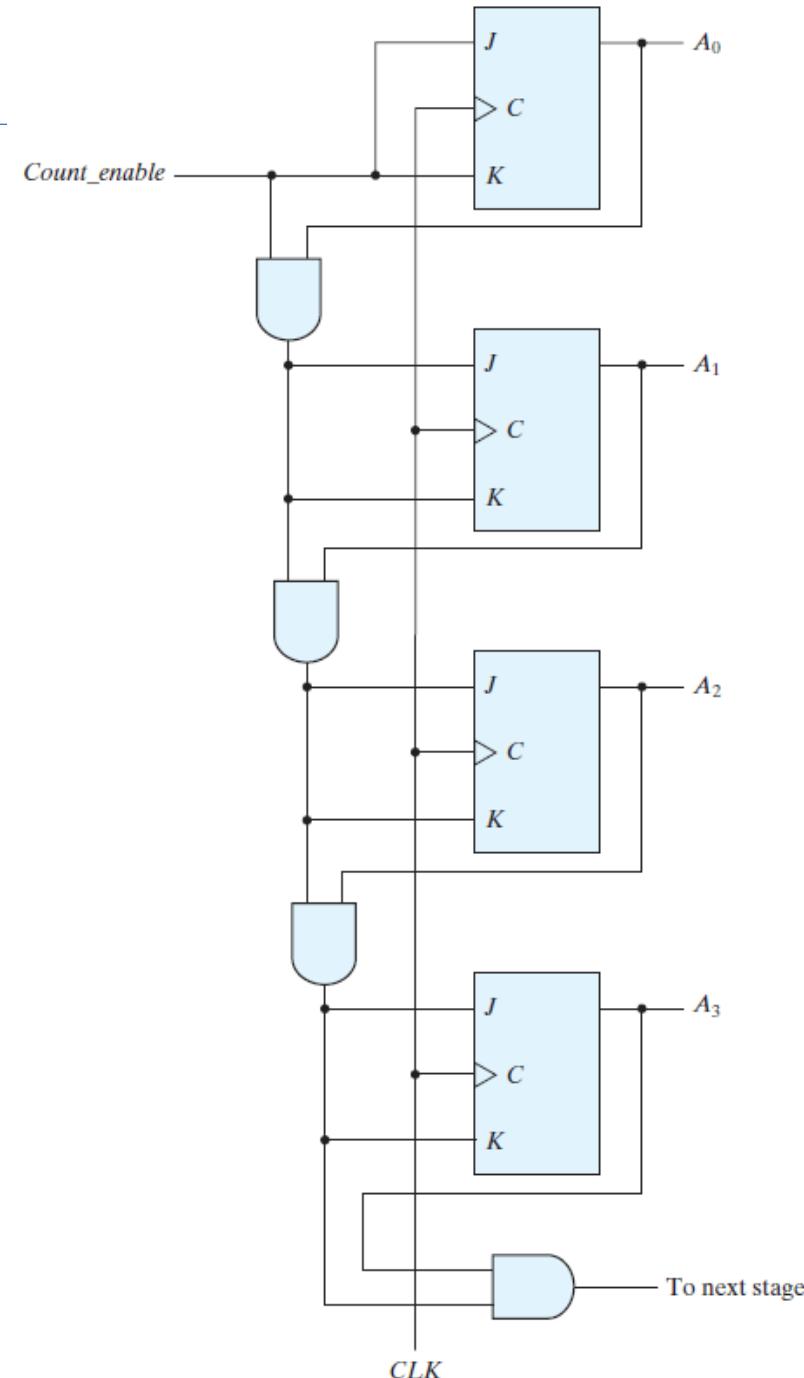


# Synchronous counter - binary

- The design of a synchronous binary counter is very simple
- In a synchronous binary counter, the flip-flop in the least significant position is complemented with every pulse
- *A flip-flop in any other position is complemented when all the bits in the lower significant positions are equal to 1*
- For example, if the present state of a four-bit counter is  $A_3A_2A_1A_0 = 0011$ , the next count is 0100
  - $A_0$  is always complemented
  - $A_1$  is complemented because the present state of  $A_0 = 1$
  - $A_2$  is complemented because the present state of  $A_1A_0 = 11$
  - However,  $A_3$  is not complemented, because the present state of  $A_2A_1A_0 = 011$ , which does not give an all-1's condition

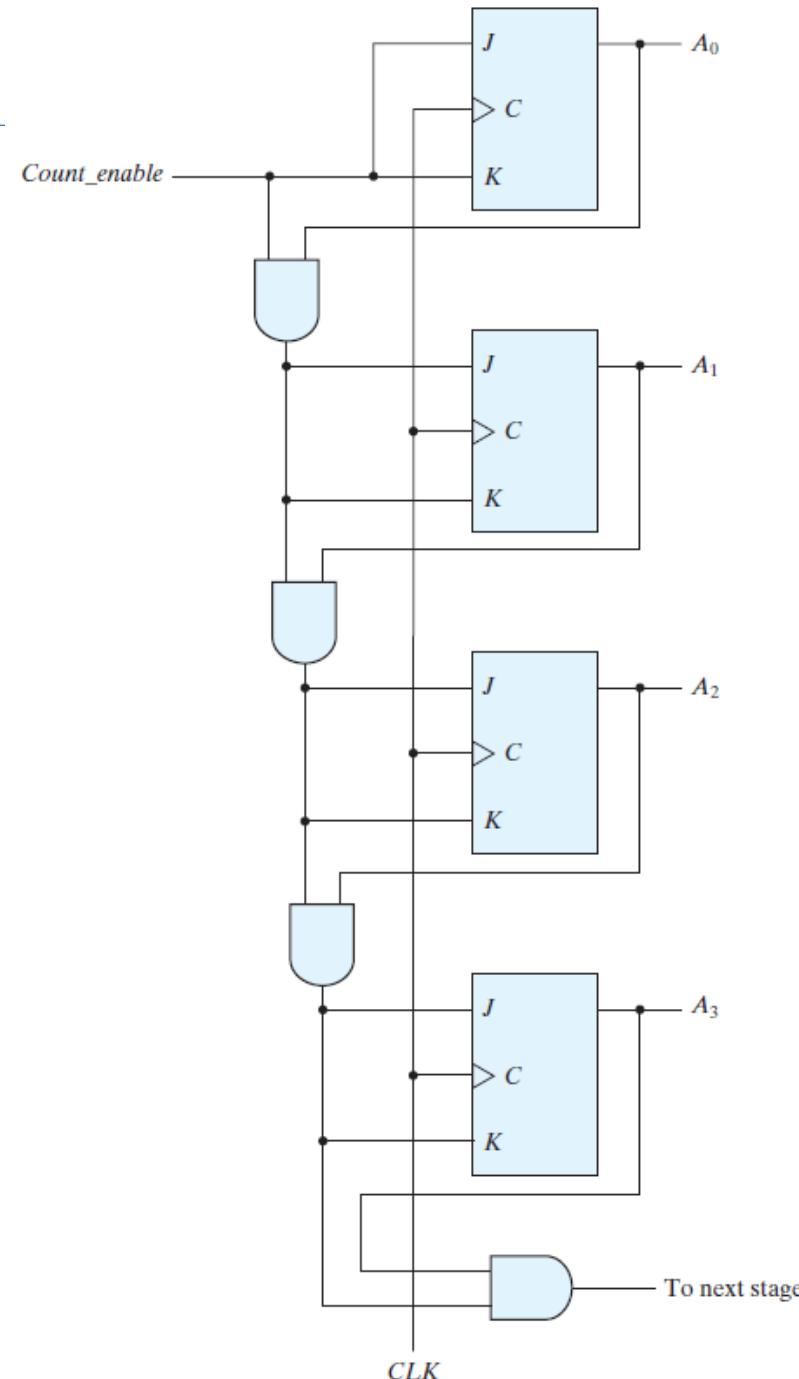
# Synchronous counter - binary

- Synchronous binary counters have a regular pattern and can be constructed with complementing flip-flops and gates
- The regular pattern can be seen from the four-bit counter
- The  $C$  inputs of all flip-flops are connected to a common clock
- The counter is enabled by  $Count\_enable$
- If the enable input is 0, all  $J$  and  $K$  inputs are equal to 0 and the clock does not change the state of the counter
- The first stage,  $A_0$ , has its  $J$  and  $K$  equal to 1 if the counter is enabled
- The other  $J$  and  $K$  inputs are equal to 1 if all previous least significant stages are equal to 1 and the count is enabled



# Synchronous counter - binary

- Note that the flip-flops trigger on the positive edge of the clock
- The synchronous counter can be triggered with either the positive or the negative clock edge
- The complementing flip-flops in a binary counter can be of either the *JK* type, the *T* type, or the *D* type with XOR gates



# Synchronous counter – down counter

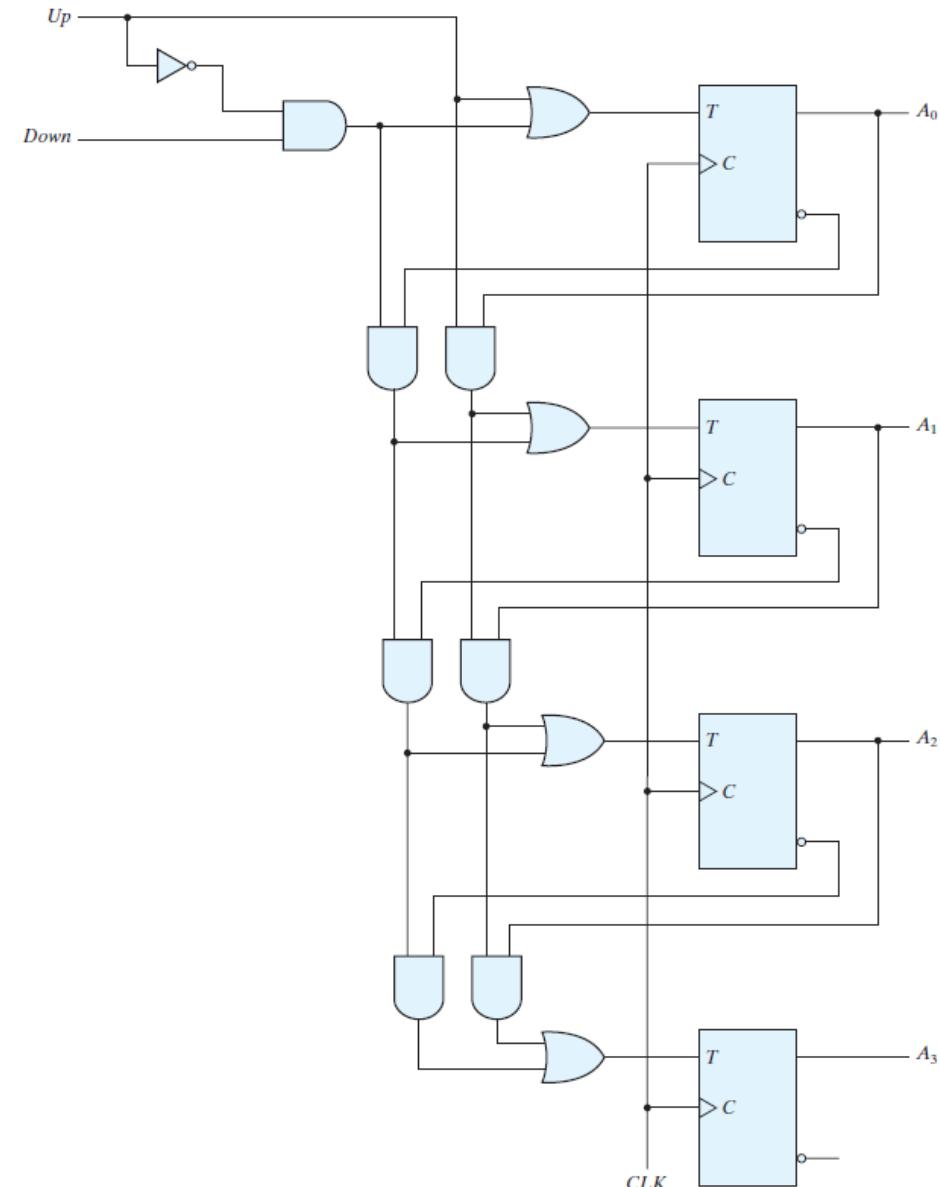
- A synchronous countdown binary counter goes through the binary states in reverse order, from 1111 down to 0000 and back to 1111 to repeat the count
- It is possible to design a countdown counter in the usual manner, but the result is predictable by inspection of the downward binary count
- The bit in the least significant position is complemented with each pulse
- *A bit in any other position is complemented if all lower significant bits are equal to 0*
- For example, the next state after the present state of 0100 is 0011
- The second significant bit is complemented because the first bit was 0
- The third significant bit is complemented because the first two bits were equal to 0
- But the fourth bit does not change, because not all lower significant bits are equal to 0
- A countdown binary counter can be constructed as a regular counter, except that the inputs to the AND gates must come from the complemented outputs, instead of the normal outputs, of the previous flip-flops

# Synchronous counter – up/down counter

- Let us see if we can design a counter that counts up/down based on a control input
- The counter should count up if  $U=1$ ,  $D=0$ ; down if  $U=0$ ,  $D=1$ ; no change if  $U=D=0$ ; and up if  $U=D=1$
- We can cascade this logic to make the synchronous up/down counter

# Synchronous counter – up/down counter

- Let us see if we can design a counter that counts up/down based on an input
- The counter should count up if  $U=1$ ,  $D=0$ ; down if  $U=0$ ,  $D=1$ ; no change if  $U=D=0$ ; and up if  $U=D=1$
- We can cascade this logic to make the synchronous up/down counter



# Synchronous counter – BCD

- BCD counter counts in binary-coded decimal from 0000 to 1001 and back to 0000
- Because of the return to 0 after a count of 9, a BCD counter does not have a regular pattern, unlike a straight binary count
- The input conditions for the 7 flip-flops are obtained from the present- and next-state conditions
- Also shown in the table is an output  $y$ , which is equal to 1 when the present state is 1001
- In this way,  $y$  can enable the count of the next-higher significant decade while the same pulse switches the present decade from 1001 to 0000

Present State				Next State				Output
$Q_8$	$Q_4$	$Q_2$	$Q_1$	$Q_8$	$Q_4$	$Q_2$	$Q_1$	$y$
0	0	0	0	0	0	0	1	0
0	0	0	1	0	0	1	0	0
0	0	1	0	0	0	1	1	0
0	0	1	1	0	1	0	0	0
0	1	0	0	0	1	0	1	0
0	1	0	1	0	1	1	0	0
0	1	1	0	0	1	1	1	0
0	1	1	1	1	0	0	0	0
1	0	0	0	1	0	0	1	0
1	0	0	1	0	0	0	0	1

# Synchronous counter – BCD

- The flip-flop input equations can be simplified by means of maps
- The unused states for minterms 10 to 15 are taken as don't-care terms
- The simplified functions are:

$$TQ_1 = 1$$

$$TQ_2 = Q'_8 Q_1$$

$$TQ_4 = Q_2 Q_1$$

$$TQ_8 = Q_8 Q_1 + Q_4 Q_2 Q_1$$

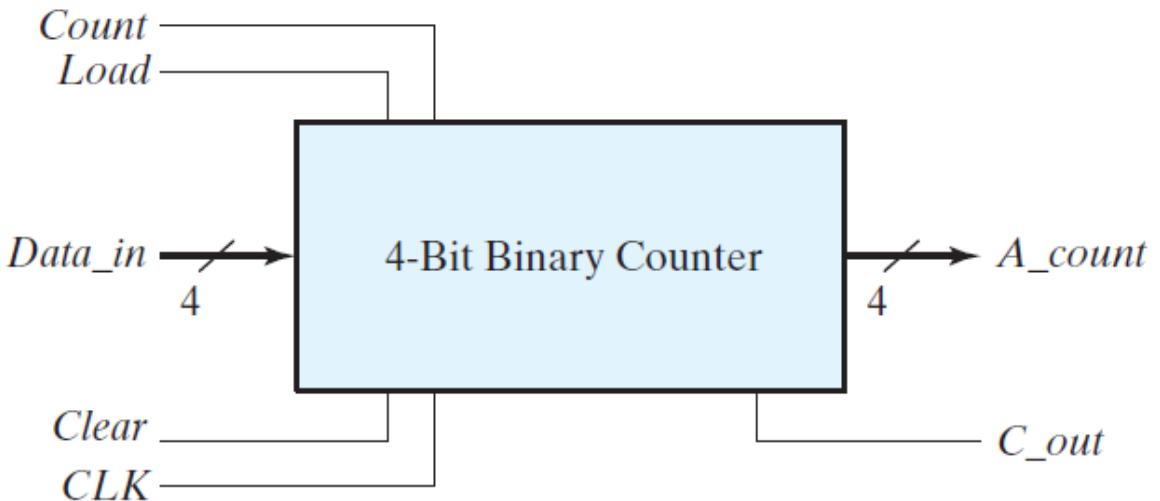
$$y = Q_8 Q_1$$

- The circuit can be made using these expressions

Present State				Next State				Output	Flip-Flop Inputs			
$Q_8$	$Q_4$	$Q_2$	$Q_1$	$Q_8$	$Q_4$	$Q_2$	$Q_1$	$y$	$TQ_8$	$TQ_4$	$TQ_2$	$TQ_1$
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	1
0	0	1	0	0	0	1	1	0	0	0	0	0
0	0	1	1	0	1	0	0	0	0	0	1	1
0	1	0	0	0	1	0	1	0	0	0	0	0
0	1	0	1	0	1	1	0	0	0	0	0	1
0	1	1	0	0	1	1	1	0	0	0	0	0
0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	0	0	1	0	0	0	0	0
1	0	0	1	0	0	0	0	1	1	0	0	1

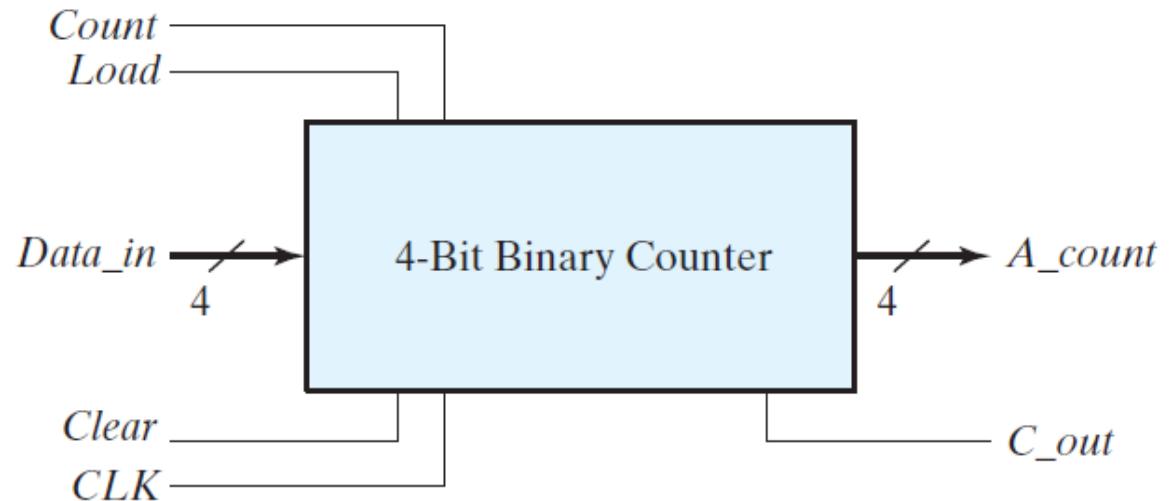
# Synchronous counter – with parallel load

- Counters employed in digital systems quite often require a parallel-load capability for transferring an initial binary number into the counter prior to the count operation
- When equal to 1, the input load control disables the count operation and causes a transfer of data from the four data inputs into the four flip-flops
- If both control inputs are 0, clock pulses do not change the state of the counter



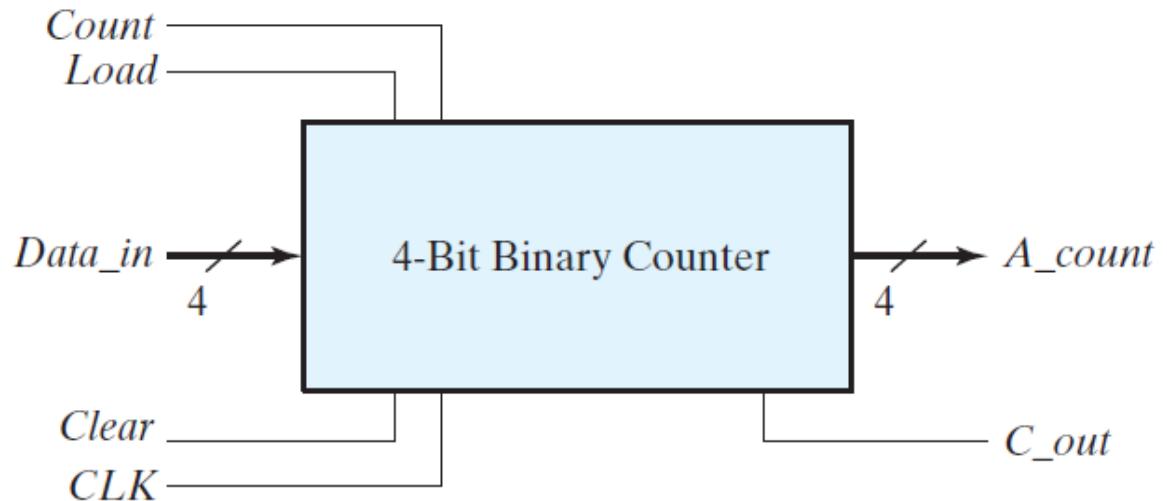
# Synchronous counter – with parallel load

- The four control inputs— *Clear*, *CLK*, *Load*, and *Count* —determine the next state
- The *Clear* input is asynchronous and, when equal to 0, causes the counter to be cleared regardless of the presence of clock pulses or other inputs
- With the *Load* and *Count* inputs both at 0, the outputs do not change, even when clock pulses are applied
- A *Load* input of 1 causes a transfer from inputs  $I_0 - I_3$  into the register during a positive edge of *CLK*
- The *Load* input must be 0 for the *Count* input to control the operation of the counter

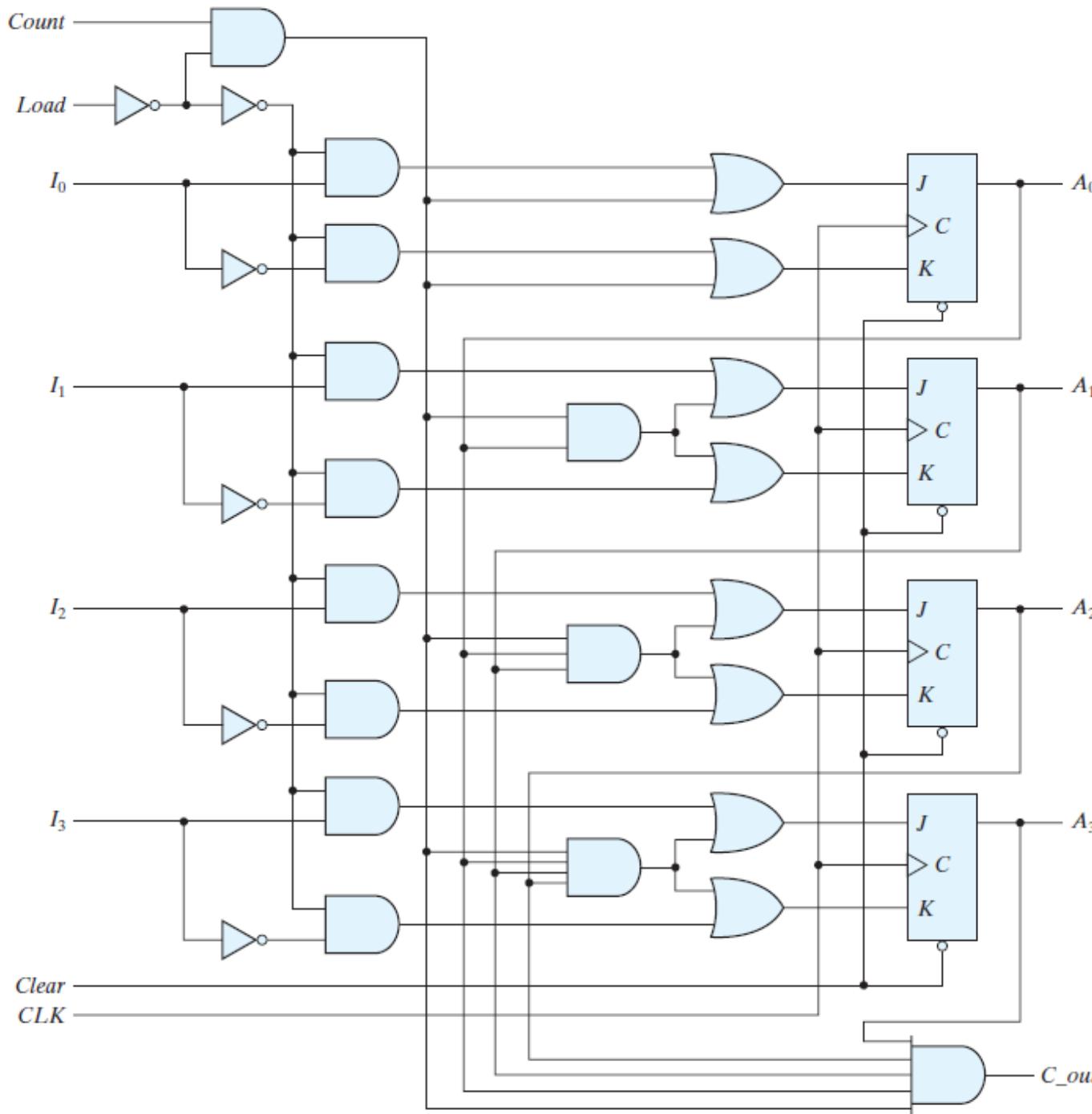


Clear	CLK	Load	Count	Function
0	X	X	X	Clear to 0
1	↑	1	X	Load inputs
1	↑	0	1	Count next binary state
1	↑	0	0	No change

# Synchronous counter – with parallel load



<b>Clear</b>	<b>CLK</b>	<b>Load</b>	<b>Count</b>	<b>Function</b>
0	X	X	X	Clear to 0
1	↑	1	X	Load inputs
1	↑	0	1	Count next binary state
1	↑	0	0	No change



<b>Clear</b>	<b>CLK</b>	<b>Load</b>	<b>Count</b>	<b>Function</b>
0	X	X	X	Clear to 0
1	↑	1	X	Load inputs
1	↑	0	1	Count next binary state
1	↑	0	0	No change

# Lecture 24 – Memory architecture 1

Chapter 7

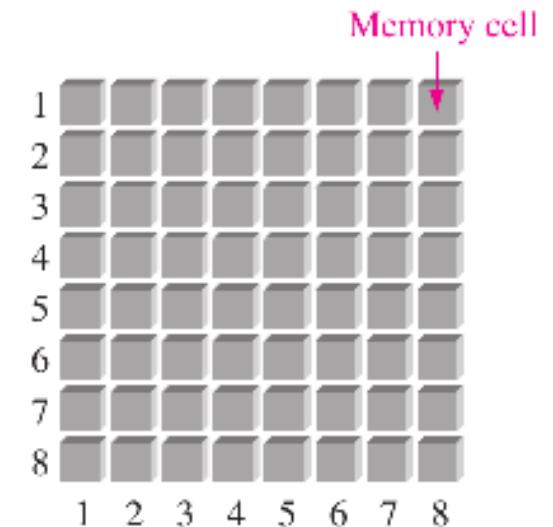
# Memory

---

- Two major types of semiconductor memories are **random-access memory (RAM)** and **read-only memory (ROM)**.
- RAM is a type of memory in which all contents are accessible in an equal amount of time and can be selected in any order for a read or write operation.
  - RAM has both read and write capability.
  - RAMs lose stored data when the power is turned off, they are *volatile memories*.
- ROM is a type of memory in which data are stored permanently or semi-permanently.
  - Data can be read from a ROM, but there is no write operation as in the RAM.
  - The ROM, like the RAM, is a random-access memory but the term RAM traditionally means a random-access read/write memory
  - Because ROMs retain stored data even if power is turned off, they are *nonvolatile* memories.

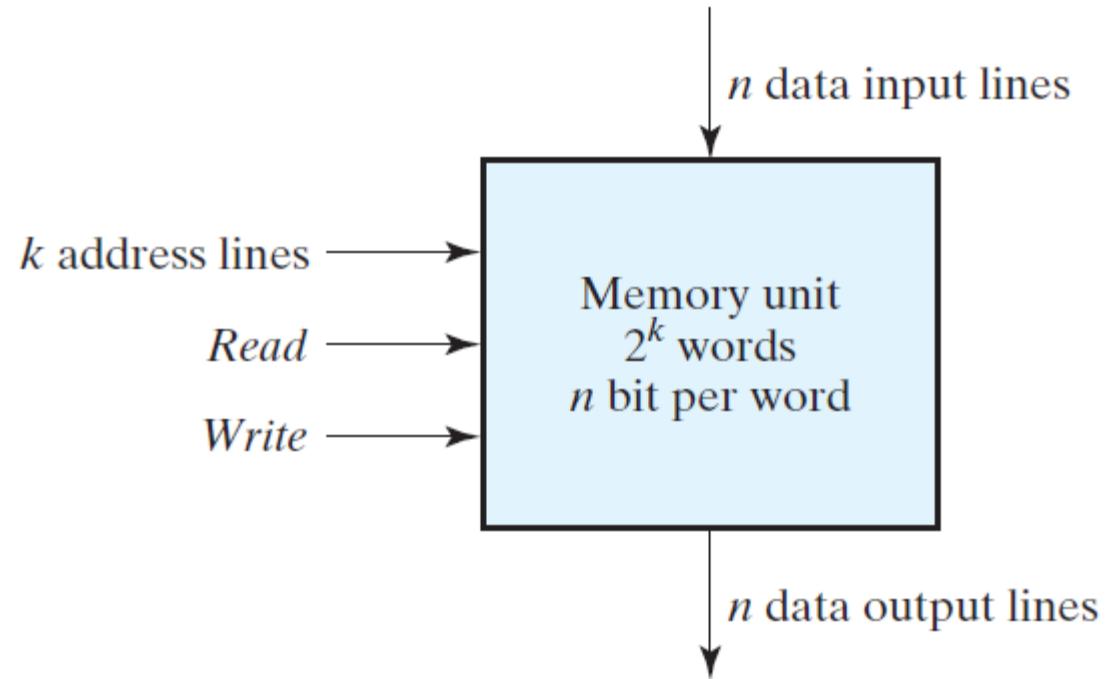
# Random Access Memory (RAM)

- A memory unit is a collection of storage cells, together with associated circuits needed to transfer information into and out of a device
- The architecture of memory is such that information can be selectively retrieved from any of its internal locations
- The time it takes to transfer information to or from any desired random location is always the same—hence the name *random-access memory, abbreviated RAM*
- In contrast, the time required to retrieve information that is stored on magnetic tape, optical disks, etc, depends on the location of the data
- A memory unit stores binary information in groups of bits called *words*
- *A word in memory is an entity of bits that move in and out of storage as a unit*
- Most computer memories use words that are multiples of 8 bits in length
- The capacity of a memory unit is usually stated as the total number of bytes that the unit can store



# Random Access Memory (RAM)

- Communication between memory and its environment is achieved through **data input** and **output lines**, **address selection lines**, and **control lines** that specify the direction of transfer
- The  $n$  data input lines provide the information to be stored in memory, and the  $n$  data output lines supply the information coming out of memory
- The  $k$  address lines specify the particular word chosen among the many available
- The two control inputs specify the direction of transfer desired: The **Write** input causes binary data to be transferred into the memory, and the **Read** input causes binary data to be transferred out of memory
- The **address lines** select one particular word

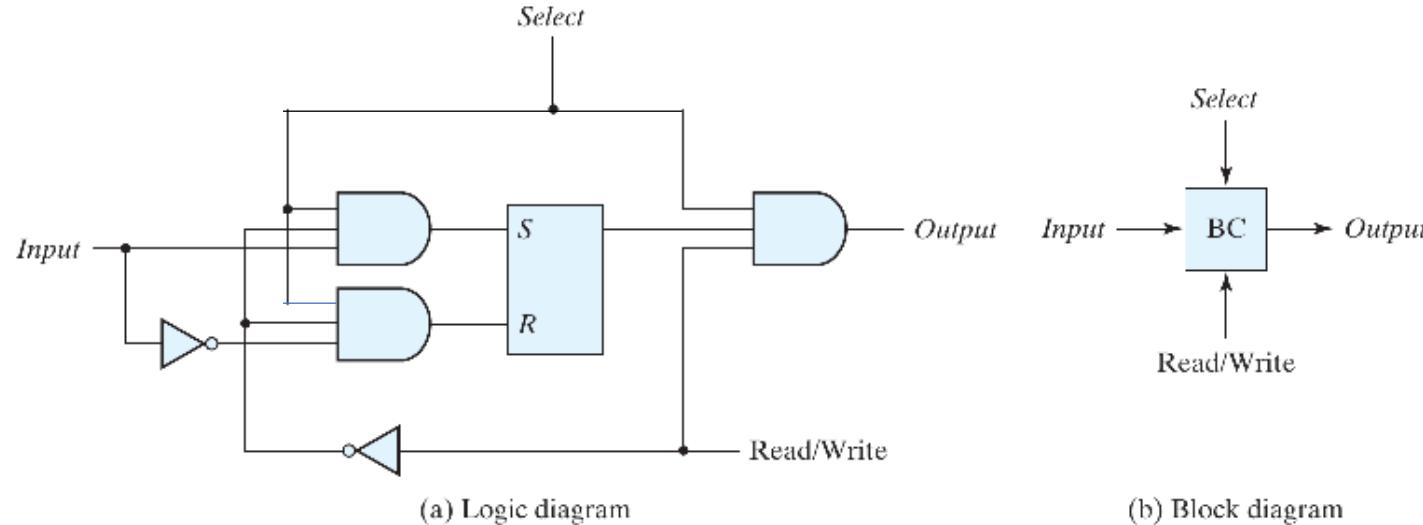


# Random Access Memory (RAM)

- Consider, for example, a memory unit with a capacity of 1K words of 16 bits each
- Since  $1K = 1,024 = 2^{10}$  and 16 bits constitute two bytes, we can say that the memory can accommodate  $2,048 = 2\text{Kb}$
- The words are recognized by their decimal address from 0 to 1,023
- The equivalent binary address consists of 10 bits
- A word in memory is selected by its binary address
- When a word is read or written, the memory operates on all 16 bits as a single unit*

Memory address	Binary	Decimal	Memory content
	0000000000	0	1011010101011101
	0000000001	1	1010101110001001
	0000000010	2	0000110101000110
		⋮	⋮
		⋮	⋮
	1111111101	1021	1001110100010100
	1111111110	1022	0000110100011110
	1111111111	1023	1101111000100101

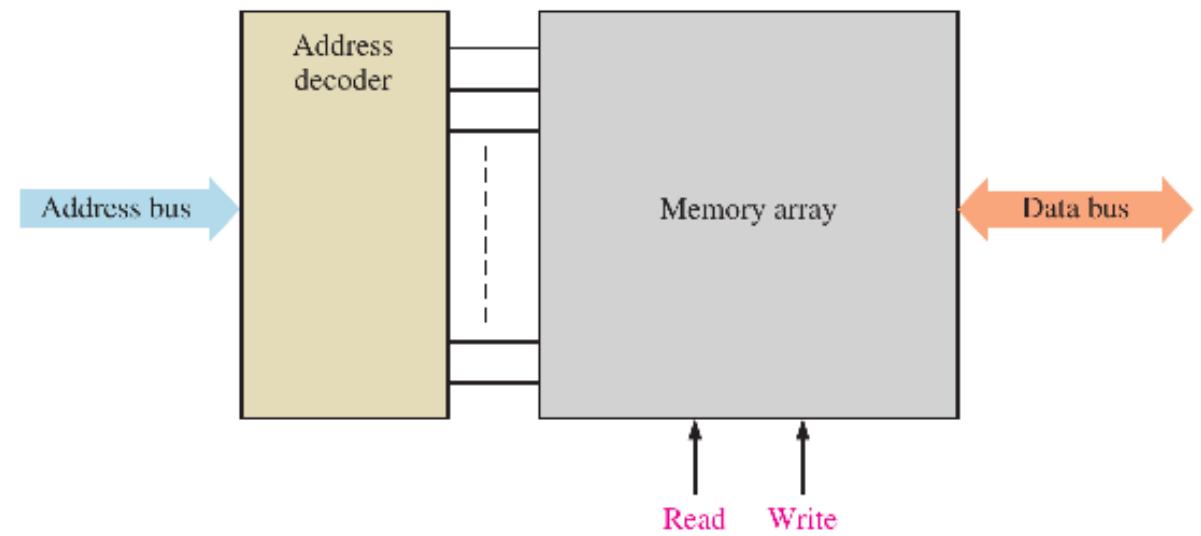
# Random Access Memory (RAM) – internal structure



- The internal construction of a RAM of  $m$  words and  $n$  bits per word consists of  $m * n$  binary storage cells and associated decoding circuits for selecting individual words
- The binary storage cell (SR latch) is the basic building block of a memory unit
- The storage part of the cell is modeled by an *SR* latch with associated gates to form a *D* latch
- *Note that this is not a D flip-flop*

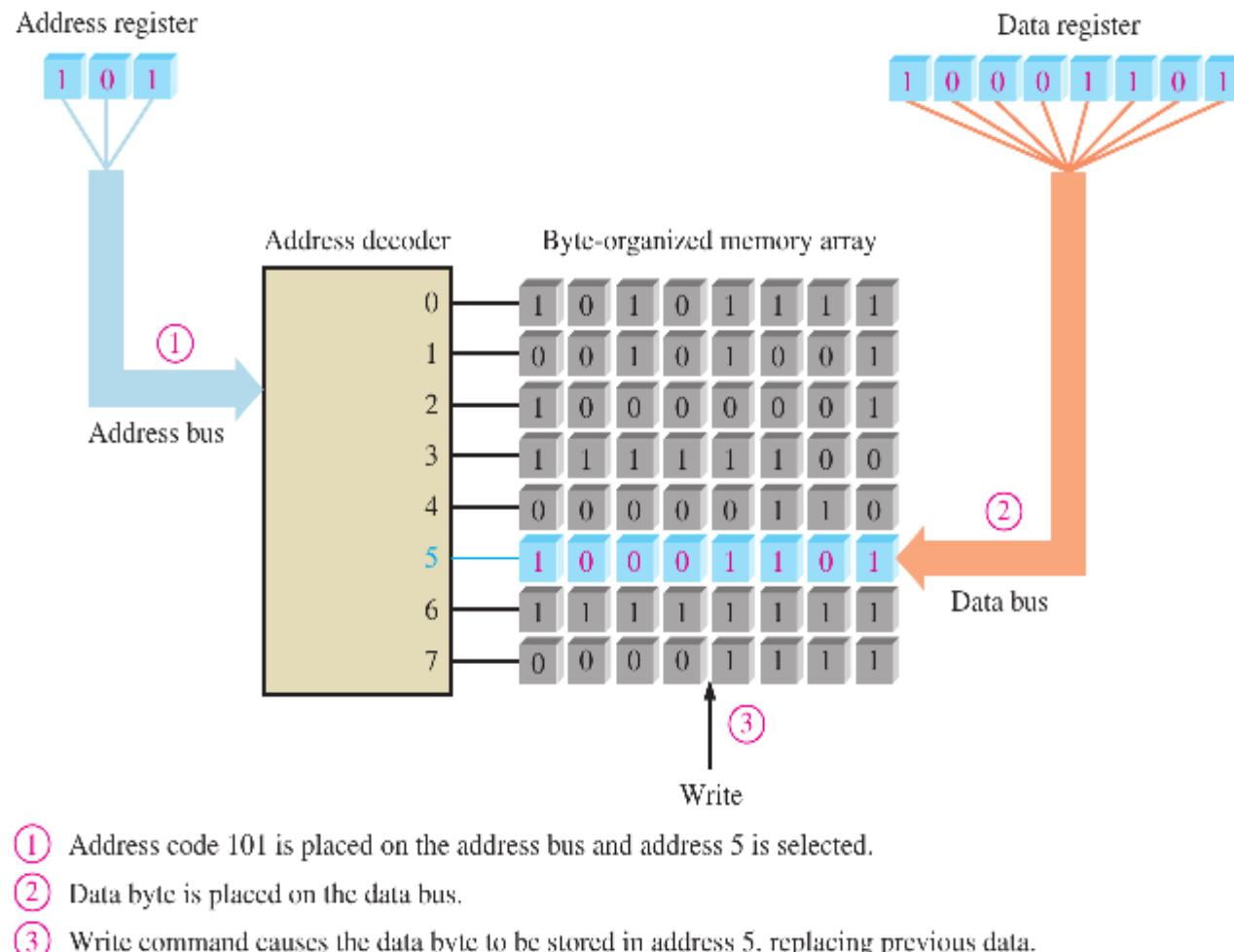
# Write and Read operations

- The two operations that RAM can perform are the write and read operations
- The write signal specifies a transfer-in operation and the read signal specifies a transfer-out operation
- On accepting one of these control signals, the internal circuits inside the memory provide the desired operation



# Write operation

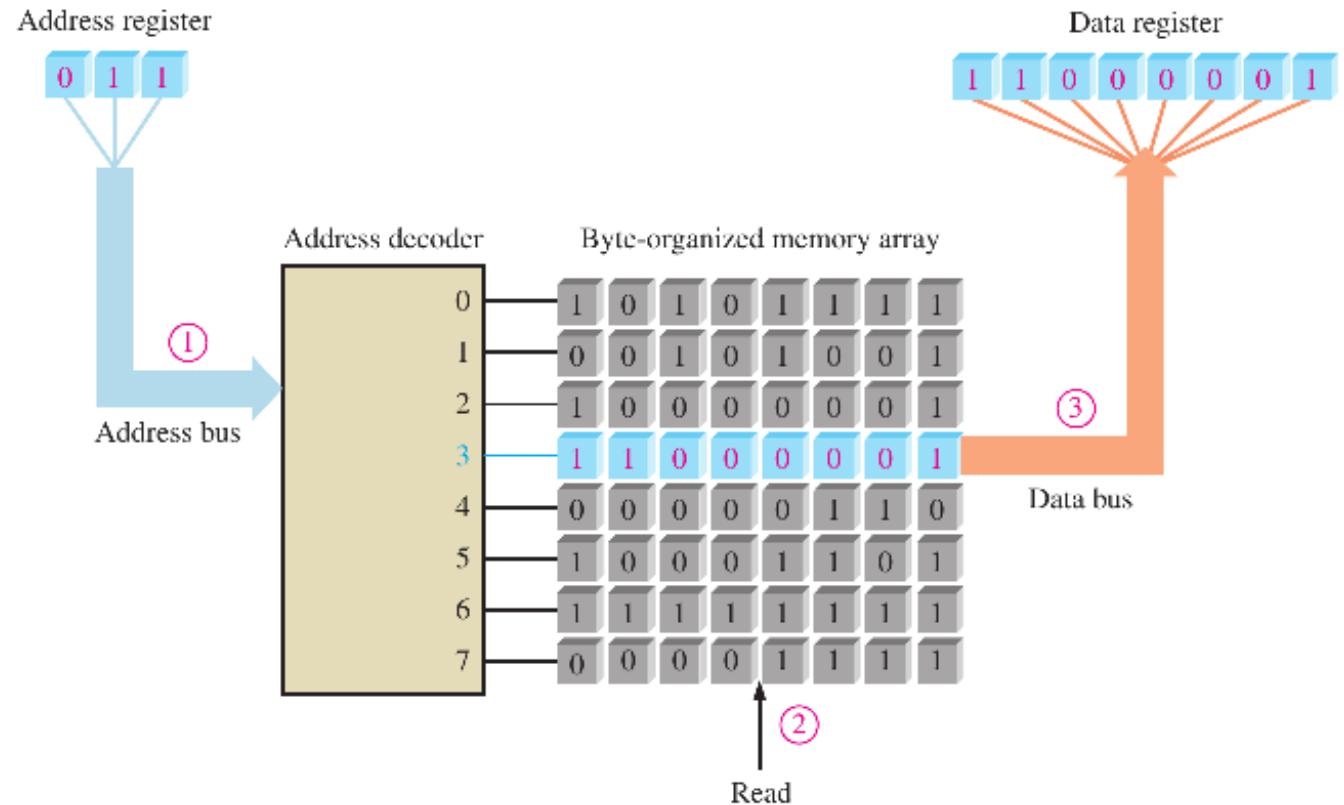
- The steps that must be taken for the purpose of transferring a new word to be stored into memory are as follows:
  - Apply the binary address of the desired word to the address lines.
  - Activate the *write* input.
  - Apply the data bits that must be stored in memory to the data input lines.
- The memory unit will then take the bits from the input data lines and store them in the word specified by the address lines



# Read operation

- The steps that must be taken for the purpose of transferring a stored word out of memory are as follows:

1. Apply the binary address of the desired word to the address lines
2. The data shows up on the output lines after a certain delay from the application of the stable address (selecting the word)



- ① Address code 011 is placed on the address bus and address 3 is selected.
- ② Read command is applied.
- ③ The contents of address 3 is placed on the data bus and shifted into data register. The contents of address 3 is not erased by the read operation.

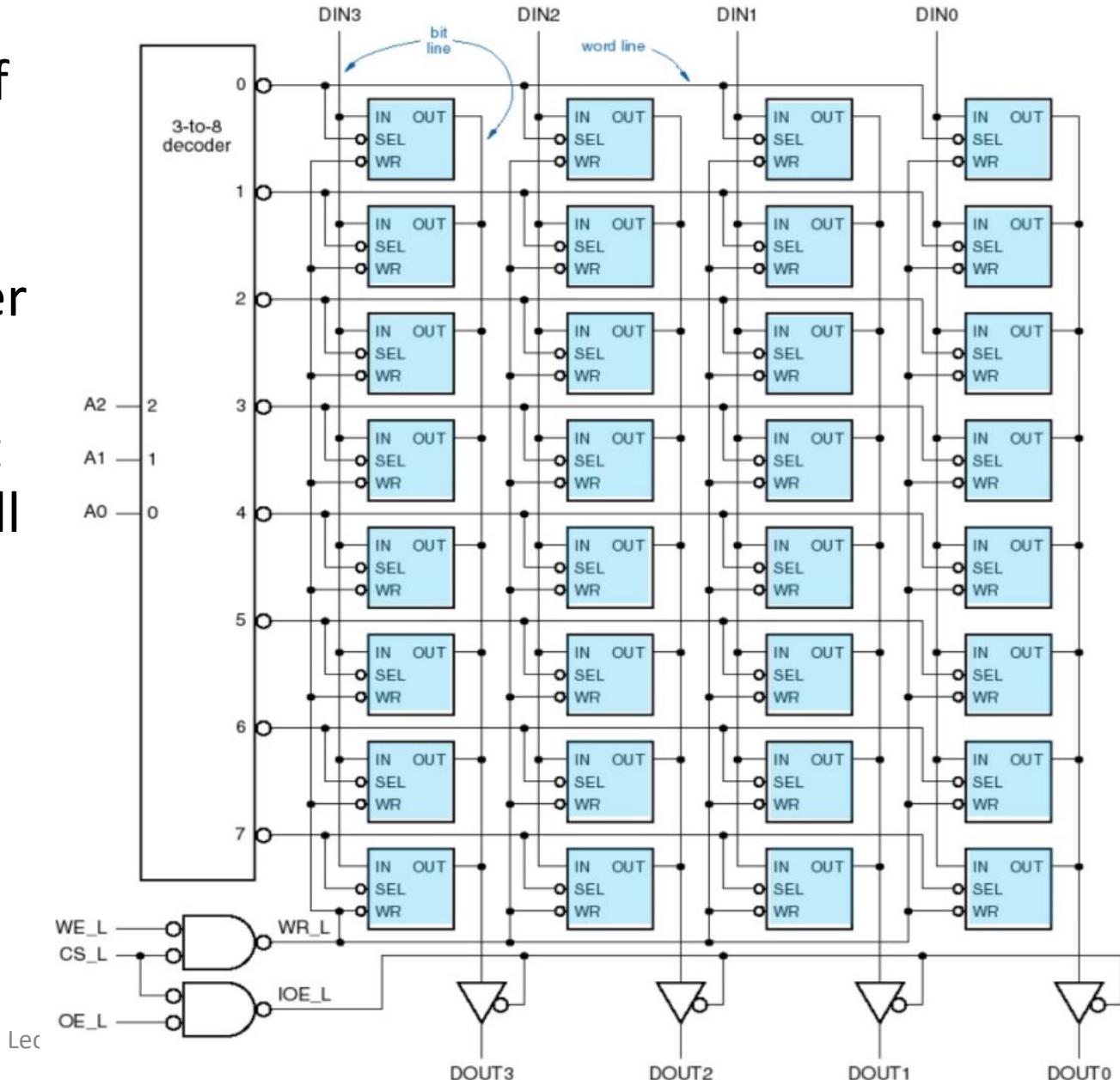
# Memory design

- The design of a memory will consist of a decoder circuit to select a particular “**word line**” based on the address
- The write enable, chip select and other inputs are common to all the latches
- The “**bit line**” determines what the bit at a particular position in the word will be

WE – write enable

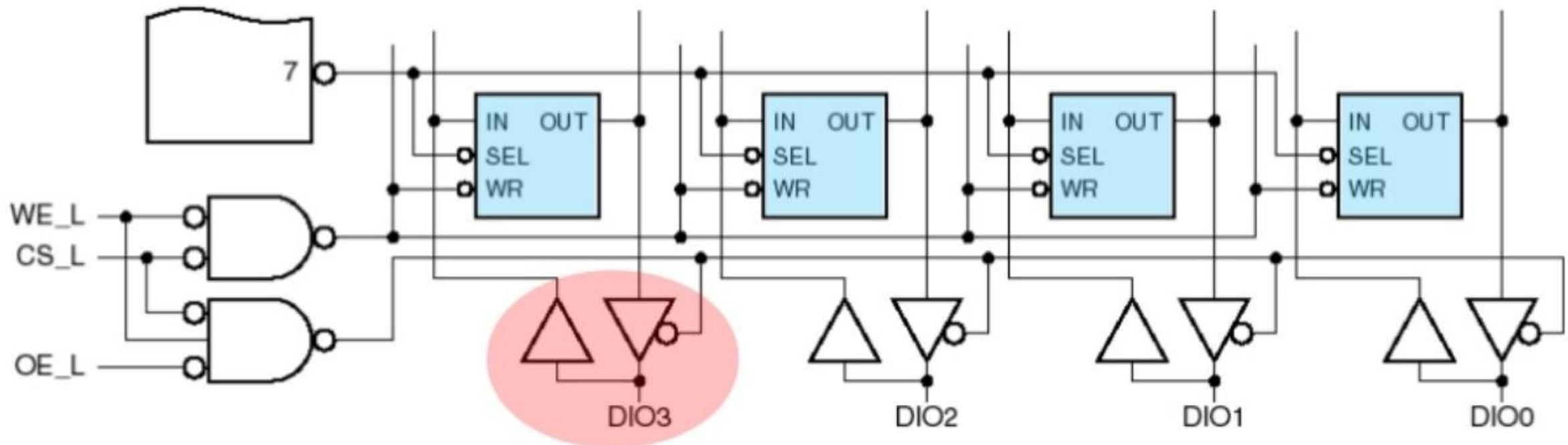
CS – chip select

OE – output enable



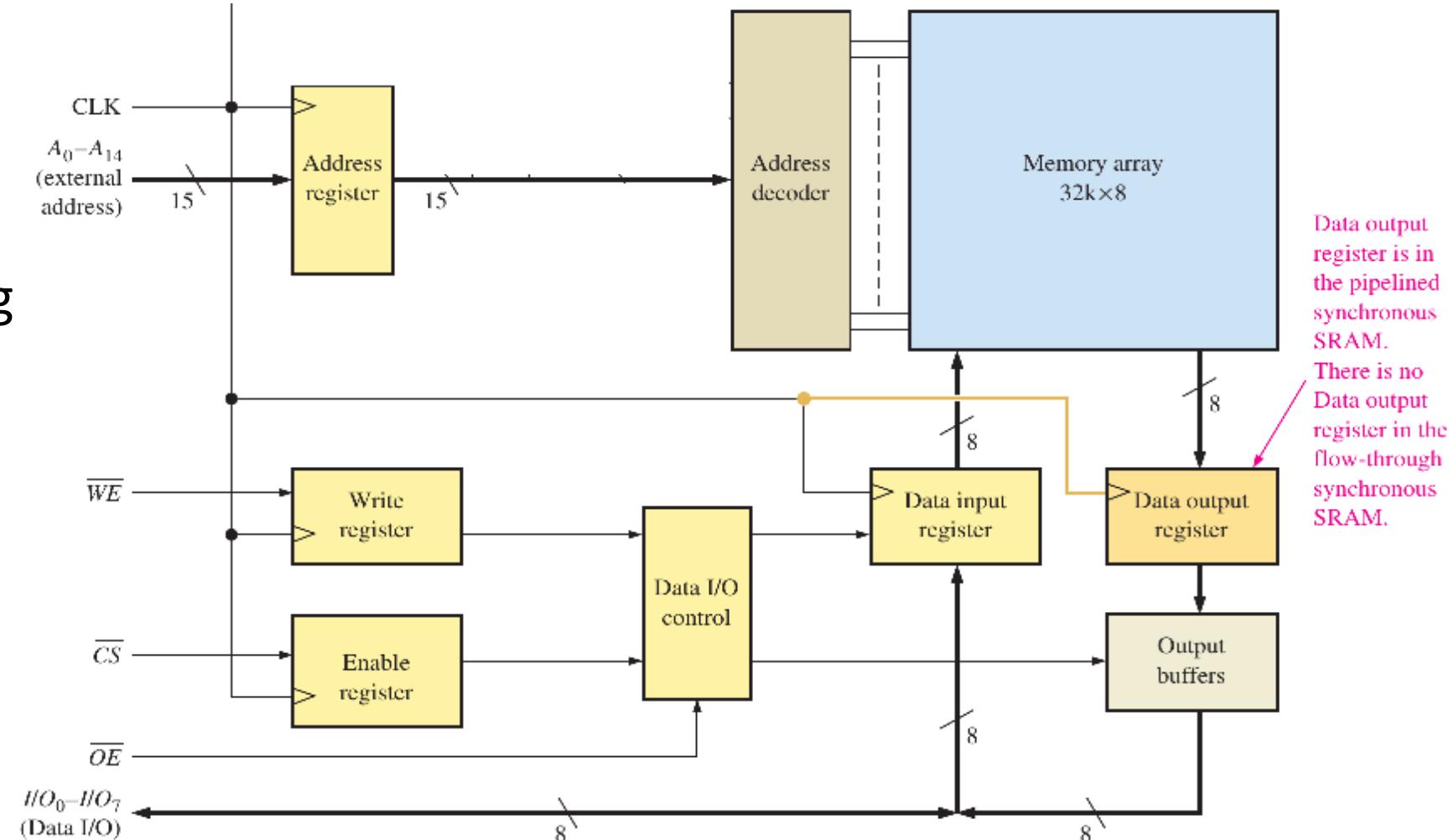
# Memory design

- We can modify the output such that we have the same lines for both input and output case
- This is very common both inside the processor (internal buses) and outside the processor (such as digital IO or GPIO)

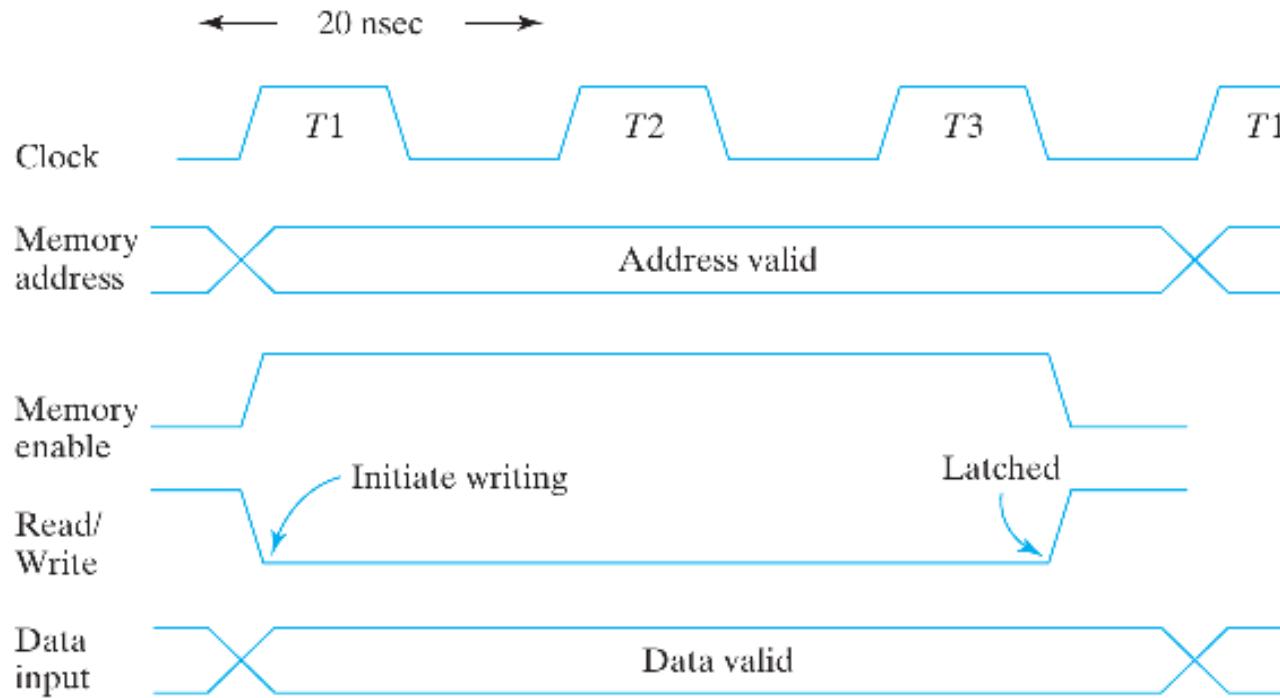


# Synchronous RAMs

- We can modify the input output behaviour of the RAM array to make it synchronous
- We can easily do it using registers at appropriate points
- We can make the output *synchronous* or *flow through*
- They are all controlled using a common clock

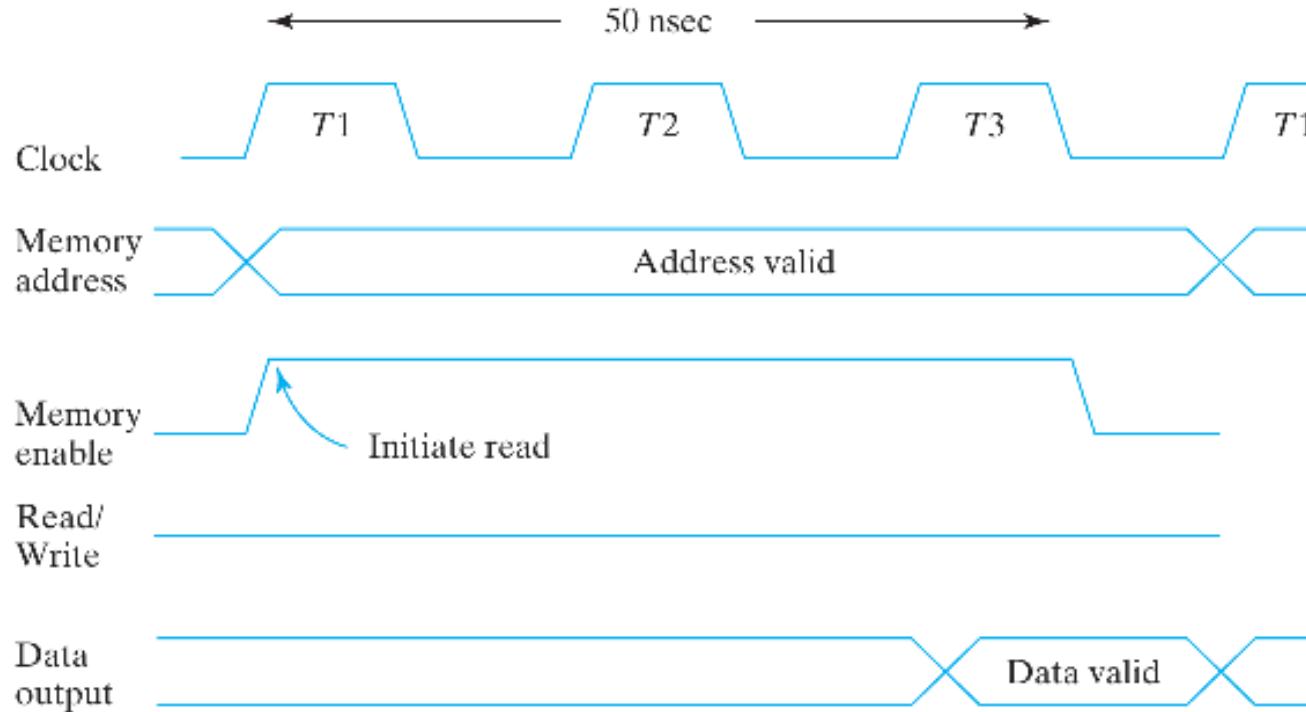


# Memory read write timing



Write cycle

# Memory read write timing



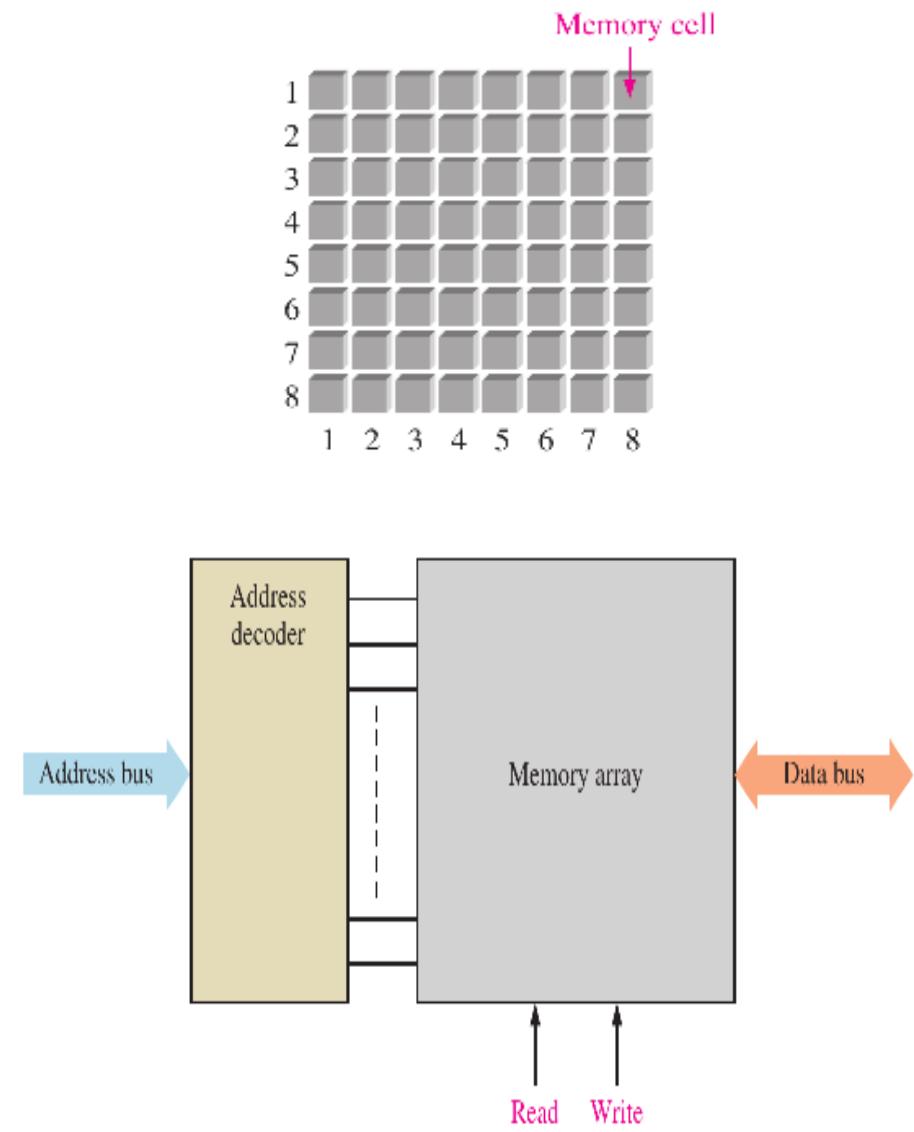
Read cycle

# Lecture 25 – Memory architecture 2

Chapter 7

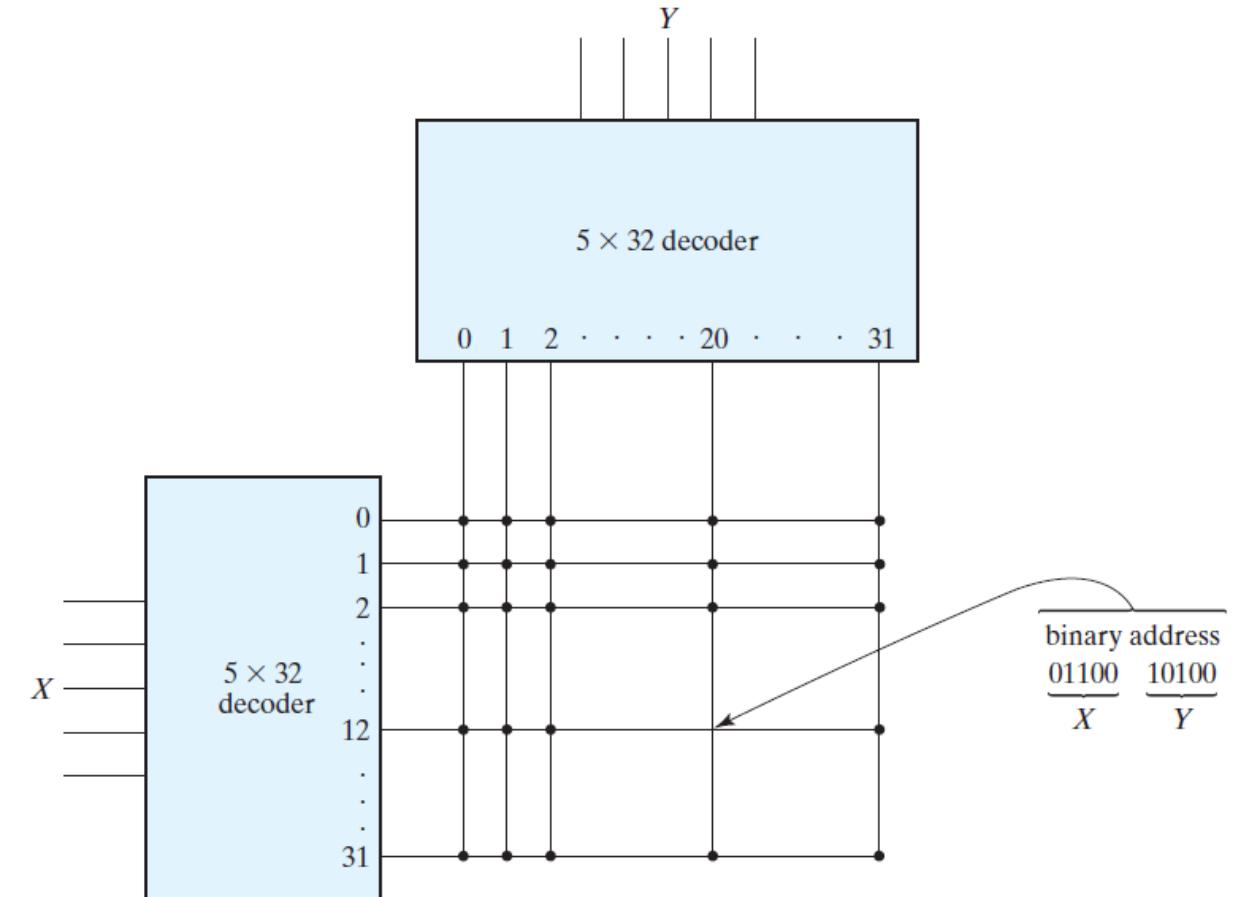
# Address decoding: coincident decoding

- A decoder with  $k$  inputs and  $2^k$  outputs requires  $2^k$  AND gates with  $k$  inputs per gate
- The total number of gates and the number of inputs per gate can be reduced by employing *two decoders in a two-dimensional selection scheme*
  - The basic idea in two-dimensional decoding is to arrange the memory cells in an array that is close as possible to square
  - In this configuration, two  $k/2$ -input decoders are used instead of one  $k$ -input decoder
  - One decoder performs the row selection and the other the column selection in a two-dimensional matrix configuration



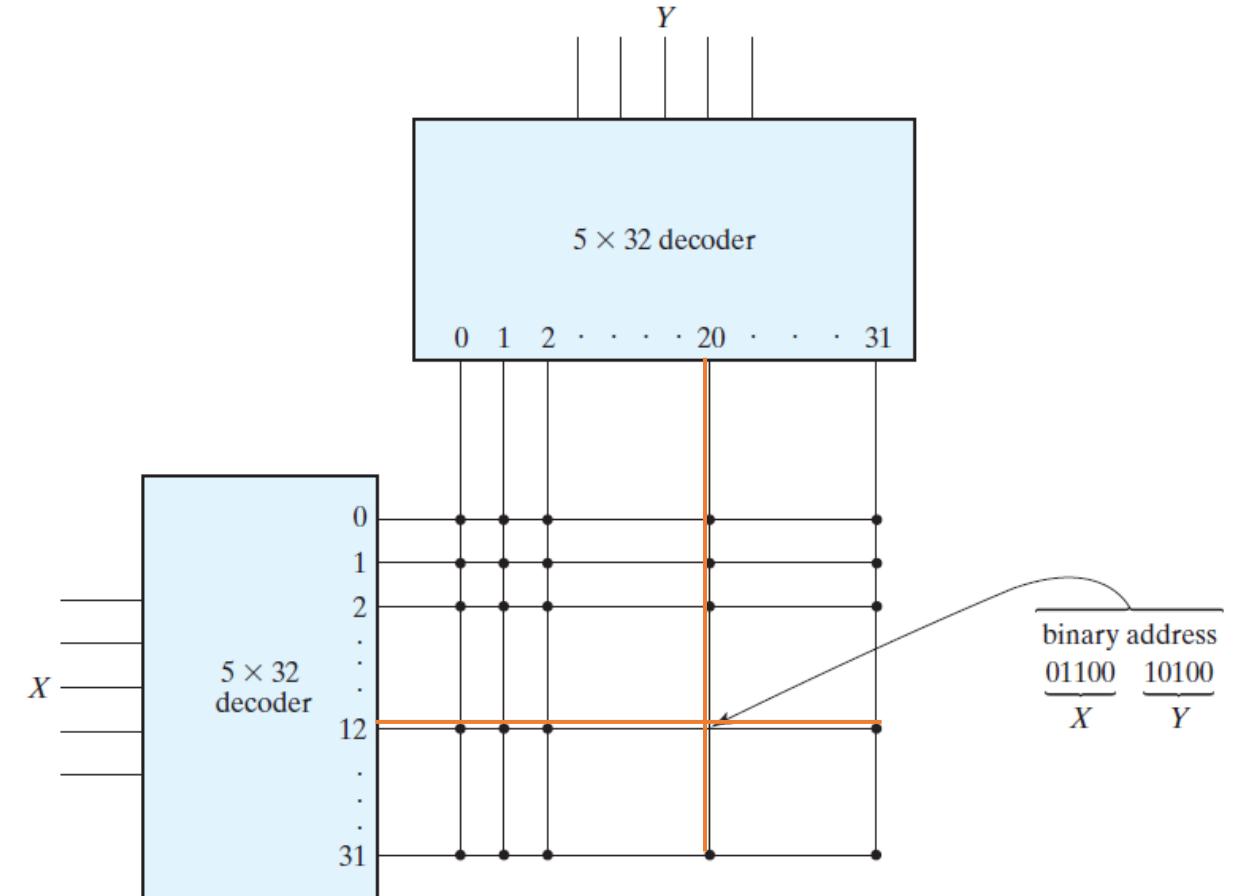
# Coincident decoding

- For example, instead of using a single  $10 * 1,024$  decoder, we use two  $5 * 32$  decoders
  - With the single decoder, we would need **1,024 AND gates with 10 inputs in each**
  - In the two-decoder case, **we need 64 AND gates with 5 inputs in each**
- The five most significant bits of the address go to input  $X$  and the five least significant bits go to input  $Y$
- *Each word within the memory array is selected by the coincidence of one  $X$  line and one  $Y$  line*
- Thus, each word in memory is selected by the coincidence between 1 of 32 rows and 1 of 32 columns, for a total of 1,024 words



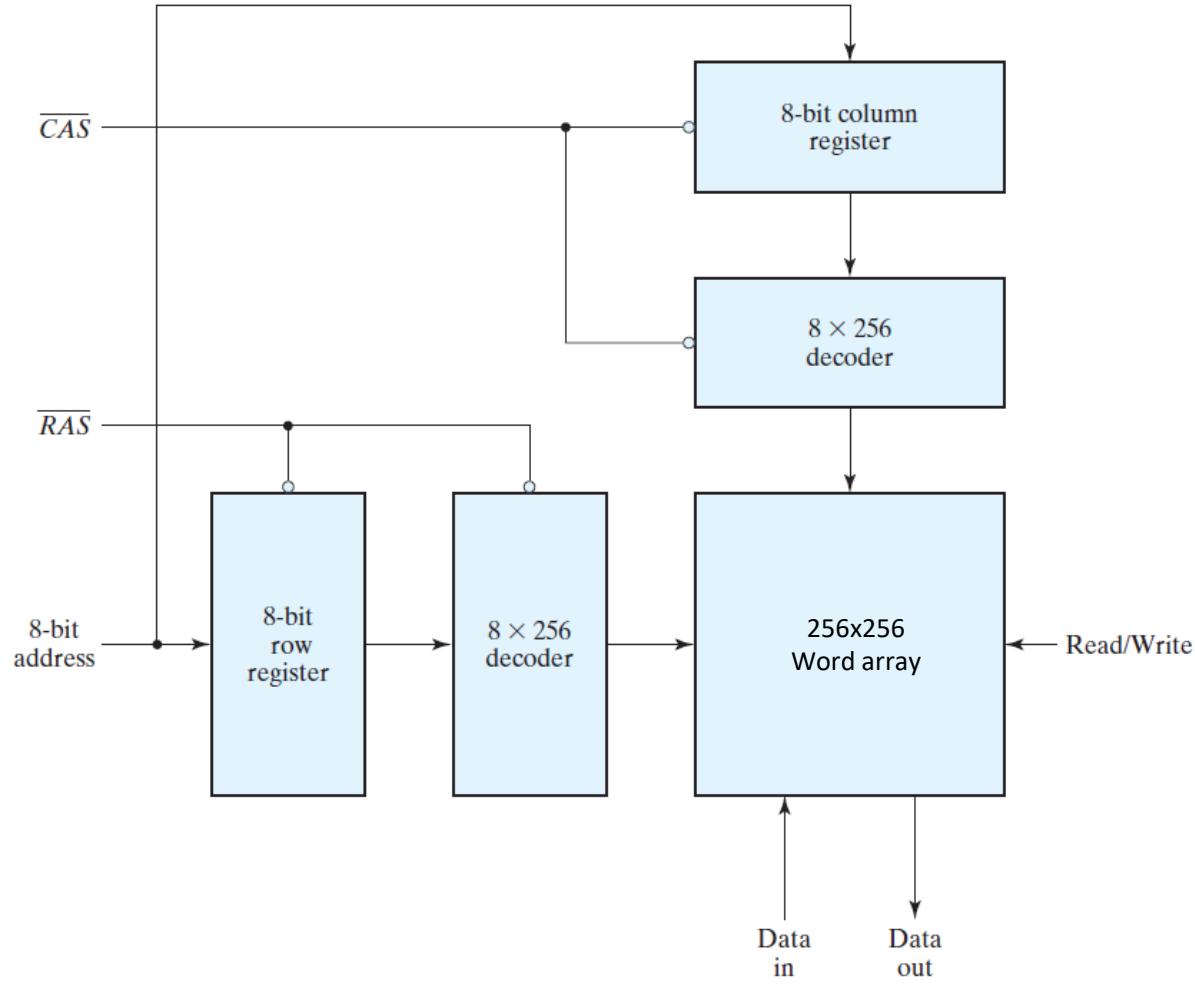
# Coincident decoding

- As an example, consider the word whose address is 404. The 10-bit binary equivalent of 404 is 01100 10100. This makes  $X = 01100$  (binary 12) and  $Y = 10100$  (binary 20)
- The  $n$ -bit word that is selected lies in the  $X$  decoder output number 12 and the  $Y$  decoder output number 20
- All the bits of the word are selected for reading or writing



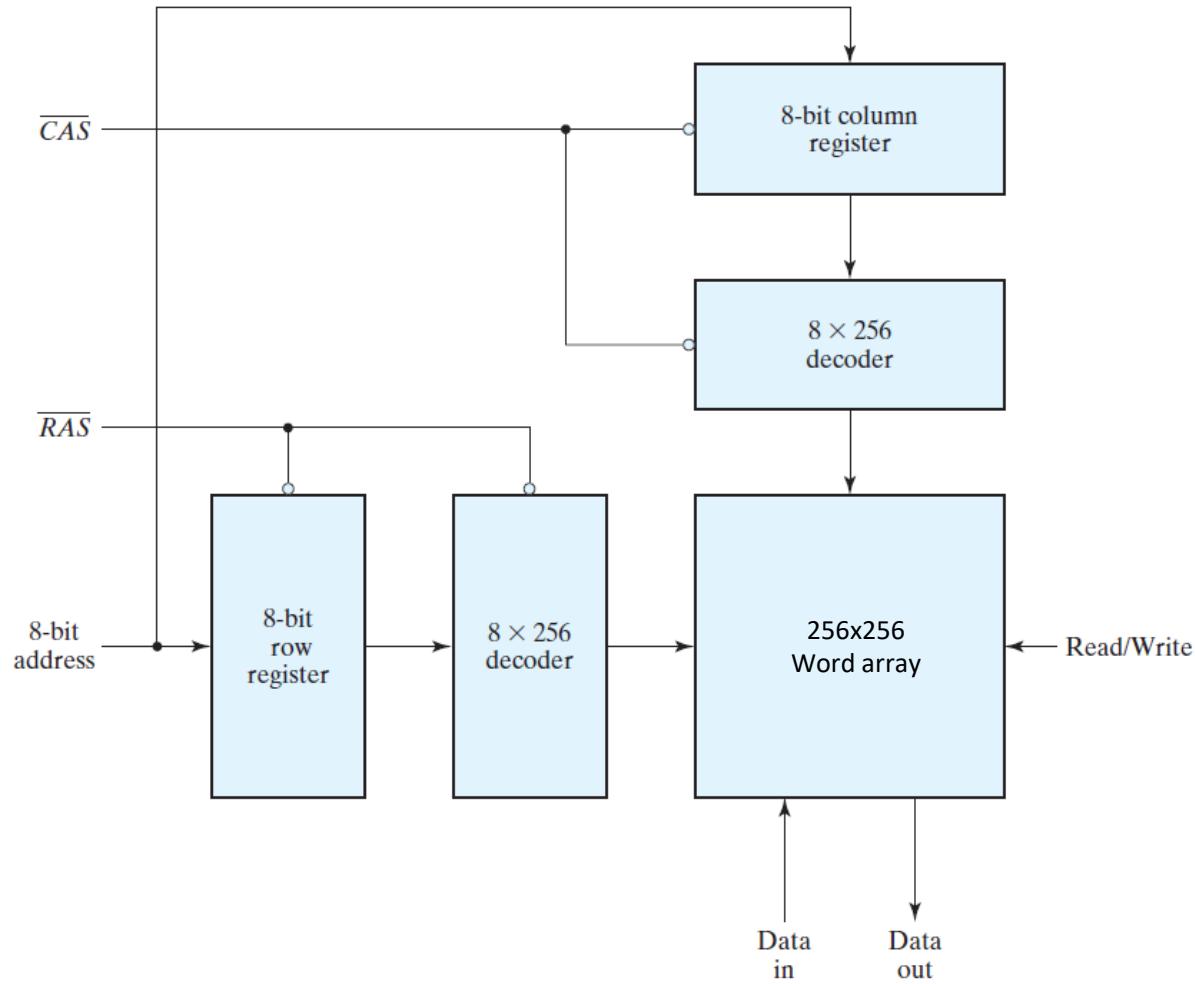
# Address multiplexing

- To reduce the number of pins in the IC package, designers utilize address multiplexing whereby one set of address input pins accommodates the address components
- In a two-dimensional array, the address is applied in two parts at different times, with the row address first and the column address second
- Since the same set of pins is used for both parts of the address, the size of the package is decreased significantly



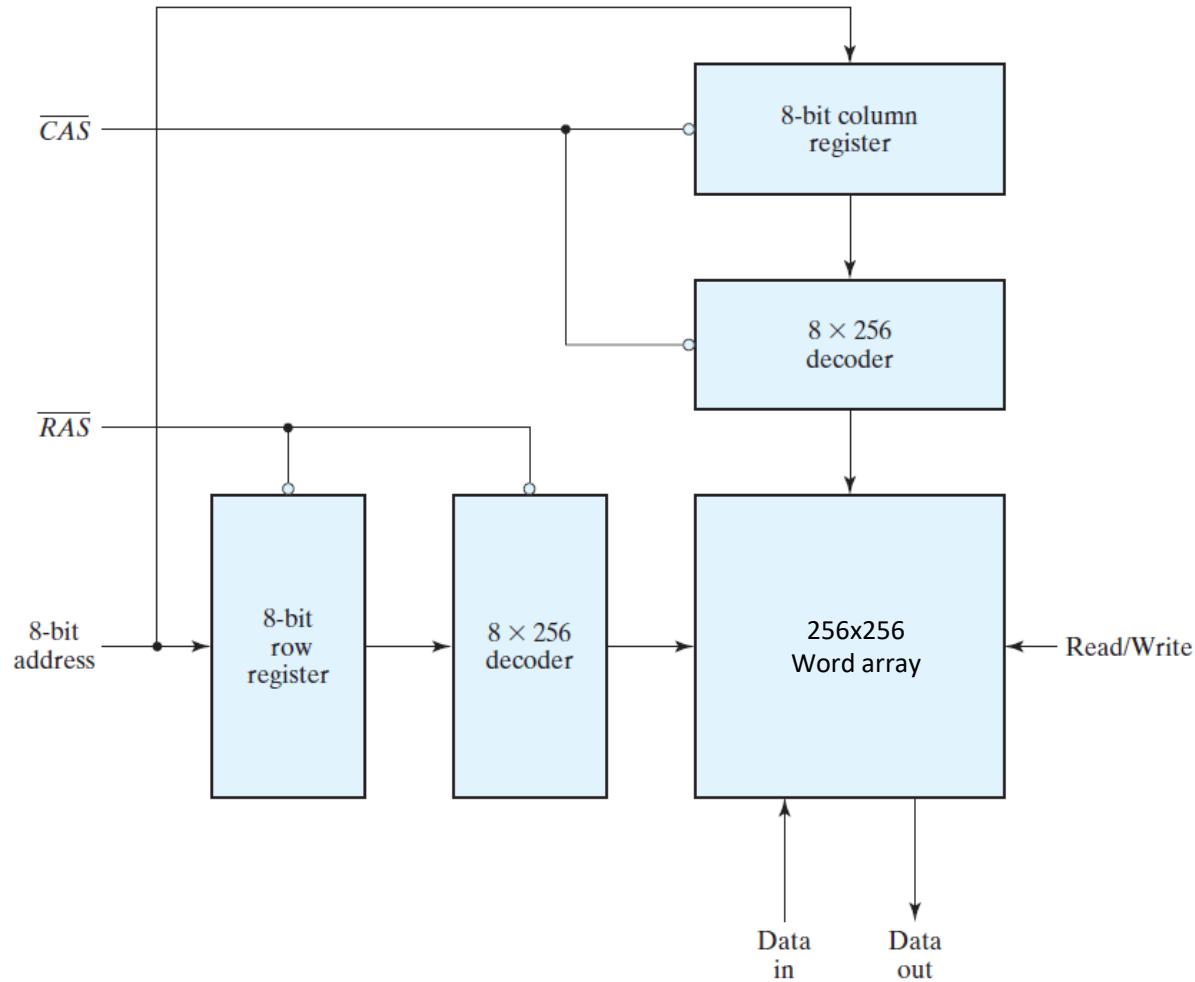
# Address multiplexing

- We will use a 64K-word memory to illustrate the address-multiplexing idea
- The memory consists of a two-dimensional array of cells arranged into 256 rows by 256 columns, for a total of  $2^8 * 2^8 = 2^{16} = 64K$  words
- There is a single data input line, a single data output line, and a read/write control, as well as an eight-bit address input and two address strobes, the latter included for enabling the row and column address into their respective registers
- The row address strobe (RAS) enables the eight-bit row register, and the column address strobe (CAS) enables the eight-bit column register



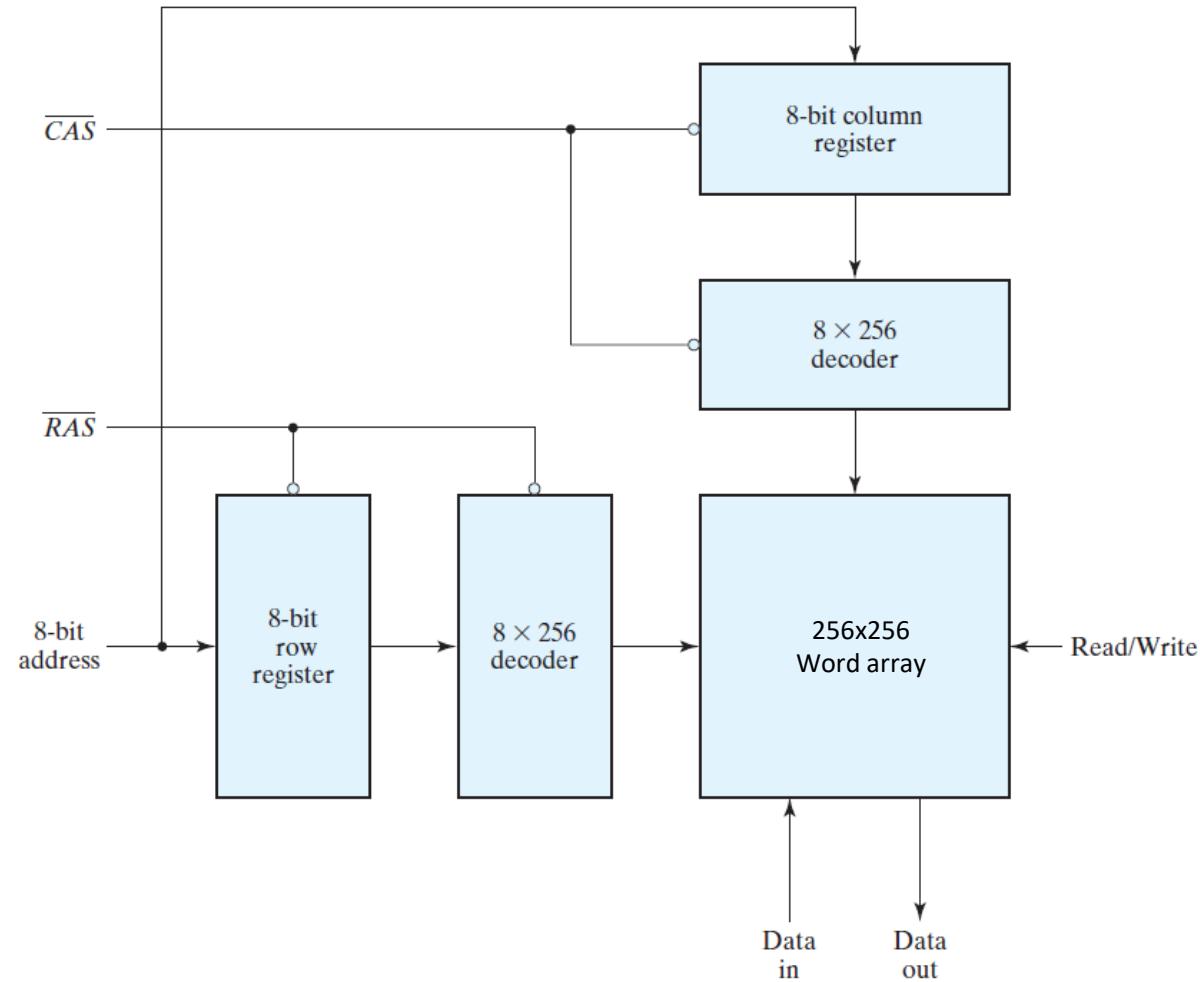
# Address multiplexing

- The 8-bit row address is applied to the address inputs and RAS is activated
- This loads the row address into the row address register
- RAS also enables the row decoder so that it can decode the row address and select one row of the array



# Address multiplexing

- The 8-bit column address is again applied to the address inputs, and CAS activated
- This transfers the column address into the column register and enables the column decoder
- Now the two parts of the address are in their respective registers, the decoders have decoded them to select the one cell corresponding to the row and column address, and a read or write operation can be performed on that cell
- CAS must go back to the logic 1 level before initiating another memory operation



# Error detection and correction

---

- The dynamic physical interaction of the electrical signals affecting the data path of a memory unit may cause occasional errors in storing and retrieving the binary information
- The reliability of a memory unit may be improved by employing error-detecting and error-correcting codes
- The most common error detection scheme is the *parity bit*

# The parity bit

- A **parity bit** is generated and stored along with the data word in memory
- The parity of the word is checked after reading it from memory
- The data word is accepted if the parity of the bits read out is correct
- If the parity checked results in an inversion, an error is detected
- However, it cannot be corrected
- Error correction requires more complex mechanisms such as the *Hamming code*

# The Hamming code

- One of the most common error-correcting codes used in RAMs was devised by R. W. Hamming
- In the **Hamming code**,  $k$  parity bits are added to an  $n$ -bit data word, forming a new word of  $n + k$  bits
  - The bit positions are numbered in sequence from 1 to  $n + k$
  - Those positions numbered as a power of 2 are reserved for the parity bits
  - The code can be used with words of any length
- Consider, for example, the 8-bit data word **11000100**
- We include 4 parity bits with the 8-bit word and make a 12 bit word

While storing:

Bit position:	1	2	3	4	5	6	7	8	9	10	11	12
	$P_1$	$P_2$	1	$P_4$	1	0	0	$P_8$	0	1	0	0

$$P_1 = \text{XOR of bits } (3, 5, 7, 9, 11) = 1 \oplus 1 \oplus 0 \oplus 0 \oplus 0 = 0$$

$$P_2 = \text{XOR of bits } (3, 6, 7, 10, 11) = 1 \oplus 0 \oplus 0 \oplus 1 \oplus 0 = 0$$

$$P_4 = \text{XOR of bits } (5, 6, 7, 12) = 1 \oplus 0 \oplus 0 \oplus 0 = 1$$

$$P_8 = \text{XOR of bits } (9, 10, 11, 12) = 0 \oplus 1 \oplus 0 \oplus 0 = 1$$

While reading:

Bit position:	0	0	1	1	1	0	0	1	0	1	0	0
	1	2	3	4	5	6	7	8	9	10	11	12

$$C_1 = \text{XOR of bits } (1, 3, 5, 7, 9, 11)$$

$$C_2 = \text{XOR of bits } (2, 3, 6, 7, 10, 11)$$

$$C_4 = \text{XOR of bits } (4, 5, 6, 7, 12)$$

$$C_8 = \text{XOR of bits } (8, 9, 10, 11, 12)$$

# The Hamming code

- A 0 check bit designates even parity over the checked bits and a 1 designates odd parity
- Since the bits were stored with even parity, the result,  $C = C_8C_4C_2C_1 = 0000$ , indicates that no error has occurred
- Here is some magic: However, if  $C \neq 0$ , then the 4-bit binary number formed by the check bits gives the position of the erroneous bit!

While storing:

Bit position:	1	2	3	4	5	6	7	8	9	10	11	12
	$P_1$	$P_2$	1	$P_4$	1	0	0	$P_8$	0	1	0	0

- A 0 check bit designates even parity over the checked bits and a 1 designates odd parity
- Since the bits were stored with even parity, the result,  $C = C_8C_4C_2C_1 = 0000$ , indicates that no error has occurred
- Here is some magic: However, if  $C \neq 0$ , then the 4-bit binary number formed by the check bits gives the position of the erroneous bit!

While reading:

Bit position:	0	1	2	3	4	5	6	7	8	9	10	11	12
	1	0	0	1	1	0	0	1	0	1	0	0	0

$C_1 = \text{XOR of bits } (1, 3, 5, 7, 9, 11)$

$C_2 = \text{XOR of bits } (2, 3, 6, 7, 10, 11)$

$C_4 = \text{XOR of bits } (4, 5, 6, 7, 12)$

$C_8 = \text{XOR of bits } (8, 9, 10, 11, 12)$

# The Hamming code

## Detecting the position of erroneous bit

Bit position:	1	2	3	4	5	6	7	8	9	10	11	12	
	0	0	1	1	1	0	0	1	0	1	0	0	No error
	<u>1</u>	0	1	1	1	0	0	1	0	1	0	0	Error in bit 1
	0	0	1	1	<u>0</u>	0	0	1	0	1	0	0	Error in bit 5

	$C_8$	$C_4$	$C_2$	$C_1$	$C_1 = \text{XOR of bits } (1, 3, 5, 7, 9, 11)$
For no error:	0	0	0	0	$C_2 = \text{XOR of bits } (2, 3, 6, 7, 10, 11)$
With error in bit 1:	0	0	0	1	$C_4 = \text{XOR of bits } (4, 5, 6, 7, 12)$
With error in bit 5:	0	1	0	1	$C_8 = \text{XOR of bits } (8, 9, 10, 11, 12)$

# Lecture 26 – Processor design 1

# The Hamming code

- A 0 check bit designates even parity over the checked bits and a 1 designates odd parity
- Since the bits were stored with even parity, the result:

$$C = C_8 C_4 C_2 C_1 = 0000$$

indicates that no error has occurred

- Here is some magic: *However, if  $C \neq 0$ , then the 4-bit binary number formed by the check bits gives the position of the erroneous bit!*

While storing:												
Bit position:	1	2	3	4	5	6	7	8	9	10	11	12
	$P_1$	$P_2$	1	$P_4$	1	0	0	$P_8$	0	1	0	0

$$P_1 = \text{XOR of bits } (3, 5, 7, 9, 11) = 1 \oplus 1 \oplus 0 \oplus 0 \oplus 0 = 0$$

$$P_2 = \text{XOR of bits } (3, 6, 7, 10, 11) = 1 \oplus 0 \oplus 0 \oplus 1 \oplus 0 = 0$$

$$P_4 = \text{XOR of bits } (5, 6, 7, 12) = 1 \oplus 0 \oplus 0 \oplus 0 = 1$$

$$P_8 = \text{XOR of bits } (9, 10, 11, 12) = 0 \oplus 1 \oplus 0 \oplus 0 = 1$$

While reading:												
Bit position:	0	0	1	1	1	0	0	1	0	1	0	0
	1	2	3	4	5	6	7	8	9	10	11	12

$$C_1 = \text{XOR of bits } (1, 3, 5, 7, 9, 11)$$

$$C_2 = \text{XOR of bits } (2, 3, 6, 7, 10, 11)$$

$$C_4 = \text{XOR of bits } (4, 5, 6, 7, 12)$$

$$C_8 = \text{XOR of bits } (8, 9, 10, 11, 12)$$

# The Hamming code

## Detecting the position of erroneous bit

Bit position:	1	2	3	4	5	6	7	8	9	10	11	12	
	0	0	1	1	1	0	0	1	0	1	0	0	No error
	<u>1</u>	0	1	1	1	0	0	1	0	1	0	0	Error in bit 1
	0	0	1	1	<u>0</u>	0	0	1	0	1	0	0	Error in bit 5

	$C_8$	$C_4$	$C_2$	$C_1$	$C_1 = \text{XOR of bits } (1, 3, 5, 7, 9, 11)$
For no error:	0	0	0	0	$C_2 = \text{XOR of bits } (2, 3, 6, 7, 10, 11)$
With error in bit 1:	0	0	0	1	$C_4 = \text{XOR of bits } (4, 5, 6, 7, 12)$
With error in bit 5:	0	1	0	1	$C_8 = \text{XOR of bits } (8, 9, 10, 11, 12)$

# Hamming code

- The Hamming code can be used for data words of any length.
- Hamming code consists of  $k$  check bits and  $n$  data bits, for a total of  $n + k$  bits.
- The syndrome value  $C$  consists of  $k$  bits and has a range of  $2^k$  values between 0 and  $2^k - 1$ .
- One of these values, usually zero, is used to indicate that no error was detected, leaving  $2^k - 1$  values to indicate which of the  $n + k$  bits was in error.
- Each of these  $2^k - 1$  values can be used to uniquely describe a bit in error.
- Therefore,  $2^k - 1 \geq n + k$
- Solving for  $n$  in terms of  $k$ , we obtain  $2^k - 1 - k \geq n$ 
  - Eg:  $k=4$  check bits  $\Rightarrow n \leq 11$  data bits

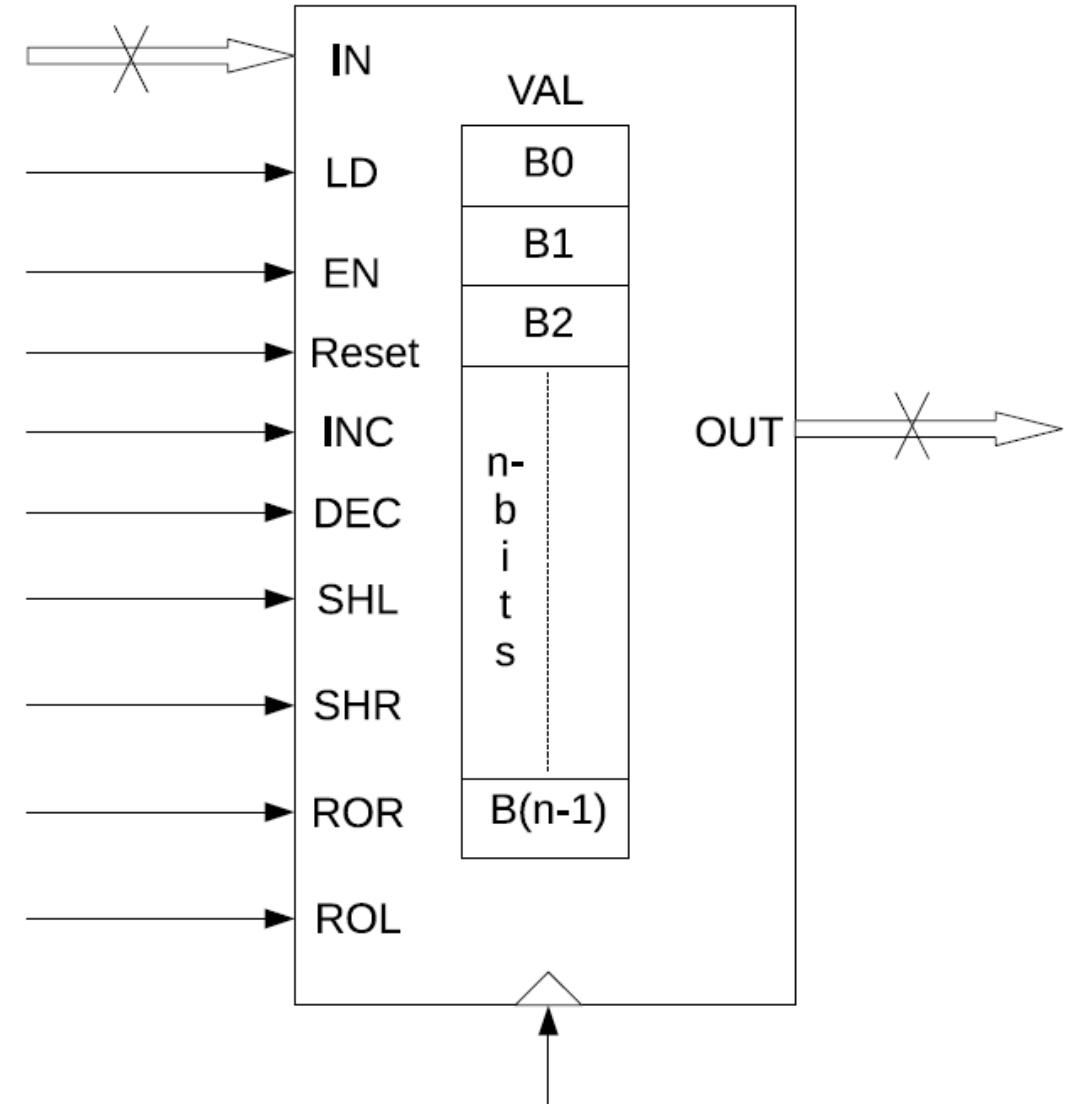
# Single correct, double detect

- The Hamming code can detect and correct only a single error
- By adding another parity bit to the coded word, the Hamming code can be used to correct a single error and detect double errors
- If we include this additional parity bit, then the previous 12-bit coded word becomes  $001110010100P_{13}$ , where  $P_{13}$  is evaluated from the exclusive-OR of the other 12 bits
- This produces the 13-bit word  $0011100101001$  (even parity)
- When the 13-bit word is read from memory, the check bits are evaluated, as is the parity  $P$  over the entire 13 bits
- The following four cases can arise:
  1. If  $C = 0$  and  $P = 0$ , no error occurred
  2. If  $C = 0$  and  $P = 1$ , an error occurred in the  $P_{13}$  bit!
  3. If  $C \neq 0$  and  $P = 1$ , a single error occurred that can be corrected
  4. If  $C \neq 0$  and  $P = 0$ , a double error occurred, but that cannot be corrected

# Processor design 1

# Multipurpose register

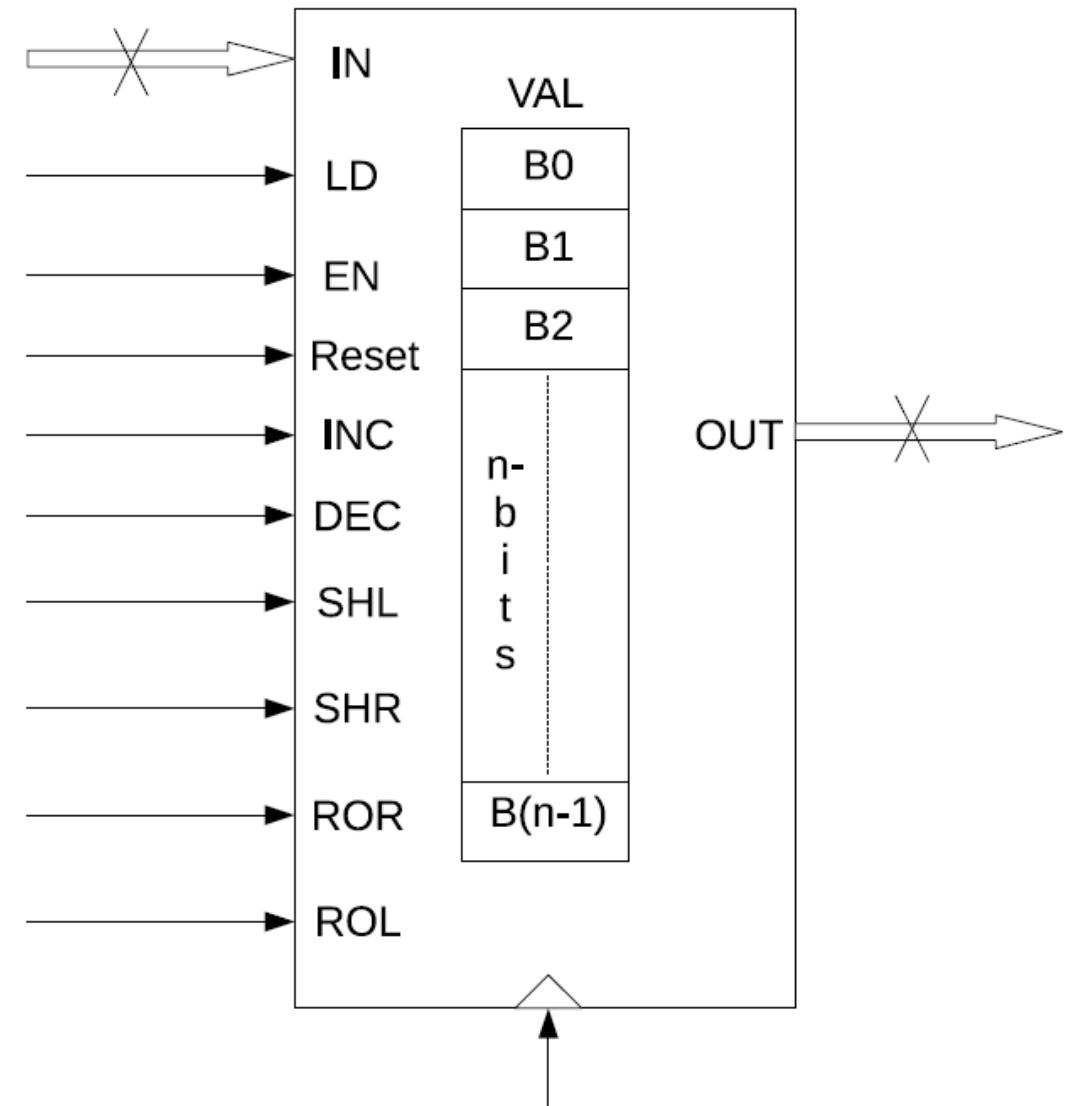
- A **multipurpose/universal register** is a very useful entity in processor design
- It can store value as well as have some basic arithmetic functions such as increment/decrement etc.
- The register has two input/output lines (sometimes coupled into one IO bus)
- Input lines **IN** are logically connected internally to the inputs of the  $n$  flip-flops inside the register
- Output lines **OUT** are the lines that hold the value stored in the register when the register is accessed for read. The **OUT** lines are **tristate-capable** so that they can be connected to a common data bus



# Multipurpose register

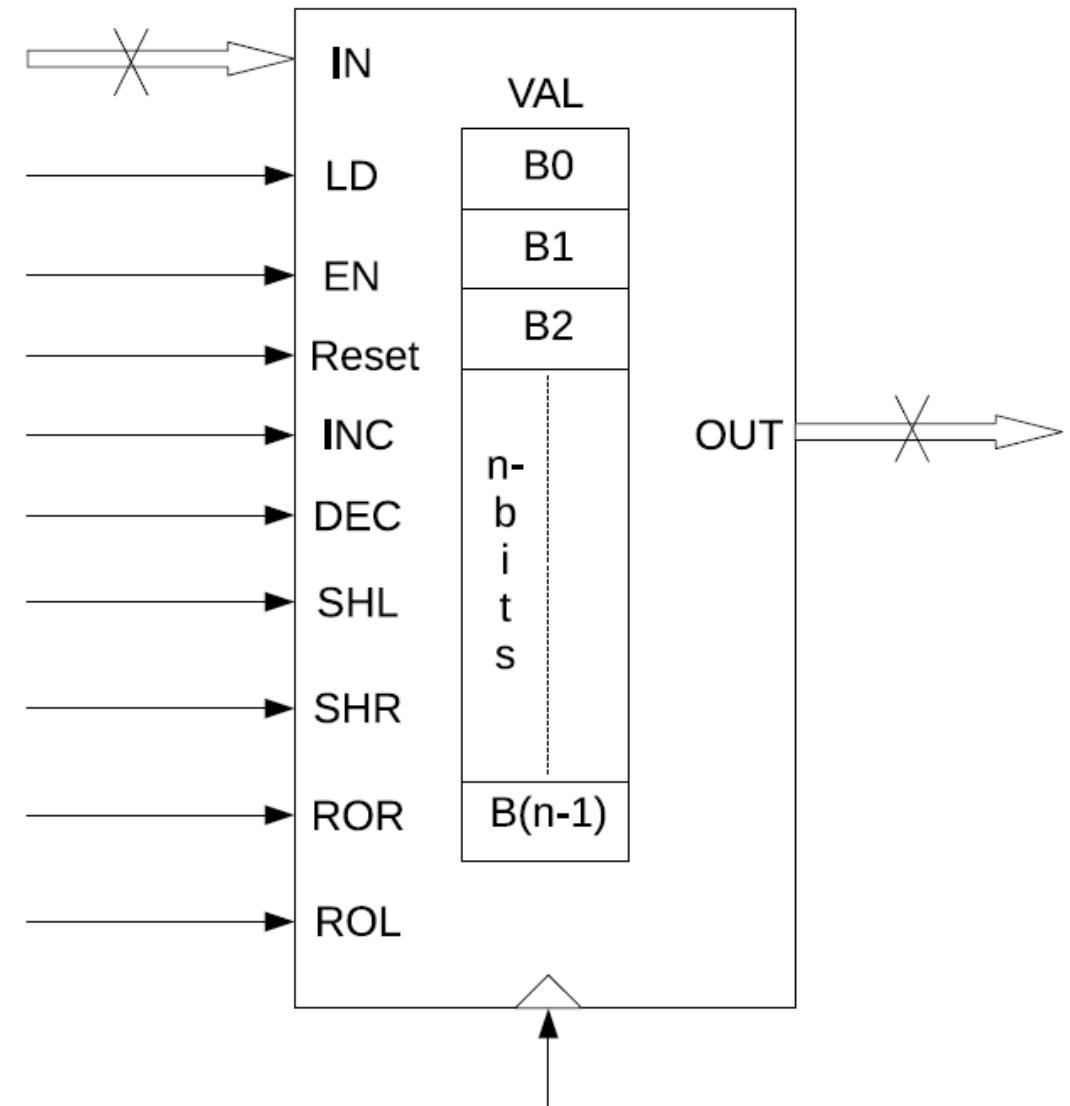
- The control bits are as follows:

- The input **EN** controls the high-impedance state of the output lines OUT. When EN is 0, the OUT lines will be in the high-impedance state. *When EN is 1, the OUT lines will be driven to the electric levels corresponding to the value stored in the register*
- The input **LD** loads the register from the input, which changes the value stored in it. *When LD is 1, the value indicated by the electric levels of the IN lines will replace the previous value stored in the register*
- The input **RESET** controls resetting of the register. *When RESET is 1, a 0 value will be written to all bits stored in the register. We will assume a synchronous reset.*



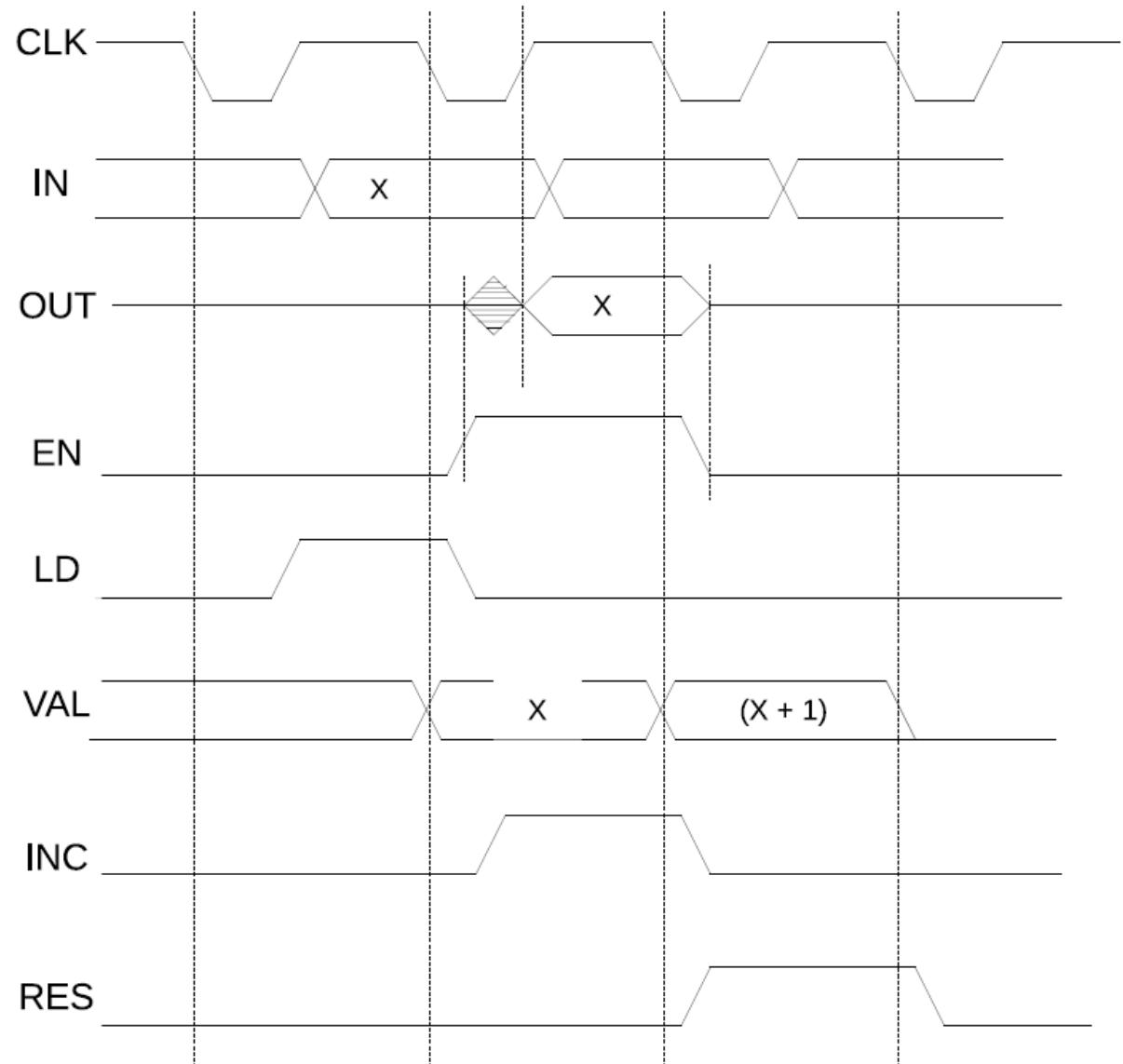
# Multipurpose register

- The control bits are as follows:
  - Inputs **INC** and **DEC** control the incrementing or decrementing the value stored in the register, interpreted as a number.
  - Inputs **ROL**, **ROR** control the left or right rotation of the register contents. This is shifting such that the last bit is feedback to the first bit
  - Inputs **SHL**, **SHR** control the left or right shift of the register contents. We assume a 0 will fill the void during the shift



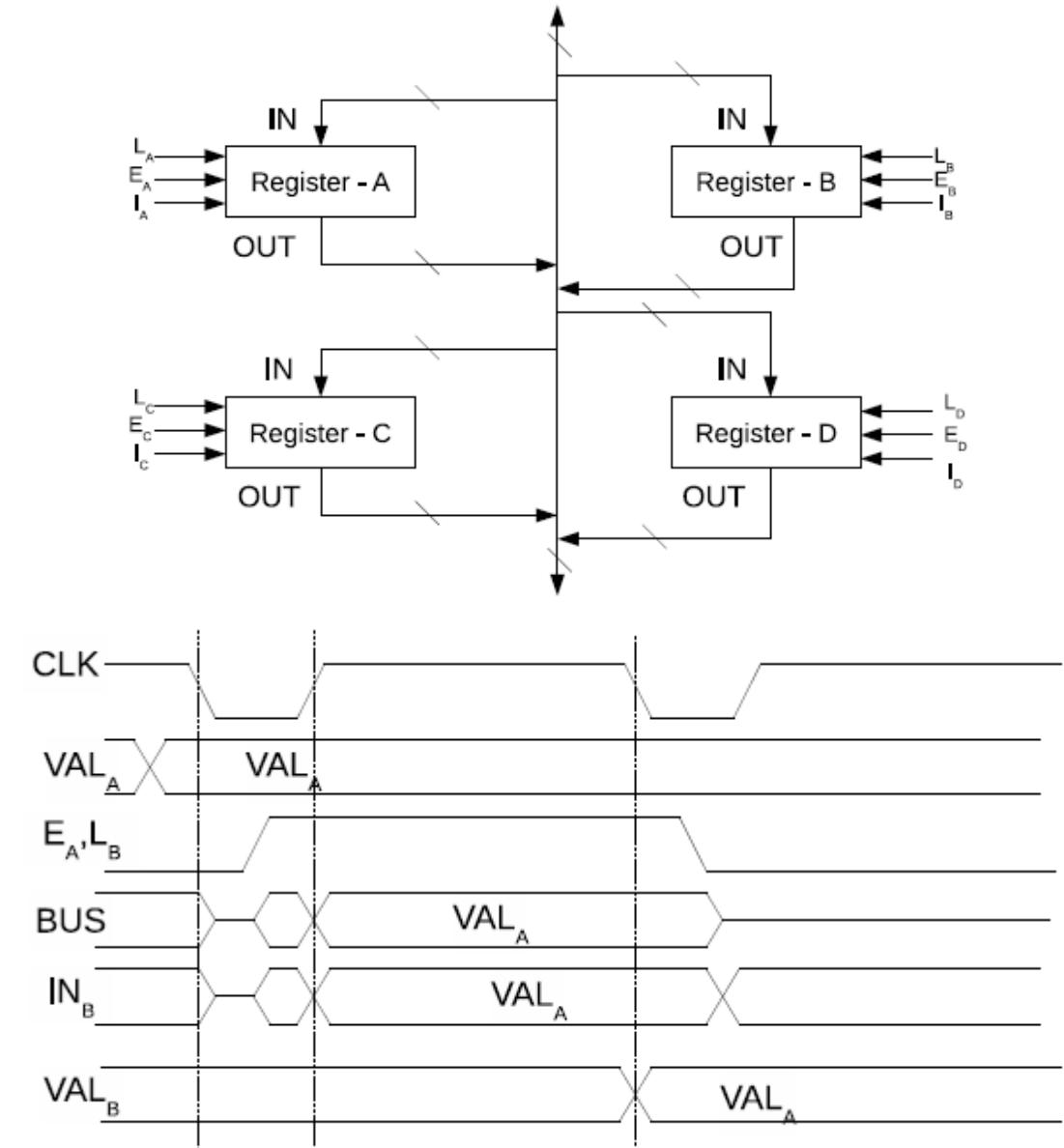
# Multipurpose register

- The timing diagram depends on how we design the register
- In this case, we have designed such that all instructions (actions) happen to the stored value at the **negative edge** of the clock
- While the EN puts the data on the output bus at the **positive edge** of the clock



# Creating a common bus

- Using tristate buffers, we can connect multiple input/outputs of multiple registers on the same bus
- Consider registers A and B
- The **enable**, **load**, and **increment** control signals of register A are respectively  $E_A$ ,  $L_A$ , and  $I_A$  and for register B are labelled  $E_B$ ,  $L_B$ , and  $I_B$
- Let us say we need to transfer from A to B
- We enable  $E_A$  and  $L_B$  and in one clock cycle, the contents get transferred



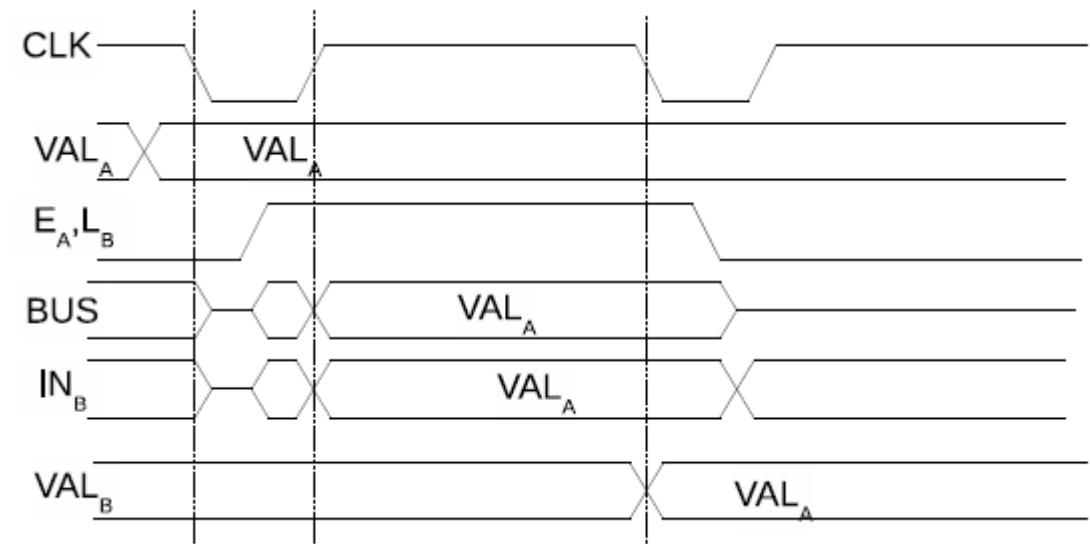
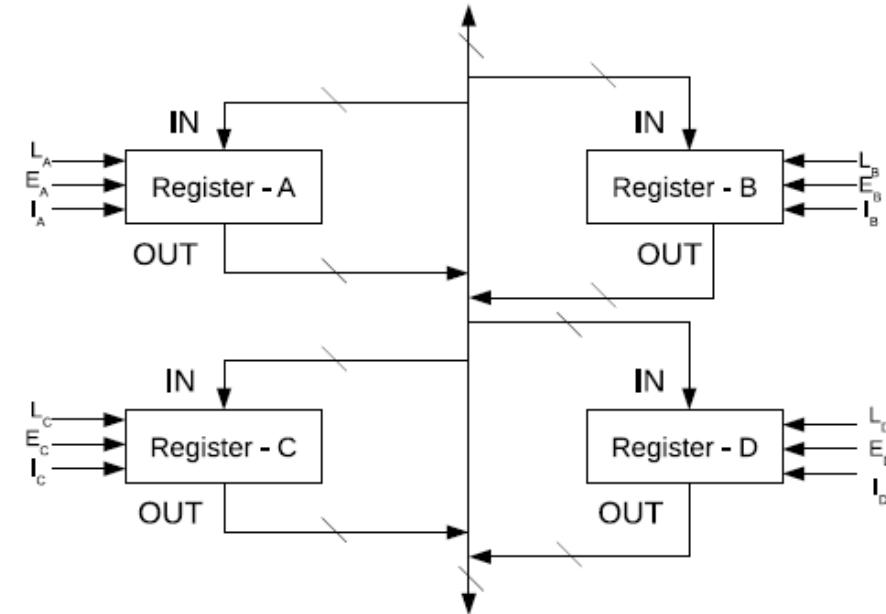
# Creating a common bus

- In effect, we have moved the data to register B from register A, like a assignment instruction!

$A=B;$

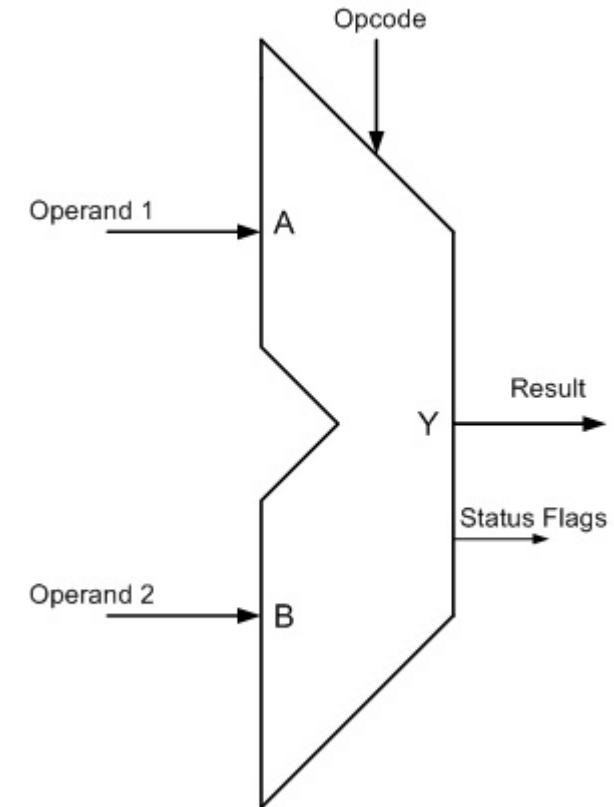
$MOV\ B,\ A$

- At the level of the hardware, activating 2 signals simultaneously was all that was done
- The rest happened due to the bus connection, the clock, and the design of the register
- What if we activate  $L_B, E_D, L_C, L_A$  together?
- What if we activate  $E_B, L_A, I_B$



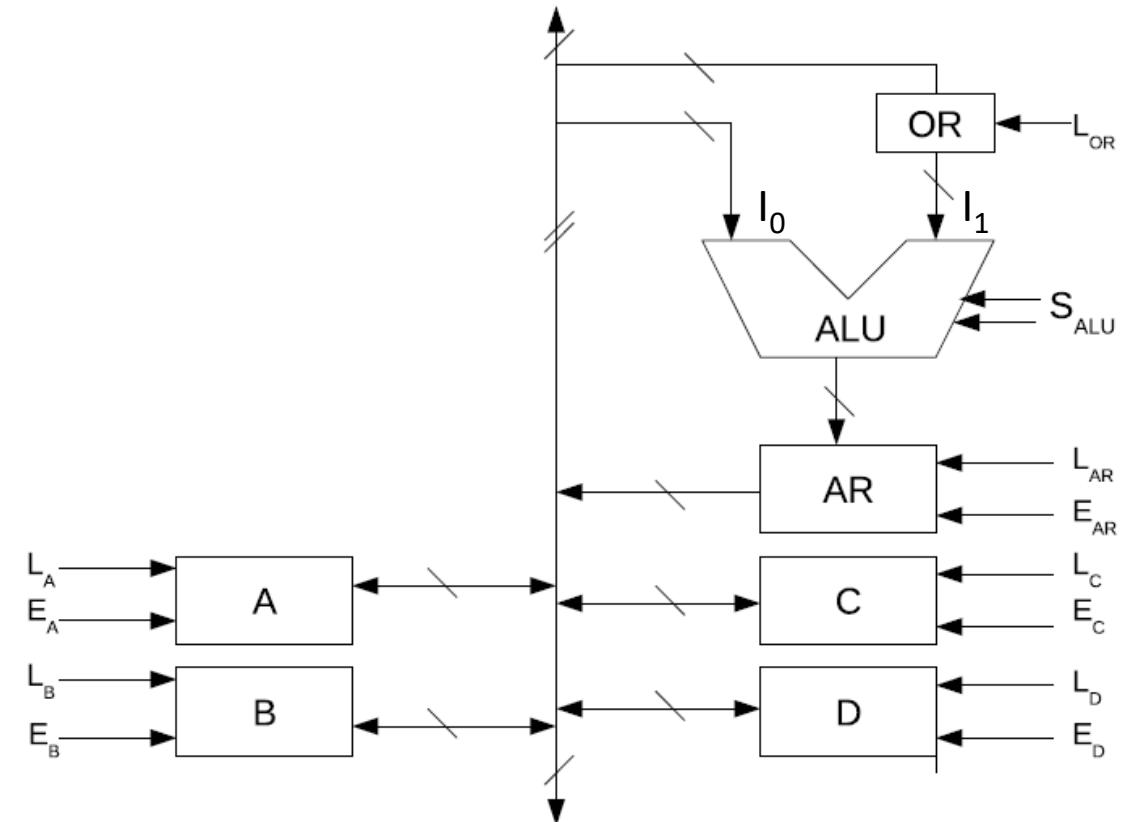
# ALU design

- Enter, The ALU!
- We generally consider an ALU as a “simple” combinational circuit capable of performing several basic functions on a set of two binary numbers – **add, sub, multiply, increment, AND, OR, XOR**, and many other operations based on a select input
- This is generally implemented using a combination of MUXes within the ALU



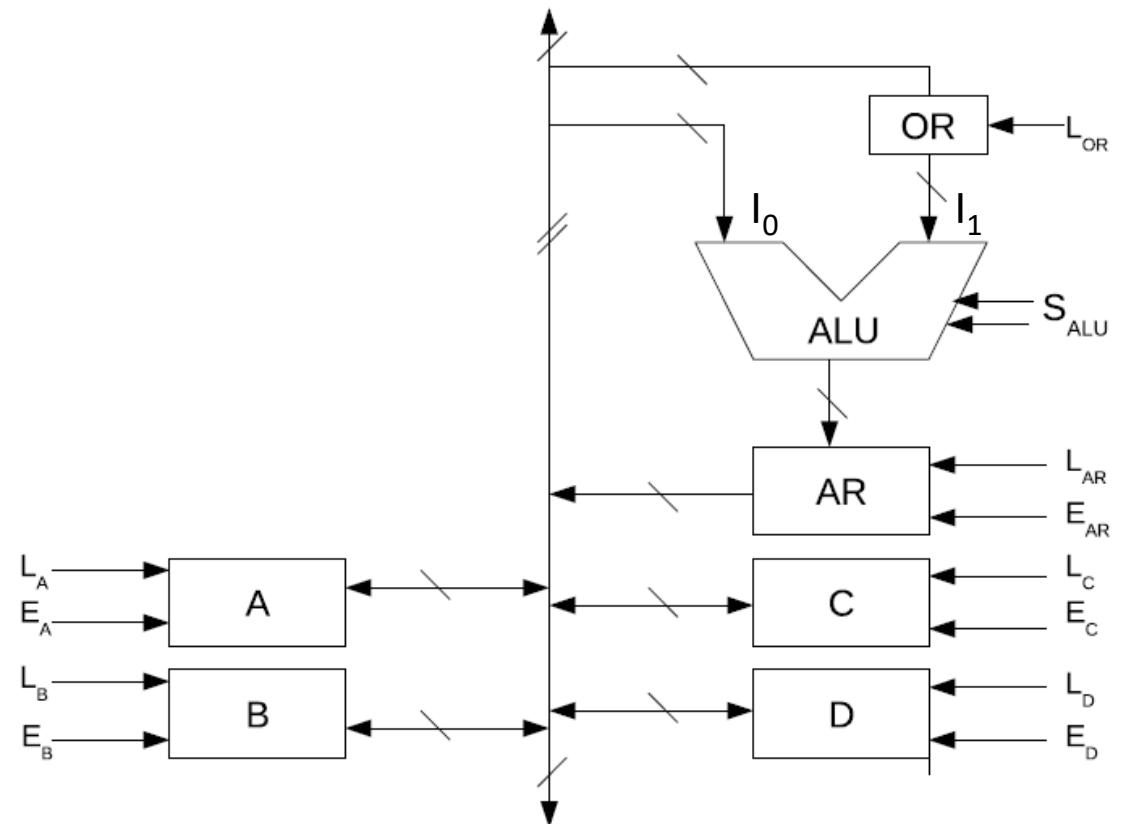
# ALU on a bus

- Consider the configuration which has a few registers and an ALU
- Assume all the registers are made using the multipurpose registers
- The ALU is a combinational circuit with 2 inputs and two lines to select the function to be performed
- The two select lines can be in one of 4 combinations
  - We refer to them symbolically as **ADD, SUB, AND, PASSO**



# ALU on a bus

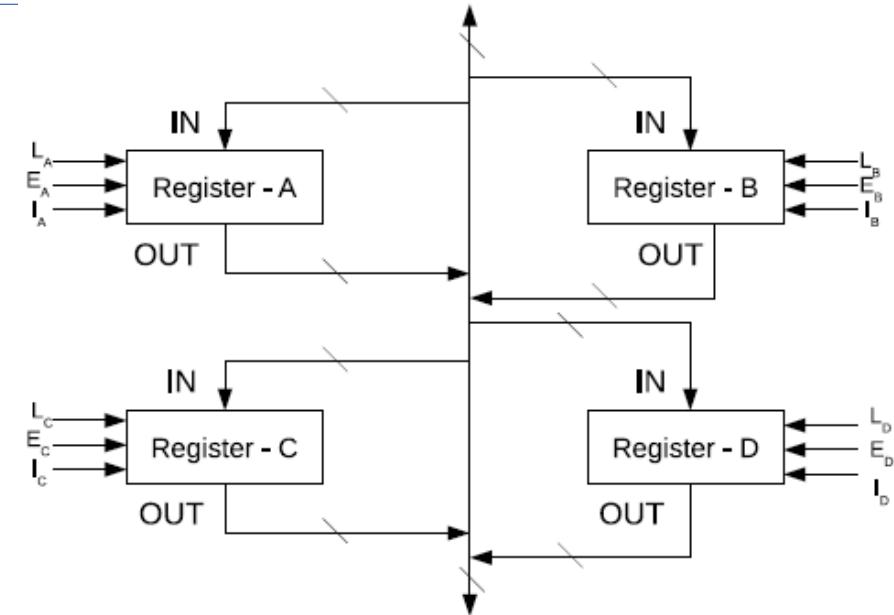
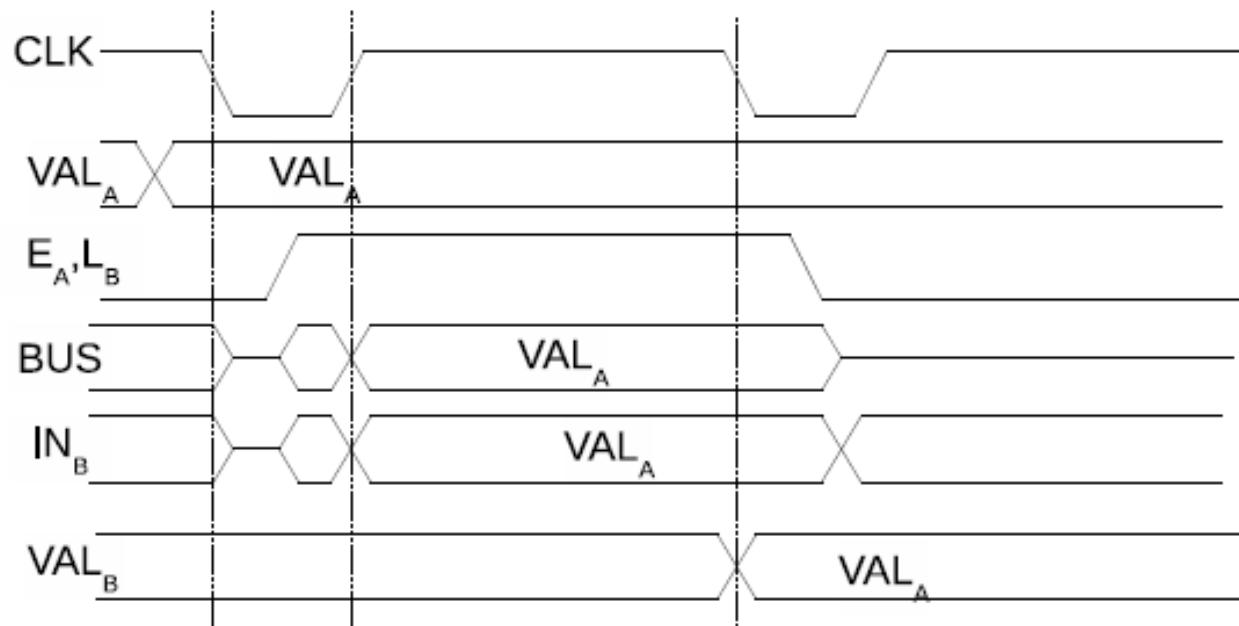
- The ALU has its left input connected to the bus and the right input to a register called **OR or operand register**, whose inputs are connected to the bus
- The output of the ALU is connected to the input of a register called **AR or accumulator register**, whose output is connected to the bus
- Every register has all the control signals of our generic multipurpose register



# Lecture 27 – Processor design 2

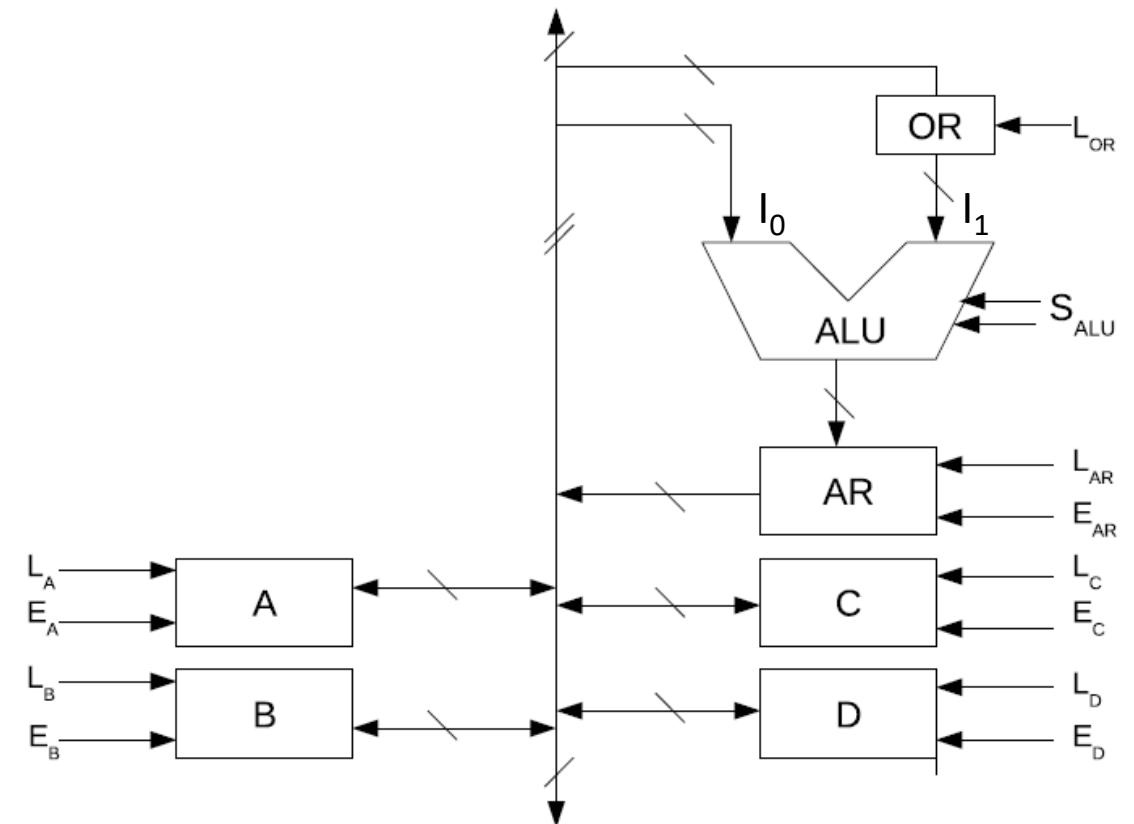
# Creating a common bus

- If we enable  $E_A$  and  $L_B$  and in one clock cycle, the contents of register A is copied to register B
- What if we activate  $E_A$ ,  $L_B$ ,  $I_A$



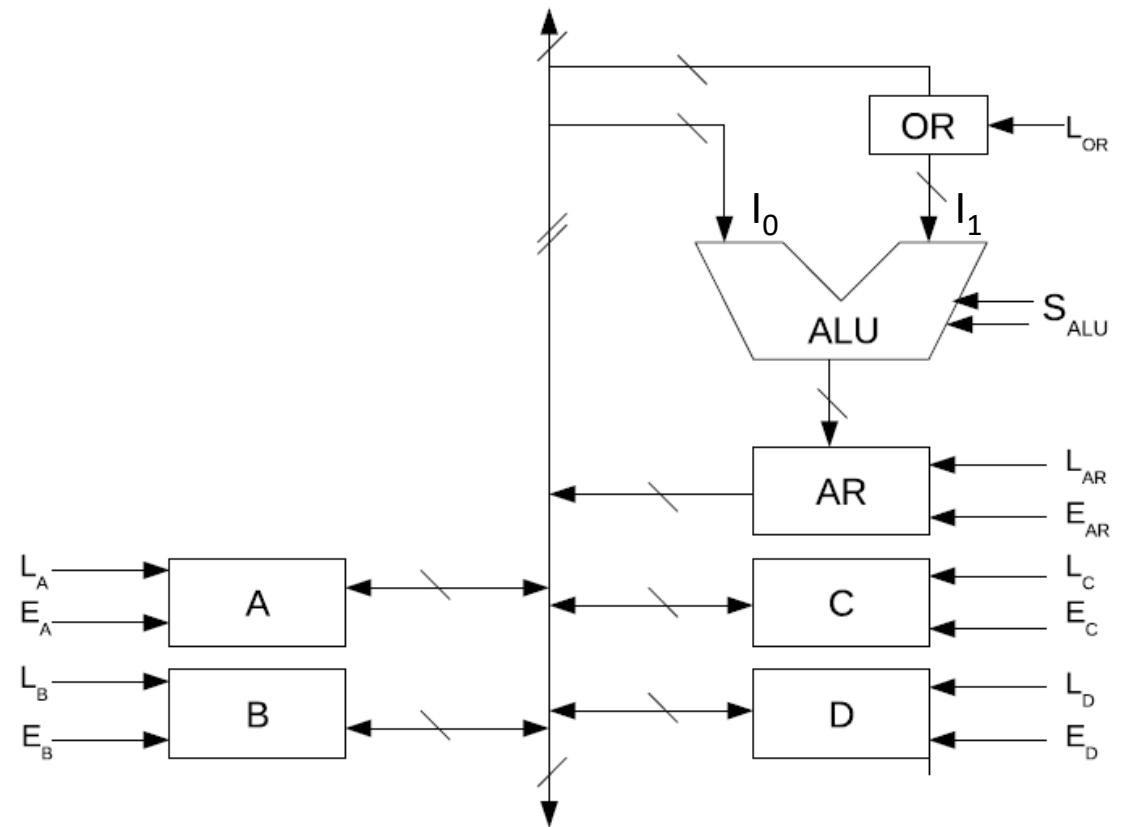
# ALU on a bus

- Consider the configuration which has a few registers and an ALU
- Assume all the registers are made using the multipurpose registers
- The ALU is a **combinational circuit** with 2 inputs and two lines to select the function to be performed
- The two select lines (  $S_{ALU}$  ) can be in one of 4 combinations:
  - We refer to them symbolically as ADD, SUB, AND, PASSO



# ALU on a bus

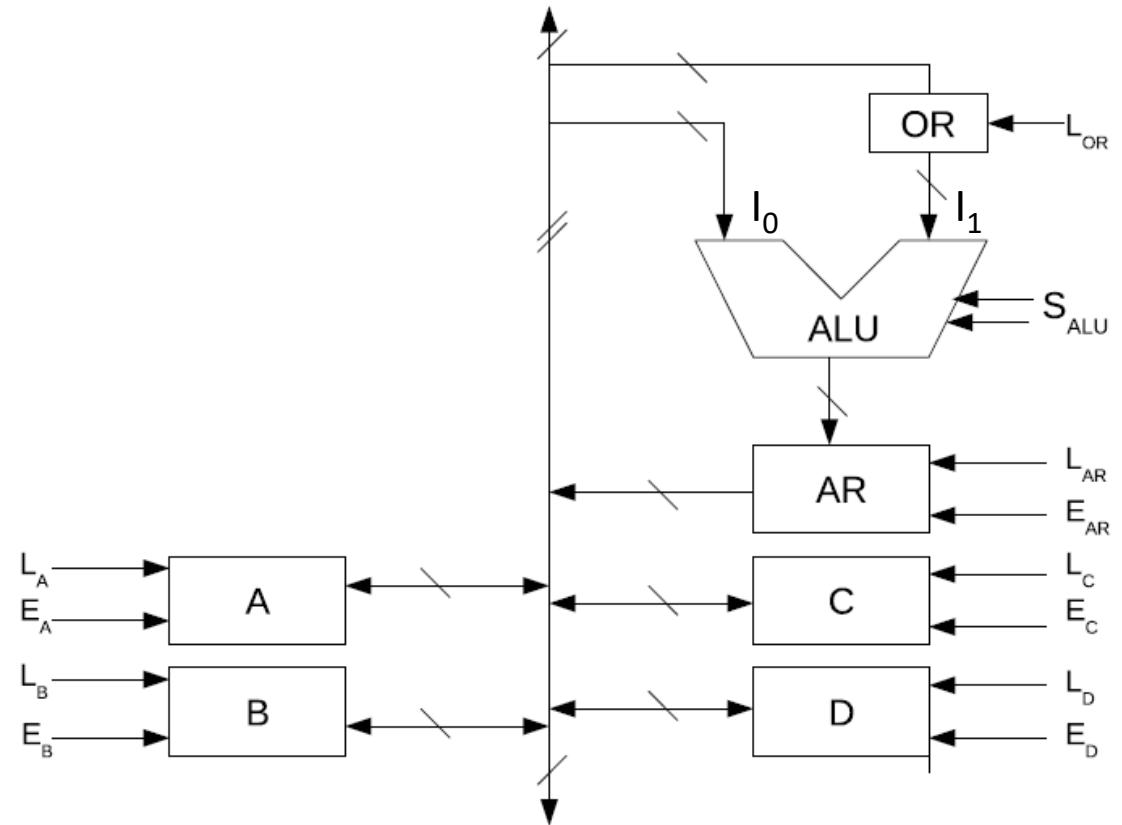
- The ALU has its left input connected to the bus and the right input to a register called **OR or operand register**, whose inputs are connected to the bus
- The output of the ALU is connected to the input of a register called **AR or accumulator register**, whose output is connected to the bus
- Every register has all the control signals of our generic multipurpose register



# ALU on a bus

- What will happen if we activate the following controls:  $E_{OR}, E_A, L_{AR}, S_{ALU\_ADD}$
- Two enables are simultaneously active
- Does it cause conflict at the bus?
- No, since the output of OR is not connected to the bus, there will be no conflict
- The last term above indicates that the select lines of the ALU are set to the bits corresponding to “ADD”
- The ALU performs addition of its inputs, which are the contents of OR register and register A
- The sum is written to AR register

$$AR \leftarrow A + OR$$



# ALU on a bus

- Consider this multiclock instruction:

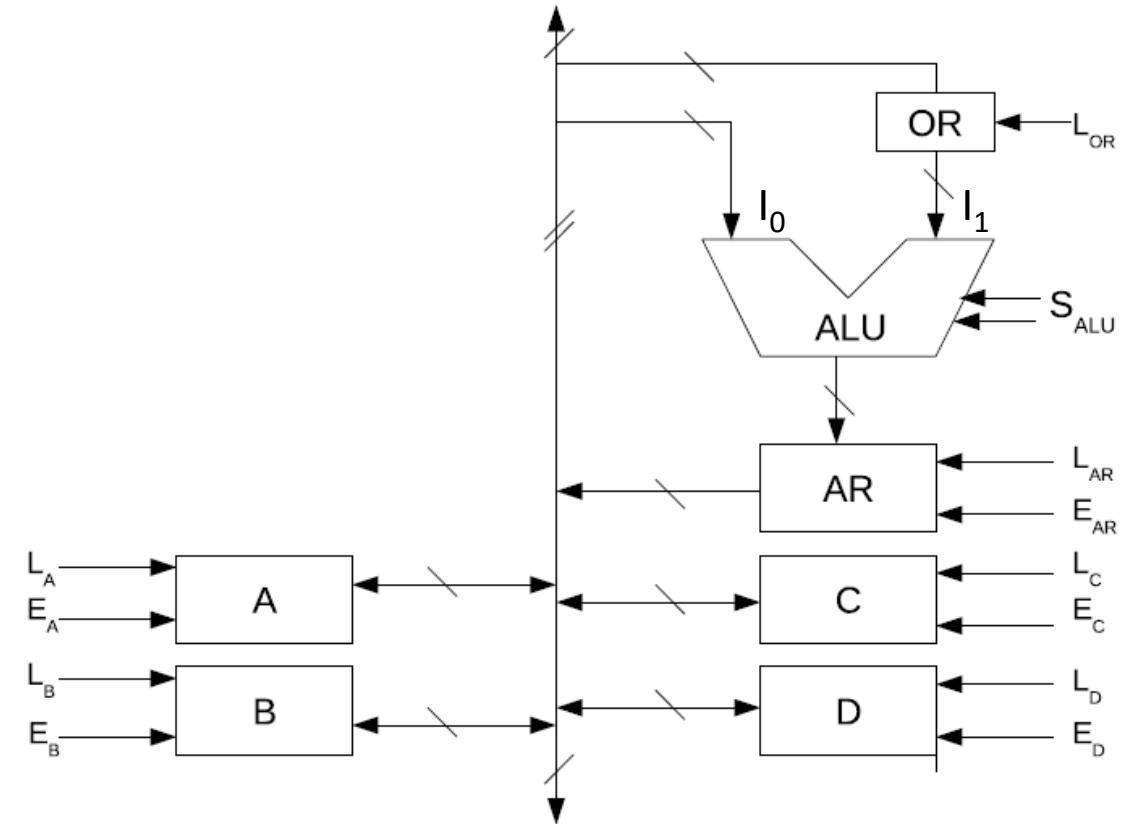
Ck 10:  $E_A, L_{OR}$

Ck 11:  $E_B, E_{OR}, L_{AR}, SALU_{SUB}$

Ck 12:  $E_{AR}, L_C$

- What does the 3 clock cycle combination achieve?
- The contents of A are copied to OR in cycle-10
- The contents of OR are subtracted from the contents of B and the result is loaded to AR in cycle-11
- The contents of AR are copied to register C in cycle-12

$$C \leftarrow B - A$$



# ALU on a bus

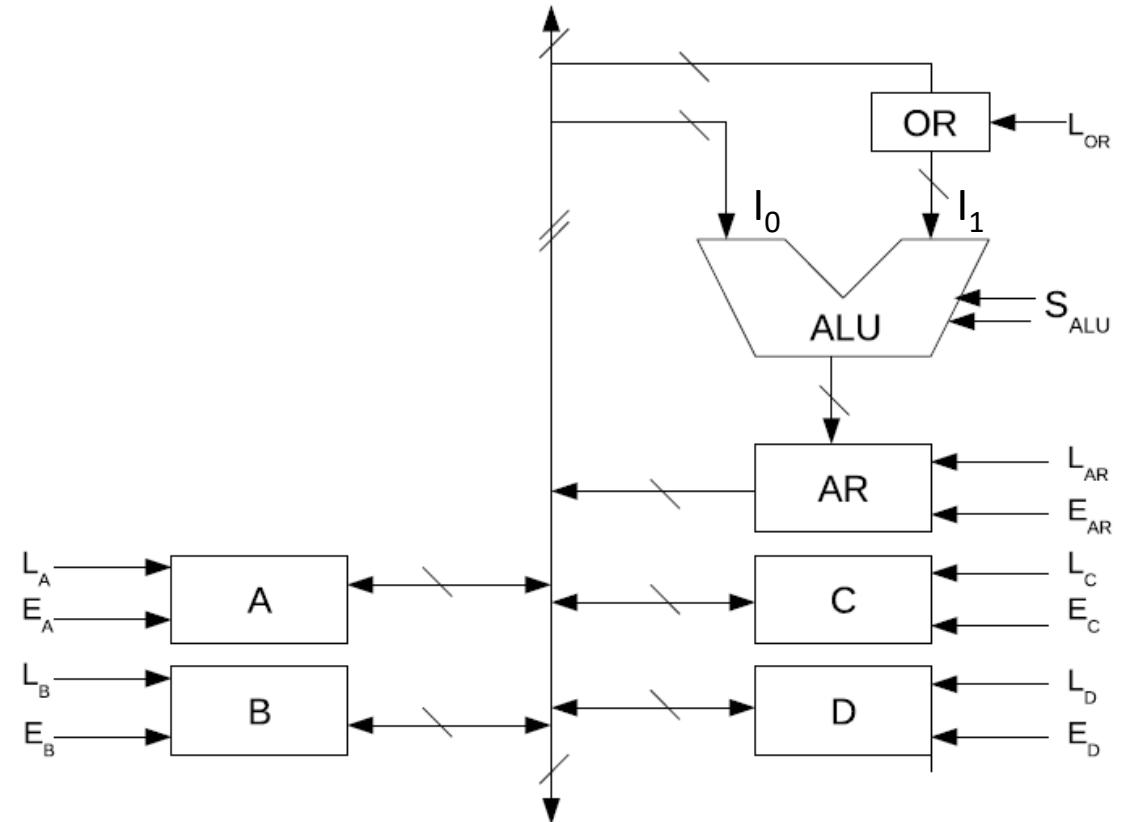
- It is easy to see how addition, subtraction and logical AND with any combination of 2 registers as input and any register as output can be implemented using a very similar 3-cycle sequence of combinations of signals
- Let us see if we can write control signals for this instruction:

$D \leftarrow C \text{ AND } D$

Ck 1:  $E_C, L_{OR}$

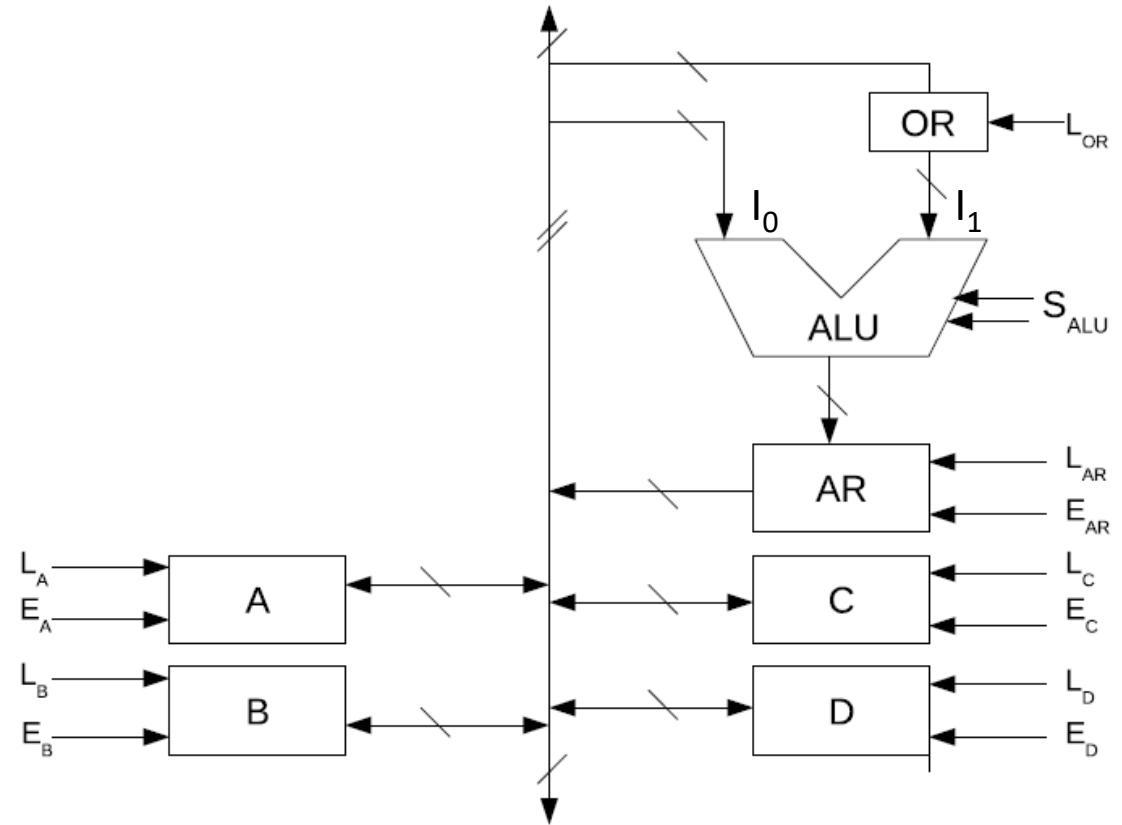
Ck 2:  $E_D, E_{OR}, L_{AR}, S_{ALU\_AND}$

Ck 3:  $E_{AR}, L_D$



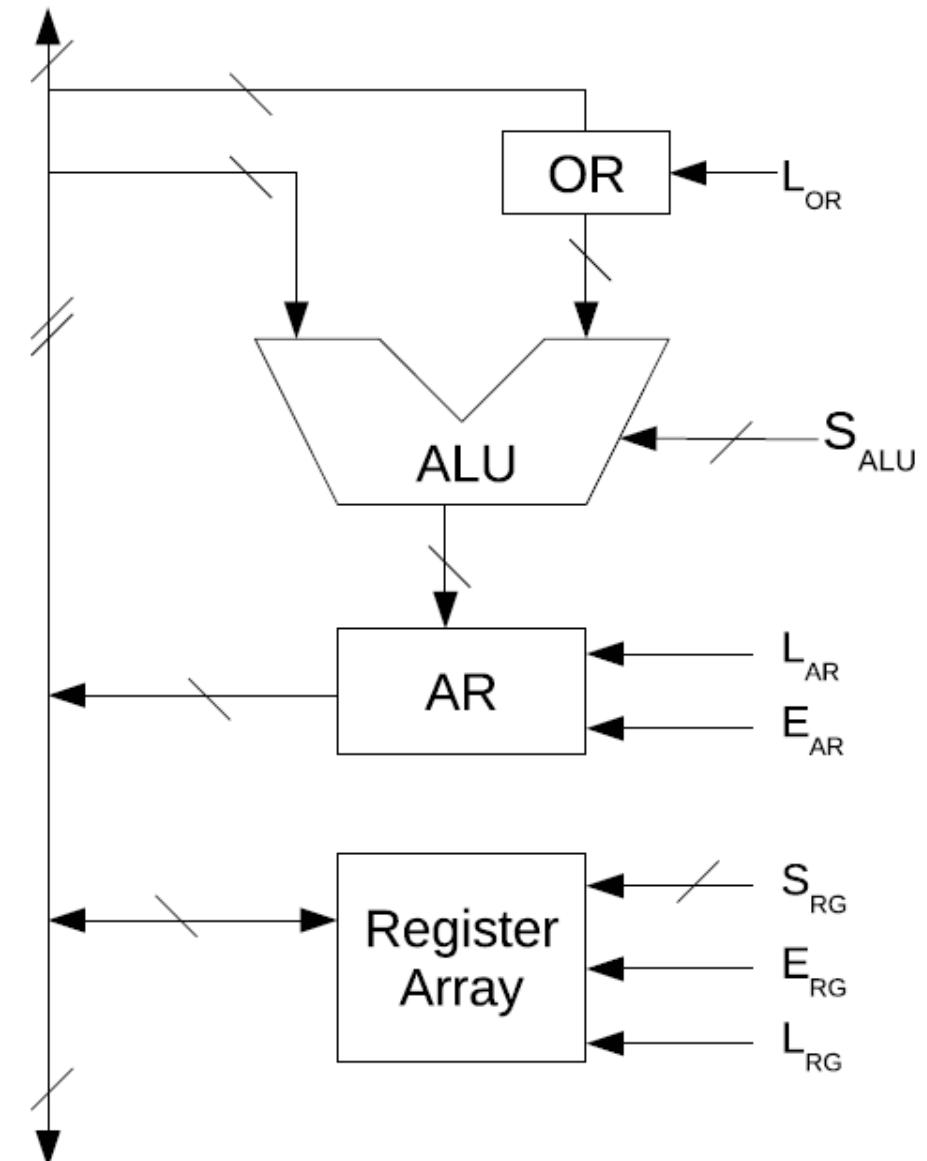
# ALU on a bus

- It is clear that we can make the hardware perform several steps by carefully selecting the control signals to different units that are active in each clock cycle
- We can also get larger “operations” implemented using multiclock sequences of such combinations
- Each such clock cycle is typically referred to as a **microcycle**, which is the basic time unit in which something happens within the processor



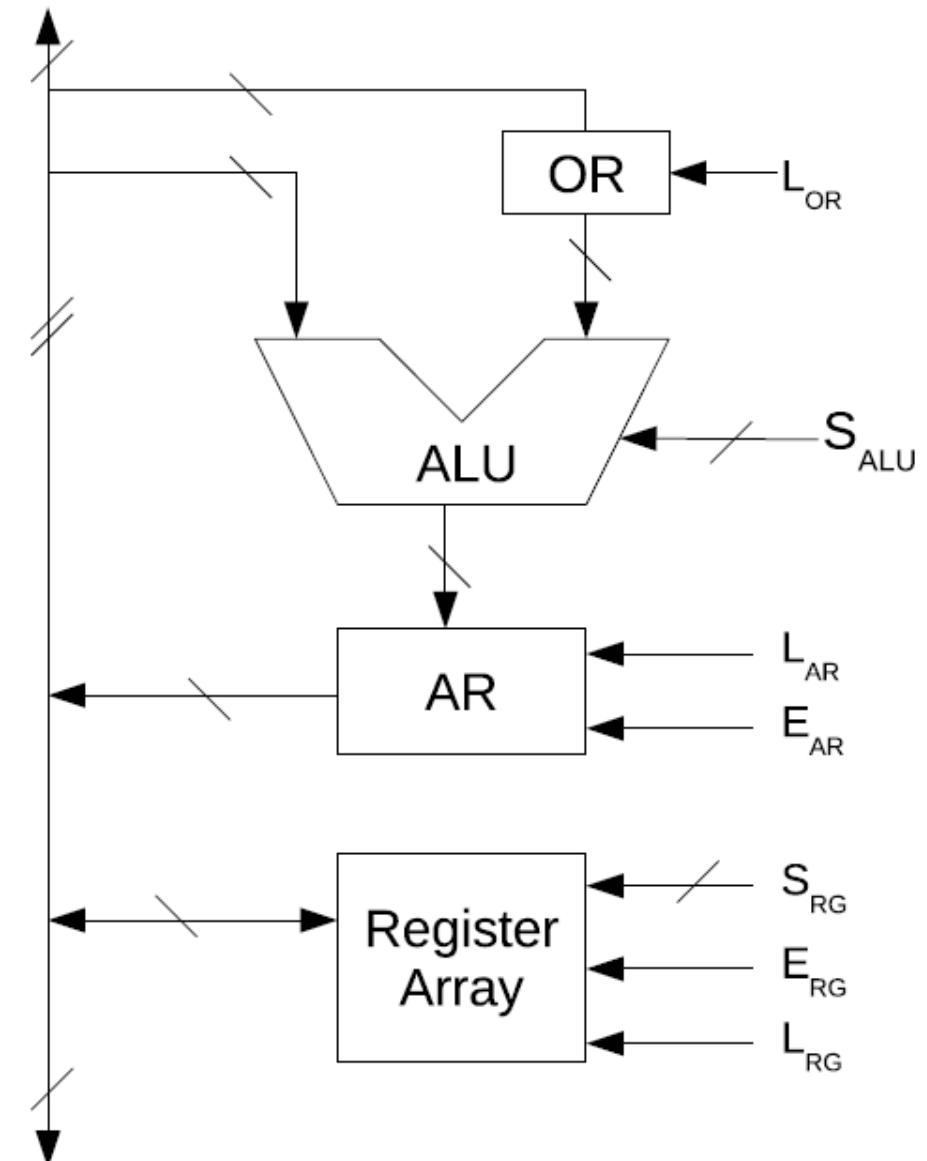
# Enhanced singlebus architecture

- We can consolidate the registers into a *register array* or a *register file*
- These are numbered  $R_0$  through  $R_{11}$
- The register file has a single enable input  $E_{RG}$  and a single load  $L_{RG}$  and 4 select lines  $S_{RG}$
- The select lines identify which register of the file is being operated on, with enable or load controlling the action performed on it
- *The new design of the register array needs only 6 control signals – 4 for select and 2 for enable/load – as opposed to 24 that would be needed were each register to have its individual enable and load signals*
- Some generality is, however, lost as only one register can be selected for writing



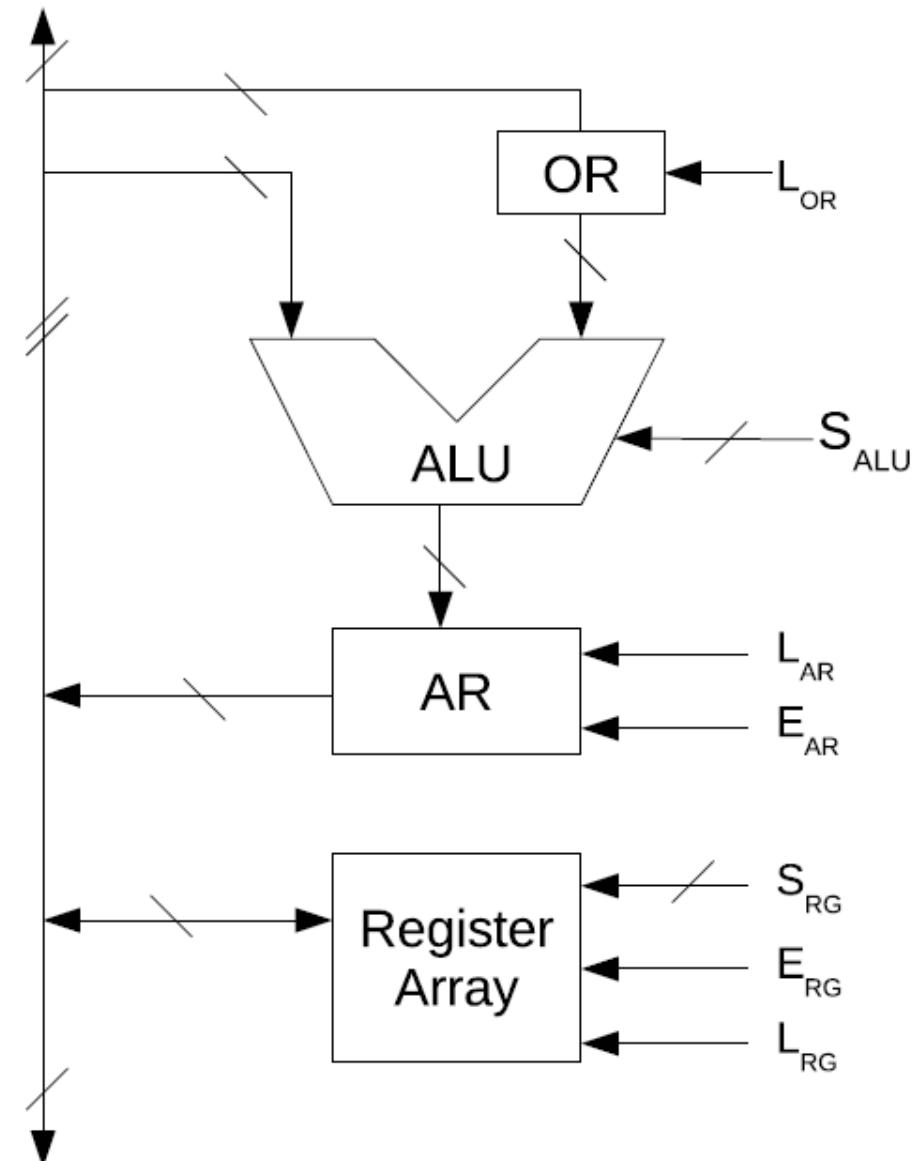
# Enhanced singlebus architecture

- We also decide to make the registers simpler devices as they need to do only parallel load and read
- Reset, increment, shift, etc., are not possible on them
- These registers are temporary store of information
- We also have an enhanced ALU in place, with 3 select lines and supporting 8 operations: **none, add, subtract, logical AND, OR, XOR, and pass left**. The ALU takes the left operand from the bus and the right one from OR



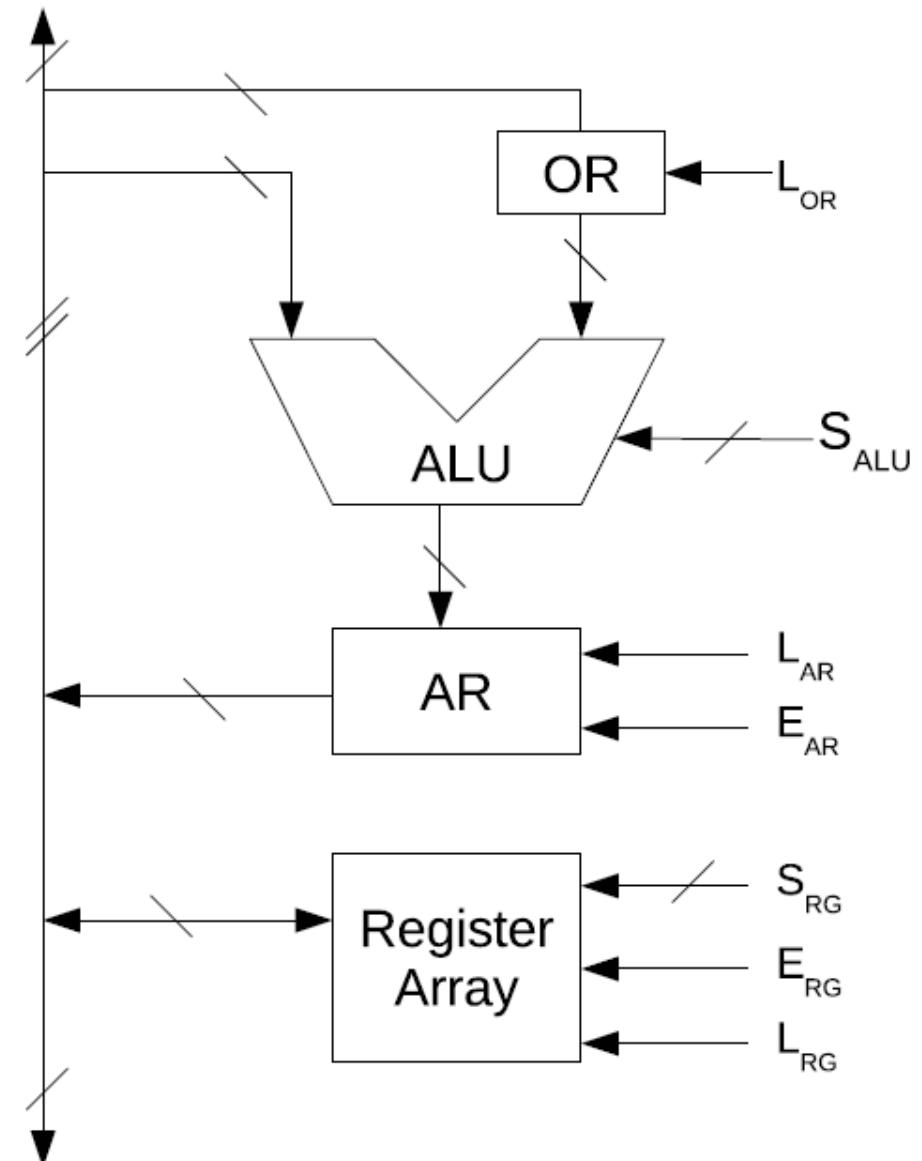
# Enhanced singlebus architecture

- ADD R1: ( $AR = AR + R1$ )
  - Ck 0:  $E_{RG}, L_{OR}, S_{RG} \leftarrow 1$
  - Ck 1:  $E_{AR}, L_{AR}, E_{OR}, S_{ALU} \leftarrow ADD$
- SUB R7: ( $AR = AR - R7$ )
  - Ck 0:  $E_{RG}, L_{OR}, S_{RG} \leftarrow 7$
  - Ck 1:  $E_{AR}, L_{AR}, E_{OR}, S_{ALU} \leftarrow SUB$
- XOR R11: ( $AR = AR \text{ XOR } R11$ )
  - Ck 0:  $E_{RG}, L_{OR}, S_{RG} \leftarrow 11$
  - Ck 1:  $E_{AR}, L_{AR}, E_{OR}, S_{ALU} \leftarrow XOR$
- OR R0: ( $AR = AR \text{ OR } R0$ )
  - Ck 0:  $E_{RG}, L_{OR}, S_{RG} \leftarrow 0$
  - Ck 1:  $E_{AR}, L_{AR}, E_{OR}, S_{ALU} \leftarrow OR$



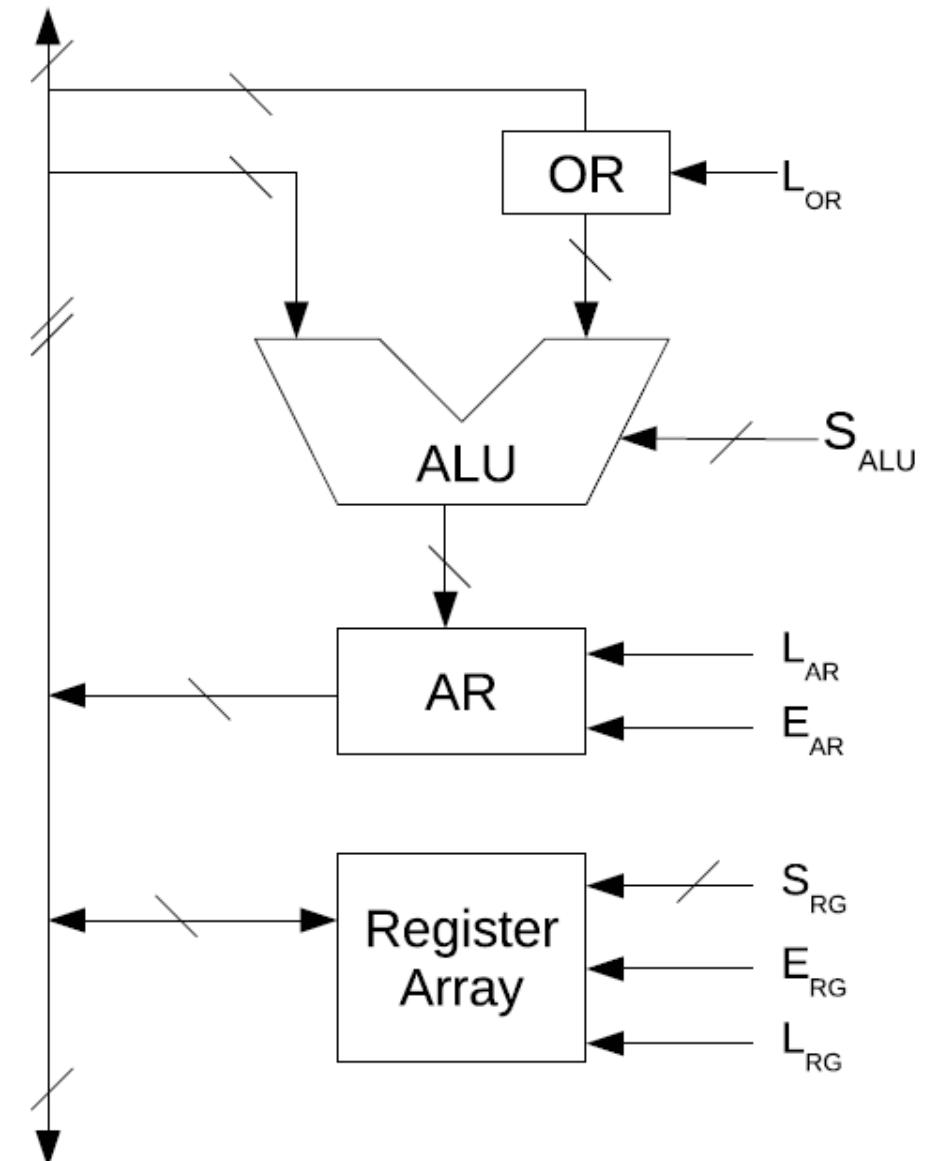
# Enhanced singlebus architecture

- We also need a way to load values from register to AR and to save results from AR to a different register
- We call these **load** and **store** respectively
- Eg: **LOAD R4** and **STOR R9** can be implemented as follows:
- **LOAD R4:**
  - Ck0:  $E_{RG}, L_{AR}, S_{RG} \leftarrow 4, SALU \leftarrow PASS0$
- **STOR R9:**
  - Ck0:  $E_{AR}, L_{RG}, S_{RG} \leftarrow 9$



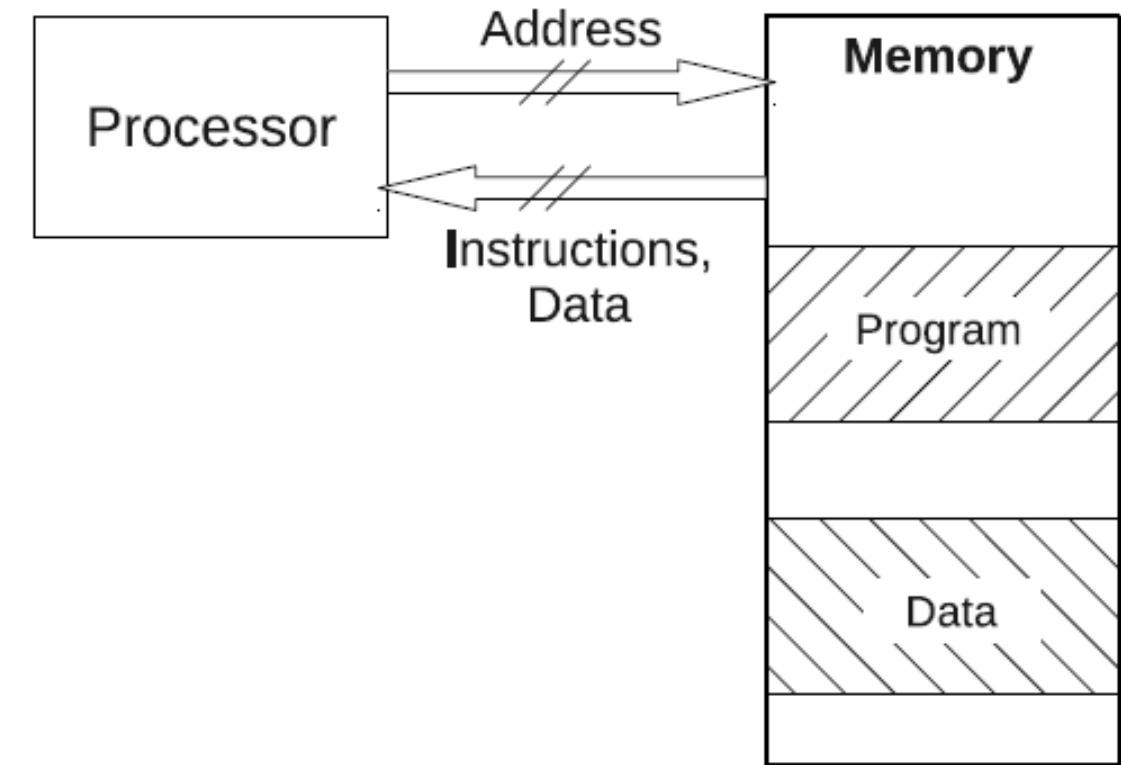
# Instructions

- A digital processor can handle only binary strings at the very lowest level
- Thus, all instructions to be carried out by a digital processor needs to be coded or represented as binary strings
- Different basic instructions have to be coded as unambiguous binary strings
- The hardware is capable of looking at a string, decoding it, and carrying out the corresponding instruction
- **A sequence of such strings forms a program**



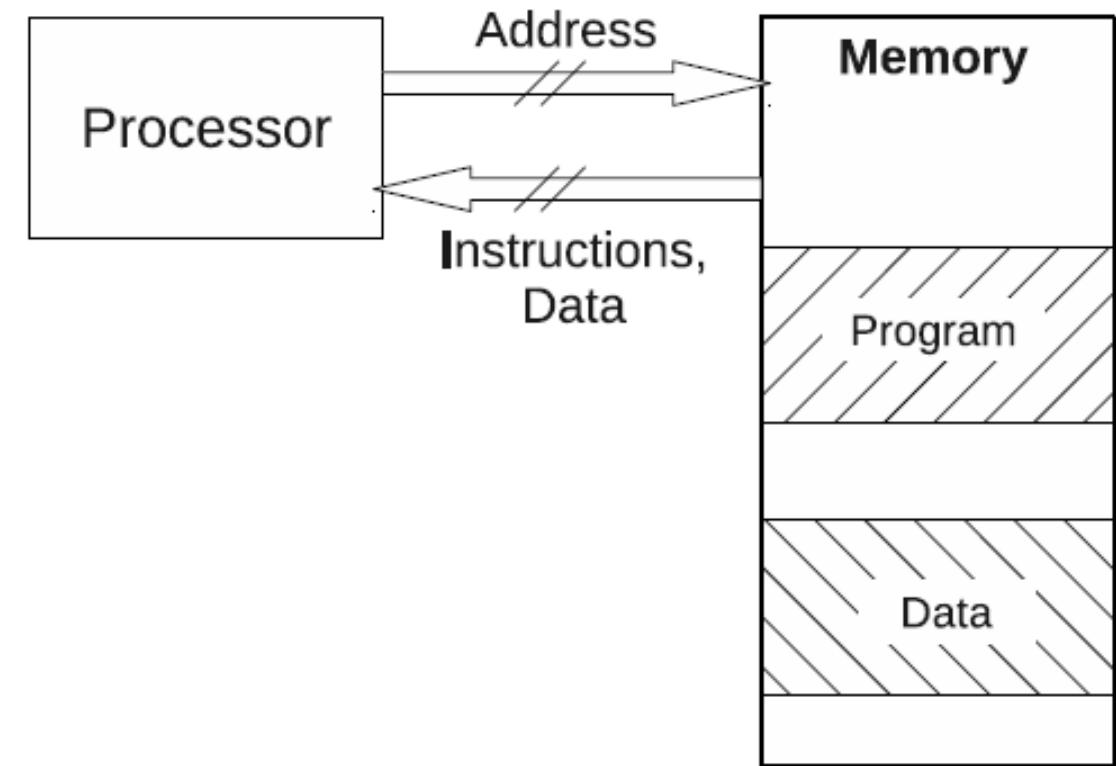
# Instructions

- Both the data to be processed and the program can be stored in the same memory (the *von Neumann* or stored-program model)
- Instructions that make up the program are stored sequentially in memory
- Each instruction is a binary string that encodes the operations to be performed without ambiguity
- The processor **fetches** the instructions one by one from the memory and **executes** it or carries out the corresponding actions



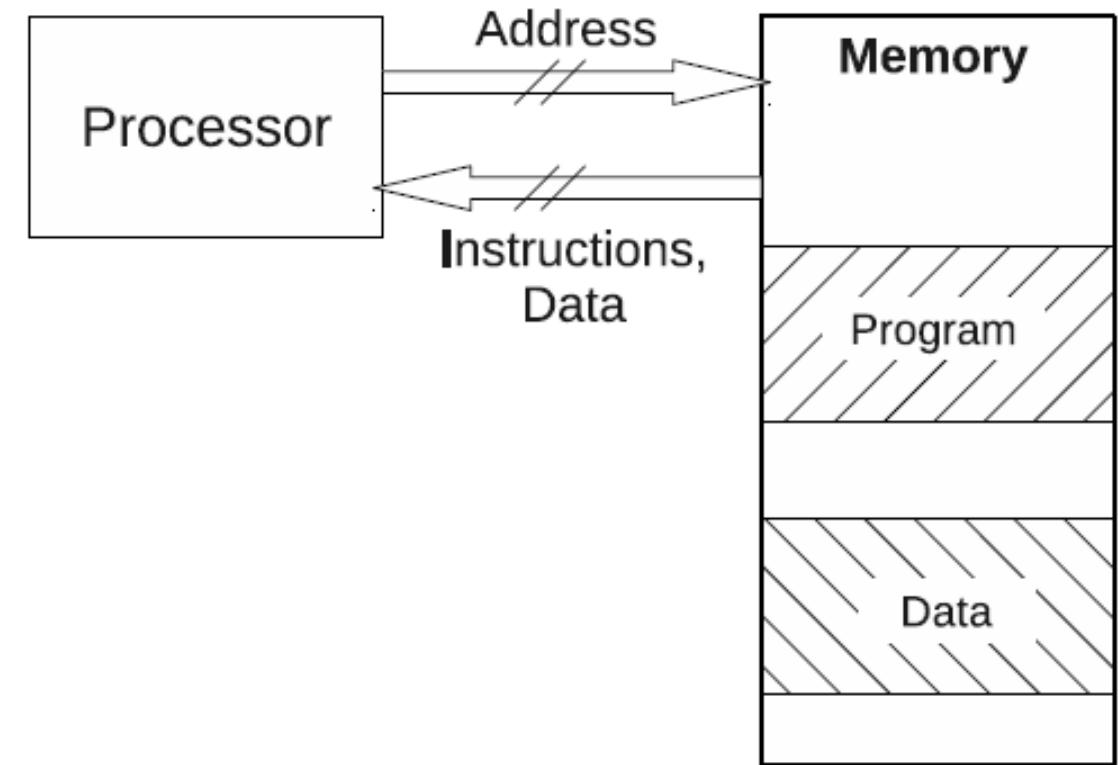
# Instructions

- Meaningful work gets done as a side-effect of executing these instructions, as the instructions can read data stored in memory, perform arithmetic, logic, and other operations on the data, and store the results back into the memory
- In fact, the processor is engaged in a perpetual loop of **fetch** and **execute**, with the real work done as the side effect of executing the instructions
- Special instructions can also control the input and output from the processor, but we will not consider those for the simple processor



# Instructions

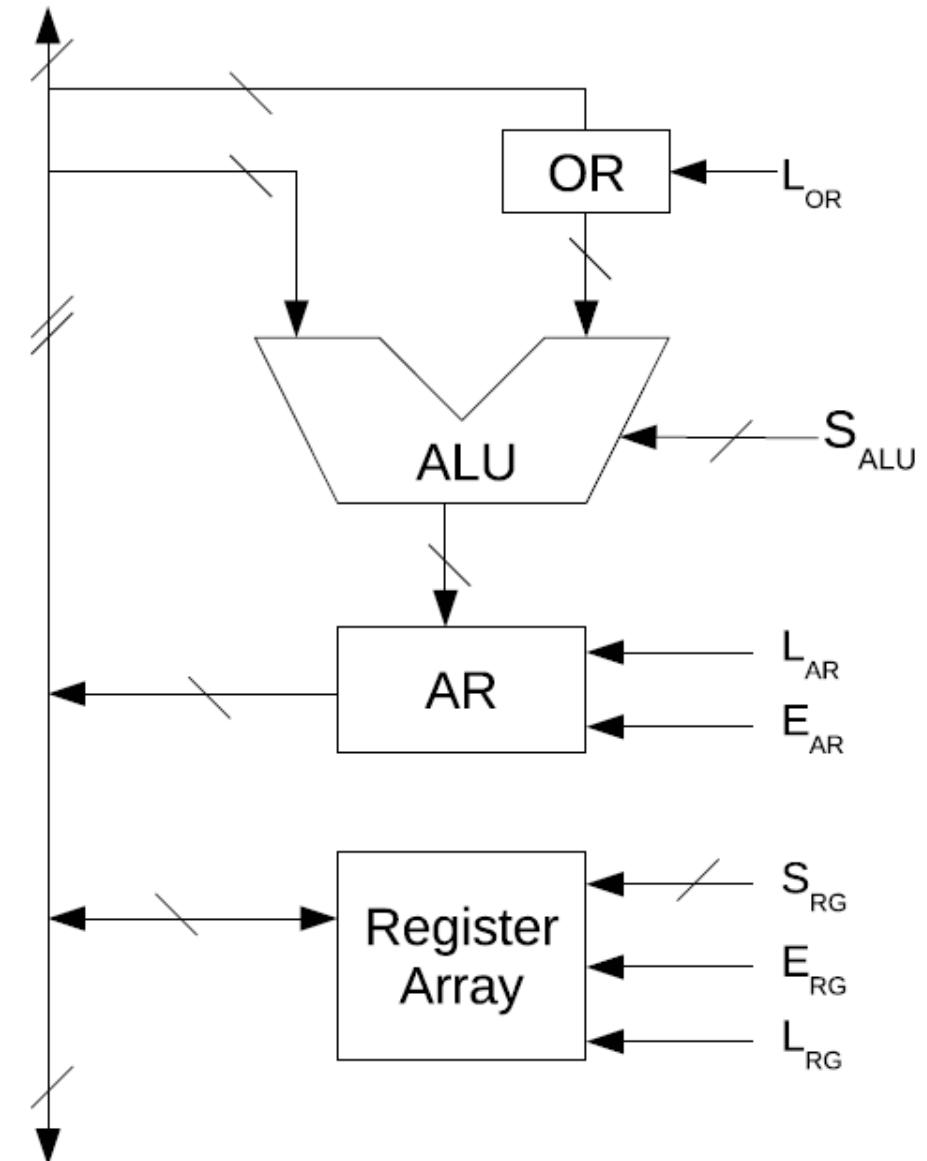
- We will follow this scheme for our simple processor
- Coded instructions are fetched from memory and executed by our processor
- We will first look at the execute step of the processor
- We will discuss the mechanism of fetching later
- We will also discuss how the endless fetch-execute loop is realized within the processor



# Lecture 28 – Processor design 3

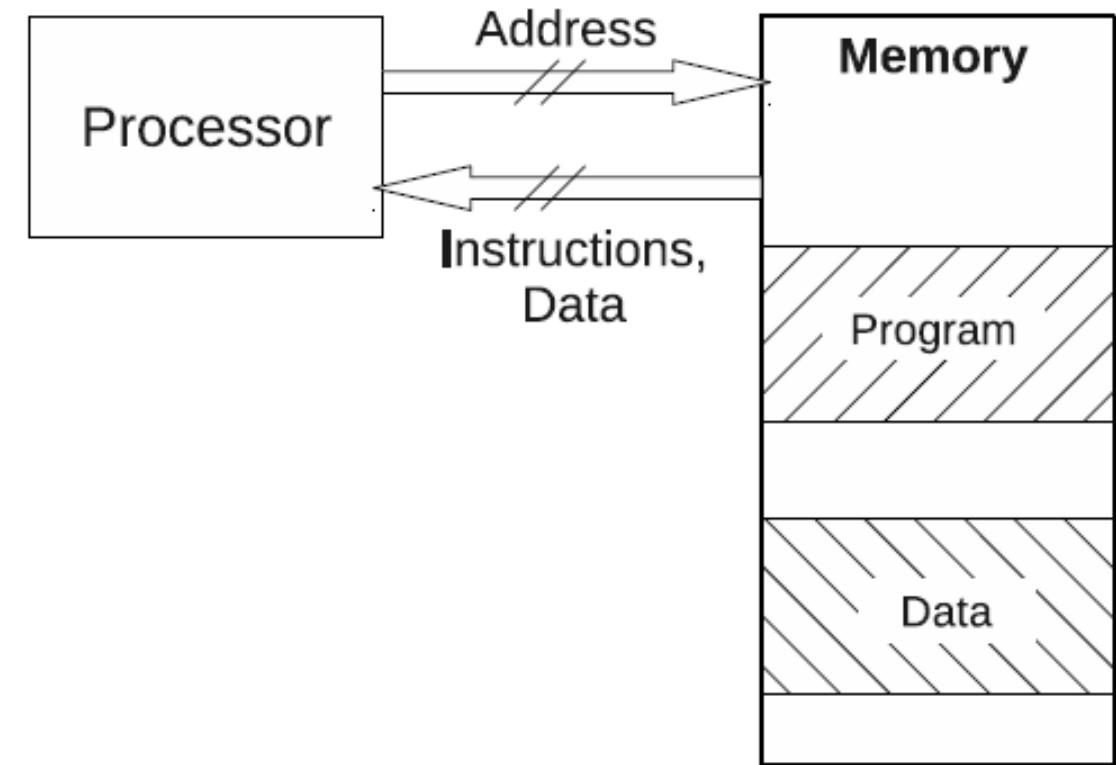
# Instructions

- A digital processor can handle only binary strings at the very lowest level
- Thus, all instructions to be carried out by a digital processor needs to be coded or represented as binary strings
- Different basic instructions have to be coded as unambiguous binary strings
- The hardware is capable of looking at a string, decoding it, and carrying out the corresponding instruction
- A sequence of such strings forms a program



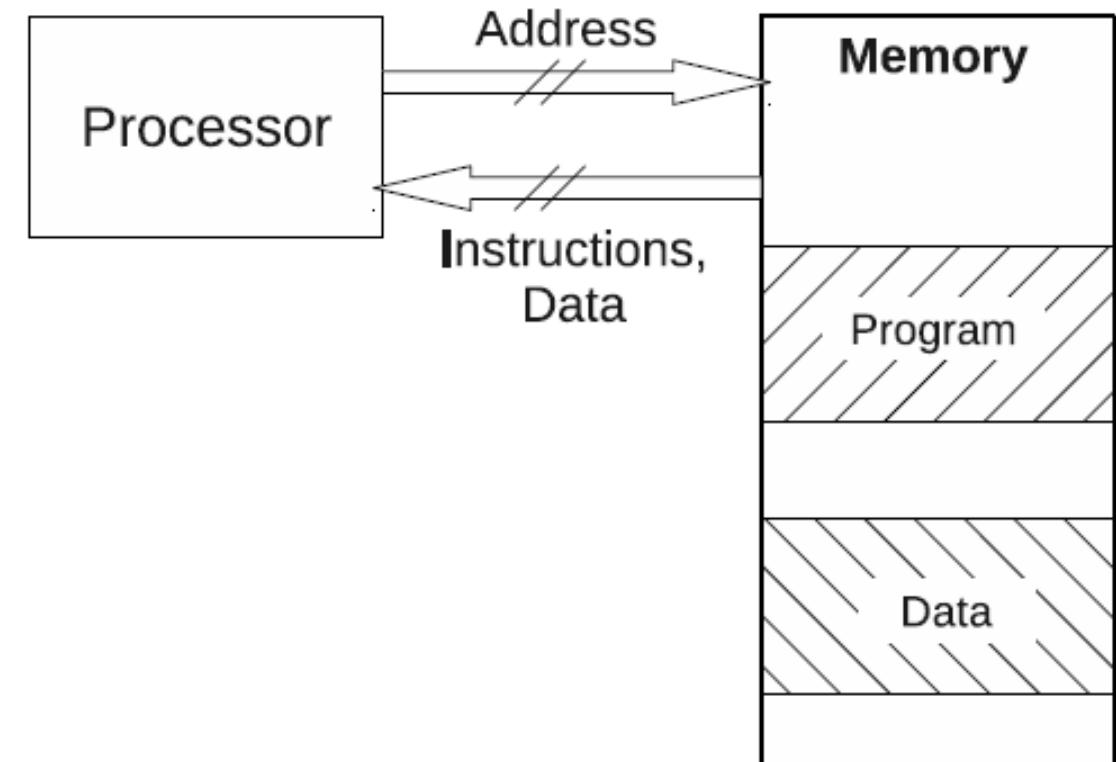
# Instructions

- Both the data to be processed and the program can be stored in the same memory ([the von Neumann or stored-program model](#))
- Instructions that make up the program are stored sequentially in memory
- **Each instruction is a binary string that encodes the operations to be performed without ambiguity**
- The processor fetches the instructions one by one from the memory and executes it or carries out the corresponding actions



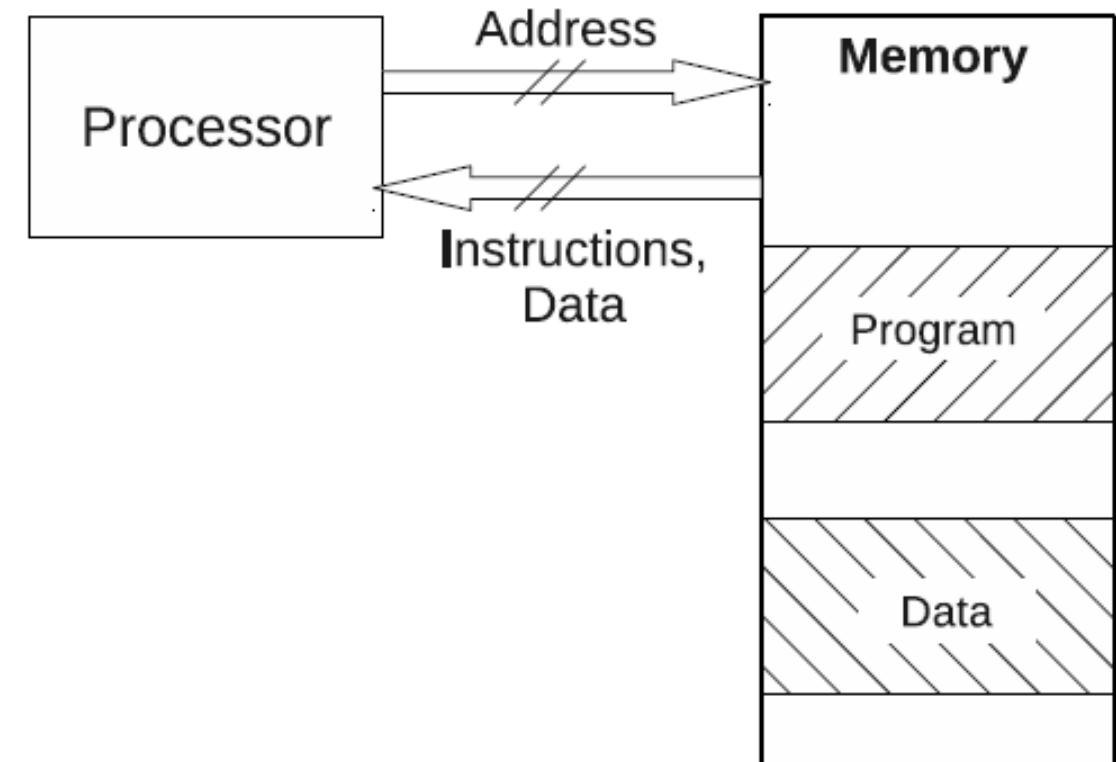
# Machine language

- The binary coded instructions are referred to as *machine instructions*, following the *machine language*
- This is really no “language” but an encoding scheme that makes unique decoding of the instructions possible
- Encoded instructions are called *machine code or opcode* for operation code
- These are understood by the processor naturally
- Importantly, that is the only “language” understood by the processor as it cannot understand high level language (like C++/Python)



# Assembly language

- Machine instructions are meant only for the processor; they require tremendous effort to interpret by us
- A mapping of the machine instructions for easier grasp by humans is used widely by processors
- This representation is essentially a one-to-one mapping from machine instructions, using mnemonics or nearly comprehensible short words and symbolic representation of internal resources like the registers
- Such a representation of the basic instructions is called *the assembly language*



# The instruction set – ALU

- Let us assume the word length of our processor is 8 bits
- Thus, all entities we will handle are 8-bits wide, which includes the coded instructions as well as data elements
- Our instruction set will have the arithmetic and logic instructions, namely: **add, subtract, and, or, xor**
- We can come up with an arbitrary **assembly to machine code mapping** as shown in the table

Assembly Instruction	Machine Code	Action
add <R>	10-1F	$[AR] \leftarrow [AR] + [<R>]$
sub <R>	20-2F	$[AR] \leftarrow [AR] - [<R>]$
xor <R>	30-3F	$[AR] \leftarrow [AR] \oplus [<R>]$
and <R>	40-4F	$[AR] \leftarrow [AR] \wedge [<R>]$
or <R>	50-5F	$[AR] \leftarrow [AR] \vee [<R>]$
cmp <R>	60-6F	$[AR] = [<R>]$

# The instruction set – ALU

- The  $\langle R \rangle$  in the first column of the table is a parameter that can be replaced by one of **R0 to R11**, with the corresponding number appearing in the lower significant half of the machine code, given in the second column
- Thus, **ADD R1** will be coded as **0x11**, XOR R8 as **0x38**, and OR R11 as **0x5B**
- Any opcode in that range can be unambiguously understood too
- Thus, **0x27** stands for **SUB R7**, **0x42** for **AND R2**, etc.

Assembly Instruction	Machine Code	Action
add $\langle R \rangle$	10-1F	$[AR] \leftarrow [AR] + [\langle R \rangle]$
sub $\langle R \rangle$	20-2F	$[AR] \leftarrow [AR] - [\langle R \rangle]$
xor $\langle R \rangle$	30-3F	$[AR] \leftarrow [AR] \oplus [\langle R \rangle]$
and $\langle R \rangle$	40-4F	$[AR] \leftarrow [AR] \wedge [\langle R \rangle]$
or $\langle R \rangle$	50-5F	$[AR] \leftarrow [AR] \vee [\langle R \rangle]$
cmp $\langle R \rangle$	60-6F	$[AR] = [\langle R \rangle]$

# The instruction set – ALU

- The last instruction performs a comparison of the register and AR without changing the value of the accumulator
- This may seem pointless as the results are not used
- However, the arithmetic and logic operations have other side-effects based on the results of the operation
- This could include overflow, carry generation, value being negative, etc. These find use in controlling loops in conjunction with conditional branching instructions we will encounter later

Assembly Instruction	Machine Code	Action
add <R>	10-1F	$[AR] \leftarrow [AR] + [<R>]$
sub <R>	20-2F	$[AR] \leftarrow [AR] - [<R>]$
xor <R>	30-3F	$[AR] \leftarrow [AR] \oplus [<R>]$
and <R>	40-4F	$[AR] \leftarrow [AR] \wedge [<R>]$
or <R>	50-5F	$[AR] \leftarrow [AR] \vee [<R>]$
cmp <R>	60-6F	$[AR] \leftarrow [AR] - [<R>]$

# The instruction set – ALU

- We should have another variation of the above arithmetic and logic instructions in which the actual operand is specified in the instruction itself as a constant
- Such instructions are frequently needed to initialize variables to a constant, such as the loop counter to 0
- Such instructions are said to provide their arguments in the *immediate mode*
- Here is the problem – all our registers are 8 bits, so these constants should be 8 bits:
  - We assume the operand is stored in the word that immediately follows the machine code that indicates such an operation

Assembly Instruction	Machine Code	Action
adi xx	01	$[AR] \leftarrow [AR] + xx$
sbi xx	02	$[AR] \leftarrow [AR] - xx$
xri xx	03	$[AR] \leftarrow [AR] \oplus xx$
ani xx	04	$[AR] \leftarrow [AR] \wedge xx$
ori xx	05	$[AR] \leftarrow [AR] \vee xx$
cmi xx	06	$[AR] \leftarrow xx$

# The instruction set – data movement

- We need instructions to move data from and to the accumulator to get our work done
- We have seen the instructions to move contents of AR from or to a register
- We also need instructions to move from AR to and from the memory, which lies outside the processor
- Registers are not sufficient to hold all our data, such as the array of marks obtained by all students
- These are kept in the memory and is brought in and out of the processor as needed
- The *movs* instruction moves the content of a register to the accumulator and the *movd* instruction moves the accumulator to a register
- The register number involved is embedded into the opcode as a parameter as before
- An additional instruction *movi* is provided to move an immediate constant directly to a register

# The instruction set – data movement

- The *load* and *stor* instructions involve a register and a memory location
- Memory resides outside of the processor
- To access the memory, one needs to give it an address to indicate which of its words is to be accessed
- The contents of the corresponding memory word will be given to the processor on a read
- The processor has to supply the contents to be written to memory for a write
- The number of bits of address determines the maximum capacity of memory that can be used
- We assume that the memory address is represented using one word of 8 bits in our processor
- Thus, the maximum memory capacity is  $2^8 = 256$  words in our simple processor

# The instruction set – data movement

- The *load* and *stor* instructions use the contents of AR as the address
- The word read from the memory is stored into the register specified in the instruction for the *load* instruction
- The value to be written is available in such a register for the *stor* instruction

Assembly Instruction	Machine Code	Action
movs <R>	70-7F	[AR] $\leftarrow$ [<R>]
movd <R>	80-8F	[<R>] $\leftarrow$ [AR]
movi <R> xx	90-9F	[<R>] $\leftarrow$ xx
stor <R>	A0-AF	[[AR]] $\leftarrow$ [<R>]
load <R>	B0-BF	[<R>] $\leftarrow$ [[AR]]

# The fetch-execute cycle

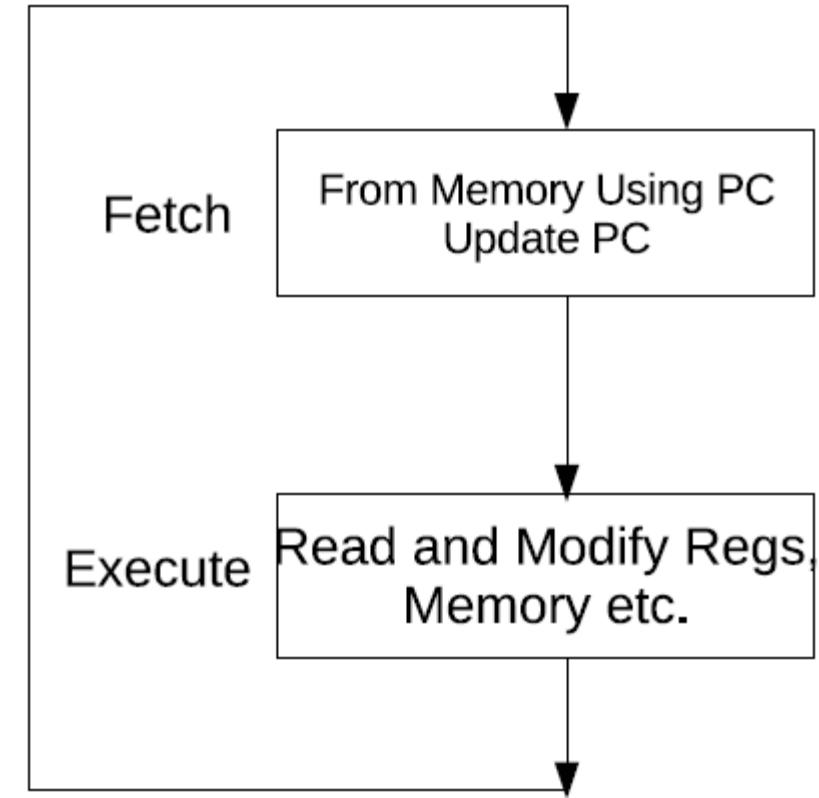
- We will look at the process of instruction fetching and execution
- The processor works autonomously as a continuous **fetch-and-execute** engine, with no other input than an external clock
- Since instructions as in the machine code are stored in memory, they have to be brought to the processor one by one and executed
- The instruction at **address  $(i + 1)$**  has to be fetched and executed after instruction at address  $i$ , since the instructions of a program are stored consecutively in the memory
- The processor has to do all these by itself

# The fetch-execute cycle

- Processors have a special register inside them that manages the process of instruction fetch by keeping track of the address of the next instruction to be fetched at all times
- This register is called **program counter or PC**
- The processing of an instruction begins with fetching its opcode from the memory word whose address is in the PC
- The contents of the PC are incremented while this happens to hold the address of the next instruction in the sequential order
- The opcode is brought to the processor and appropriate action is performed in the execution phase
- Once this is completed, the next instruction is processed by fetching it from the memory using PC as the address
- **This goes on for ever inside the processor until a special STOP instruction is encountered**
- Executing this instruction stops all activities of the processor

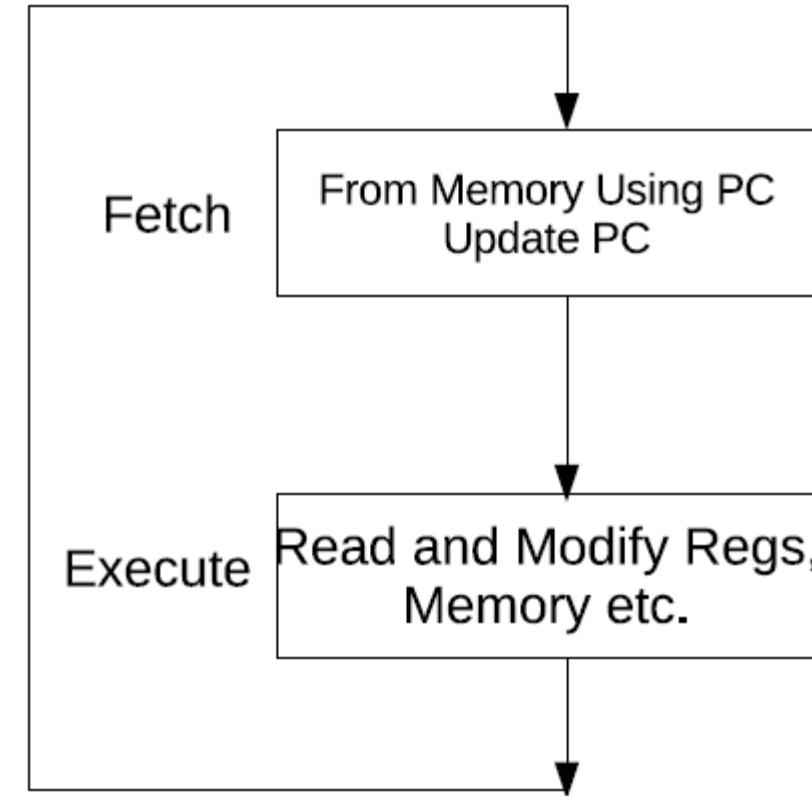
# The fetch-execute cycle

- So how do we start the process?
- It is clear that once one instruction is done with, the next one is taken up by incrementing the PC
- Thus, once the execution of a program starts, everything goes on as the program indicates
- So how to start a program?
- A program can be started by loading the address of its first instruction into the PC
- However, how does the very first program start when the computer's power is turned on?



# The fetch-execute cycle

- We know Operating System (OS) is the program that controls our computer
- The OS itself is loaded into the processor's memory from the hard disk on boot up prior to taking over the system
- Which program loads the operating system? How does that program get the control at the very beginning?
- Modern PCs have a program called the **BIOS (Basic Input Output System)**, which is the very first one to get control of the processor
- How does the BIOS get control?



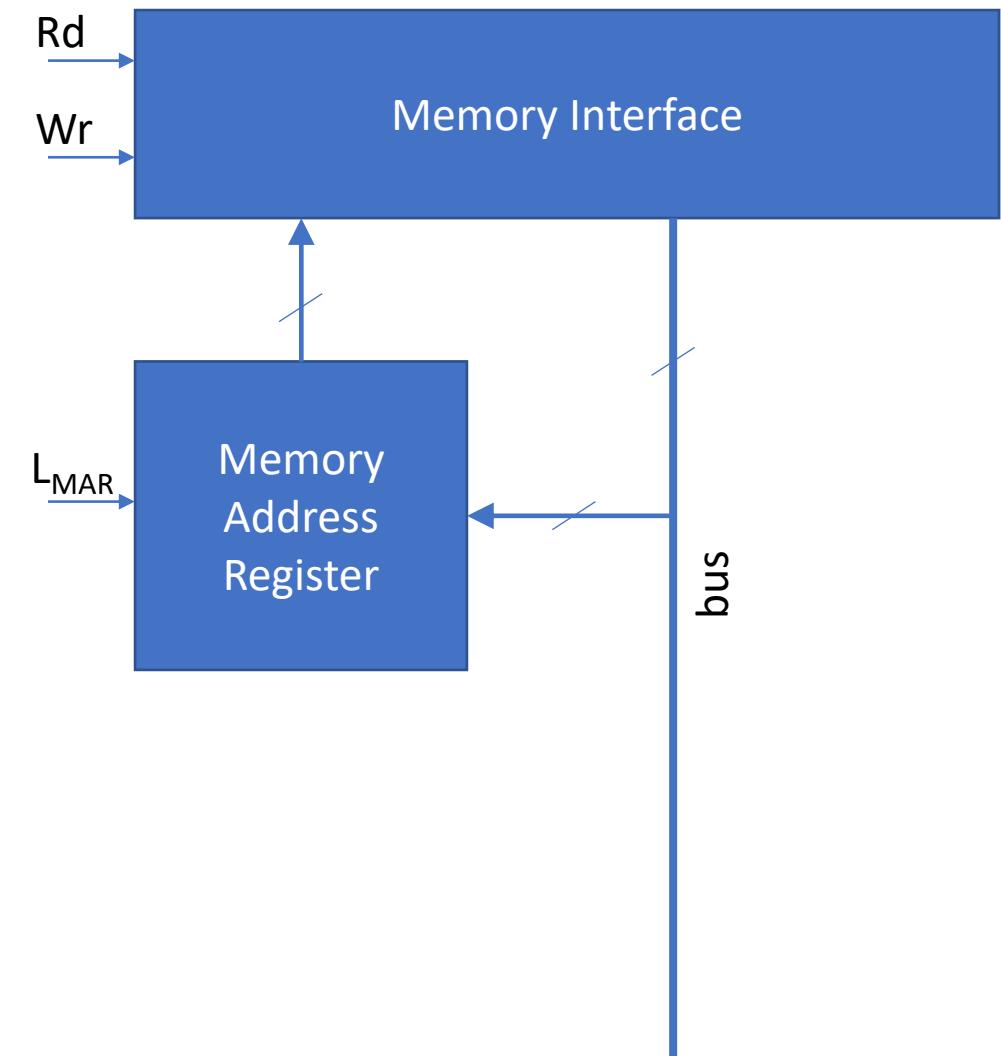
# The fetch-execute cycle

- The processor hardware has a special feature to load a value of 0 to the PC when power is turned on or when the reset button of the computer is pressed (*literally* “resets” the “PC”)
- Thus, the very first program that gets control is the one that is saved at memory address 0
- Computer manufacturers place a special program at address 0 that has the BIOS program, which knows how to load the operating system from the boot record and proceed accordingly
- Any corruption in the BIOS can be very detrimental to booting the computer

# Lecture 29 – Processor design 4

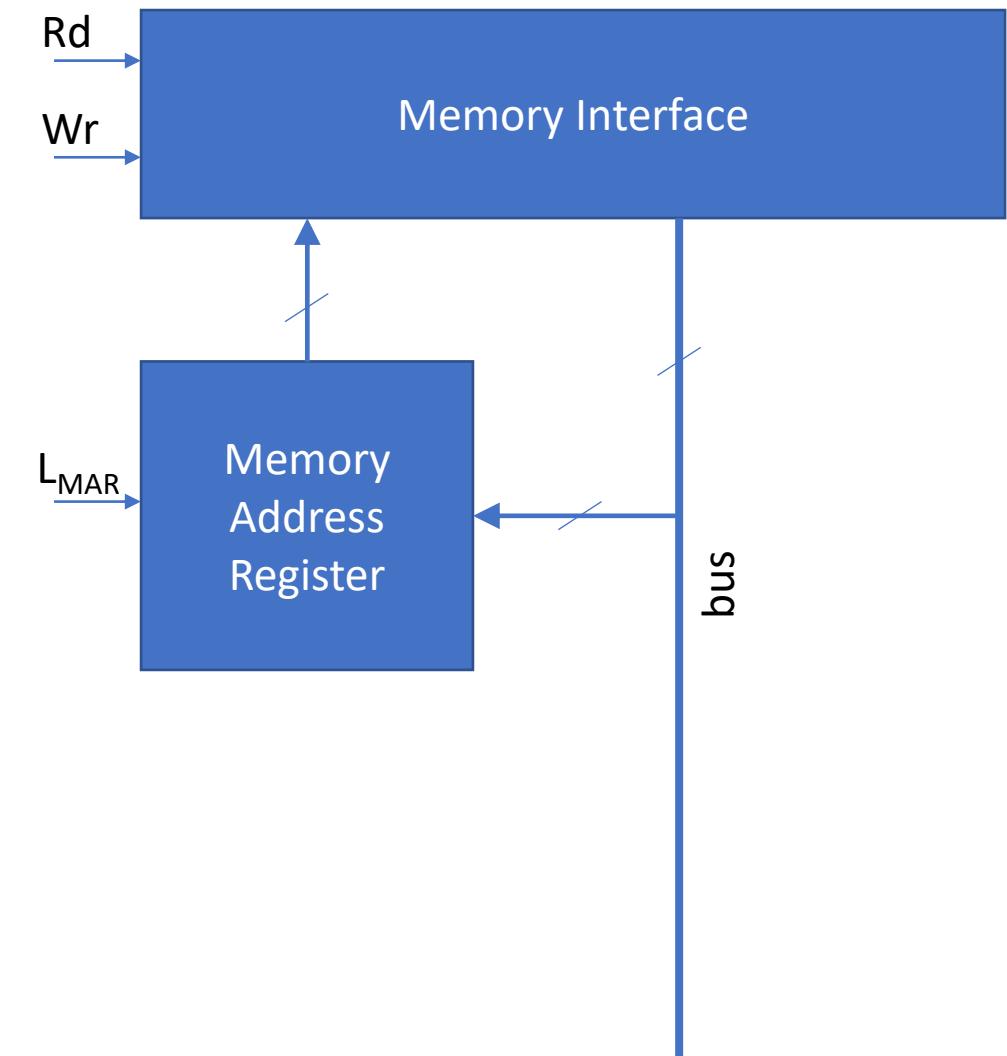
# Memory access

- How does the simple processor access the memory?
- We will use a memory interface that supplies the address to the memory along with the signals to indicate if a read or a write is desired
- Data should be presented separately for writes; data supplied by the memory should be used inside the processor for reads
- We assume an external memory interface consisting of address lines, data lines, and two control lines
- The data lines are connected directly to the data lines of the bus, as if the memory is a large register array, but outside of the processor



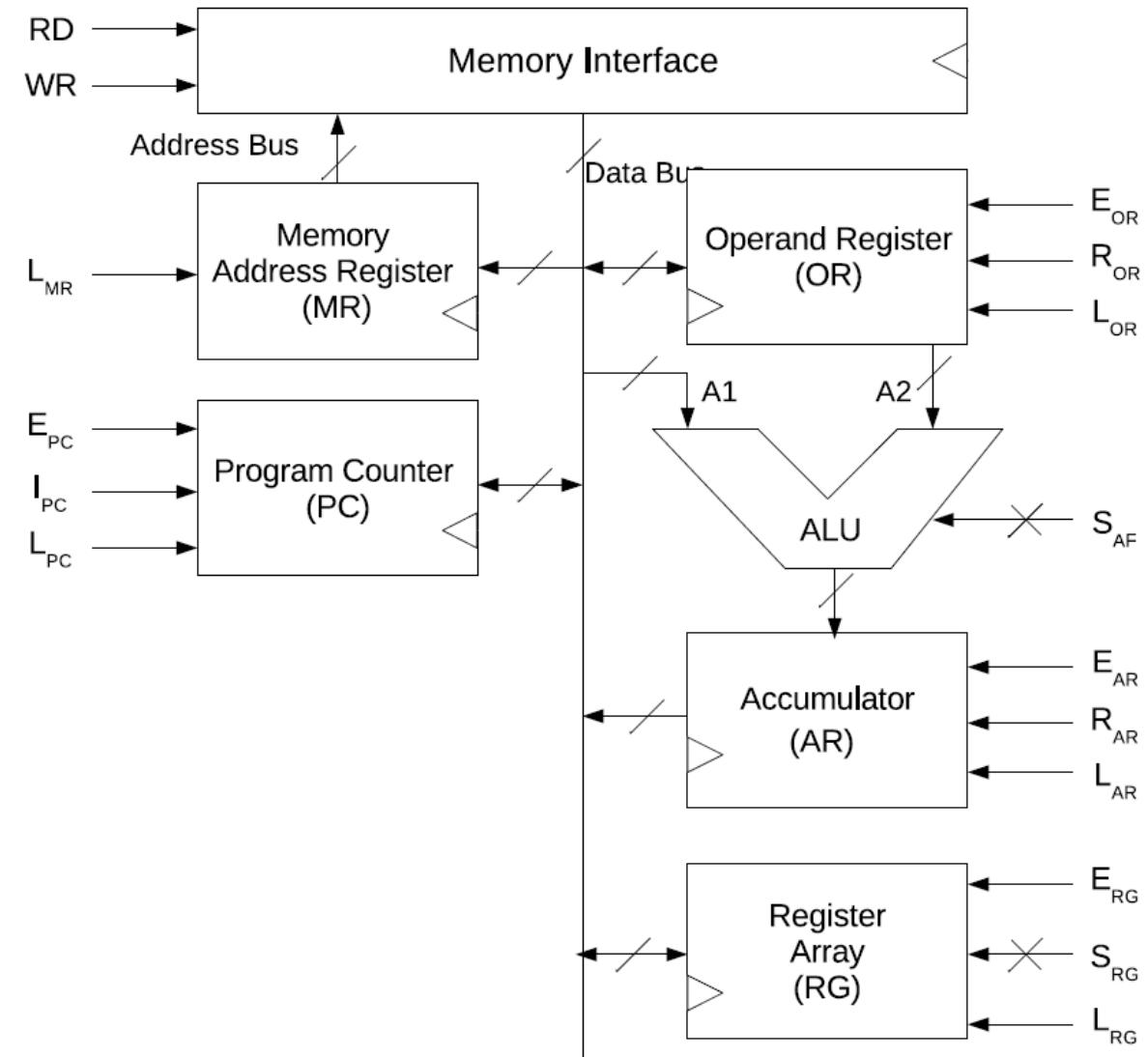
# Memory access

- The address has to be supplied separately, prior to the read or write operation
- We assign a memory address register (MAR) to hold the address
- The MAR is connected to the bus like other registers and can be written to from the bus
- There is usually no need to enable the MAR to the internal bus
- It can be assumed to be enabled always to the external memory interface
- Two control lines RD and WR are sent to the memory to indicate memory read and write respectively



# Enhanced enhanced single bus architecture

- With this information, the enhanced single bus architecture is modified to include additional components
- The memory address register to store the next memory address to be accessed
- The program counter to store the current address of the instruction being performed



# Two more instructions

- We will introduce two more simple instructions: **NOP** instruction for no operation and **STOP** instruction to stop the processor.
- NOP does nothing: when it is executed nothing at all changes in the processor or memory.
- It may seem superfluous, but comes into use when nothing is required from the processor other than spending the required clocks to fetch and execute this instruction.
- The STOP instruction terminates the endless fetch-execute cycle that the processor is engaged in.
- The processor enters a state of complete inaction when the STOP instruction is executed.
- There is no way to come out of this state through a program as the processor has stopped looking at programs!
- Thus, the only way to come out is a hard reset through a reset button or through power cycling.

Assembly Instruction	Machine Code	Action
nop	00	-
stop	07	(Stops fetch)

# Implementing instructions – ALU

- Consider the simple instruction:

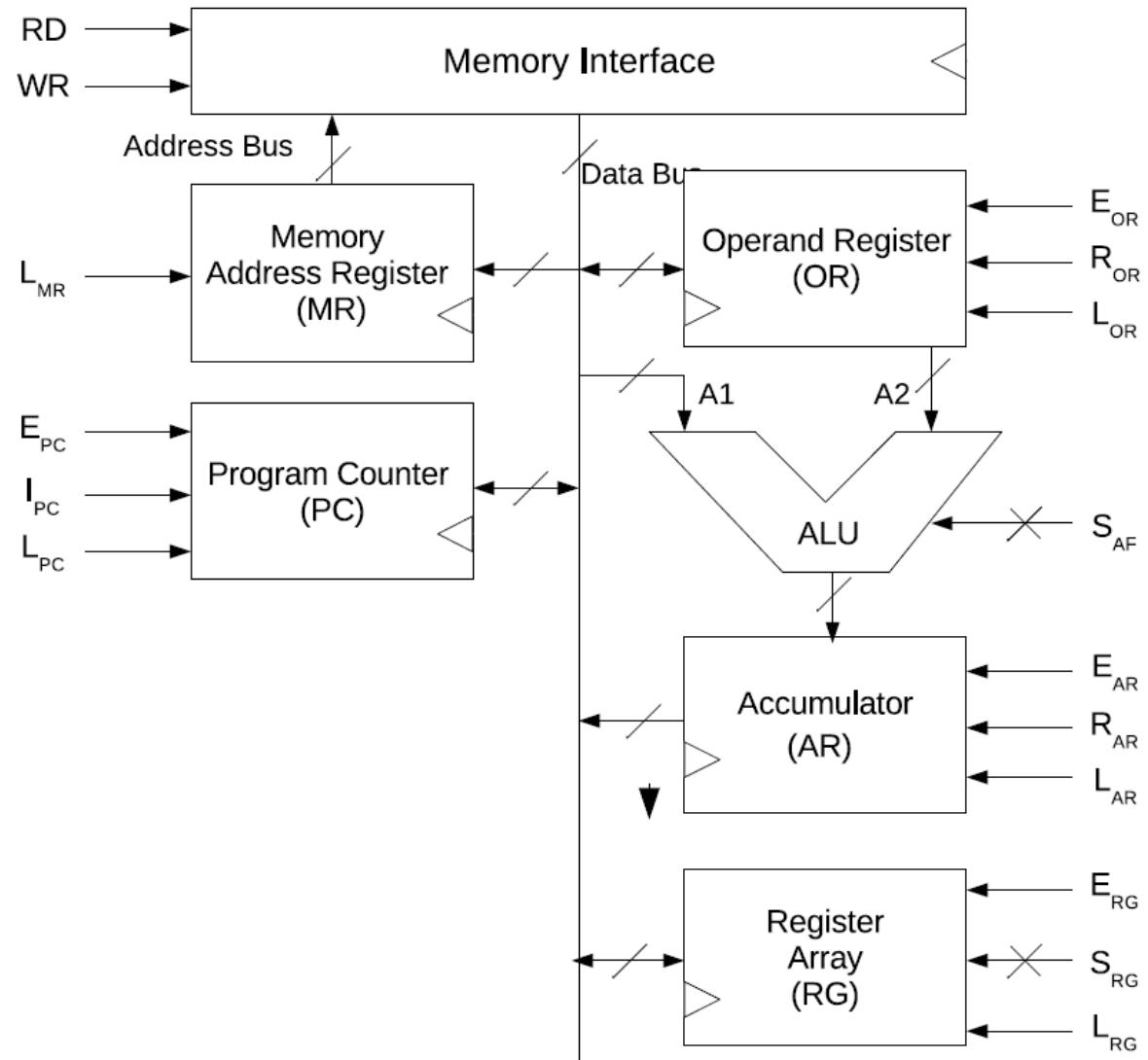
**ADD <R>**

- This instruction can be executed in two clock cycles:

Ck 3: We need to enable RG and load the OR with the value [<R>] using the select lines for RG

Ck 4: Enable AR, set the ALU select lines for ADD operation, activate load AR

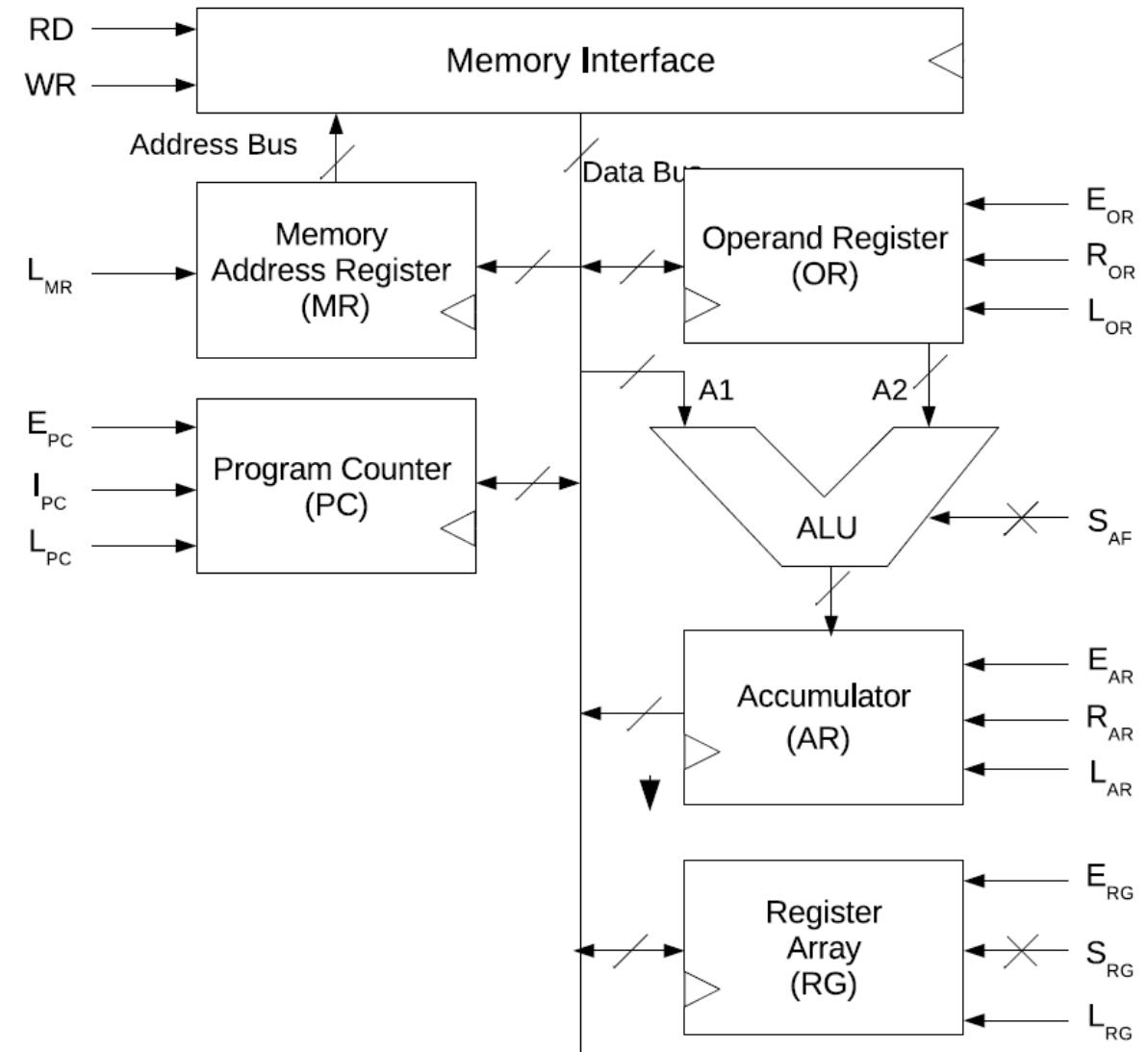
add <R>	Ck 3. $E_{RG}$ , $L_{OR}$ Ck 4: $E_{AR}$ , $L_{AR}$ , End	$S_{RG} \leftarrow <R>$ $S_{ALU} \leftarrow ADD$
---------	--	---



# Implementing instructions – ALU

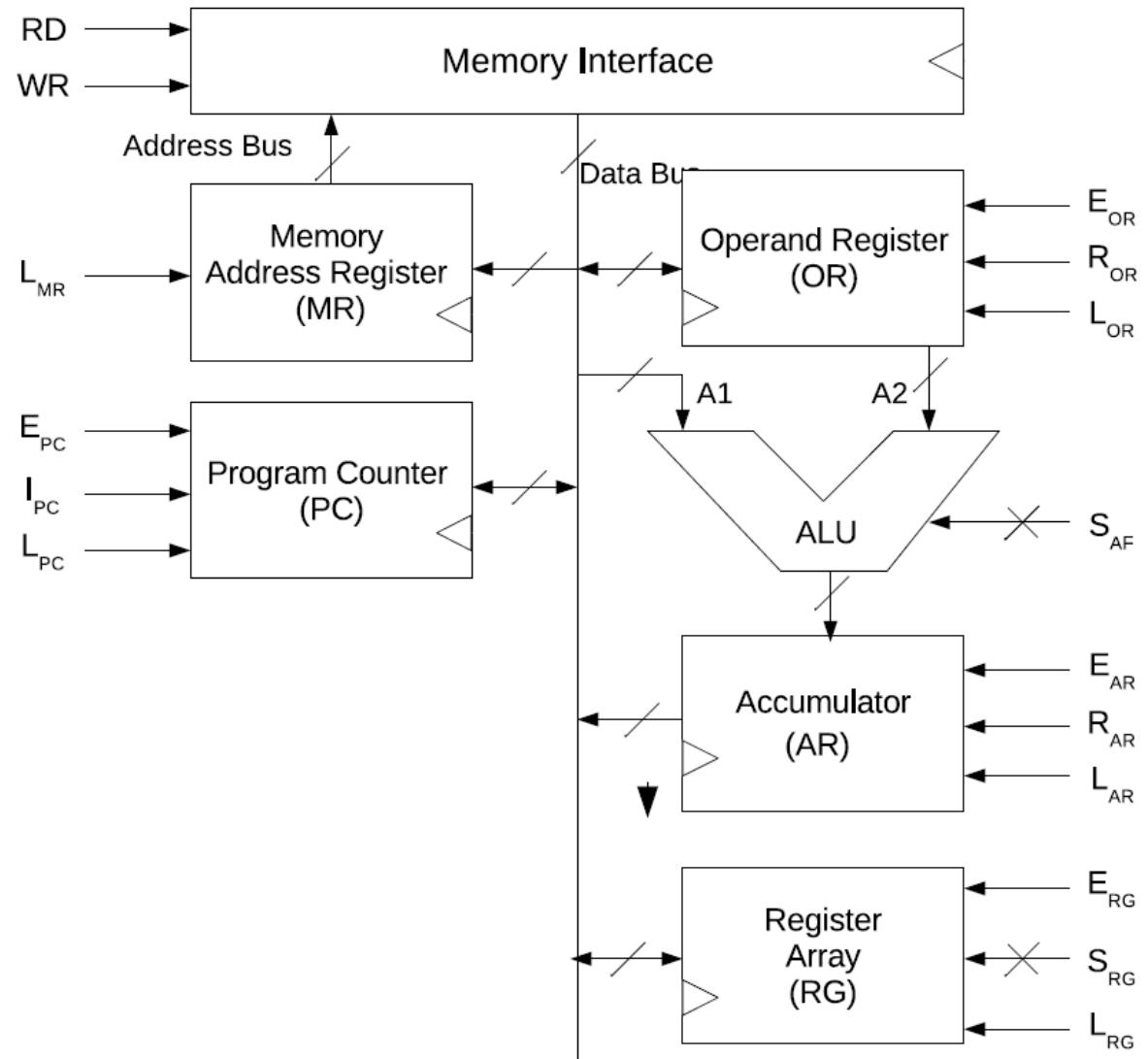
ADD <R>

CLK  
 $E_{RG}, L_{OR}$   
 $S_{RG}$



# Implementing instructions – ALU

Instruction	Control Signals	Select Signals
add <R>	Ck 3: E <sub>RG</sub> , L <sub>OR</sub> Ck 4: E <sub>AR</sub> , L <sub>AR</sub> , End	S <sub>RG</sub> ← <R> S <sub>ALU</sub> ← ADD
sub <R>	Ck 3: E <sub>RG</sub> , L <sub>OR</sub> Ck 4: E <sub>AR</sub> , L <sub>AR</sub> , End	S <sub>RG</sub> ← <R> S <sub>ALU</sub> ← SUB
xor <R>	Ck 3: E <sub>RG</sub> , L <sub>OR</sub> Ck 4: E <sub>AR</sub> , L <sub>AR</sub> , End	S <sub>RG</sub> ← <R> S <sub>ALU</sub> ← XOR
and <R>	Ck 3: E <sub>RG</sub> , L <sub>OR</sub> Ck 4: E <sub>AR</sub> , L <sub>AR</sub> , End	S <sub>RG</sub> ← <R> S <sub>ALU</sub> ← AND
or <R>	Ck 3: E <sub>RG</sub> , L <sub>OR</sub> Ck 4: E <sub>AR</sub> , L <sub>AR</sub> , End	S <sub>RG</sub> ← <R> S <sub>ALU</sub> ← OR
cmp <R>	Ck 3: E <sub>RG</sub> , L <sub>OR</sub> Ck 4: E <sub>AR</sub> , End	S <sub>RG</sub> ← <R> S <sub>ALU</sub> ← CMP
nop	Ck 3: End	-



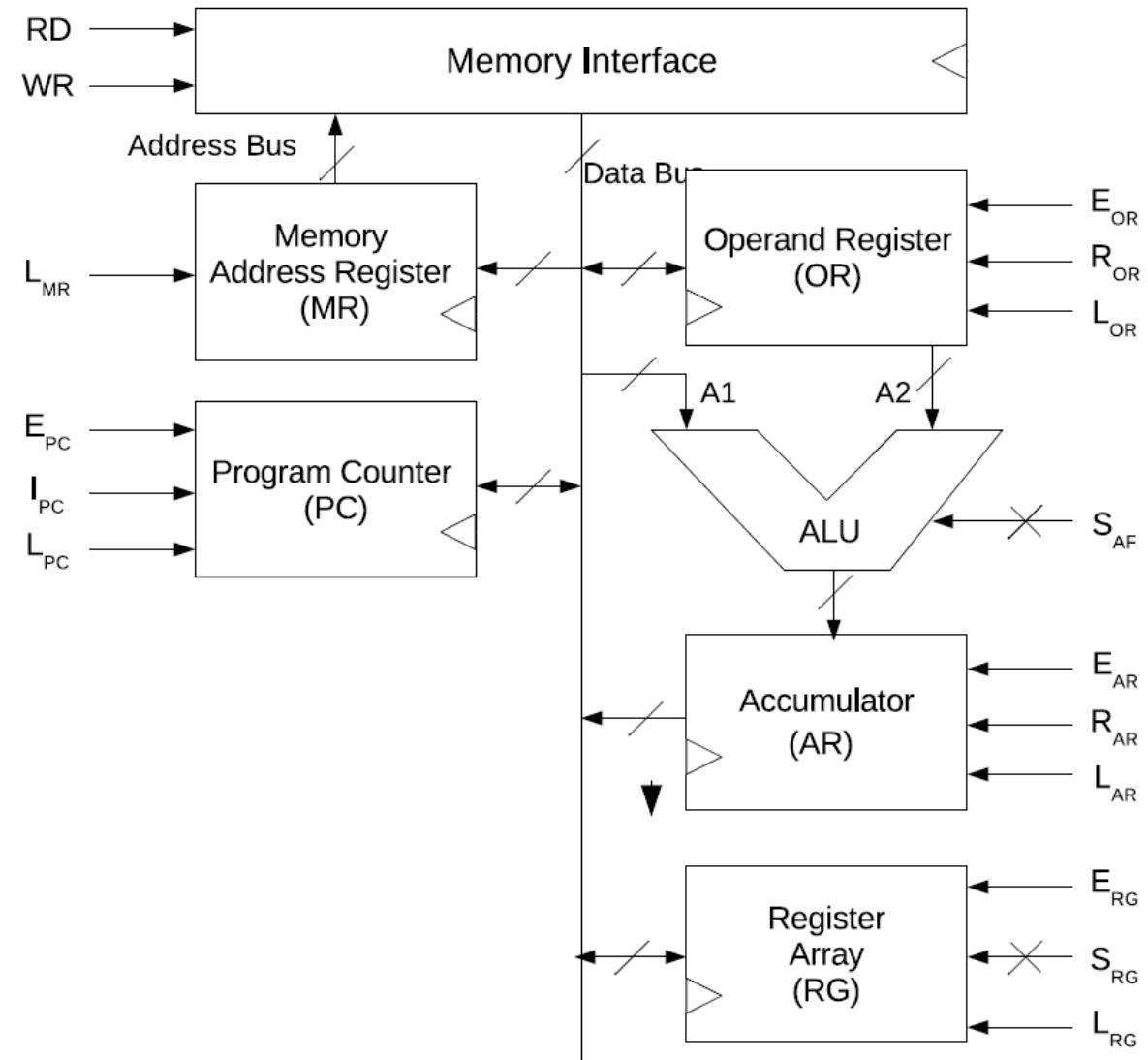
# Implementing instructions

---

- A clock cycle in which one basic operation is performed is called a *microcycle*
- The combination of control signals that are active (or at level 1) in a microcycle determines what operation is performed in that cycle
- The operation performed in a microcycle is often referred to as a *microinstruction*
- The execution of each machine instruction (such as ADD <R>) needs one or more microcycles
- Faster instructions take fewer microcycles and vice versa
- The number of microcycles needed for different machine instructions depends on the processor architecture – some processors are “hardwired” to perform certain instructions very rapidly

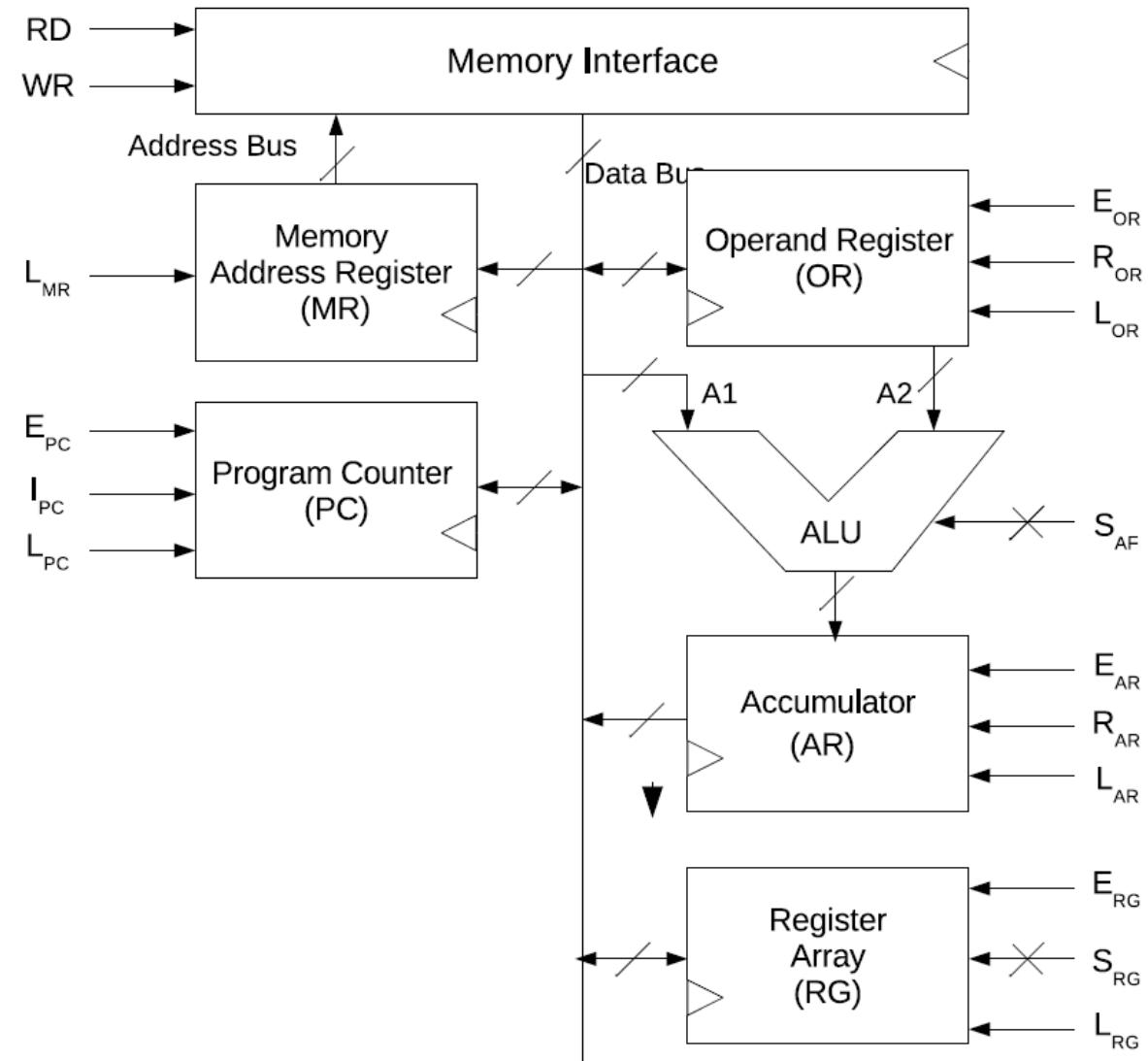
# Implementing instructions – data movement

- Moving from AR to a register is achieved using the *movd* instruction
- It is quite straightforward to implement, enable AR and load RG, and needs only one cycle to execute
- To load from register to ALU: we load the register value to the bus by setting  $S_{RG}$  and choosing the pass option of the ALU
- The register contents are available at the input of AR in the same clock cycle
- If  $L_{AR}$  is also active in that clock, the data will go from the register to ALU input through the bus, pass through the ALU to AR and be stored into it all in *one clock cycle!*



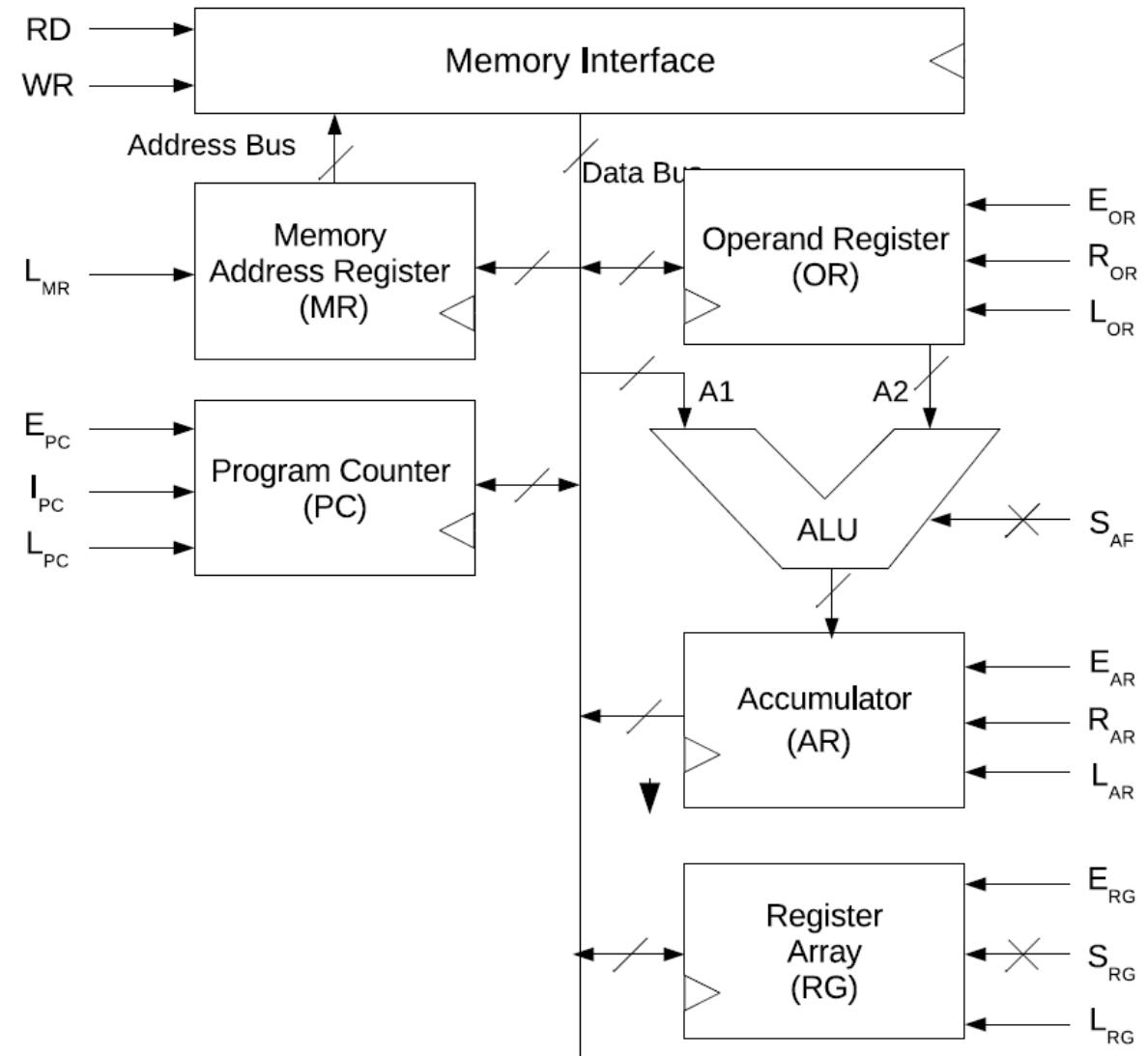
# Implementing instructions – data movement

- Now, we look at the two data movement instructions, namely, *load* and *stor*
- The load instruction reads a value from the memory to a register
- The address of the memory location is given in AR
- The memory sits outside of the processor and is accessed by giving it an address through the MAR register and a command through the RD and WR lines, as is appropriate
- The first microcycle of execution moves the address from AR to MAR for both instructions
- This is done using the  $E_{AR}$  and  $L_{MR}$  signals



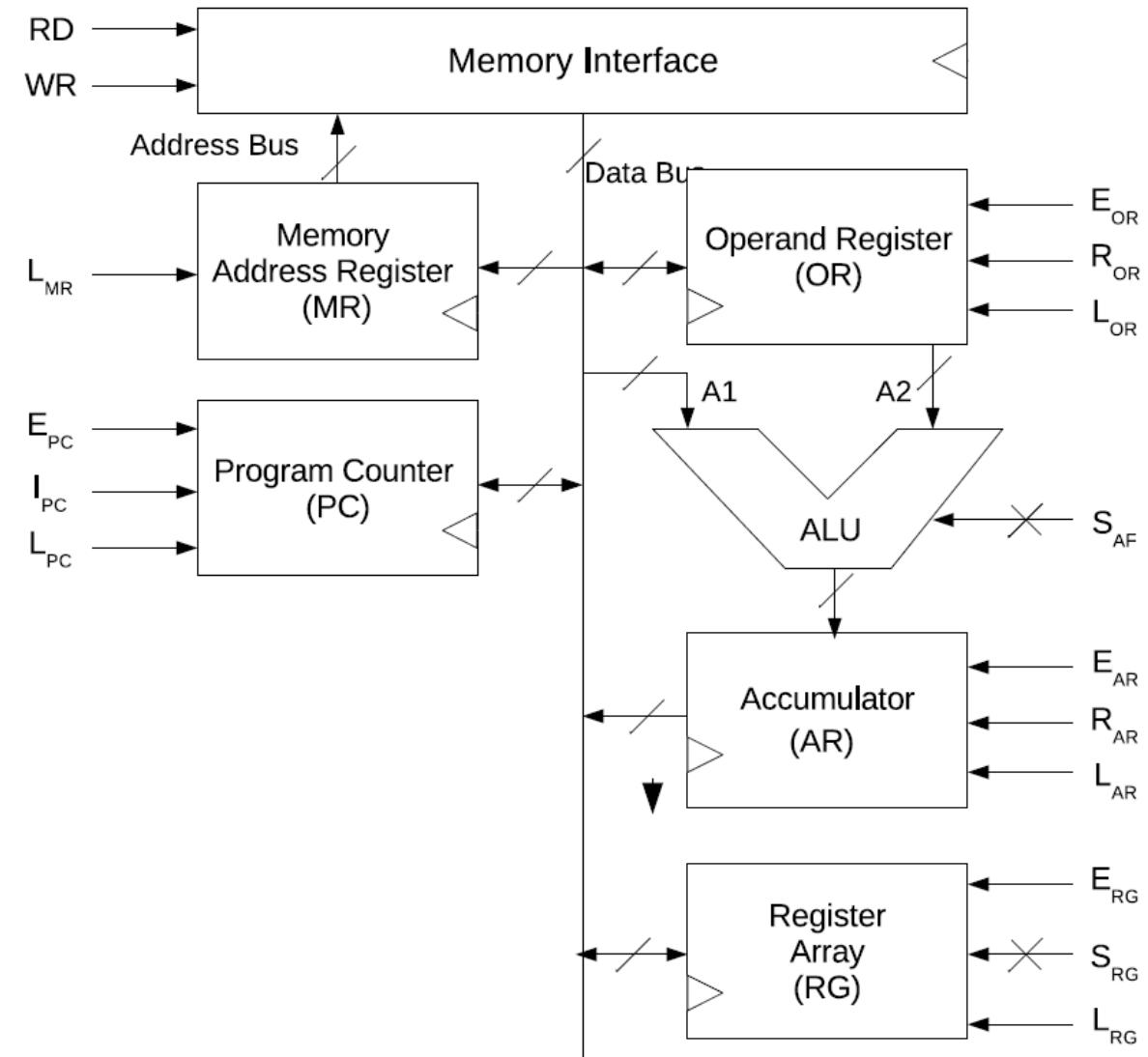
# Implementing instructions – data movement

- In case of **load**, the next microcycle asks the memory to read the location using RD and  $L_{RG}$  is activated
- In case of **stor**, the next microcycle activates WR and  $E_{RG}$
- In case of **load**, we assume the value will be available on the data bus before the end of the clock cycle
- Thus, the memory is treated like an external register file, whose address (or select) is given through MAR
- However, in practice, the memory is significantly slower than the registers and the read cannot complete in the same clock cycle
- We will ignore that aspect as we are designing a very simple processor
- Thus, the data movement instructions only take 2 clock cycles for their execution on our architecture



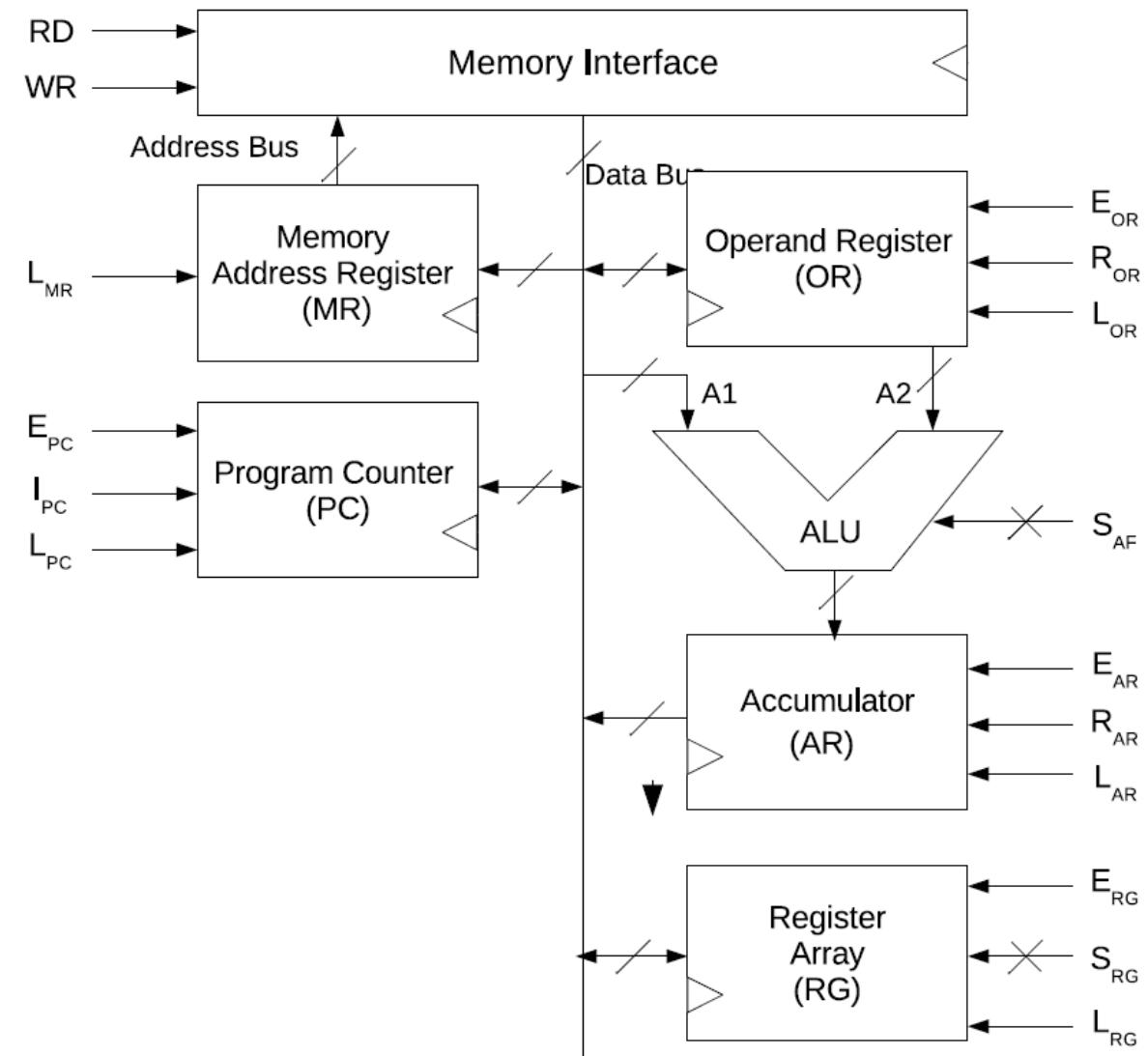
# Implementing instructions – data movement

- The third data movement operation uses an **immediate argument**
- This is very similar to *load* except for the specification of the source memory address
- The source value is stored immediately along with the instruction
- As we have seen before, the immediate argument  $xx$  is stored in a memory location with address  $(k+1)$  if the opcode for *movi* is stored in a memory location with address  $k$
- Moreover, we assume that as the opcode for *movi* is fetched from  $k$ , the PC value is incremented by 1 to point to the next instruction



# Implementing instructions – data movement

- Thus, when the execution of *movi* starts, the PC is pointing to the word following the opcode
- This word holds the **immediate operand xx**
- Thus, the situation is similar to *load*, except for the **PC supplying the address of the operand instead of AR**
- Thus, the execution of *movi* proceeds very similarly:  $E_{PC}$ ,  $L_{MR}$
- However, the PC needs to point to the next real opcode at the end of executing *movi*
- We achieve this by incrementing PC while it is loaded onto MAR, by enabling the  $I_{PC}$  control signal
- Thus, the microcycle activates:  $E_{PC}$ ,  $L_{MR}$ ,  $I_{PC}$
- Then the memory can be read as before



# Lecture 31 – Processor design 6

# Instruction fetch

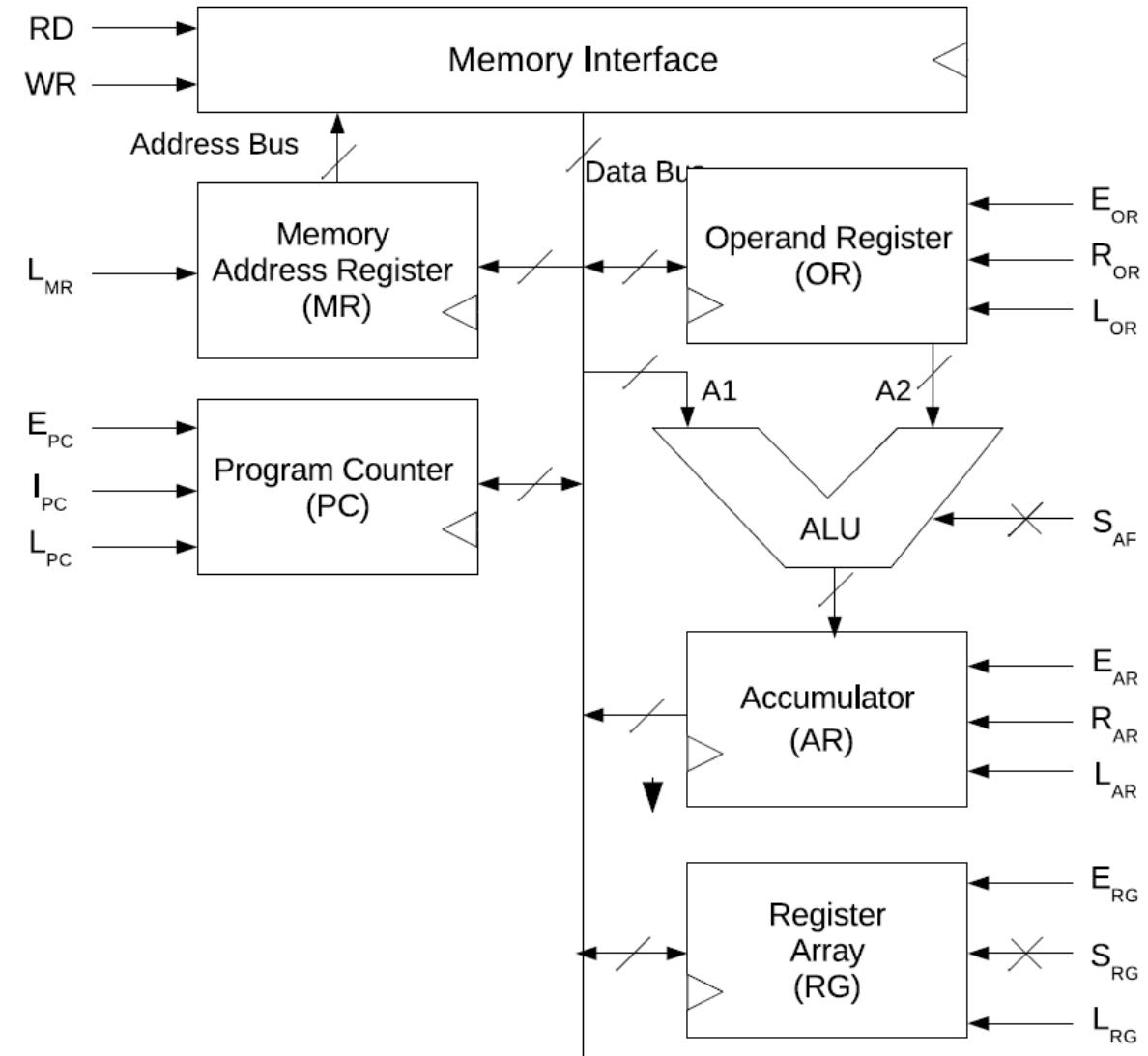
- We are now ready to tackle **instruction fetch** - we know it involves reading a word from the memory, using the PC value as the address.
- This can be achieved using the following two microinstructions:

Ck 1:  $E_{PC}$ ,  $L_{MR}$ ,  $I_{PC}$   
Ck 2:  $RD$ ,  $L_{IR}$

- In the first cycle, PC value is loaded to MAR and the PC is simultaneously incremented, so that the next fetch will be from the next word in memory.
- In the second cycle, the memory word at the address given by MAR is read and the value obtained is loaded into a special **instruction register or IR**.
- The instruction register holds the entire opcode, which then needs to be decoded or deciphered to select one of the possible actions.

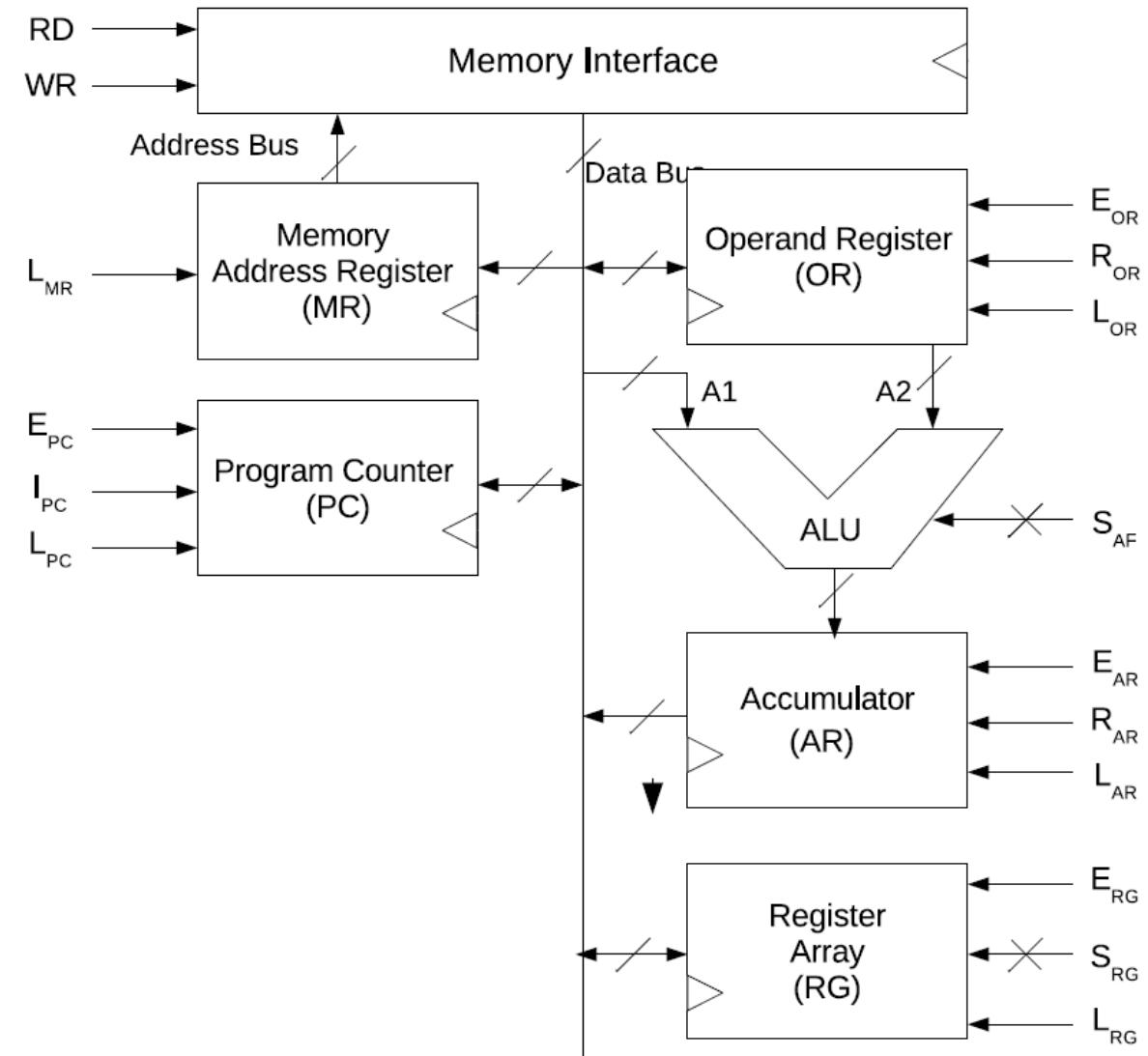
# Implementing instructions – data movement

- *movi xx* data movement operation uses an **immediate argument**
- This is very similar to *load* except for the specification of the source memory address
- The source value is stored immediately along with the instruction
- As we have seen before, the immediate argument *xx* is stored in a memory location with address (*k+1*) if the opcode for *movi* is stored in a memory location with address *k*
- Moreover, we assume that as the opcode for *movi* is fetched from *k*, the PC value is incremented by 1 to point to the next instruction



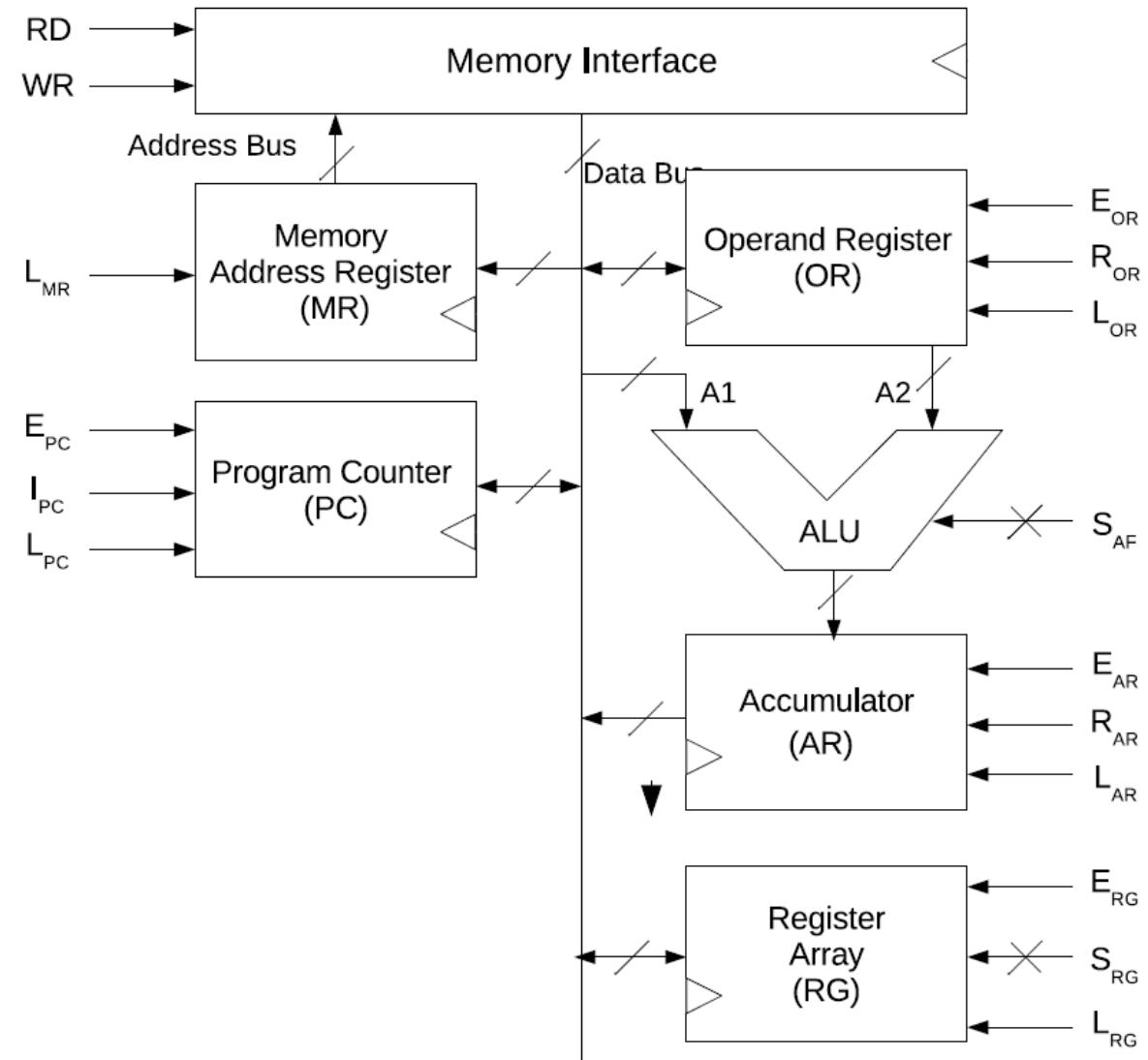
# Implementing instructions – data movement

- Thus, when the execution of *movi* starts, the PC is pointing to the word following the opcode
- This word holds the **immediate operand xx**
- Thus, the situation is similar to *load*, except for the **PC supplying the address of the operand instead of AR**
- Thus, the execution of *movi* proceeds very similarly:  $E_{PC}$ ,  $L_{MR}$
- However, the PC needs to point to the next real opcode at the end of executing *movi*
- We achieve this by incrementing PC while it is loaded onto MAR, by enabling the  $I_{PC}$  control signal
- Thus, the microcycle activates:  $E_{PC}$ ,  $L_{MR}$ ,  $I_{PC}$
- Then the memory can be read as before



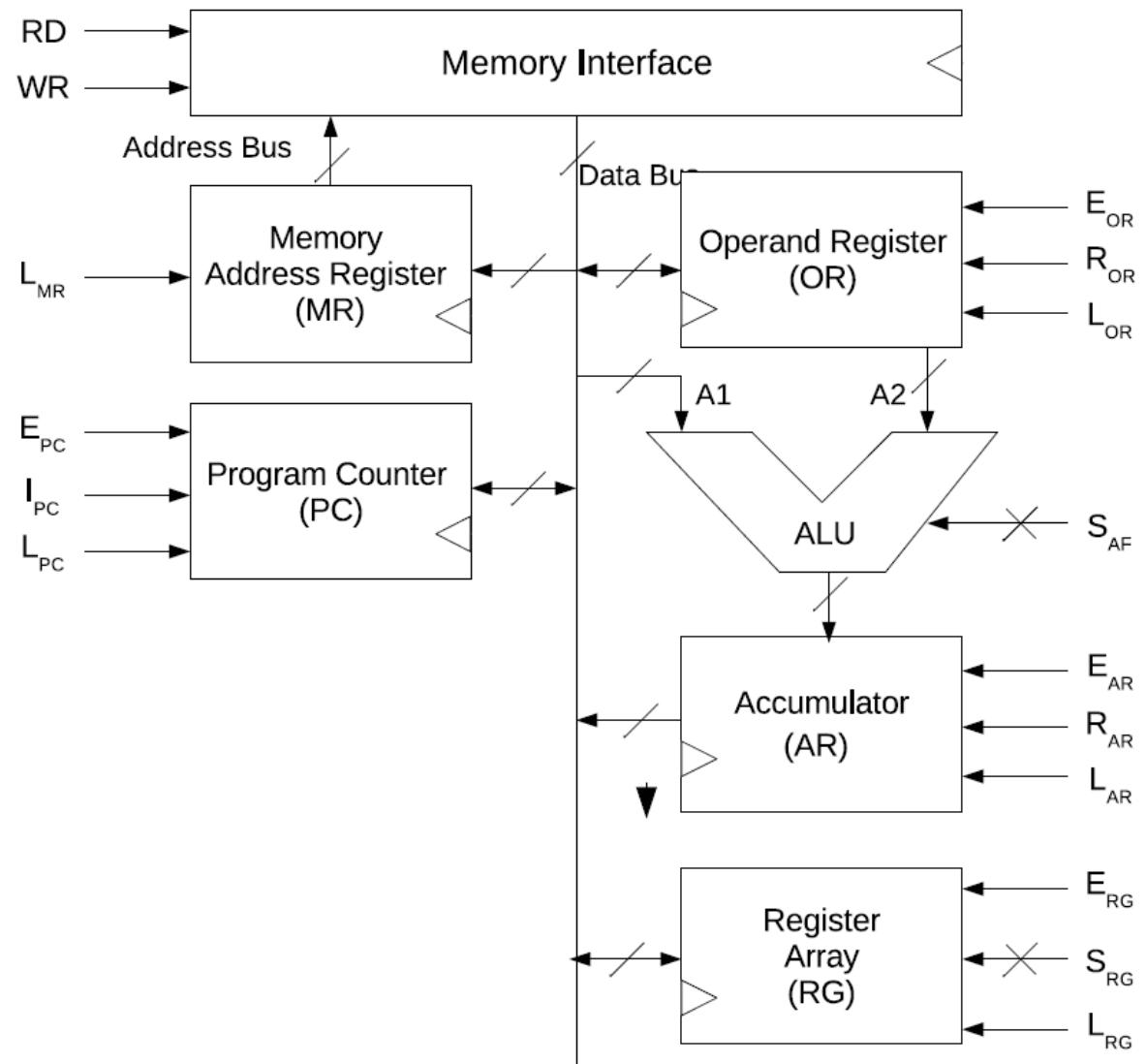
# Implementing instructions – ALU immediate

- The only difference between an instruction that uses a register argument and one that uses an immediate argument is the source of the argument
- Earlier, we loaded the source from the selected register in one clock to OR through the bus
- In immediate case, we have to get it from the memory, and we know that the PC holds the operand's address when execution starts
- All arithmetic and logic instructions can be implemented keeping this in mind
- Note that these instructions require 3 clock cycles each for their execution



# Implementing instructions

Instruction	Control Signals	Select Signals
movs <R>	Ck 3: E <sub>RG</sub> , L <sub>AR</sub> , End	S <sub>RG</sub> ← <R>, S <sub>ALU</sub> ← PASS0
movd <R>	Ck 3: E <sub>AR</sub> , L <sub>RG</sub> , End	S <sub>RG</sub> ← <R>
load <R>	Ck 3: E <sub>AR</sub> , L <sub>MR</sub> Ck 4: RD, L <sub>RG</sub> , End	- S <sub>RG</sub> ← <R>
stor <R>	Ck 3: E <sub>AR</sub> , L <sub>MR</sub> Ck 4: E <sub>RG</sub> , WR, End	- S <sub>RG</sub> ← <R>
movi <R> xx	Ck 3: E <sub>PC</sub> , L <sub>MR</sub> , I <sub>PC</sub> Ck 4: RD, L <sub>RG</sub> , End	- S <sub>RG</sub> ← <R>
adi xx	Ck 3: E <sub>PC</sub> , L <sub>MR</sub> , I <sub>PC</sub> Ck 4: RD, L <sub>OR</sub> Ck 5: E <sub>AR</sub> , L <sub>AR</sub> , End	- - S <sub>ALU</sub> ← ADD
sbi xx	Ck 3: E <sub>PC</sub> , L <sub>MR</sub> , I <sub>PC</sub> Ck 4: RD, L <sub>OR</sub> Ck 5: E <sub>AR</sub> , L <sub>AR</sub> , End	- - S <sub>ALU</sub> ← SUB
xri xx	Ck 3: E <sub>PC</sub> , L <sub>MR</sub> , I <sub>PC</sub> Ck 4: RD, L <sub>OR</sub> Ck 5: E <sub>AR</sub> , L <sub>AR</sub> , End	- - S <sub>ALU</sub> ← XOR
ani xx	Ck 3: E <sub>PC</sub> , L <sub>MR</sub> , I <sub>PC</sub> Ck 4: RD, L <sub>OR</sub> Ck 5: E <sub>AR</sub> , L <sub>AR</sub> , End	- - S <sub>ALU</sub> ← AND
ori xx	Ck 3: E <sub>PC</sub> , L <sub>MR</sub> , I <sub>PC</sub> Ck 4: RD, L <sub>OR</sub> Ck 5: E <sub>AR</sub> , L <sub>AR</sub> , End	- - S <sub>ALU</sub> ← OR
cni xx	Ck 3: E <sub>PC</sub> , L <sub>MR</sub> , I <sub>PC</sub> Ck 4: RD, L <sub>OR</sub> Ck 5: E <sub>AR</sub> , End	- - S <sub>ALU</sub> ← CMP

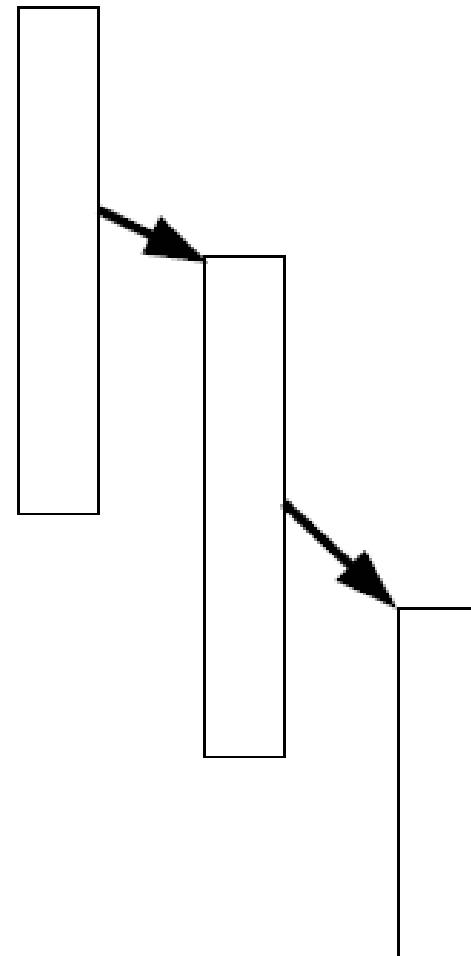


# Jump instructions

- We will now look at the remaining few instructions to make our processor more complete
- We had no branching instructions so far; all programs have to be a strictly linear sequence of instructions
- That is obviously not a very desirable situation
- We need the capability to **branch** or to **break** the sequential flow of instructions to implement any sort of loops
- *Additionally, we need the capability to branch based on some condition based on the values of registers*
- We use two forms of branching: one **based on an immediate address** and the other **based on a register value (flag register)**

# Jump instructions

- **Branching** involves the shifting of the program execution from one point in the program to another
- This is a change in the control flow of the program and may be used to perform different actions on the basis of the results achieved so far
- **Jump is a type of branching** where the control is transferred absolutely, without any memory of the branching point
- Branching is natural in our every day activities



# The flag register

- The branching may be conditional, based on a current state of the processor
- We include a **flag register** in the processor to indicate particular conditions
- The condition of one of the flag register bits can be used for branching
- If the jump is conditioned on a flag bit being set, the branching happens only if that flag has a value of 1 and the program proceeds with the instruction at the branch address
- If the flag is at state 0, execution proceeds normally with the next instruction as if the conditional branch instruction is a *nop* instruction
- The flag register stores limited history of the results of the computations performed by the processor
- This may include aspects like: Did the last arithmetic operation result in an overflow? Did it result in a carry from the most significant bit? Was the result of the previous operation a zero?
- These, in conjunction with branching, are essential to control the program based on the results of operations

# The flag register

- For example, if we want to run a loop ten times, we can repeat the code 10 times, which makes the code long
- It also allows no flexibility to run the code 12 times, if we desire it
- An alternative is to use a **count** (typically stored in a register) that is initialized to 10
- After one set of computations is over, the count can be reduced by 1
- The program can branch to the start of the computations if the count is still not zero
- It is clear that the second option results in shorter code
- Even better, if the count is initialized to 12 or 25, the code remains exactly the same

# The flag register

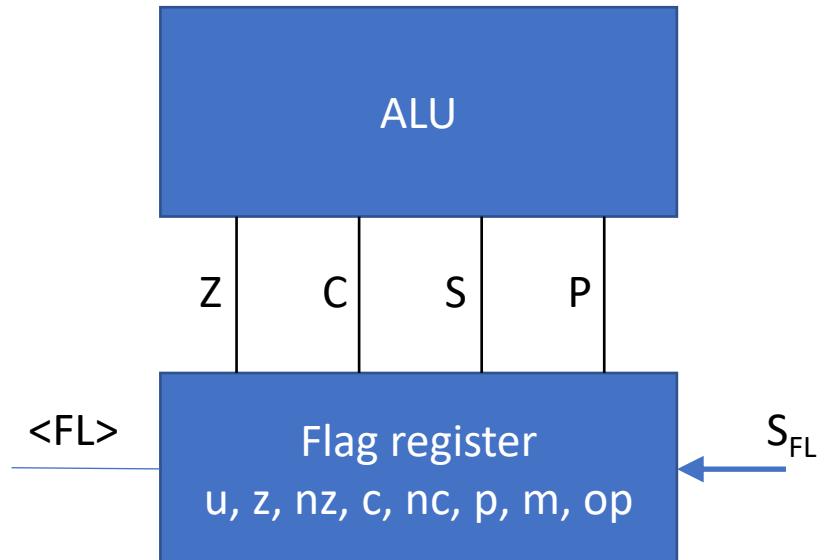
- Our simple processor has the following 4 flag bits: zero, carry, sign, and parity, with respective flags Z, C, S, and P
- The zero flag is set if the previous ALU operation produced an exact 0 as the result (i.e., if AR is zero)
- Similarly, the carry flag is set if the previous operation resulted in a carry-out or borrow-in from the most significant bit
- The S bit copies the sign bit of the last arithmetic operation and becomes 1 if the result was negative
- The parity bit counts the number of 1 bits in the result of the last operation
- If that number is odd, the parity bit is 1

# The flag register

- Instructions that do not use the ALU – such as the data movement instructions, branching instructions, and the like – do not change the flag values
- Some processors group all flags into a special register word known as the **Program Status Word (PSW)**
- Special instructions may move the PSW to or from internal registers or memory
- This allows their manipulation as data

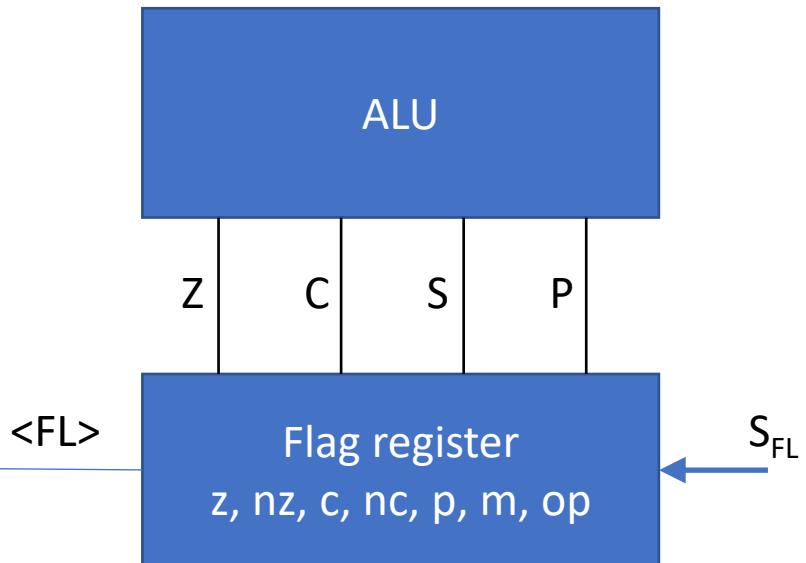
# Jump instruction

- For the simple flag register defined, the **<FL>** flag for conditional instructions can take one of the following values: **z, nz, c, nc, p, m, op**
- These respectively stand for zero, non-zero, carry, no-carry, positive, minus, and odd-parity
- Zero condition is true when the Z bit is set and the non-zero condition is true otherwise
- Similarly, the carry and no-carry conditions are true when the C bit of the flags is 1 and 0 respectively
- The plus condition is true when the sign bit S is 0 and the minus condition is true otherwise
- The odd-parity condition is true if the flag bit P is 1



# Implementing the jump instruction

- To implement the jump function, we need to change the PC to the contents of the AR if the flag condition is satisfied
- If not, it should have no effect
- The conditional jump instructions use a modified control signal, labelled “*End if <FL>*”
- This means that the End control signal is activated only if the selected flag is at a 0 level
- This is the case where the condition is not satisfied and hence the instruction has no impact

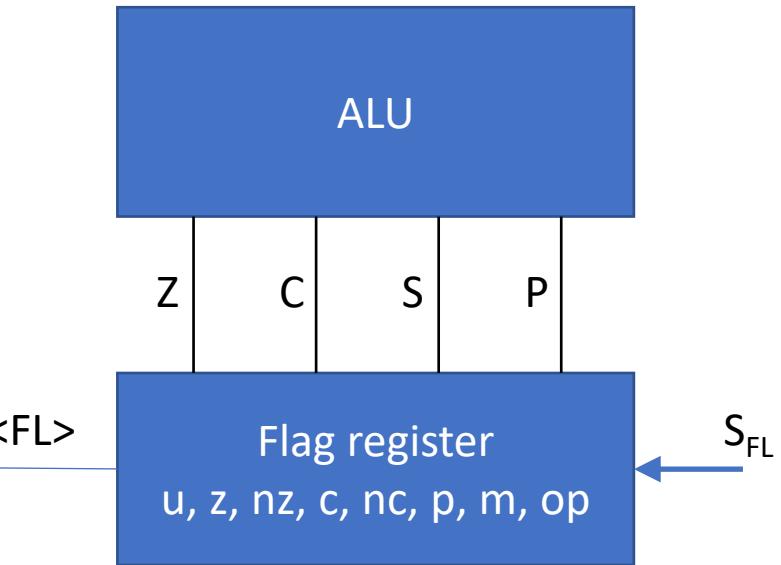


Assembly Instruction	Machine Code	Action
jmpd<FL> xx	E0-E7	[PC] $\leftarrow$ xx if <FL> = 1
jmpr<FL>	E8-EF	[PC] $\leftarrow$ [AR] if <FL> = 1

Instruction	Control Signals	Select Signals
jmpd<FL> xx	Ck 3: E <sub>PC</sub> , L <sub>MR</sub> , I <sub>PC</sub> , E <sub>FL</sub> , End if <FL> Ck 4: RD, L <sub>PC</sub> , End	S <sub>FL</sub> $\leftarrow$ <FL> -
jmpr<FL>	Ck 3: E <sub>FL</sub> , End if <FL> Ck 4: E <sub>AR</sub> , L <sub>PC</sub> , End	S <sub>FL</sub> $\leftarrow$ <FL> -

# Implementing the jump instruction

- However, for instructions with immediate operands (such as `jmpd`), the next word has to be jumped over so that the PC points to the next real instruction
- At the same time, the value of PC should be incremented whether the value of flag is true or not
- Hence, the PC value is saved in MR and PC is incremented
- Then if the flag condition is satisfied, the value at the address is loaded into PC



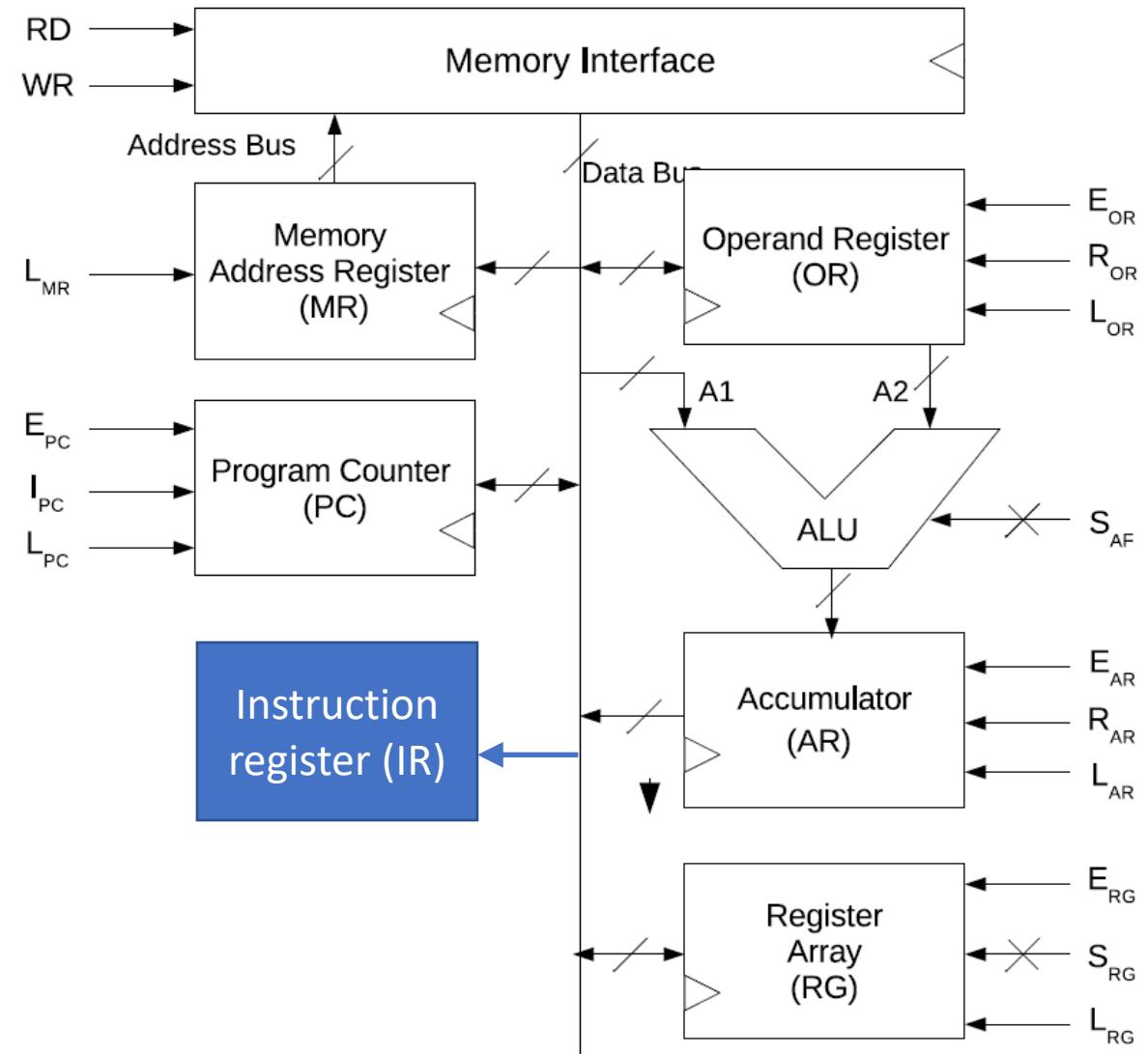
Assembly Instruction	Machine Code	Action
<code>jmpd&lt;FL&gt; xx</code>	E0-E7	$[PC] \leftarrow xx$ if $<FL> = 1$
<code>jmpr&lt;FL&gt;</code>	E8-EF	$[PC] \leftarrow [AR]$ if $<FL> = 1$

Instruction	Control Signals	Select Signals
<code>jmpd&lt;FL&gt; xx</code>	Ck 3: E <sub>PC</sub> , L <sub>MR</sub> , I <sub>PC</sub> , E <sub>FL</sub> , End if $<FL>$ Ck 4: RD, L <sub>PC</sub> , End	S <sub>FL</sub> $\leftarrow <FL>$ -
<code>jmpr&lt;FL&gt;</code>	Ck 3: E <sub>FL</sub> , End if $<FL>$ Ck 4: E <sub>AR</sub> , L <sub>PC</sub> , End	S <sub>FL</sub> $\leftarrow <FL>$ -

# Lecture 31 – Processor design 6

# Instruction fetch

- Now let us look at instruction fetch
- We know it involves reading a word from the memory, using the PC value as the address
- This can be achieved using the following two microinstructions:
  - Ck 1:  $E_{PC}$ ,  $L_{MR}$ ,  $I_{PC}$
  - Ck 2: RD,  $L_{IR}$
- Instruction fetch requires 2 microcycles; in the first cycle, PC value is loaded to MAR
- The PC is simultaneously incremented, so that the next fetch will be from the next word in memory
- In the second cycle, the memory word at the address given by MAR is read and the value obtained is loaded into a special *instruction register* or IR
- The instruction register holds the entire opcode, which then needs to be decoded or deciphered to activate the control signals

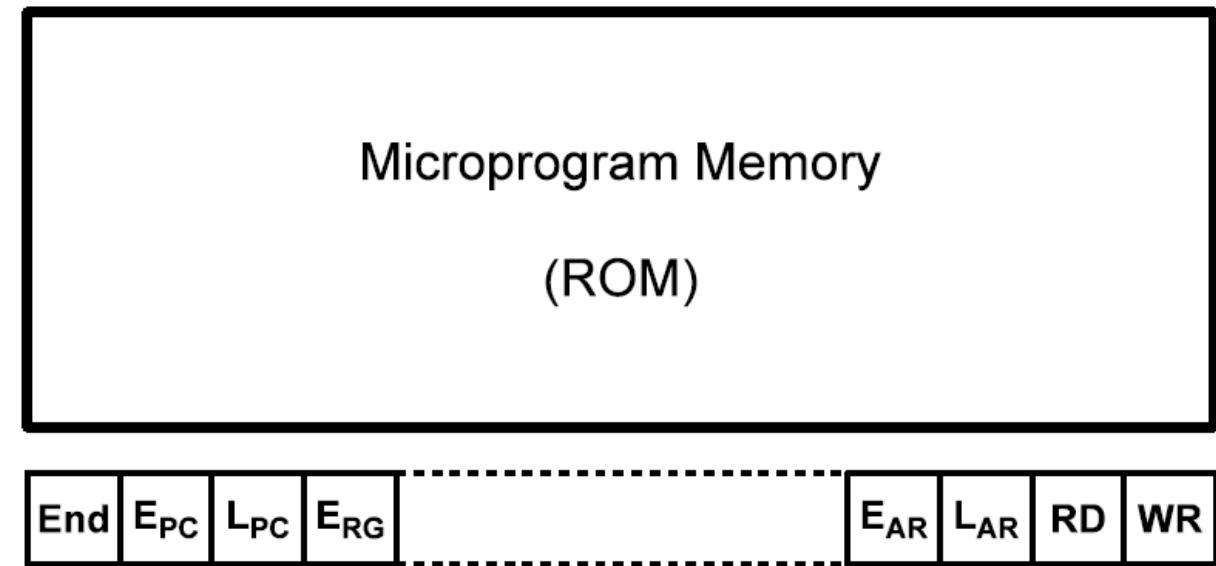


# Instruction decoder

- Now, we complete the story of the processor
- It is clear that the main task is to generate the combination of control signals for each clock cycle for each instruction as per its implementation
- We can think of the control signals being generated using a combinational circuit, whose input is the opcode of the current instruction and the clock cycle number
- The opcode is loaded into the IR register by the fetch process which needs to be decoded into the control signals
- Further, a single opcode can have multiple microcycles associated with it and all of these will have different control signals active
- The process needs to continue until the END signal line is activated

# Instruction decoder

- We can use a special ROM, to generate all control signals
- The word-width of the ROM equals the number of control signals used by the processor
- The number of words in the ROM should exceed the number of distinct input combinations to have enough space to encode all the possibilities
- Each bit of the ROM output can directly serve as the control input line
- The set of control signals treated as a word is often referred to as the *microprogram* word or the control word
- Each clock cycle of each instruction maps to a separate microprogram word



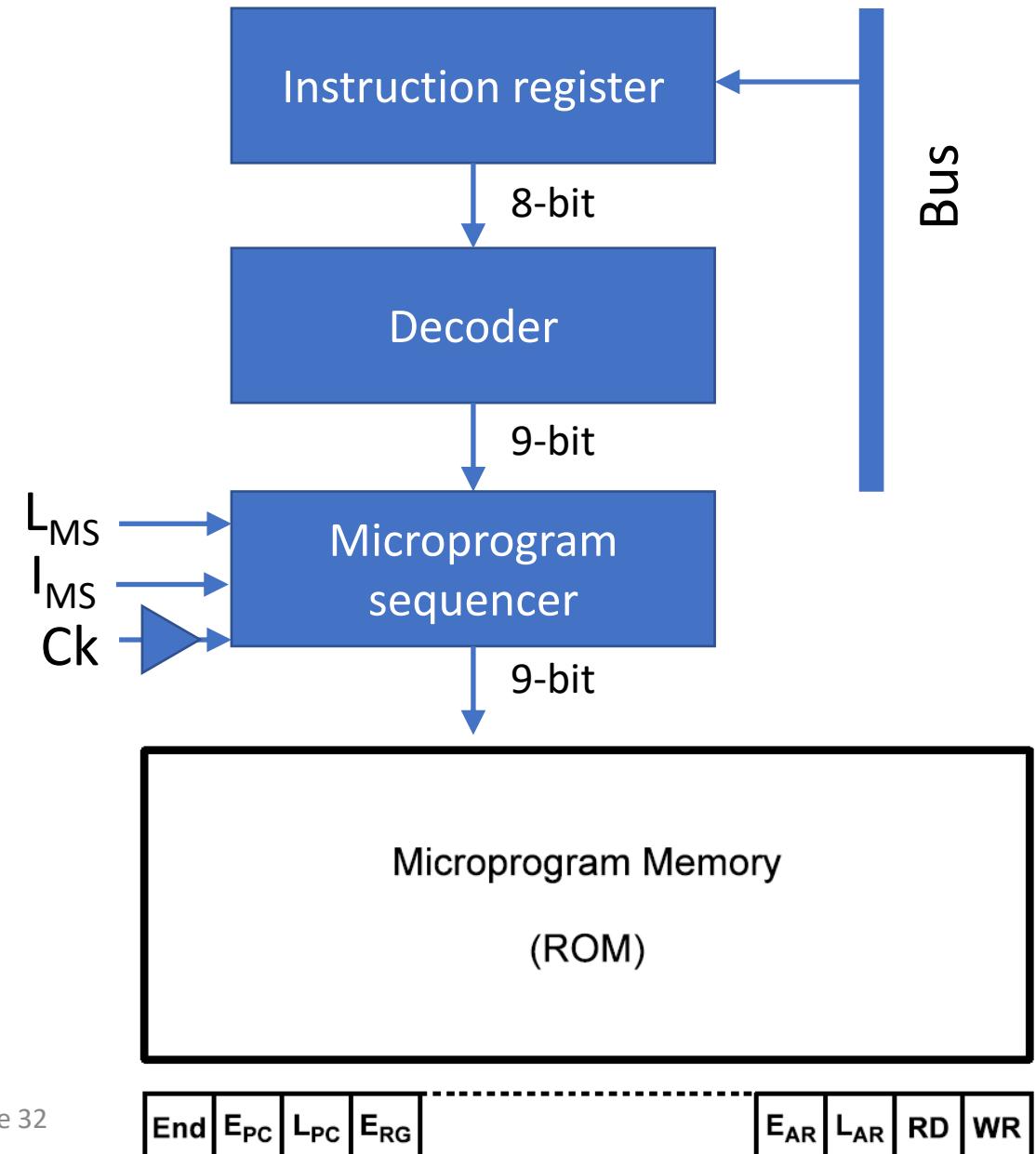
# Instruction decoder

- The width of the word is clear from the number of control signals in the processor (close to 30 in this case)
- How to decide the address lines?
- The number of address lines is decided from the number of microprogram words needed to execute all the instructions
- This includes the separate microcycles within a given instruction
- For instance, ADD <R> will have two separate microprogram words associated with it
- Let's say we have around 500 words, so we can go with 9 address lines

Instruction	Opcode	Clk	Control Signals	Select Signals
Fetch	-	1	EPC, L <sub>MR</sub> , I <sub>PC</sub>	-
		2	RD, L <sub>IR</sub> , L <sub>MS</sub>	-
nop	00	3	End	-
adi xx	01	3	EPC, L <sub>MR</sub> , I <sub>PC</sub>	-
		4	RD, L <sub>OR</sub>	-
		5	EAR, L <sub>AR</sub> , End	S <sub>ALU</sub> ← ADD
		3	EPC, L <sub>MR</sub> , I <sub>PC</sub>	-
sbi xx	02	4	RD, L <sub>OR</sub>	-
		5	EAR, L <sub>AR</sub> , End	S <sub>ALU</sub> ← SUB
		3	EPC, L <sub>MR</sub> , I <sub>PC</sub>	-
		4	RD, L <sub>OR</sub>	-
xri xx	03	5	EAR, L <sub>AR</sub> , End	S <sub>ALU</sub> ← XOR
		3	EPC, L <sub>MR</sub> , I <sub>PC</sub>	-
		4	RD, L <sub>OR</sub>	-
		5	EAR, L <sub>AR</sub> , End	-
ani xx	04	3	EPC, L <sub>MR</sub> , I <sub>PC</sub>	-
		4	RD, L <sub>OR</sub>	-
		5	EAR, L <sub>AR</sub> , End	S <sub>ALU</sub> ← AND
		3	EPC, L <sub>MR</sub> , I <sub>PC</sub>	-
ori xx	05	4	RD, L <sub>OR</sub>	-
		5	EAR, L <sub>AR</sub> , End	S <sub>ALU</sub> ← OR
		3	EPC, L <sub>MR</sub> , I <sub>PC</sub>	-
		4	RD, L <sub>OR</sub>	-
cmi xx	06	5	EAR, End	S <sub>ALU</sub> ← CMP
		3	EPC, L <sub>MR</sub> , I <sub>PC</sub>	-
		4	RD, L <sub>OR</sub>	-
		5	EAR, End	-

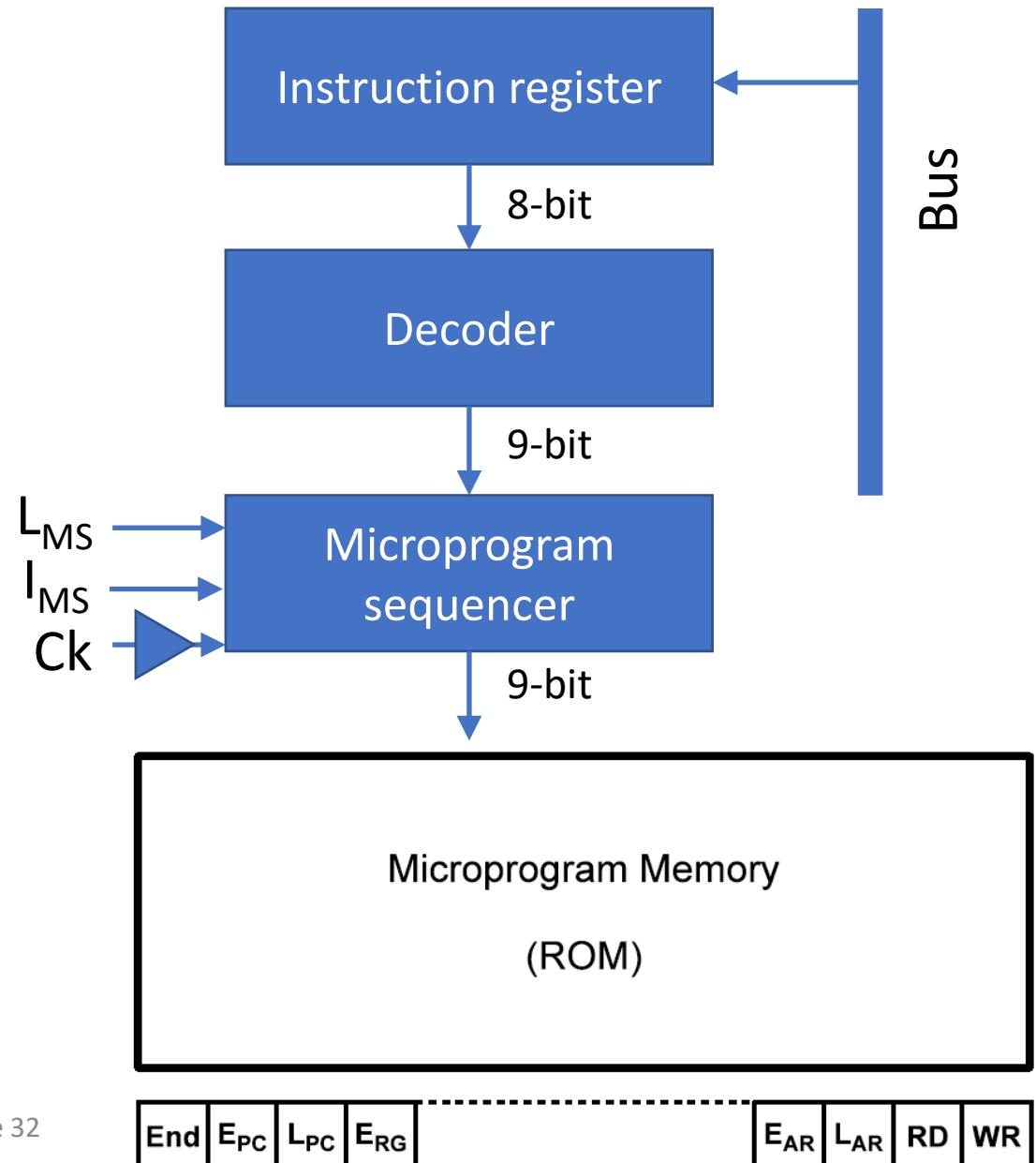
# Instruction decoder

- Now, all we need to do is to map the 8-bit opcode to the 9-bit address in the ROM
- This can be done using a simple combinational circuit
- Thus, the opcode from the instruction register (IR) is decoded and stored in a register called microprogram sequencer which has an option to load it from the decoder output ( $L_{MS}$ )
- Further, we store the microprogram words for a given instruction consecutively in the ROM
- The MS is incremented in every clock cycle to get the next microprogram word, until it is loaded again



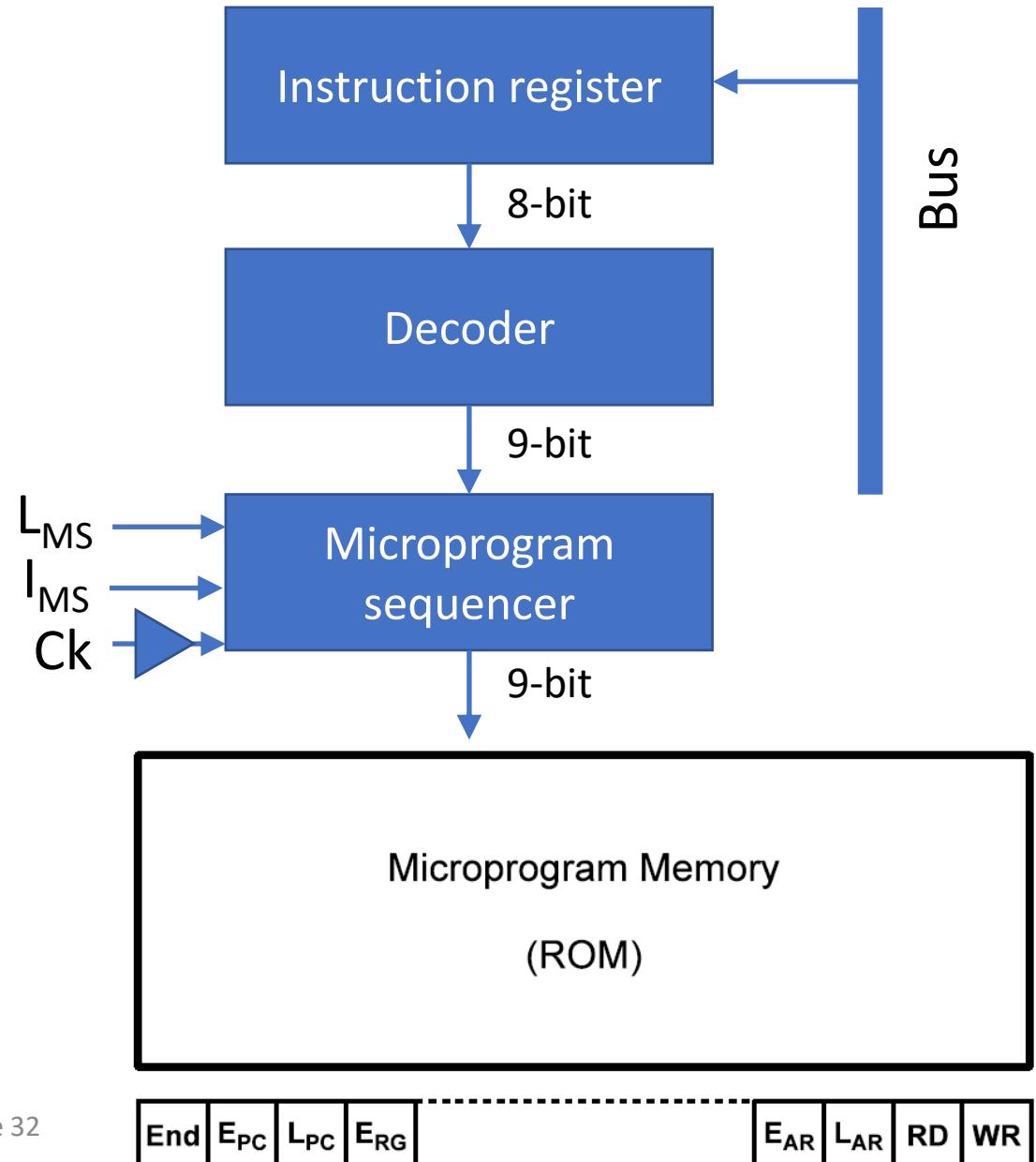
# Instruction decoder

- The starting microprogram address is loaded onto the microprogram sequencer using  $L_{MS}$  (this is done using the microprogram word after the fetch cycle)
- This will result in the first microinstruction corresponding to the current opcode to be taken up in the following clock cycle
- Since we have stored the microinstructions for each instruction in consecutive locations of the microprogram memory, the microprogram sequencer is a register which gets incremented by 1 every clock cycle at the falling edge
- This continues till the last microinstruction of each opcode and the *End* line is activated**



# The End!

- We issue a signal *End* to denote that the execution of the instruction is completed
- The processor is to proceed to fetching the next instruction
- How can we ensure this?
- We do this by putting the microprogram word corresponding to *Fetch* instruction at address 0 of the ROM
- Thus, going to the fetch phase of the next instruction can be achieved by loading 0 to the MS
- We can do that if we use the signal *End* as the reset for the MS, making it a register with load (using  $L_{MS}$ ), reset (using  $R_{MS}$ ), and increment facilities



# The final processor design

- So finally we are able to reveal the complete processor design
- This 8-bit, single bus processor can take instructions and data stored in the memory and perform tasks
- Each instruction is first fetched from memory into the IR, decoded and stored in the MS
- This is done using the address being pointed at by the PC (through the MR)
- Once, the microprogram word is obtained, the subsequent processing is done by the hardware
- Because the MS updated at the (delayed) negative edge of the clock, the control signals are available just after the negative edge , i.e., just before the positive edge

