Information Processing and Digital Systems

Objectives

In this lesson, some basic concepts regarding information processing and representation are clarified. These include:

1. "*Analog*" versus "*Digital*" parameters and systems.
2. Digitization of "*Analog*" signals.
3. Digital representation of information.
4. Effect of noise on the reliability and choice of digital system representation.

Digital versus Analog

- We live in an "*Analog*" world.
- "*Analog*" means Continuous
- We use the word "*Analog*" to express phenomena or parameters that have smooth gradual change or movement.
- For example, earth's movement around the sun is continuous or "*Analog*".
- Temperature is an "*Analog*" parameter. In making a cup of tea, the temperature of the tea kettle increases gradually or smoothly.
- In an "*Analog*" system, parameters have a continuous range of values → just like a mathematical function which is "*Continuous*" ; in other words, the function has no discontinuity points
- The word "*Digital*", *however*, means just the opposite.
- In *Digital* Systems, parameters have a limited set of "*Discrete*" Values that they can assume.

- In Other words, digital parameters don't have a "*Continuous*" range.
- This means that, digital parameters change their values by "*Jumping*" from one allowed value to another.
- As an example, the day of the month is a parameter that may only assume one value out of a set of limited discrete values {1, 2, 3, …., 31}.
- Thus, the day of the month is a parameter may not assume a value of 2.5 for example, but it rather jumps from a value of 2 to a value of 3 then to 4 and so on with no intermediate values!!!

To Summarize:

- *Analog* Systems deal with *Continuous* Range of values.
- *Digital* Systems deal with a *Discrete* set of values.
- **Q**. Which is *easier* to design *digital* systems or *analog* ones?

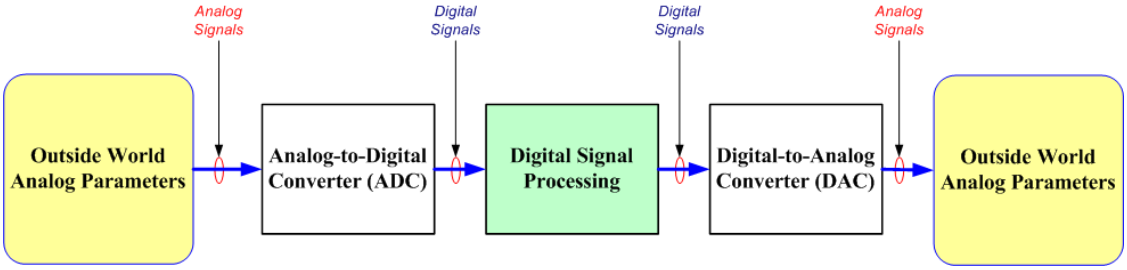
- **A**. Digital systems are *easier* to design since dealing with a limited set of values rather than an *infinite* (*or indefinitely large*) continuous range of values is significantly simpler.

Digitization/Quantization of Analog Signals

- Since the world around us is analog, and processing of digital parameters is much easier, is it is fairly common to convert analog parameters (or signals) into a digital form in order to allow for efficient transmission and processing of these parameters (or signals)
- To convert an Analog signal into a digital one, some loss of accuracy is inevitable since digital systems can only represent a finite discrete set of values.
- The process of conversion is known as *Digitization* or *Quantization*.
- Analog-to-digital-converters (ADC) are used to produce a digitized

version of analog signals.

- Digital-to-analog-converters (DAC) are used to regenerate analog signals from their digitized form.
- A typical system consists of an ADC to convert analog signals into digital ones to be processed by a digital system which produces results in digital form which is then transformed back to analog form through a DAC.

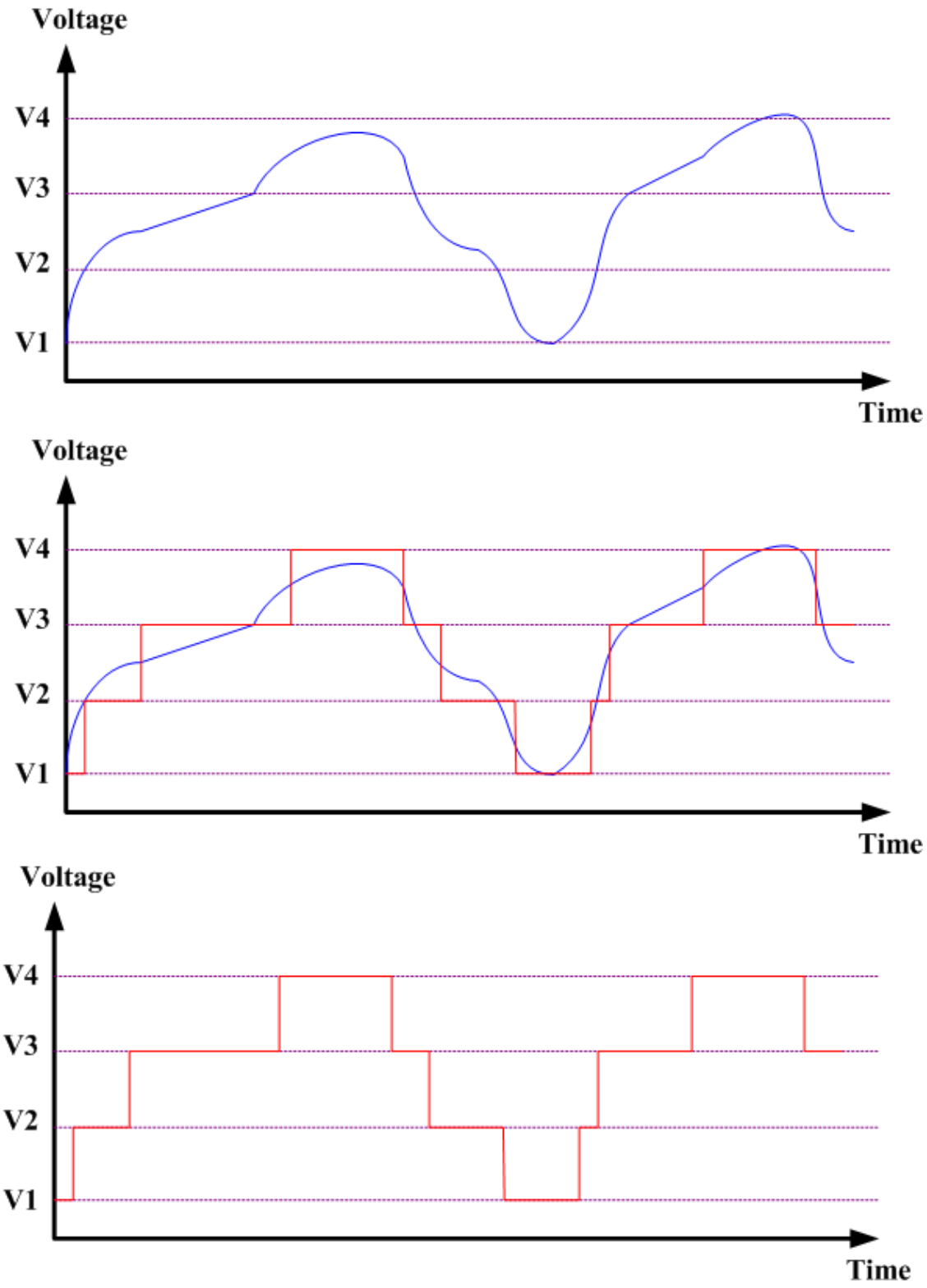


- In this course, we will only be studying digital hardware design concepts, where both the input and output signals are digital signals.

Digitization Example

- As an example, consider digitizing the shown voltage signal assuming that the digitized version allowed set of discrete voltages is {V1, V2, V3, V4}.

- Analog signal values are mapped to the closest allowed discrete voltage $\in$ {V1, V2, V3, V4} as shown in Figure.



The Resulting Digitized Waveform

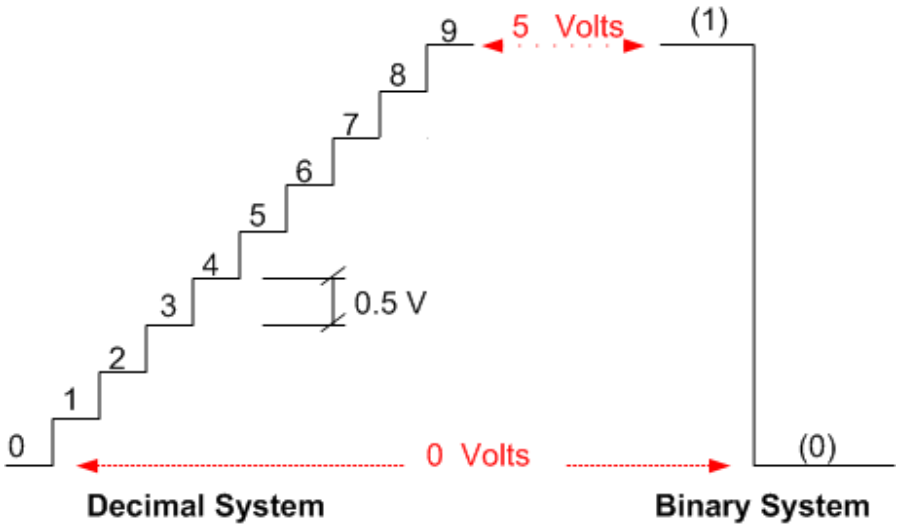
Information Representation

How Do Computers Represent Values (e.g. V1, V2, V3, V4) ?

1. Using Electrical Voltages (Semiconductor Processor, or Memory)
2. Using Magnetism (Hard Disks, Floppies, etc.)
3. Using Optical Means (Laser Disks, e.g. CD's)

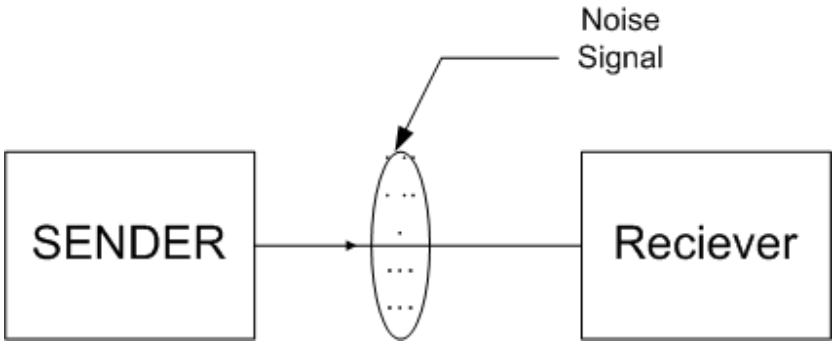
Consider the case where values are represented by voltage signals:

- Each *signal* represents a *digit* in some *Number System*
- If the *Decimal Number System* is used, each signal should be capable of representing one of *10 possible digits* (0-to-9)
- If the *Binary Number System* is used, each signal should be capable of representing only one of *2 possible digits* (0 or 1).
- Digital computers, typically use low power supply voltages to power internal signals, e.g. 5 volts, 3.3 volts, 2.5 volts, etc.
- The voltage level of a signal may be anywhere between the 0 voltage level (Ground) and the power supply voltage level (5 volts, 3.3 volts, 2.5 volts, etc.)
- Thus, for a power supply voltage of 5 volts, internal voltage signals may have any voltage value between 0 and 5 volts.
- Using a decimal number system would mean that each signal should be capable of representing 10 possible digits (0-to-9).
- With 5 volt range signals, the 10 digits of the *decimal* system are represented with each digit having a ***range*** of only 0.5 a volt
- If, however, a *binary* number system is used only 2 digits {0, 1} need to be represented by a signal, allowing much higher Voltage ***range*** of 5 volts between the 2 binary digits.



The Noise Factor

- Typically, lots of *noise signals* exist in most environments.
- Noise may cause the voltage level of a signal (which represents some digit value) to be changed (either higher or lower) which leads to misinterpretation of the value this signal represents.



- Good designs should guard against noisy environments to prevent misinterpretation of the signal information.
- **Q**. Which is more reliable for data transmission; binary signals or decimal signals ?
- **A**. Binary Signals are more reliable.

- **Q**. Why?
- **A**. The Larger the gap between voltage levels, the more reliable the system is. Thus, a signal representing a binary digit will be transmitted more reliably compared to a signal which represents a decimal digit.
- For example, with 0.25 volts *noise level* using a decimal system at 5 volts power supply is totally unreliable

Conclusions

- Information can be represented either in an analog form or in a digital form.
- Due to noise, it is more reliable to transmit information in a digital form rather than an analog one.
- Processing of digitally represented information is much more reliable, flexible and powerful.
- Today's powerful computers use digital techniques and circuitry.
- Because of its high reliability and simplicity, the binary representation of information is most commonly used.
- The coming lessons in this chapter will discuss how numbers are represented and manipulated in digital system.

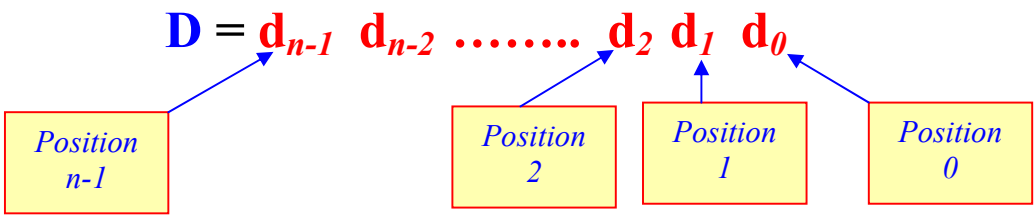
Number Systems

Introduction & Objectives:

- Before the inception of *digital* computers, the only number system that was in common use is the *decimal* number system (النظام العشري) which has a total of 10 digits (0 to 9).
- As discussed in the previous lesson, signals in *digital* computers may represent a digit in some number system. It was also found that the binary number system is more reliable to use compared to the more familiar decimal system
- In this lesson, you will learn:
  - What is meant by a weighted number system.
  - Basic features of weighted number systems.
  - Commonly used number systems, e.g. decimal, binary, octal and hexadecimal.
  - Important properties of these systems.

Weighted Number Systems:

- A number **D** consists of *n* digits with each digit has a particular *position.*



- Every digit *position* is associated with a *fixed weight*.
- If the weight associated with the i^{th} position is w_i, then the value of **D** is given by:

$$D = d_{n-1} w_{n-1} + d_{n-2} w_{n-2} + \ldots + d_2 w_2 + d_1 w_1 + d_0 w_0$$

Example of Weighted Number Systems:

- The Decimal number system (النظام العشري) is a weighted system.
- For Integer decimal numbers, the weight of the rightmost digit (*at position 0*) is **1**, the weight of *position 1* digit is **10**, that of *position 2* digit is **100**, *position 3* is **1000**, etc.

Thus,

$w_0 = 1,\ w_1 = 10,\ w_2 = 100,\ w_3 = 1000,$ **etc.**

Example Show how the value of the decimal number **9375** is estimated

First Position Index

Position	**3**	**2**	**1**	**0**
Number	**9**	**3**	**7**	**5**
Weight	**1000**	**100**	**10**	**1**
Value	**9 x 1000**	**3x100**	**7x10**	**5x1**
Value	**9000 + 300 + 70 + 5**			

First Position Index (0)

The Radix (Base)

1. For *digit position i*, most weighted number systems use weights ($\mathbf{w}_i$) that are ***powers of some constant value*** called the **radix (r)** or the **base** such that $\mathbf{w}_i = \mathbf{r}^i$.
2. A number system of radix ***r***, typically has a set of ***r*** allowed digits $\in$ {0,1, …,(r-1)} → *See the next example*
3. The leftmost digit has the highest weight → **Most Significant Digit (MSD)** → *See the next example*
4. The rightmost digit has the lowest weight → **Least Significant Digit (LSD)** → *See the next example*

Example **Decimal Number System**

1. **Radix (Base) = *Ten***
2. **Since $w_i = r^i$., then**
   - **$w_0 = 10^0 = 1$,**
   - **$w_1 = 10^1 = 10$,**
   - **$w_2 = 10^2 = 100$,**
   - **$w_3 = 10^3 = 1000$, etc.**
3. **Number of Allowed Digits is Ten →** {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

Thus: MSD LSD

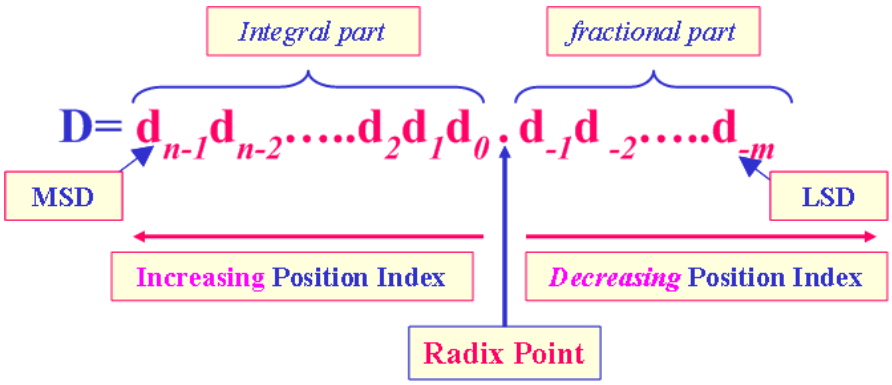
$$9375 = 5x10^0 + 7x10^1 + 3x10^2 + 9x10^3$$
$$= 5x1 + 7x10 + 3x100 + 9x1000$$

Position	3	2	1	0
Weight	1000 $= 10^3$	100 $= 10^2$	10 $= 10^1$	1 $= 10^0$

The Radix Point

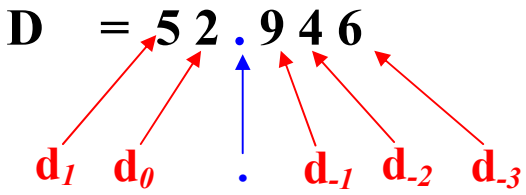
Consider a number system of radix r,

- A number D of *n integral* digits and *m fractional* digits is represented as shown



- Digits to the left of the radix point (*integral digits*) have positive position indices, while digits to the right of the radix point (*fractional digits*) have negative position indices
- Position *indices* of digits to the *left* of the ***radix point*** (*the integral part of* ***D***) start with a **0** and are incremented **as we move lefts ($d_{n-1}d_{n-2}.....d_2d_1d_0$.)**
- Position *indices* of digits to the *right* of the ***radix point*** (*the fractional part of* ***D***) are *negative* starting with **–1** and are decremented **as we move rights ($d_{-1}d_{-2}.....d_{-m}$).**
- The *weight* associated with digit position *i* is given by $\mathbf{w}_i = \mathbf{r}^i$, *where i is the position index*
  - $\forall \boldsymbol{i}$= *-m, -m+1, …, -2, -1, 0, 1, ……, n-1*
- The Value of **D** is Computed as : $D = \sum_{i=-m}^{n-1} d_i \, r^i$

Example Show how the value of the following decimal number is estimated



Number	5	2	.	9	4	6
Position	1	0	.	-1	-2	-3
Weight	10^1 = 10	10^0 = 1	.	10^{-1} = 0.1	10^{-2} = 0.01	10^{-3} = 0.001
Value	5 x 10	2 x 1	.	9 x 0.1	2 x 0.01	6 x 0.001
Value	50 +	2 +		0.9 +	0.02 +	0.006

$$D = 5x10^1 + 2x10^0 + 9x10^{-1} + 4x10^{-2} + 6x10^{-3}$$

Notation

- **Let $(D)_r$** denotes a number **D** expressed in a number system of radix **r.**

Note: *In this notation, r will be expressed in decimal*

Example:

- $(29)_{10}$ Represents a decimal value of 29. The radix "10" here means ten.
- $(100)_{16}$ is a Hexadecimal number since r = "16" here means sixteen. This number is equivalent to a decimal value of 16^2.
- $(100)_2$ is a Binary number (radix =2, i.e. two) which is equivalent to a decimal value of $2^2 = 4$.

Important Number Systems

The Decimal System

- **r = 10 (*ten* → *Radix is not a Power of 2*)**
  - **Ten** Possible Digits {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

The Binary System

- **r = 2**
- **Two** Allowed Digits {**0**, **1**}
- A **B**inary Dig**it** is referred to as **Bit**
- The leftmost bit has the highest weight → **Most Significant Bit (MSB)**
- The rightmost bit has the lowest weight → **Least Significant Bit (LSB)**

Examples

Find the decimal value of the two Binary numbers $(101)_2$ *and* $(1.101)_2$

MSB LSB

- $(1\ 0\ 1)_2 = 1x2^0 + 0x2^1 + 1x2^2$
- $= 1x1 + 0x2 + 1x4$
- $= (5)_{10}$

MSB LSB

- $(1\ .\ 1\ 0\ 1)_2 = 1x2^0 + 1x2^{-1} + 0x2^{-2} + 1x2^{-3}$
- $= 1 + 0.5 + 0 + 0.125$
- $= (1\ .\ 6\ 2\ 5)_{10}$

Octal System:

- **r = 8 (*Eight* = 2^3)**
  - **Eight** Allowed Digits {0, 1, 2, 3, 4, 5, 6, 7}

Examples

Find the decimal value of the two Octal numbers $(375)_8$ *and* $(2.746)_8$

MSD LSD

$(375)_8 = 5x8^0 + 7x8^1 + 3x8^2$

$= 5x1 + 7x8 + 3x64$

$= (253)_{10}$

MSD LSD

$(2.746)_8 = 2x8^0 + 7x8^{-1} + 4x8^{-2} + 6x8^{-3}$

$= (2.94921875)_{10}$

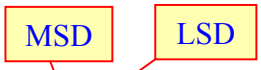
Hexadecimal System:

- **r = 16 (*Sixteen* = 2^4)**
- **Sixteen** Allowed Digits {*0-to-9 and A, B, C, D, E, F*}
  - *Where:* ***A = ten, B = Eleven, C = Twelve, D = Thirteen, E = Fourteen & F = Fifteen.***

- **Q:** Why is the digit following **9** assigned the character **A** and not "**10**"?
- **A:** What we need is a *single* digit whose value is *ten*, but "**10**" is actually *two digits* not *one*.
  - Thus, in Hexadecimal system the 2-digit number $\mathbf{(10)_{16}}$ actually represents a value of *sixteen* not *ten* $\{\mathbf{(10)_{16} = 0x16^0 + 1x16^1 .= (16)_{10}}\}$.

Examples

Find the decimal value of the two Hexadecimal numbers $(9E1)_{16}$ *and* $(3B.C)_{16}$



$$(9E1)_{16} = 1x16^0 + Ex16^1 + 9x16^2$$
$$= 1x1 + 14x16 + 9x256$$
$$= (2529)_{10}$$

$$(3B.C)_{16} = Cx16^{-1} + Bx16^0 + 3x16^1$$
$$= 12x16^{-1} + 11x16^0 + 3x16$$
$$= (59.75)_{10}$$

Important Properties

1. The number of possible digits in any number system with radix *r* equals *r*. (*Give examples in decimal, binary, octal and hexadecimal*)
2. The smallest digit is 0 and the largest possible digit has a value = (*r-1*)
3. The Largest value that can be expressed in *n integral* digits is ($\mathbf{r^n-1}$) → *Prove (Hint add 1 to the LSD position of the largest number)*
4. The Largest value that can be expressed in *m fractional* digits is ($\mathbf{1-r^{-m}}$) → *Prove* (*Hint add 1 to the LSD position of the largest number*)
5. The Largest value that can be expressed in *n integral* digits and *m fractional* digits is ($r^n - r^{-m}$) → *Prove* (*Hint- add results of properties 3 &4 above*)
6. Total number of values (patterns) representable in *n* digits is r^n

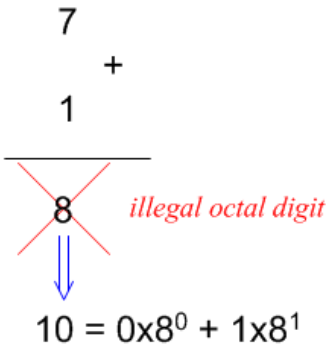
Clarification (a)

Q. What is the result of adding 1 to the largest digit of some number system??

A.

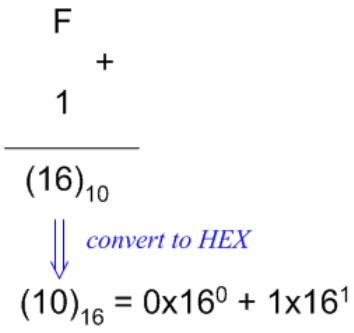
- For the decimal number system, $(1)_{10} + (9)_{10} = (10)_{10}$
- For the octal number system, $(1)_8 + (7)_8 = (10)_8 = (8)_{10}$

OCTAL System



- For the hex number system, $(1)_{16} + (F)_{16} = (10)_{16} = (16)_{10}$

HEX System



- For the binary number system, $(1)_2 + (1)_2 = (10)_2 = (2)_{10}$

Conclusion. Adding 1 to the largest digit in any number system always has a result of (10) in that number system.

- This is easy to prove since the largest digit in a number system of radix *r* has a value of (*r-1*). Adding 1 to this value the result is ***r*** *which is always equal to* $(10)_r = 0x\ r^0 + 1x\ r^1 = (r)_{10}$

Clarification (b)

Q. What is the largest value representable in 3-integral digits?

A. The largest value results when all 3 positions are filled with the largest digit in the number system.

- **For** the decimal system, it is $(999)_{10}$
- **For** the octal system, it is $(777)_8$
- **For** the hex system, it is $(FFF)_{16}$
- **For** the binary system, it is $(111)_2$

Clarification (c)

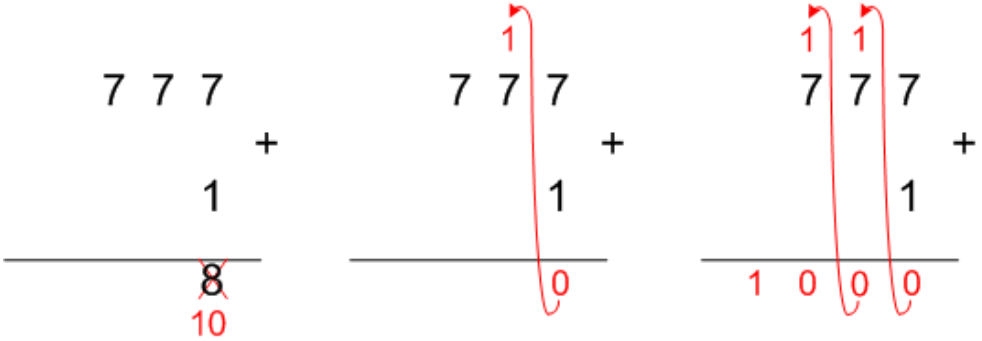
Q. What is the result of adding 1 to the largest 3-digit number?

?

A.

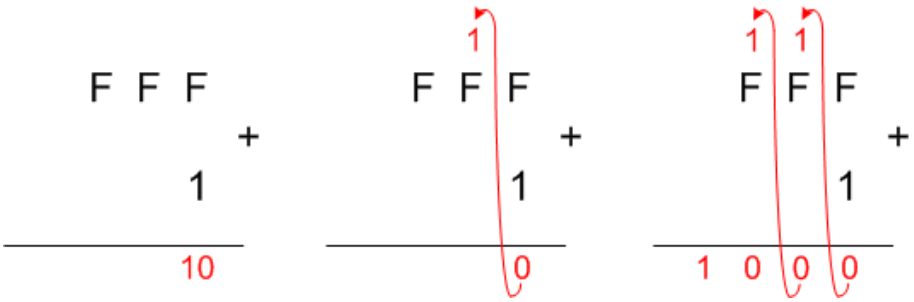
- **For** the decimal system, $(1)_{10} + (999)_{10} = (1000)_{10} = (\mathbf{10^3})_{10}$
- **For** the octal system, $(1)_8 + (777)_8 = (1000)_8 = (\mathbf{8^3})_{10}$

OCTAL System



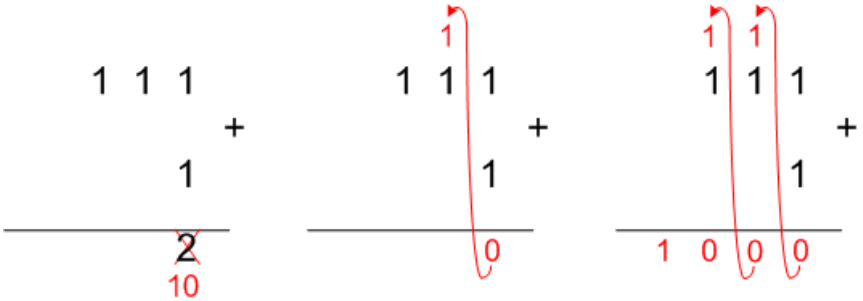
- **For** the hex system, $(1)_{16} + (FFF)_{16} = (1000)_{16} = (16^3)_{16}$

HEX System



- **For** the binary system, $(1)_2 + (111)_2 = (1000)_2 = (2^3)_{10}$

Binary System



In general, for a number system of radix **r**, adding 1 to the largest *n*-digit number = $\mathbf{r}^{n}$

Accordingly, the value of largest *n*-digit number = $\mathbf{r}^{n}$ -1

Conclusions.

1. In any number system of radix ***r***, the result of adding 1 to the *largest n-digit* number equals r^{n}.
2. Thus, the value of the *largest n-digit* number is equal to (r^{n} *-1*)
3. Thus, *n digits* can represent r^{n} different values (digit combinations) starting from a 0 value up to the largest value of r^{n} *-1*.

Appendix A. Summary of Number Systems Properties

The following table summarizes the basic features of the Decimal, Octal, Binary, and Hexadecimal number systems as well as a number system with a general radix ***r***

	Decimal 10	**Octal 8**	**Binary 2**	**Hexadecimal 16**	**General r**
Allowed Digits	{0-9}	{0-7}	{0-1}	{0-9, A-F}	{ 0 - R} where R = (r-1)
Value of $a_{n-1}\ldots a_2\, a_1\, a_0 . a_{-1}a_{-2}\ldots a_{-m}$	$a_{n-1}x10^{n-1} + a_{n-2}x10^{n-2} + \ldots + a_2x10^2 + a_1x10^1 + a_0 \times 10^0 + a_{-1} \times 10^{-1} + a_{-2} \times 10^{-2} + \ldots + a_{-m}x10^{-m}$ $a_i \in \{0\text{-}9\}$ i=-m,…., 0, 1, …, n-1	$a_{n-1}8^{n-1}+\ldots+ a_2 8^2+a_1 8^1+a_0 8^0+a_{-1}8^{-1}+ a_{-2}8^{-2}+\ldots+a_{-m}8^{-m}$ $a_i \in \{0\text{-}7\}$	$a_{n-1}2^{n-1}+\ldots+ a_2 2^2+a_1 2^1+a_0 2^0+a_{-1}2^{-1}+ a_{-2}2^{-2}+\ldots+a_{-m}2^{-m}$ $a_i \in \{0,1\}$		$a_{n-1}r^{n-1}+\ldots+ a_2r^2 + a_1r^1 + a_0r^0 + a_{-1}r^{-1} + a_{-2}r^{-2} +\ldots+a_{-m}r^{-m}$ $a_i \in \{0 - (r\text{-}1)\}$
Smallest n-digit number	000…..0	000…..0	000…..0	000…..0	000…..0
Largest n-digit number	999……9 = $10^n - 1$	77…..7 = $8^n - 1$	11…..1 = $2^n - 1$	FF…….F = $16^n - 1$	RR…..R = $r^n - 1$
Range of n-digit integers	0 - $(10^n\text{-}1)$	0- $(8^n\text{-}1)$	0- $(2^n\text{-}1)$	0- $(16^n\text{ -}1)$	0 – $(r^n - 1)$
# of Possible Combinations of n-digits	10^n	8^n	2^n	16^n	r^n
Max Value of m Fractional Digits	$1\text{-}10^{-m}$	$1\text{-}8^{-m}$	$1\text{-}2^{-m}$	$1\text{-}16^{-m}$	$1\text{-}r^{-m}$

Appendix B. First 16 Binary Numbers & Their Decimal Equivalent
(*All Possible Binary Combinations in 4-Bits*)

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

Appendix C. Decimal Values of the First 10 Powers of 2

- One Kilo is defined as 1000.
- For example, one Kilogram is 1000 grams. A kilometer is 1000 meters.
- In the Binary system, the power of 2 value closest to 1000 is 2^{10} which equals 1024. This is referred to as one Kilo (or in short 1K) in binary systems.
- Thus, one Kilo (or 1K) in Binary systems is not exactly 1000 but rather equals 1024 or 2^{10}
- Thus, in binary systems 2K= 2 x 1024 = 2048, 4K=4 x 1024= 4096, and so on
- Similarly, a one Meg (one million) in binary systems is 2^{20} which equals 1,048,576.

Powers of 2	Decimal. Value
2^0	1
2^1	2
2^2	4
2^3	8
2^4	16
2^5	32
2^6	64
2^7	128
2^8	256
2^9	512
2^{10}	1024

1 Kilo = 1K
2K = 2048
4K = 4096

Number Systems Arithmetic

Objectives

- In this lesson, we will study basic arithmetic operations in various number systems with a particular stress on the binary system.

Approach

- Arithmetic in the Binary number system (addition, subtraction and multiplication).
- Arithmetic in other number systems

Binary Addition

$$0 + 0 = 0$$

$$1 + 0 = 1$$

$$0 + 1 = 1$$

$$1 + 1 = 2$$

2 is not an allowed digit in binary

$$1 + 1 = (10)_2$$

$$(3)_{10} + (7)_{10} = (\text{ten})_{10}$$

$$(3)_{10} + (7)_{10} = (10)_{10}$$

Example

Show the result of adding:

$$(27)_{10} + (43)_{10}$$

Carry	1		
1st Number	2	7	
2nd Number	4	3	+
Result	**7**	**0**	

Position	$i+1$	i	
weight	$r^{(i+1)}$	$w = r^i$	
Digit 1		D_1	
Digit 2		D_2	+
Result	D_{Carry}	D_{Sum}	

Position	1	$i=0$	
weight	$w = 10^1 = 10$	$w = 10^0 = 1$	
Digit 1		**5**	
Digit 2		**7**	+
Result	**1**	**2**	

1x10 2x1

- Likewise, in case of the binary system, if the weight of the sum bit is 2^i, then the weight of the carry bit is 2^{i+1}.

- Thus, adding 1 + 1 in the *binary* system results in a Sum bit of 0 and a carry bit of 1.
- The shown table summarizes the *Sum* and *Carry* results for binary addition

Binary Addition Table

	Carry	Sum
Weight	2^1	2^0
0 + 0	**0**	**0**
0 + 1	**0**	**1**
1 + 0	**0**	**1**
1 + 1	**1**	**0**

$\equiv 1x2^1$ $\equiv 0x2^0$

$\equiv +2$

Example

Carries

	5	4	3	2	1	0	
		1	1	1	1		
	1	0	1	1	0	1	
	1	0	0	1	1	1	+
	1	**0**	**1**	**0**	**1**	**0**	**0**

Result of Binary Addition (SUM)

Binary Subtraction

1 - 0 = 1

1 - 1 = 0

0 - 0 = 0

0 - 1 = ?

Position	1	0	
weight	10	1	
1st Number	**7**	**5**	
2nd Number		**8**	**-**
Result	**?**	**?**	

Position	1	0	
weight	10	1	
1st Number	6 ~~7~~	~~5~~ 15	
2nd Number		**8**	**-**
Result	**6**	**7**	

$(5)_{10} - (8)_{10} = (7)_{10}$ Borrow 1

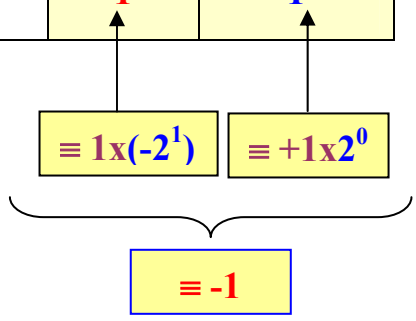
- For Binary subtraction

0 - 1 = 1 Borrow 1

- In general, the result of subtracting two digits each of weight *w* is two digits. One is the "**Difference**" digit and the other is the "**Borrow**" digit.

- The **difference** digit has the *same weight* *w* as the operand digits.
- The **borrow** digit is considered negative and has the weight of the next ***higher*** *digit* (*wr*).

	Borrow	Difference
Weight	-2^1	$+2^0$
0 - 0	**0**	**0**
1 - 1	**0**	**0**
1 - 0	**0**	**1**
0 - 1	**1**	**1**



Q. What is **1 – 1 – 1 = ?**

A. The answer is **1 borrow 1**.

Explanation: We perform the operation in 2 steps:

- **1 – 1 = 0**
- We then *subtract* **1** from the above result, i.e. **0 – 1** which is **1 borrow 1**.

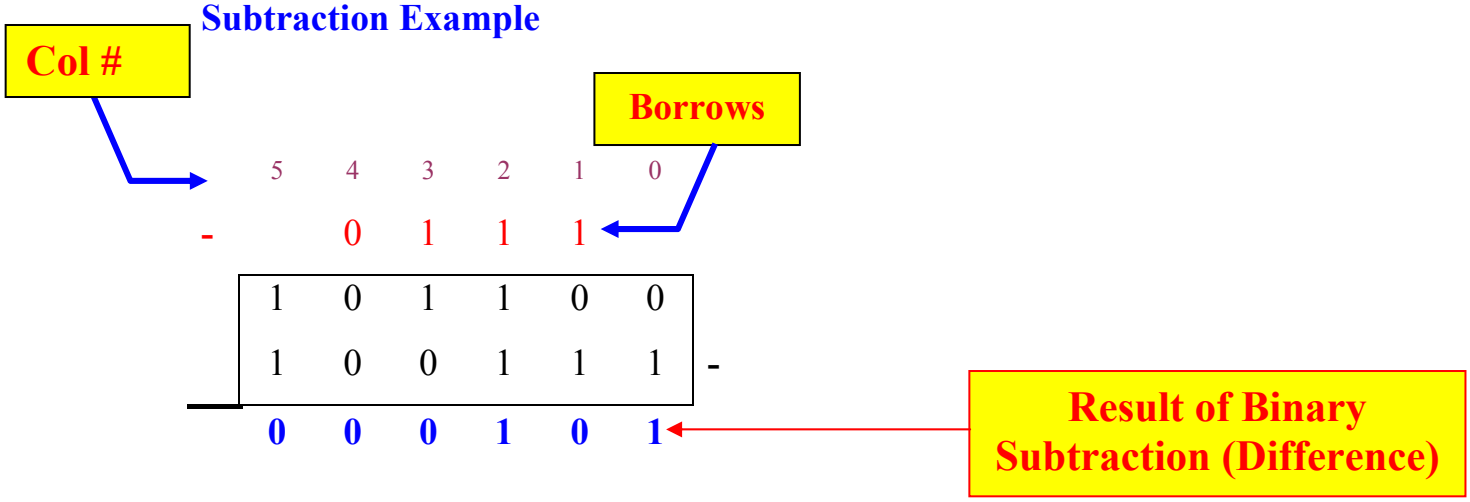
Q. What is **0 – 1 – 1 = ?**

A. The answer is **0 borrow 1**.

Explanation: We perform the operation in 2 steps:

- **0 - 1= 1 borrow 1**
- We then *subtract* **1** from the above result, which yields **0 borrow 1**.

Subtraction Example



Binary Multiplication (example)

Multiplicand		**1**	**0**	**1**	**1**	
Multiplier			**1**	**0**	**1**	**x**
		1	**0**	**1**	**1**	
	0	**0**	**0**	**0**		**+**
1	**0**	**1**	**1**			**+**
1	**1**	**0**	**1**	**1**	**1**	

Arith. With Bases Other Than 10

Example: **Base 5** → **Digit Set= {0, 1, 2, 3, 4}**

$(2)_5 + (3)_5 = (5)_{10}$

$= (?)_5$

$= (10)_5$

Addition Table

+	0	1	2	3	4
0	0				
1	1	2			
2	2	3	4		
3	3	4	10	11	
4	4	10	11	12	13

$=5= 0x5^0 + 1x5^1$

$=6= 1x5^0 + 1x5^1$

$=8= 3x5^0 + 1x5^1$

Multiplication Table

*	0	1	2	3	4
0	0				
1	0	1			
2	0	2	4		
3	0	3	11	14	
4	0	4	13	22	31

$=6= 1x5^0 + 1x5^1$

$=9= 4x5^0 + 1x5^1$

$=16= 1x5^0 + 3x5^1$

Number Base Conversion

Objectives

Given the representation of some number (X_B) in a number system of radix B, this lesson will show how to obtain the representation of the same number (X) in another number system of radix A, i.e. (X_A).

Converting Whole (Integer) Numbers

Assuming X to be an Integer,

1. Assume that X_B has *n* digits $(b_{n-1}\ldots\ldots\ldots b_2\, b_1\, b_0)_B$,

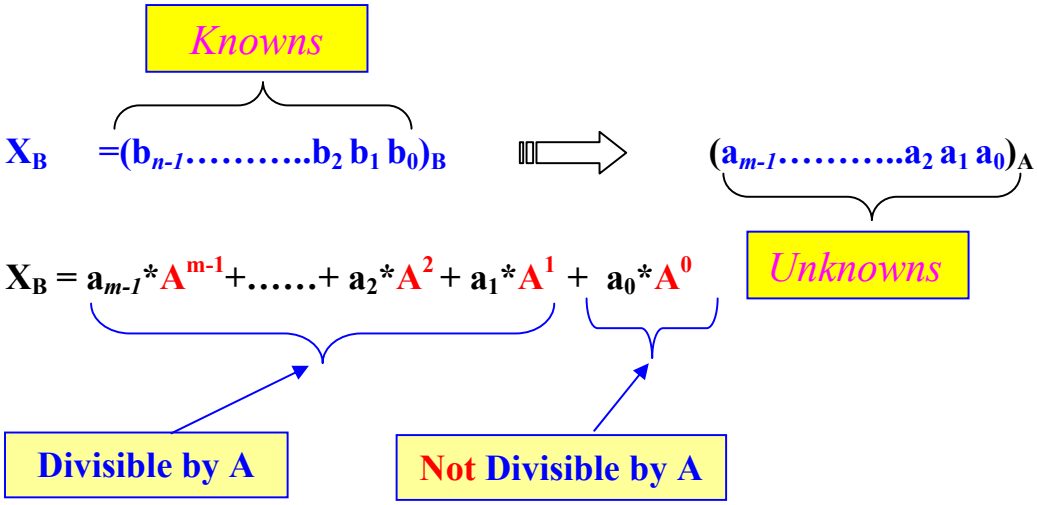
   where $\mathbf{b_i}$ *is a digit in radix B system,*

   i.e. $\mathbf{b_i} \in$ **{0, 1, ….., "B-1"}**

2. Assume that X_A has *m* digits $(a_{m-1}\ldots\ldots\ldots a_2\, a_1\, a_0)_A$

   where $\mathbf{a_i}$ *is a digit in radix A system,*

   i.e. $\mathbf{a_i} \in$ **{0, 1, ….., "A-1"}**



Where $a_i \in \{0\text{-}(A\text{-}1)\}$

Accordingly, dividing X_B by A, the remainder will be a_0.

In other words, we can write

$$X_B = Q_0.A+a_0$$

Where, $Q_0 = a_{m-1}*A^{m-2} + \ldots + a_2*A^1 + a_1*A^0$

Divisible by A — **Not Divisible by A**

$$Q_0 = Q_1A+a_1$$

$$Q_1 = Q_2A+a_2$$

……..……

$$Q_{m-3}=Q_{m-2}A+a_{m-2}$$

$$Q_{m-2}=a_{m-1} < A \text{ (not divisible by A)}$$

$$=Q_{m-1}A+a_{m-1}$$

Where $Q_{m-1} = 0$

- This division procedure can be used to convert an integer value from some radix number system to any other radix number system
- An important point to remember is the first digit we get using the division process is a_0, then a_1, then a_2, till a_{m-1}

- In other words, we get the digits of the integer number starting from the radix point and moving lefts

Example :

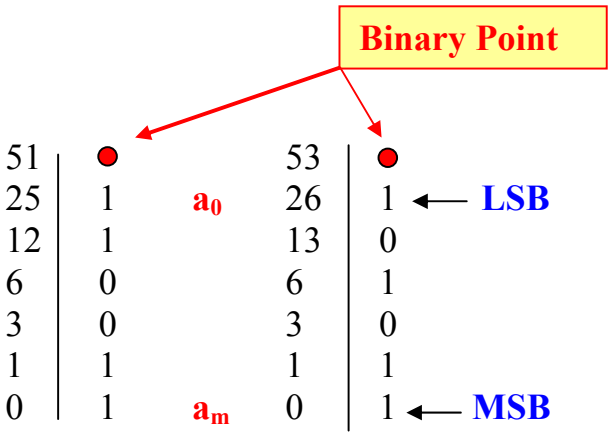
Convert $(53)_{10}$ ⟶ $(?)_2$

Division Step			Quotient	Remainder	
53	÷	2	$Q_0=26$	$1 = a_0$	LSB
26	÷	2	$Q_1=13$	$0 = a_1$	
13	÷	2	$Q_2=6$	$1 = a_2$	
6	÷	2	$Q_3=3$	$0 = a_3$	
3	÷	2	$Q_4=1$	$1 = a_4$	
1	÷	2	0	$1 = a_5$	MSB

Stopping Point

Thus $(53)_{10}=(110101.)_2$

Since we always divide by the radix, and the quotient is re-divided again by the radix, the solution table may be compacted into 2 columns only as shown:



$(51)_{10}=(110011.)_2$

$(53)_{10}=(110101.)_2$

Example :

Convert $(755)_{10}$ ⟹ $(?\ )_8$

Division Step	Quotient	Remainder	
755 ÷ 8	$Q_0=94$	$3 = a_0$	LSB
94 ÷ 8	$Q_1=11$	$6 = a_1$	
11 ÷ 8	$Q_2=1$	$3 = a_2$	
1 ÷ 8	0	$1 = a_3$	MSB

Thus, $(755)_{10}$ ⟹ $(1363.)_8$

The above method can be more compactly coded as follows:

755	●
94	3
11	6
1	3
0	1

Example :

Convert $(1606)_{10}$ ⟹ $(?\ )_{12}$

For radix twelve, the allowed digit set is:

{0-9, A, B}

1606 ÷12	●	
133 ÷12	10 = A	LSB
11 ÷12	1	
0	11 = B	MSB

$(1606)_{10} \Rightarrow (\ B1A.)_{12}$

Converting Fractions

Assuming X to be a fraction (< 1),

1. Assume that X_B has *n* digits

$$X_B = (0.b_{-1}\ b_{-2}\ b_{-3}\ldots\ldots b_{-n})_B$$

2. Assume that X_A has *m* digits

$$X_A = (0.a_{-1}\ a_{-2}\ a_{-3}\ldots\ \ldots a_{-m})_A$$

Thus, $X_B = (0.b_{-1}\ b_{-2}\ b_{-3}\ldots\ldots b_{-n})_B \Rightarrow (0.a_{-1}\ a_{-2}\ a_{-3}\ldots\ \ldots a_{-m})_A$

Knowns — $(0.b_{-1}\ b_{-2}\ b_{-3}\ldots\ldots b_{-n})_B$

Unknowns — $(0.a_{-1}\ a_{-2}\ a_{-3}\ldots\ \ldots a_{-m})_A$

$$X_B = a_{-1}*A^{-1}+a_{-2}*A^{-2}+\ldots\ldots\ldots a_{-m}*A^{-m}$$

Integer | Fraction

$$X_B*A = a_{-1} + X_{B1}$$

Repeating:

$$X_{B1} * A = a_{-2} + X_{B2}$$

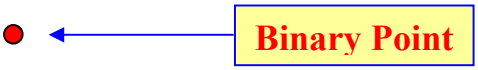
.....

$$X_{Bm-2} * A = a_{-m-1} + X_{Bm-1}$$

$$X_{Bm-1} * A = a_{-m}$$

Example :

Convert $(0.731)_{10}$ ⟶ $(?)_2$



0.731*2=**1**.462

0.462*2=**0**.924

0.924*2=**1**.848

0.848*2=**1**.696

0.696*2=**1**.392

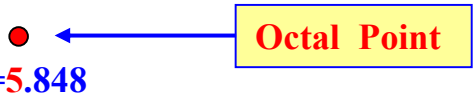
0.392*2=**0**.784

0.784*2=**1**.568

$(0.731)_{10} = (.1011101)_2$

Example :

Convert $(0.731)_{10} \longrightarrow (?)_8$



8*0.731 =5.848

8*0.848 = 6.784

8*0.784 = 6.272

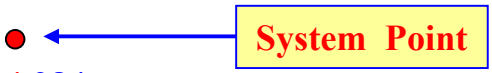
8*0.272 = 2.176

$(0.731)_{10}$ =$(0.5662)_8$

Example :

Convert $(0.357)_{10} \Longrightarrow (?)_{12}$

- **For radix twelve, the allowed digit set is:**
  - **{0-9, A, B}**



12*0.357 =4.284

12*0.284 = 3.408

12*0. 408= 4.896

12*0. 896= 10.752 $\Longrightarrow$ A=10

=A

$(0.357)_{10} \Longrightarrow (0.434A)_{12}$

IMPORTANT NOTE

For a number that has both integral and fractional parts, conversion is done separately for both parts, and then the result is put together with a system point in between both parts.

Conversion From Bases Other Than 10

Example

$(\quad)_7 \Rightarrow (\quad)_5$

$(\quad)_9 \Rightarrow (\quad)_{12}$

2 Approaches

Perform arith. in original base system (in the above example bases 7 & 9)

1. Convert to Decimal
2. Convert from Decimal to new base (in the above example bases 5&12)

Binary To Octal Conversion

$(b_n \ldots b_5\ b_4\ b_3\ b_2\ b_1\ b_0 . b_{-1}\ b_{-2}\ b_{-3}\ b_{-4}\ b_{-5} \ldots\ldots)_2$ ⇨ $(?)_8$

$(b_n \ldots b_5\ b_4\ b_3\ b_2\ b_1\ b_0 . b_{-1}\ b_{-2}\ b_{-3}\ b_{-4}\ b_{-5} \ldots)_2$

3-bits 3-bits 3-bits 3-bits

Starting Point

Group of 3 Binary Bits $b_{i+2}\ b_{i+1}\ b_i$	Octal Equivalent
0 0 0	0
0 0 1	1
0 1 0	2
0 1 1	3
1 0 0	4
1 0 1	5
1 1 0	6
1 1 1	7

Example :

Convert $(1110010101.1011011)_2$ ino Octal.

We first partition the Binary number into groups of 3 bits

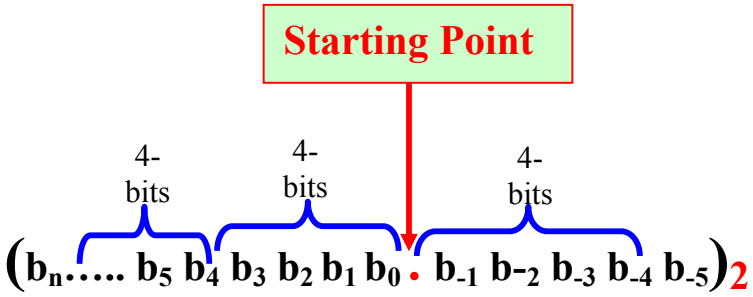
001 110 010 101 . 101 101 100

1 6 2 5 5 5 4

001__110__010__101_._101__101__100 = $(1625.554)_8$

Binary To Hexadecimal Conversion

$(b_n \ldots\ldots b_5\ b_4\ b_3\ b_2\ b_1\ b_0 . b_{-1}\ b_{-2}\ b_{-3}\ b_{-4}\ b_{-5} \ldots\ldots)_2$ ⇒ $(\ ?\ )_{16}$



Group of 4 Binary Bits $b_{i+3}\ b_{i+2}\ b_{i+1}\ b_i$	Hexadecimal Equivalent
0 0 0 0	0
0 0 0 1	1
0 0 1 0	2
0 0 1 1	3
0 1 0 0	4
0 1 0 1	5
0 1 1 0	6
0 1 1 1	7
1 0 0 0	8
1 0 0 1	9
1 0 1 0	A
1 0 1 1	B
1 1 0 0	C
1 1 0 1	D
1 1 1 0	E
1 1 1 1	F

Example :

Convert $(1110010101.1011011)_2$ into Hexadecimal.

$$\underbrace{0011}_{3}\ \underbrace{1001}_{9}\ \underbrace{0101}_{5}\ .\ \underbrace{1011}_{B}\ \underbrace{0110}_{6}$$

$= (395.B6)_{16}$

To Convert Between Octal && Hexadecimal Convert to Binary as an Intermediate Step

Machine Representation of Numbers

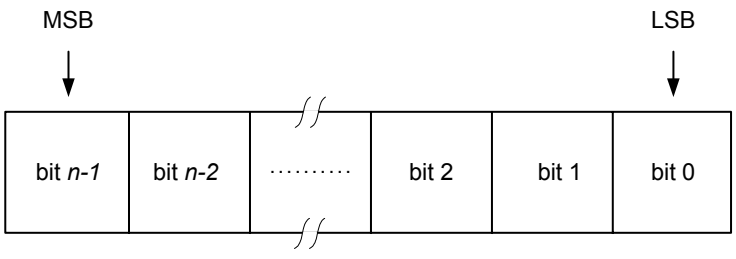
Objectives

- In this lesson, you will learn how signed numbers (positive or negative) are represented in digital computers.
- You will learn the 2 main methods for signed number representation:
  a. The signed-magnitude method, and
  b. The complement method.

Registers

- Digital computers store numbers in special digital electronic devices called **Registers**
- **Registers** consist of a fixed number *n* of *storage elements.*
- Each storage element is capable of storing one bit of data (either 0 or 1}.
- The register ***size*** is the number of storage bits in this register (*n*).
- Thus, registers are capable of holding *n*-bit binary numbers
- Register size (*n*) is typically a power of 2, e.g. 8, 16, 32, 64, etc.
- An *n-bit* register can represent (store) one of 2^n *Distinct Values*.
- Numbers stored in registers may be either *unsigned* or *signed* numbers. For example, **13** is an unsigned number but **+13** and **–13** are signed numbers.

Unsigned Number Representation



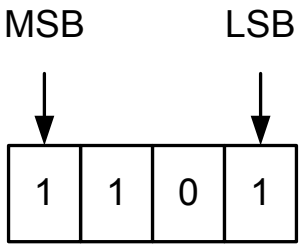
N-Bit Register holding an n-Bit Unsigned Number

- A register of n-bits, can store any unsigned number that has n-bits or less.

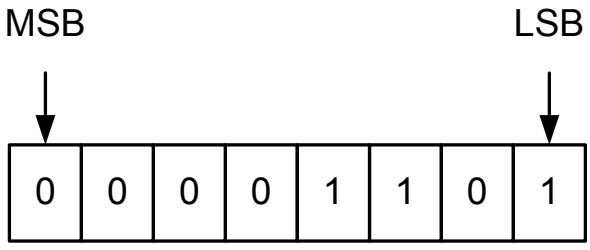
- Typically, the *rightmost* bit of the register is designated to be the least significant bit (LSB), while the *leftmost* bit is designated to be the most-significant bit (MSB).
- When representing an integer number , this *n-bit* register can hold values from 0 up to ($2^n - 1$).

Example

Show how the value $(13)_{10}$ (or **D** in Hexadecimal) is stored in a 4-bit register and in an 8-bit register



4-Bit Register Storing 13

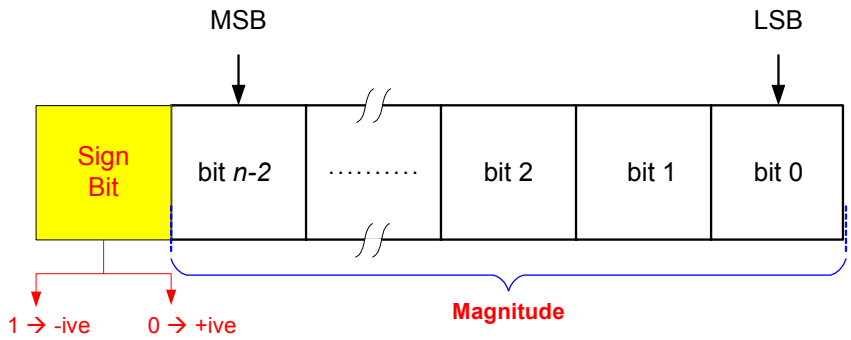


8-Bit Register Storing 13

Signed Number Representation

- The n-bits of the register holding an *unsigned number* need only represent the value (magnitude) of the number. No sign information needs to be represented in this case.
- In the case of a signed number, the *n-bits* of the register should represent both the magnitude of the number and its sign as well.
- Two major techniques are used to represent signed numbers:
  1. Signed Magnitude Representation
  2. Complement method
     - Radix (R's) Complement (2's Complement)
     - Diminished Radix (R-1's) Complement (1's Complement)

Signed Magnitude Number Representation



Signed-Magnitude Number Representation in *n*-Bit Register

- Independent Representation of The Sign and The Magnitude
- The leftmost bit is used as a *Sign Bit*.
- The *Sign Bit* :
  - = 0 → +ive number
  - = 1 → -ive number.
- The remaining (*n-1*) bits are used to represent the magnitude of the number.
- Thus, the *largest* representable *magnitude*, in this method, is $(2^{n-1}-1)$

Example

Show the signed-magnitude representations of +6, -6, +13 and –13 using a 4-Bit register and an 8-Bit register

Solution

- **For a 4-bit register**, the leftmost bit is a sign bit, which leaves 3 bits only to represent the magnitude.
- The largest magnitude representable in 3-bits is 7. Accordingly, we cannot use a 4-bit register to represent +13 or –13.

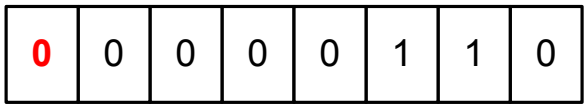
0	1	1	0

Signed-Magnitude Representation of +6

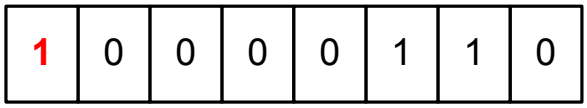
1	1	1	0

Signed-Magnitude Representation of -6

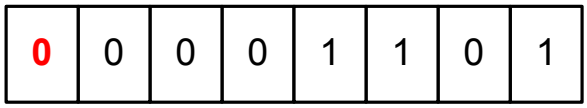
- **For an 8-bit register**, the leftmost bit is a sign bit, which leaves 7 bits to represent the magnitude.
- The largest magnitude representable in 7-bits is 127 (= 2^7-1).



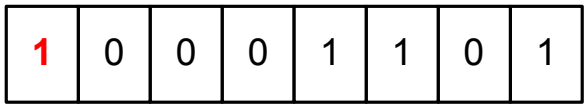
Signed-Magnitude Representation of +6



Signed-Magnitude Representation of -6



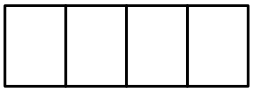
Signed-Magnitude Representation of +13



Signed-Magnitude Representation of -13

Notes

1. Signed magnitude method has Two representations for 0 → {+0 , -0} → nuisance for implementation.



2. Signed magnitude method has a symmetric range of representation {-(2^{n-1} -1) : +(2^{n-1} - 1)}
3. Harder to implement addition/subtraction.
    a) *The sign and magnitude parts have to be processed independently.*
    b) *Sign bits of the operands have to be examined to determine the actual operation (addition or subtraction).*
    c) *Separate circuits are required to perform the addition and subtraction operations.*
4. Multiplication & division are less problematic.

Complement Representation

- Positive numbers (+N) are represented in ***exactly*** the same way as in signed magnitude system
- Negative numbers (-N) are represented by the *complement* of N (**N'**)

Define the Complement N' of some number N as:

N' = *M* -*N* *where,* ***M*** *= Some Constant*

- Applying a negative sign to a number (N→ **-N**) is equivalent to *Complementing* that number (N → **N'**)
- Thus, given the representation of some number N, the representation of –N is equivalent to the representation of the complement N'.

Important Property:

- The Complement of the Complement of some number *N* is the original number *N*.

$$N' = M-N$$

$$(N')' = M-(M-N) = N$$

- This is a required property to match the negation process since a number negated twice must yield the original number {- (-**N**) = N}

Why Use the Complement Method ?

Through the proper choice of the constant **M**, the complement operation can be fairly *simple* and quite *fast*. A simple complement process allows:

i. Simplified arithmetic operations since. subtraction can be totally replaced by addition and complementing.
ii. Lower cost, since no subtractor circuitry will be required and only an adder is needed.

Complement Arithmetic

Basic Rules

1. Negation is replaced by complementing (**- N → N'**)
2. Subtraction is replaced by addition to the complement.
   - *Thus,* (X – Y) *is replaced by* (X + Y')

Choice of M

The value of M should be chosen such that:

1. It simplifies the computation of the complement of a number.
2. It results in simplified arithmetic operations.

- Consider the operation

$$Z = X - Y,$$

where *both X and Y are positive* numbers

- In complement arithmetic, Z is computed by *adding* X to the *complement* of Y

$$Z = X + Y'$$

Consider the following two possible cases:

First case Y > X → (Negative Result)

- The result **Z** is **–ive**, *where*

$$Z = -(Y-X) \rightarrow$$

- Being –ive**, Z** should be represented in the *complement form* as **M-(Y-X)**
- Using the complement method:

$$Z = X - Y$$

$$Z = X + Y'$$

$$= X + (M-Y)$$

$$= M - (Y-X)$$

= *Correct Answer in the Complement Form*

- Thus, in the case of a *negative result*, any value of M may be used.

Second case Y < X → (Positive Result)

The result **Z** is **+ive** *where,*

$$Z = +(X-Y).$$

Using complement arithmetic we get:

$$Z = X-Y$$

$$Z = X + Y'$$

$$= X + (M-Y)$$

$$Z = M + (X-Y)$$

- which is *different* from the *expected correct result* **of +(X-Y)**

- In this case, a *correction step* is required for the final result.
- The choice of the value of M affects the complexity of this correction step.

To summarize,

There are two constraints on the choice of M

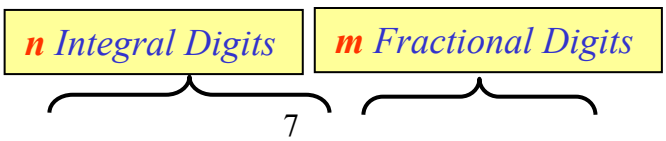
1. Simple and fast complement operation.
2. Elimination or simplification of the correction step.

R's and (R-1)'s Complements

- Two complement methods have generally been used.
- The two methods differ in the choice of the value of M.

1. The diminished radix complement method {(R-1)'s Complement }, and
2. The radix complement method (R's Complement).

- Consider the number **X**, with *n* integral digits and *m* fractional digits, where



$$\mathbf{X} = X_{n-1}\, X_{n-2}\ \ldots.\ X_1\, X_0 \,.\, X_{-1}\, X_{-2}\ \ldots\ldots\ X_{-m}$$

- Next, we will show how to compute the (R-1)'s and the R's complements of X

The Diminished Radix Complement (R-1)'s Complement:

$$\mathbf{M}_{R-1} = r^n - r^{-m}$$

where; $r^n = 1\underbrace{000\ldots00}_{n\ digits}.\underbrace{00\ldots000}_{m\ digits}$

$(n+1)^{th}$ *Position*

and; $r^{-m} = \underbrace{000\ldots00}_{n\ Positions}.\underbrace{00\ldots001}_{m\ Positions}$

- *Note that, if X is integer, then* $m=0$ *and* $r^{-m} = \mathbf{1}$.

Thus; $r^{-m} = 000\ldots00\,.\,00\ldots001$

$=$ *Unit (one) in Least Position (ulp)*

OR $\mathbf{M}_{R-1} = r^n - ulp$

where; ulp = Unit (one) in Least Position $= r^{-m}$

Important Notes:

- The (R-1)'s complement of X will be denoted by X'_{r-1}.
- $(r^n - r^{-m})$ **is the** *largest number representable* in *n integral* digits and *m fractional* digits.
- $X'_{r-1} = L - X$, *where* L is *largest number representable* in *n integral* digits and *m fractional* digits

The shown table shows how to compute the (r-1)'s complement of X for various number systems

Number System	(R-1)'s Complement	Complement of X (X'_{r-1})
Decimal	**9's** Complement	$X'_9 = (10^n - 10^{-m}) - X$ $= 99...9.99...9 - X$
Binary	**1's** Complement	$X'_1 = (2^n - 2^{-m}) - X$ $= 11...1.111...1 - X$
Octal	**7's** Complement	$X'_7 = (8^n - 8^{-m}) - X$ $= 77...7.77...7 - X$
Hexadecimal	**F's** Complement	$X'_F = (16^n - 16^{-m}) - X$ $= FF...F.FF...F - X$

n-integral digits

m-fractional digits

Radix Complement (R's Complement):

$$M_R = r^n$$

Note that $r^n = 1000...00.000$

$(n+1)^{th}$ Position

n Positions

Notes:

1. The R's complement of X will be denoted by X'_r.
2. M_R depends only on the number of integral digits (n), but is independent of the number of fractional digits (m).
3. $X'_r = r^n - X$
4. $X'_{r-1} = (r^n - ulp) - X$

5. **Thus**, $X'_r = X'_{r-1} + ulp$, *i.e R's complement =(R-1)'s complement + ulp*

The shown table summarizes the radix complement computation of X for various number systems

Number System	R's Complement	Complement of X (X'_r)
Decimal	**10's** Complement	$X'_{10} = 10^n - X$
Binary	**2's** Complement	$X'_2 = 2^n - X$
Octal	**8's** Complement	$X'_8 = 8^n - X$
Hexa-decimal	**16's** Complement	$X'_{16} = 16^n - X$

Examples

Find the 9's and the 10's complement of the following decimal numbers:

a- 2357

b- 2895.786

Solution:

a- X = 2357→ *n=4,*

- $X'_9 = (10^4 - ulp) - 2357$
  $= 9999 - 2357 = 7642$
- $X'_{10} = 10^4 - 2357 = 7643$;
- *Alternatively*, $X'_{10} = X'_9 + 0001 = 7643$

b- X = 2895.786→ *n=4, m=3*

- $X'_9 = (10^4 - ulp) - 2895.786$
  $= 9999.999 - 2895.786 = 7104.213$

- $X'_{10}=10^4 - 2895.786= 7104.214$;
- *Alternatively*, $X'_{10} = X'_9 + 0000.001= 7104.214$

Example

Find the 1's and the 2's complement of the following binary numbers:

a- 110101010

b- 1010011011

c- 1010.001

Solution:

a- X = 110101010 → *n=9,*

- $X'_1=(2^9 - ulp) - 110101010 = 111111111 - 110101010 = 001010101$
- $X'_2=2^9 - 110101010 = 1000000000 - 110101010 = 001010110$
- *Alternatively*, $X'_2= X'_1 + ulp = 001010101 + 000000001 = 001010110$

b- X = 1010011011 → *n=10,*

- $X'_1=(2^{10} - ULP) - 101001101 = 1111111111 - 101001101 = 010110010$
- $X'_2=2^{10} - 101001101 = 10000000000 - 101001101 = 010110011$
- *Alternatively*, $X'_2= X'_1 + ulp = 010110010 + 0000000001 = 010110011$

c- X = 1010.001 → *n=4, m=3*

- $X'_1=(2^4 - ULP) - 1010.001 = 1111.111 - 1010.001 = 0101.110$
- $X'_2 = 2^4 - 1010.001 = 10000 - 1010.001 = 0101.111$

- *Alternatively*, $X'_2 = X'_1 + ulp$ $= 0101.110 + 0000.001$

  $= 0101.111$

Important Notes:

1. The 1's complement of a number can be directly obtained by bitwise complementing of each bit, i.e. each 1 is replaced by a 0 and each 0 is replaced by a 1.
   - Example: $X = 1\ 1\ 0\ 0\ 1\ 0\ 1\ 0\ 0\ 1$
   - $X_1' = 0\ 0\ 1\ 1\ 0\ 1\ 0\ 1\ 1\ 0$

2. The 2's complement of a number can be visually obtained as follows:
   - Scan the binary number from right to left.
   - 0's are replaced by 0's till the first 1 is encountered.
   - The first encountered 1 is replaced by a 1 but from this point onwards each bit is complemented replacing each 1 by a 0 and each 0 by a 1
     - Example: $X = 1\ 1\ 0\ 0\ 1\ 0\ 1\ 0\ 0$
     - $X_2' = 0\ 0\ 1\ 1\ 0\ 1\ 1\ 0\ 0$

Example

Find the 7's and the 8's complement of the following octal numbers:

a- 6770

b- 541.736

Solution:

a- $X = 6770 \rightarrow n=4$,

- $X'_7 = (8^4 - ULP) - 6770$ $= 7777 - 6770$

  $= 1007$
- $X'_8 = 8^4 - 6770$ $= 10000 - 6770 = 1010$
- *Alternatively*, $X'_8 = X'_7 + ulp$ $= 1007 + 0001 = 1010$

b- $X = 541.736 \rightarrow n=3, \rightarrow m=4$

- $X'_7 = (8^3 - ULP) - 541.736$ $= 777.7777 - 541.736 = 236.041$

- $X'_8 = 8^3 - 541.736$ $= 1000 - 541.736 = 236.042$
- *Alternatively*, $X'_8 = X'_7 + ulp$ $= 236.041 + 0.001 = 236.042$

Example

Find the F's and the 16's complement of the following HEX numbers:

a- 3FA9

b- 9B1.C70

Solution:

a- X = 3FA9 → *n=4,*

- $X'_F = (16^4 - ULP) - 3FA9$ $= FFFF - 3FA9 = C056$
- $X'_{16} = 16^4 - 3FA9$ $= 10000 - 3FA9 = C057$
- *Alternatively*, $X'_{16} = X'_F + ulp$ $= C056 + 0001 = C057$

b- X = 9B1.C70 → *n=3,* → *m=3*

- $X'_F = (16^3 - ULP) - 9B1.C70$ $= FFF.FFF - 9B1.C70 = 64E.38F$
- $X'_{16} = 16^3 - 9B1.C70$ $= 1000 - 9B1.C70 = 64E.390$
- *Alternatively*, $X'_{16} = X'_F + ulp$ $= 64E.38F + 000.001 = 64E.390$

Example

Show how the numbers +53 and -53 are represented in 8-bit registers using signed-magnitude, 1's complement and 2's complement representations.

	+53	**-53**
Signed Magnitude	**0**0110101	**1**0110101
1's Complement	**0**0110101	**1**1001010
2's Complement	**0**0110101	**1**1001011

Important Notes:

1. In ***all*** signed number representation methods, the leftmost bit indicates the sign of the number, i.e. it is considered as a *sign bit*
2. If the *sign bit* (leftmost) is 1, then the number is negative and if it is 0 the number is positive.

Comparison:

	Signed Magnitude	1's Complement	2's Complement
No. of 0's	2 **(± 0)**	2 **(± 0)**	1 **(+ 0)**
Symmetric	yes	yes	no
Largest +ive value	$+(2^{n-1}-1)$	$+(2^{n-1}-1)$	$+(2^{n-1}-1)$
Smallest –ive Value	$-(2^{n-1}-1)$	$-(2^{n-1}-1)$	-2^{n-1}

Quiz:

For the shown 4-bit numbers, write the corresponding decimal values in the indicated representation.

X	Un-signed	Signed Magnitude	1's Comp (X_1')	2's Comp (X_2')
0000				
0001				
0010				
0011				
0100				
0101				
0110				
0111				
1000				
1001				
1010				
1011				
1100				
1101				
1110				
1111				

End of Lessons Exercises

1. Find the binary representation in signed magnitude, 1's complement, and 2's complement for the following decimal numbers: +13, -13, +39, -39, +1, -1, +73 and -73. For all numbers, show the required representation for 6-bit and 8-bit registers
2. Indicate the decimal value corresponding to all 5-bit binary patterns if the binary pattern is interpreted as a number in the signed magnitude, 1's complement, and 2's complement representations.

Complement Arithmetic

Objectives

In this lesson, you will learn:

- How additions and subtractions are performed using the complement representation,
- What is the Overflow condition, and
- How to perform arithmetic shifts.

Summary of the Last Lesson

Basic Rules

1. Negation is replaced by complementing (**- N → N'**)
2. Subtraction is replaced by addition to the complement.
   - *Thus,* (X – Y) *is replaced by* (X + Y')
3. For some number **N,** its complement **N' is** computed as **N'** = M –N, *where*
   - M = r^n for **R's** complement representation, where *n* is the number of *integral* digits of the register holding the number.
   - M = (r^n – *ulp*) for **(R-1)'s** complement representation
4. The operation **Z= X–Y,** *where* both X and Y are positive numbers (computed as **X + Y'**) yields two different results depending on the relative magnitudes of X & Y. (*Review page 12 of the previous lesson*).

a) First case Y > X → (Negative Result)

- The result **Z** is **–ive**, *where*

$$Z = -(Y-X) \rightarrow$$

- Being –ive**, Z** should be represented in the *complement form* as

$$\boxed{Z = M-(Y-X)} \quad (1)$$

➢ Using the complement method:

$Z = X + Y'$

$= X + (M-Y)$, **i.e.**

$Z = M - (Y-X)$ (2)

= *Correct Answer in the Complement Form*

- **In this case, any value of M gives correct result.**

Note In this case the result fits in the n-digits of the operands. In other words, there is *no end carry* irrespective of the value of M.

Second case Y < X → (Positive Result)

The result **Z** is **+ive** *where,*

$$Z = +(X-Y).$$

Using complement arithmetic we get:

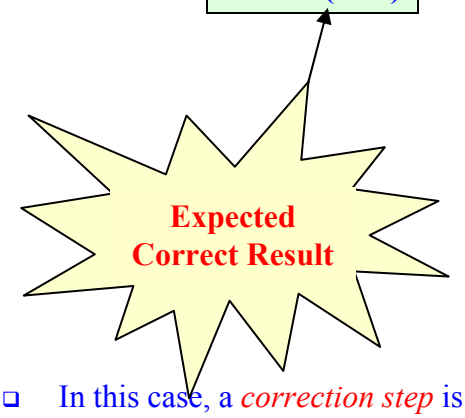
$Z = X + Y'$

$= X + (M-Y)$

$Z = M + (X-Y)$ (3)

- which is *different* from the *expected correct result* **of**

$Z = +(X-Y)$ (4)



- In this case, a *correction step* is required for the final result.
- The *correction step* depends on the value of M.

Correction Step for R's and (R-1)'s Complements

The previous analysis shows that computing **Z = (X-Y)** using complement arithmetic gives:

- The correct complement representation of the answer if the result is negative, that is **M - (Y-X)**.
- *Alternatively,* if the result is positive it gives an answer of **M + (X-Y)** which is *different* from the *correct answer* **of +(X-Y)** requiring a correction step.
- The correction step depends on the value of M

For the R's Complement

Note that $\mathbf{M}_R = r^n = 1000\ldots00.000$

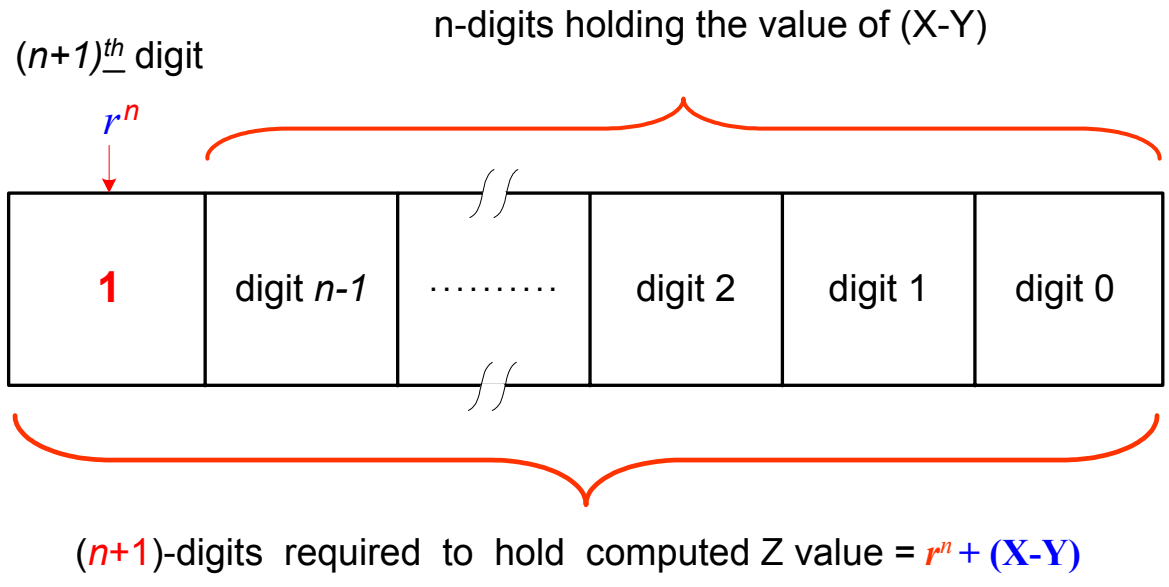
$(n+1)^{th}$ Position

n Positions

Thus, the computed result (**M + (X-Y)**) is given by

$$Z = r^n + (X-Y)$$

Since **(X-Y)** is *positive*, the computed Z **value** $\{r^n + (X-Y)\}$ requires $(n + 1)$ integral digits to be expressed as shown in Figure.



In this case, it is clear that **Z = r^n + (X-Y)** consists of the digit **1** in the $(n+1)^{th}$ digit position while the least significant *n* digits will hold the *expected correct result* of (X-Y).
Since X, Y, and the result Z are stored in registers of *n* digits, the correct result (X- Y) is simply obtained by neglecting the 1 in the $(n+1)^{th}$ digit.

The **1** in the $(n+1)^{th}$ digit is typically referred to as "*end carry*".

Conclusion:

- For the R's complement method;
  i. If the computed result has *no end carry*. This result is the correct answer.
  ii. In case the computed result has an *end carry*, this end carry is *DISACRDED* and the remaining digits represent the correct answer.

For the (R-1)'s Complement

- $\mathbf{M}_{R-1} = r^n - ulp$

Thus, the computed result (**M + (X-Y)**) is given by

$$\mathbf{Z} = (r^n - ulp) + \mathbf{(X-Y)}$$

For a *positive* value of **(X-Y)**, the computed Z **value** $\{(r^n - ulp) + \mathbf{(X-Y)}\}$ requires $(n + 1)$ integral digits for its representation.

Again, r^n represents a **1** in the $(n+1)^{th}$ digit position (i.e. an end carry) while the least significant *n* digits will hold the value (X-Y-*ulp*).

Since the *expected correct answer* is (X-Y), the correct result is obtained by adding a *ulp* to the least significant digit position.

Q. What does the computed result represent in case X=Y **?**

Conclusion:

- For the (R-1)'s complement method;
  a. If the computed result has no end carry. This result is the correct answer.
  b. In case the computed result has an end carry, this end carry is added to the least significant position (i.e., as *ulp*).

Important Note:

- *The previous conclusions are valid irrespective of the signs of X or Y and for both addition and subtraction operations.*

Add/Subtract Procedure

It is desired to compute **Z = X ± Y,** *where* **X, Y** and **Z**:

(a) are signed numbers represented in one of the complement representation methods.

(b) have *n* integral digits including the sign digit.

The procedure for computing the value of **Z** depends on the used complement representation method:

R's Complement Arithmetic

1. If the operation to be performed is addition compute **Z = X + Y**, otherwise if it is subtraction, **Z = X – Y,** compute **Z = X + Y'** instead**.**
2. If the result has no end carry, the obtained value is the correct answer.
3. If the result has an end carry, discard it and the value in the remaining digits is the correct answer.

(R-1)'s Complement Arithmetic

1. If the operation to be performed is addition compute **Z = X + Y**, otherwise if it is subtraction, **Z = X – Y,** compute **Z = X + Y'** instead**.**
2. If the result has no end carry, the obtained value is the correct answer.
3. If the result has an end carry, this end carry should be added to the least significant digit (*ulp*) to obtain the final correct answer.

Examples

RADIX COMPLEMENT

Compute (M-N) and (N-M), *where* $M=(072532)_{10}$ $N=(003250)_{10}$

Both M & N must have the same # of Digits (*Pad with 0`s if needed*).

COMPUTING (M – N)

Regular Subtraction

M		0	7	2	5	3	2
N	–	0	0	3	2	5	0
		0	6	9	2	8	2

Complement Method

Compute (M+N')

M		0	7	2	5	3	2
N'	+	9	9	6	7	5	0
	1	0	6	9	2	8	2

Correct Result

Discard End Carry

COMPUTING (N – M)

Regular Subtraction

N		0	0	3	2	5	0
M	–	0	7	2	5	3	2
	–	0	6	9	2	8	2

-ive sign

Equivalent Results The –ive Result is Represented by the 10's Complement

Complement Method

Compute (N + M')

N		0	0	3	2	5	0
M'	+	9	2	7	4	6	8
		9	3	0	7	1	8

No End Carry

This is the 10's complement representation of a –ive number, i.e. the result (930718) represents the number (-069282)

Example : (2`s Comp) $M=(01010100)_2$ $N=(01000100)_2$

Note: Both M & N are positive 8-bit numbers

COMPUTING (M – N)

Regular Subtraction

M		0	1	0	1	0	1	0	0
N	–	0	1	0	0	0	1	0	0
		0	0	0	**1**	0	0	0	0

Complement Method

Compute (M+N')

M		0	1	0	1	0	1	0	0
N'	+	1	0	1	1	1	1	0	0
	1	0	0	0	**1**	0	0	0	0

Correct Result

Sign Bit

Discard Carry Out

COMPUTING (N – M)

Regular Subtraction

N		0	1	0	0	0	1	0	0
M	–	0	1	0	1	0	1	0	0
	–	0	0	0	1	0	0	0	0

-ive sign

Equivalent Results The –ive Result is Represented by the 2's Complement

Complement Method

Compute (N + M')

N		0	1	0	0	0	1	0	0
M'	+	1	0	1	0	1	1	0	0
		1	1	1	1	0	0	0	0

Sign Bit

No End Carry

This is the 2's complement representation of a –ive number, i.e. the result (11110000) represents the number (-00010000)

DIMINISHED / (R-1)'s RADIX COMPLEMENT

Compute (M-N) and (N-M), *where* **M=$(072532)_{10}$ N=$(003250)_{10}$**

Both M & N must have the same # of Digits (*Pad with 0`s if needed*).

COMPUTING (M – N)

Regular Subtraction

M		0	7	2	5	3	2
N	–	0	0	3	2	5	0
		0	6	9	2	8	2

Complement Method

Compute (M+N')

M		0	7	2	5	3	2
N'	+	9	9	6	7	4	9
		1	0	6	9	2	8	1
	+						1
		0	6	9	2	8	2

End Carry

Correct Result

COMPUTING (N – M)

Regular Subtraction

N		0	0	3	2	5	0
M	–	0	7	2	5	3	2
	–	0	6	9	2	8	2

-ive sign

Complement Method

Compute (N + M')

N		0	0	3	2	5	0
M'	+	9	2	7	4	6	7
		9	3	0	7	1	7

No End Carry

Equivalent Results The –ive Result is Represented by the 9's Complement

This is the 9's complement representation of a –ive number, i.e. the result (930717) represents the number (-069282)

Example : (1`s Comp) $M=(01010100)_2$ $N=(01000100)_2$

Note: Both M & N are positive 8-bit numbers

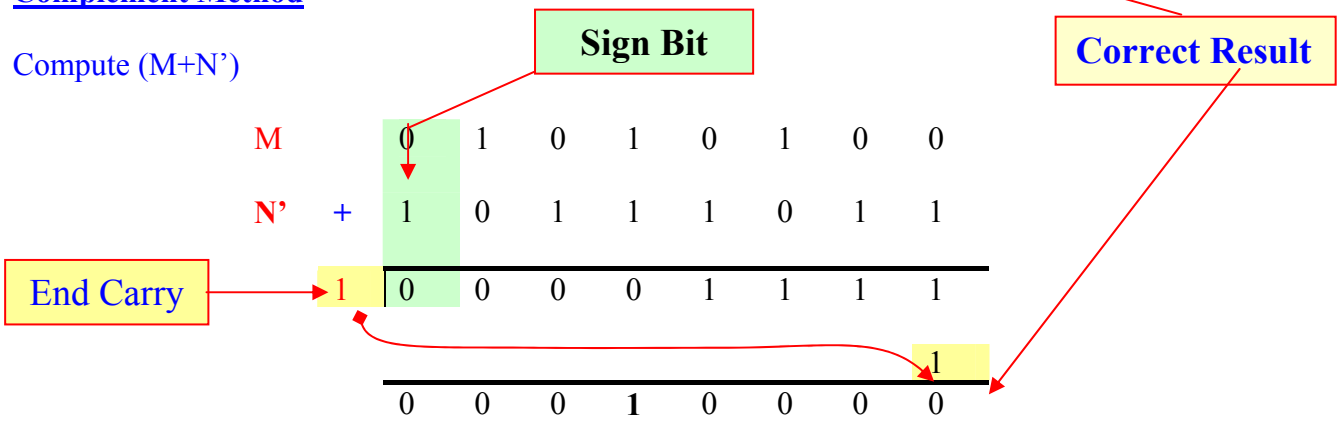
COMPUTING (M – N)

Regular Subtraction

M		0	1	0	1	0	1	0	0
N	–	0	1	0	0	0	1	0	0
		0	0	0	**1**	0	0	0	0

Complement Method

Compute (M+N')



COMPUTING (N – M)

Regular Subtraction

N		0	1	0	0	0	1	0	0
M	–	0	1	0	1	0	1	0	0
	–	0	0	0	1	0	0	0	0

-ive sign

Equivalent Results The –ive Result is Represented by the 1's Complement

Complement Method

Compute (N + M')

N		0	1	0	0	0	1	0	0
M'	+	1	0	1	0	1	0	1	1
		1	1	1	0	1	1	1	1

Sign Bit

No End Carry

This is the 1's complement representation of a –ive number, i.e. the result (11101111) represents the number (-00010000)

Overflow Condition

- If adding two *n*-digit *unsigned* numbers results in an *n*+1 digit sum, this represents an *overflow* condition.
- In digital computers, overflow represents a problem since register sizes are fixed, accordingly a result of *n*+1 bits cannot fit into an n-bit register and the most significant bit will be lost.
- Overflow condition is a problem whether the added numbers are signed or unsigned.
- In case of signed numbers, overflow may occur only if the two numbers being added have the same sign, i.e. either both numbers are positive or both are negative.
- For 2's complement represented numbers, *the sign bit is treated as part of the number and an end carry does not necessarily indicate an overflow*.
- In 2's complement system, an overflow condition always changes the sign of the result and gives an erroneous n-bit answer. Two cases are possible:
  1. Both operands are positive (sign bits=0). In this case, an overflow will result from a carry of 1 into the sign bit column; causing the sum to be interpreted as a negative number.
  2. Both operands are negative (sign bits=1). In this case, an overflow will result when no carry is received at the sign bit column causing the two sign bits to be added resulting in a 0 in the sign bit column and a carry out in the $(n+1)^{th.}$ bit position which will be discarded. This causes the sum to be interpreted as a positive number.
- Accordingly, an overflow condition is detected if one of the two following conditions occurs:
  (a) There is a carry into the sign bit column but no carry out of that column.
  (b) There is a carry out of the sign bit column but no carry into that column.

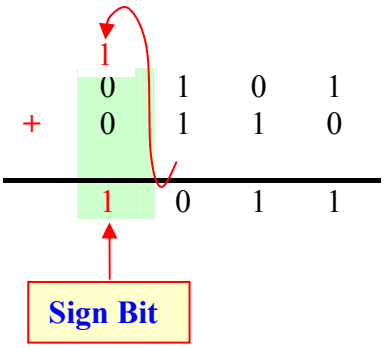
Example:

- Consider the case of adding the binary values corresponding to $(+5)_{10}$ and $(+6)_{10}$ where the correct result should be (+11).
- Even though the operands $(+5)_{10}$ & $(+6)_{10}$ can be represented in 4-bits, the result $(+11)_{10}$ cannot be represented in 4-bits.
- Accordingly, the 4-bit result will be erroneous due to "*overflow*".

Add (+5) to (+6) using 4-bit registers and 2's complement representation.

$(+5)_{10} \rightarrow (0101)_2$

$(+6)_{10} \rightarrow (0110)_2$



*There is a carry **into** the sign bit column but no carry **out** of it*

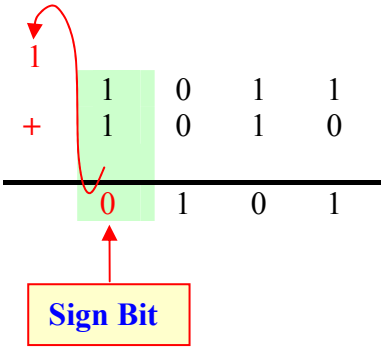
- If this overflow condition is not detected, the resulting sum would be erroneously interpreted as a negative number (1011) which equals $(-5)_{10}$.

Example:

Add (-5) to (-6) using 4-bit registers and 2's complement representation.

$(-5)_{10} \rightarrow (1011)_2$

$(-6)_{10} \rightarrow (1010)_2$



There is a carry out of the sign bit column but no carry into it.

- If this overflow condition is not detected, the resulting sum would be erroneously interpreted as a positive number (0101) which equals $(+5)_{10}$.

Example:

Using 8-bit registers, show the ***binary*** number representation of the decimal numbers (37), (-37), (54), and (-54) using the following systems:

	Signed magnitude system	Signed 1's complement System	Signed 2's complement system
37	00100101	00100101	00100101
-37	10100101	11011010	11011011
54	00110110	00110110	00110110
-54	10110110	11001001	11001010

Compute the result of the following operations in the ***signed 2's complement*** system.

I. (+37) – (+54)

Subtraction is turned into addition to the complement, i.e.

(+37) – (+54) → (+37) + (+54)'

$$\begin{array}{r} 0\ 0\ 1\ 0\ 0\ 1\ 0\ 1 \\ + \\ 1\ 1\ 0\ 0\ 1\ 0\ 1\ 0 \\ \hline 1\ 1\ 1\ 0\ 1\ 1\ 1\ 1 \end{array}$$

$= (-17)_{10}$

II. (-37) – (+54)

Subtraction is turned into addition to the complement, i.e.

(-37) – (+54) → (-37) + (+54)'

$$\begin{array}{r} 1\ 1\ 0\ 1\ 1\ 0\ 1\ 1 \\ + \\ 1\ 1\ 0\ 0\ 1\ 0\ 1\ 0 \\ \hline 1\ 1\ 0\ 1\ 0\ 0\ 1\ 0\ 1 \end{array}$$

Discard End Carry

$= -(01011011) = -(91)_{10}$

III. (54) + (-37)

$$\begin{array}{r} 0\ 0\ 1\ 1\ 0\ 1\ 1\ 0 \\ + \\ 1\ 1\ 0\ 1\ 1\ 0\ 1\ 1 \\ \hline 1\ 0\ 0\ 0\ 1\ 0\ 0\ 0\ 1 \end{array}$$

Discard End Carry

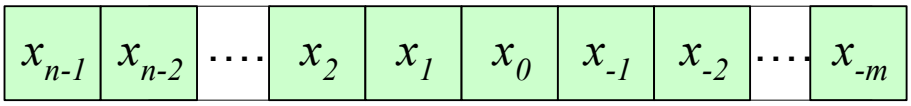
$= +(17)_{10}$

Range Extension of 2's Complement Numbers

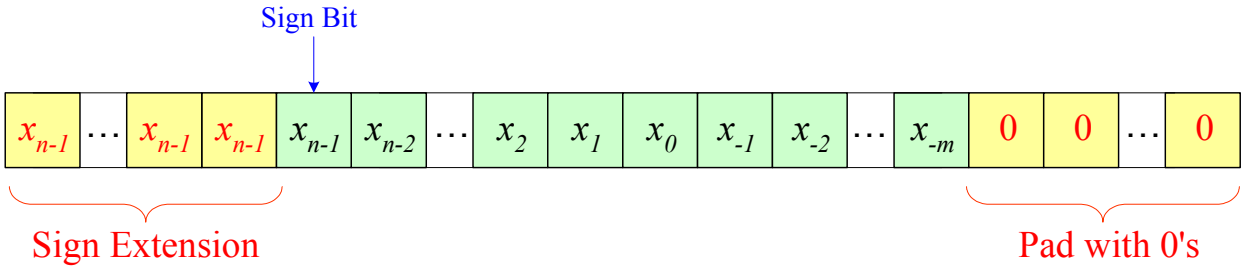
- To extend the representation of some 2's complement number X from *n-bits to n`-bits where n` > n*.
  1. If **X** is +ive → pad with 0`s to the *right* of fractional part and/or to the *left* of the integral part.
  2. If **X** is -ive → pad with 0`s to the *right* of fractional part and/or with 1`s to the *left* of the integral part.

In General

- Pad with 0`s to the *right* of fractional part and/or **extend sign bit to the *left* of the integral part** (*Sign Bit Extension*).



X- Before Extending its Range

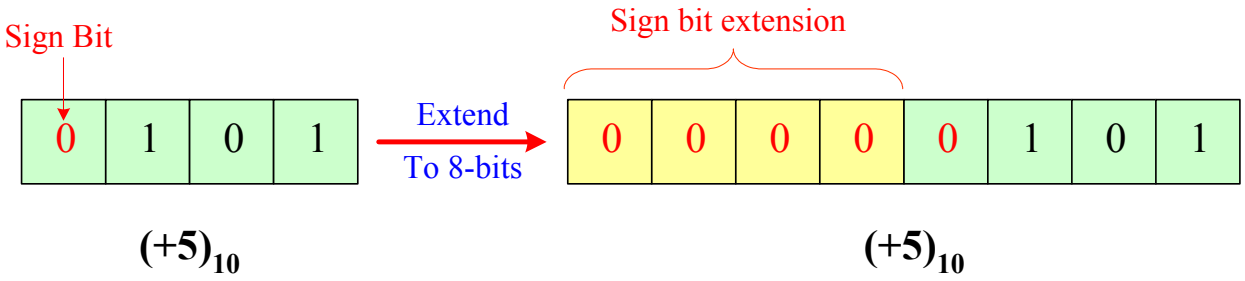


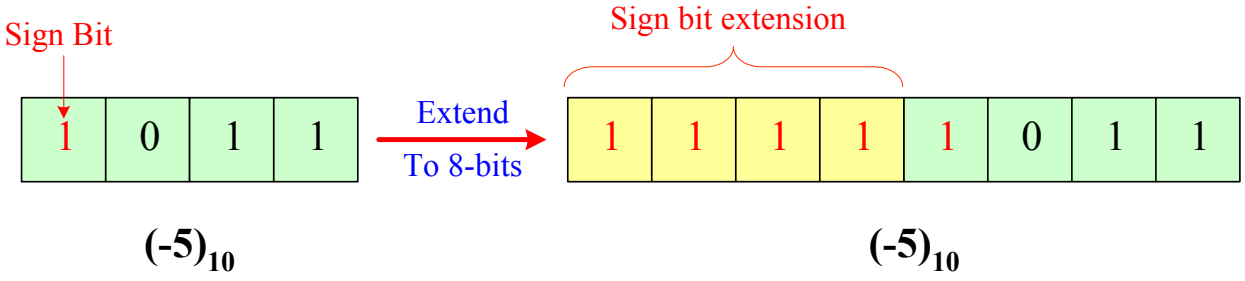
X- After Extending its Range

(*0's Padded to the Right of Fractional Part and the Sign is Extended to the Left of the Integral Part***)**

Example:

Show how the numbers $(+5)_{10}$ and $(-5)_{10}$ are represented in 2's complemenr using 4-bit registers then extend this representation to 8-bit registers.





Arithmetic Shifts

Effect of 1-Digit Shift

- Left Shift → *Multiply* by radix ***r***
- Right Shift → *Divide* by radix ***r***

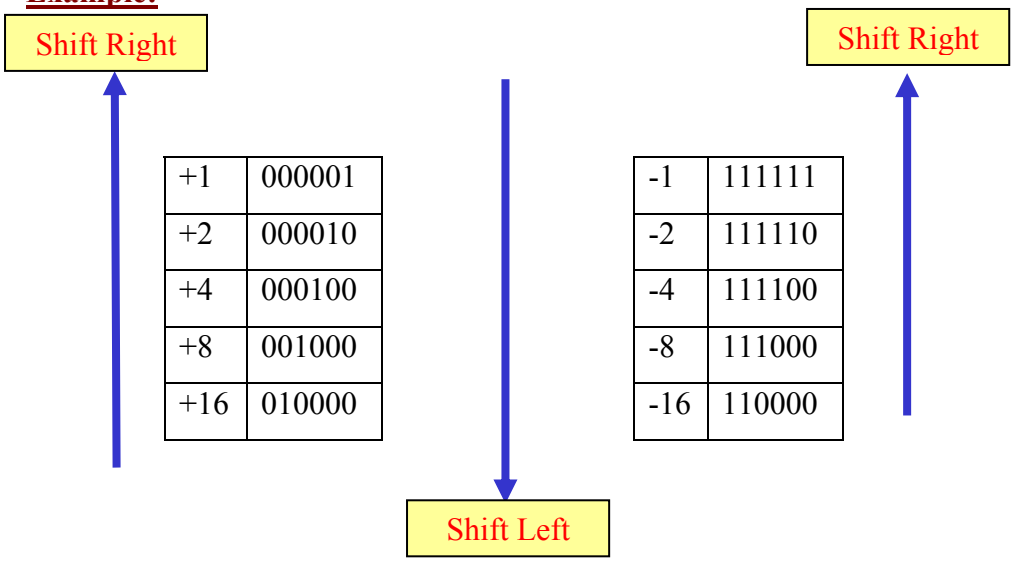
(a) Shifting Unsigned Numbers

- Shift-in **0`s** (for both Left & Right Shifts)

(b) Shifting 2's Complement Numbers

- Left Shifts: **0**`s are shifted-in
- Right Shifts: *Sign Bit Extended*

Example:



+1	000001
+2	000010
+4	000100
+8	001000
+16	010000

-1	111111
-2	111110
-4	111100
-8	111000
-16	110000

Binary Codes

Objectives

In this lesson, you will study:

1. Several binary codes including
   - Binary Coded Decimal (BCD),
   - Error detection codes,
   - Character codes
2. Coding versus binary conversion.

Binary Codes for Decimal Digits

- Internally, digital computers operate on binary numbers.
- When interfacing to humans, digital processors, e.g. pocket calculators, communication is decimal-based.
- Input is done in decimal then converted to binary for internal processing.
- For output, the result has to be converted from its internal binary representation to a decimal form.
- To be handled by digital processors, the decimal input (output) must be coded in binary in a digit by digit manner.
- For example, to input the decimal number **957**, each *digit* of the number is individually *coded* and the number is stored as **1001_0101_0111.**
- Thus, we need a specific code for each of the 10 decimal digits. There is a variety of such decimal binary codes.
- The shown table gives several common such codes.
- One commonly used code is the *Binary Coded Decimal* (**BCD**) code which corresponds to the first 10 binary representations of the decimal digits 0-9.
- The BCD code requires 4 bits to represent the 10 decimal digits.
- Since 4 bits may have up to 16 different binary combinations, a total of 6 combinations will be unused.
- The position weights of the BCD code are 8, 4, 2, 1.
- Other codes (shown in the table) use position weights of 8, 4, -2, -1 and 2, 4, 2, 1.
- An example of a non-weighted code is the *excess-3 code* where digit codes is obtained from their binary equivalent after adding 3. Thus the code of a decimal 0 is 0011, that of 6 is 1001, etc.

Decimal Digit	BCD 8	4	2	1	8	4	-2	-1	2	4	2	1	Excess-3			
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1
1	0	0	0	1	0	1	1	1	0	0	0	1	0	1	0	0
2	0	0	1	0	0	1	1	0	0	0	1	0	0	1	0	1
3	0	0	1	1	0	1	0	1	0	0	1	1	0	1	1	0
4	0	1	0	0	0	1	0	0	0	1	0	0	0	1	1	1
5	0	1	0	1	1	0	1	1	1	0	1	1	1	0	0	0
6	0	1	1	0	1	0	1	0	1	1	0	0	1	0	0	1
7	0	1	1	1	1	0	0	1	1	1	0	1	1	0	1	0
8	1	0	0	0	1	0	0	0	1	1	1	0	1	0	1	1
9	1	0	0	1	1	1	1	1	1	1	1	1	1	1	0	0
U	1	0	1	0	0	0	0	1	0	1	0	1	0	0	0	0
N	1	0	1	1	0	0	1	0	0	1	1	0	0	0	0	1
U	1	1	0	0	0	0	1	1	0	1	1	1	0	0	1	0
S	1	1	0	1	1	1	0	0	1	0	0	0	1	1	0	1
E	1	1	1	0	1	1	0	1	1	0	0	1	1	1	1	0
D	1	1	1	1	1	1	1	0	1	0	1	0	1	1	1	1

Number Conversion versus Coding

- Converting a decimal number into binary is done by repeated division (multiplication) by 2 for integers (fractions) (see lesson 4).
- Coding a decimal number into its BCD code is done by replacing each decimal digit of the number by its equivalent 4 bit BCD code.

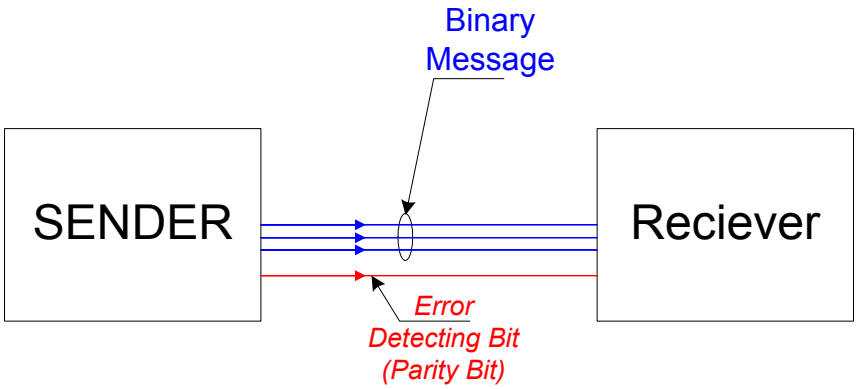
Example Converting $(13)_{10}$ into binary, we get **1101**, coding the same number into BCD, we obtain **00010011**.

Exercise: Convert $(95)_{10}$ into its binary equivalent value and give its BCD code as well.

Answer {$(1011111)_2$, and **10010101**}

Error-Detection Codes

- Binary information may be transmitted through some communication medium, e.g. using wires or wireless media.
- A corrupted bit will have its value changed from 0 to 1 or vice versa.
- To be able to detect errors at the receiver end, the sender sends an extra bit (*parity bit*) with the original binary message.



- A *parity bit* is an extra bit included with the *n-bit binary message* to make the total number of 1's in this message (*including the parity bit*) either odd or even.
- If the *parity bit* makes the total number of 1's an odd (even) number, it is called odd (even) parity.
- The table shows the ***required*** odd (***even***) ***parity*** for a 3-bit message.

Three-Bit Message			**Odd Parity Bit**	**Even Parity Bit**
X	**Y**	**Z**	**P**	**P**
0	0	0	1	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	0	1

- At the receiver end, an error is detected if the message does not match have the proper parity (odd/even).
- Parity bits can detect the occurrence 1, 3, 5 or any odd number of errors in the transmitted message.

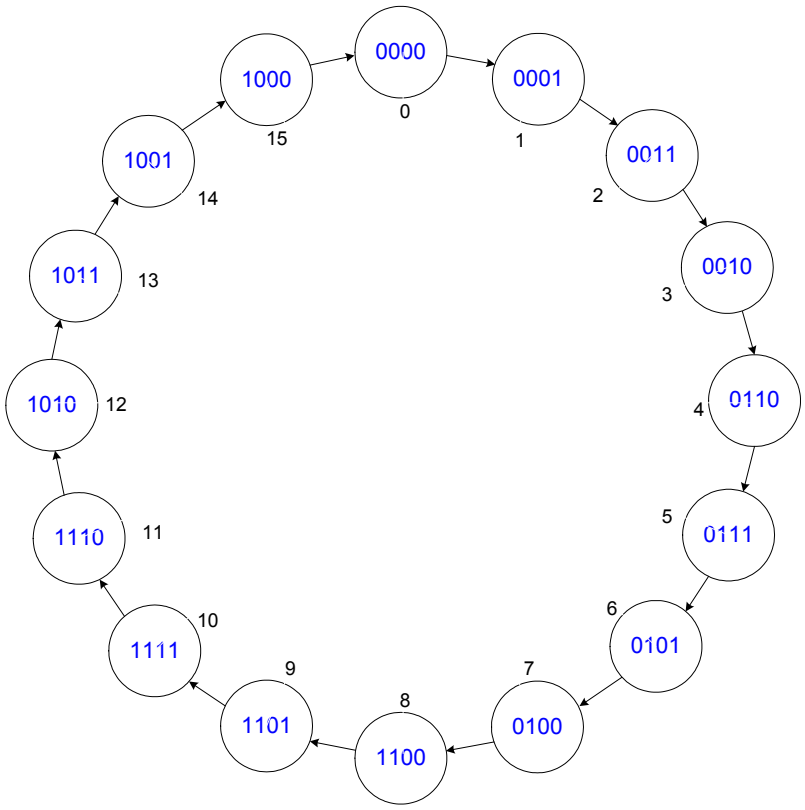
- No error is detectable if the transmitted message has 2 bits in error since the total number of 1's will remain even (or odd) as in the original message.
- In general, a transmitted message with even number of errors cannot be detected by the parity bit.

Error-Detection Codes

- Binary information may be transmitted through some communication medium, e.g. using wires or wireless media.
- Noise in the transmission medium may cause the transmitted binary message to be corrupted by changing a bit from 0 to 1 or vice versa.
- To be able to detect errors at the receiver end, the sender sends an extra bit (*parity bit*).

Gray Code

- The Gray code consist of 16 4-bit code words to represent the decimal Numbers 0 to 15.
- For Gray code, successive code words differ by only one bit from one to the next as shown in the table and further illustrated in the Figure.



Gray Code				Decimal Equivalent
0	0	0	0	0
0	0	0	1	1
0	0	1	1	2
0	0	1	0	3
0	1	1	0	4
0	1	1	1	5
0	1	0	1	6
0	1	0	0	7
1	1	0	0	8
1	1	0	1	9
1	1	1	1	10
1	1	1	0	11
1	0	1	0	12
1	0	1	1	13
1	0	0	1	14
1	0	0	0	15

Character Codes

ASCII Character Code

- ASCII code is a 7-bit code. Thus, it represents a total of 128 characters.

- Out of the 128 characters, there are 94 printable characters and 34 control (non- printable) characters.
- The printable characters include the upper and lower case letters (2*26), the 10 numerals (0-9), and 32 special characters, e.g. @, %, $, etc.
- For example, "A" is at $(41)_{16}$, while "a" is at $(61)_{16}$.
- To convert upper case letters to lower case letters, add $(20)_{16}$. Thus "a" is at $(41)_{16} + (20)_{16} = (61)_{16}$.
- The code of the character "9" at position $(39)_{16}$ is different from the binary number 9 (0001001). To convert ASCII code of a numeral to its binary number value, subtract $(30)_{16}$.

00	NUL	10	DLE	20	SP	30	0	40	@	50	P	60	`		
01	SOH	11	DC1	21	!	31	1	41	A	51	Q	61	a		
02	STX	12	DC2	22	"	32	2	42	B	52	R	62	b		
03	ETX	13	DC3	23	#	33	3	43	C	53	S	63	c		
04	EOT	14	DC4	24	$	34	4	44	D	54	T	64	d		
05	ENQ	15	NAK	25	%	35	5	45	E	55	U	65	e		
06	ACK	16	SYN	26	&	36	6	46	F	56	V	66	f		
07	BEL	17	ETB	27	'	37	7	47	G	57	W	67	g		
08	BS	18	CAN	28	(	38	8	48	H	58	X	68	h		
09	HT	19	EM	29	)	39	9	49	I	59	Y	69	i		
0A	LF	1A	SUB	2A	*	3A	:	4A	J	5A	Z	6A	j		
0B	VT	1B	ESC	2B	+	3B	;	4B	K	5B	[	6B	k		
0C	FF	1C	FS	2C	´	3C	<	4C	L	5C	\	6C	l		
0D	CR	1D	GS	2D	-	3D	=	4D	M	5D	]	6D	m		
0E	SO	1E	RS	2E	.	3E	>	4E	N	5E	^	6E	n		
0F	SI	1F	US	2F	/	3F	?	4F	O	5F	_	6F	o		

NUL	Null	FF	Form feed	CAN	Cance
SOH	Start of heading	CR	Carriage return	EM	End o
STX	Start of text	SO	Shift out	SUB	Substi
ETX	End of text	SI	Shift in	ESC	Escape
EOT	End of transmission	DLE	Data link escape	FS	File se
ENQ	Enquiry	DC1	Device control 1	GS	Group
ACK	Acknowledge	DC2	Device control 2	RS	Record
BEL	Bell	DC3	Device control 3	US	Unit se
BS	Backspace	DC4	Device control 4	SP	Space
HT	Horizontal tab	NAK	Negative acknowledge	DEL	Delete
LF	Line feed	SYN	Synchronous idle		
VT	Vertical tab	ETB	End of transmission block		

Unicode Character Code

- Unicode is a 16-bit character code that accommodates characters of various languages of the world.

Binary Logic and Gates

Introduction

- Our objective is to learn how to design ***digital*** circuits.
- These circuits use ***binary*** systems.
- **Signals** in such binary systems may represent only one of ***2 possible values*** *0 or 1*
- *Physically*, these signals are electrical *voltage* signals
- **These signals may assume either a high** or a **Low** voltage value.
- The high voltage value typically equals the voltage of the power supply **(**e.g. 5 volts or 3.3 volts), and the **Low** voltage value is typically 0 volts (or Ground).
- When a signal is at the **High** voltage value, we say that the signal has a *Logic 1* value.
- When a signal is at the **Low** voltage value, we say that the signal has a *Logic 0* value.
- Hence, the *physical* value of a signal is the actual voltage value it carries, while its *Logic* value is either **1** (**High**) or **0** (**Low**).
- Digital circuits process (or manipulate) input binary signals and produce the required output binary signals as shown in **Figure 1**

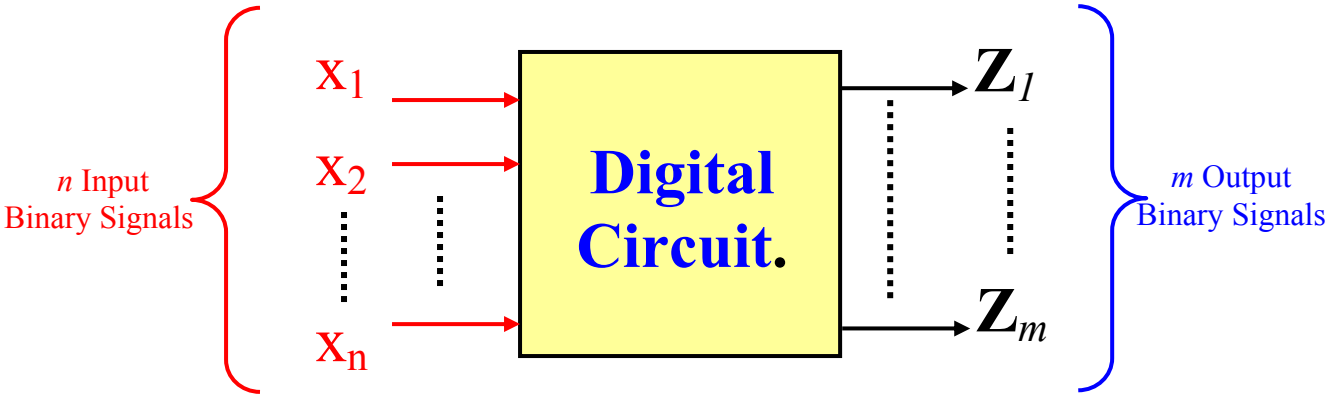


Figure 1 A Digital Circuit with *n* Input Signals and *m* Output Signals

- Generally, the circuit will have a number of input signals (say *n* of them) as shown in the Figure x_1, x_2, up to x_n, and a number of output signals (say m) Z_1, Z_2, up to Z_m.
- The value assumed by the i^{th} output signal Z_i depends on the values of the input signals x_1, x_2, up to x_n.
- In other words, we can say that Z_i is a function of the n input signals x_1, x_2, up to x_n. *Or we can write:*

$$Z_i = F_i(x_1, x_2, \ldots\ldots, x_n) \qquad \textit{for } i = 1, 2, 3, \ldots m$$

- The m output functions (F_i) are functions of binary signals and produce a single binary output signal.
- Thus, these functions are binary functions and require binary logic algebra for their derivation and manipulation. This binary system algebra is commonly referred to as ***Boolean Algebra*** after the mathematician George Boole. The functions are known as *Boolean functions* while the binary signals are represented by *Boolean variables*.
- To be able to design a digital circuit, we must learn how to *derive* the Boolean function implemented by this circuit.

Notes:

1. The two values of binary variables may be equivalently referred to as **0** and **1** or **False** (0) and **True** (1)or as **Low** (0) and **High**(1).
2. Whether we use 0 and 1 or False and True or Low and High, all these are referred to as Logic Values.
3. Systems manipulating Binary Logic Signals are commonly referred to as **Binary Logic systems.**
4. Digital circuits implementing a particular Binary (Boolean) function are commonly known as **Logic Circuits**.

CHAPTER OBJECTIVES

- **Learn** ***Binary Logic*** **and** ***BOOLEAN Algebra***
- **Learn How to Map a** ***Boolean Expressions*** **into Logic** ***Circuit Implementations***
- **Learn How To Manipulate** ***Boolean Expressions*** **and** ***Simplify Them***

Elements of Boolean Algebra (Binary Logic)

As in standard algebra, Boolean algebra has 3 main elements:

1. Constants,
2. Variables, and
3. Operators.

Logically

- ***Constant*** *Values* are either 0 or 1 *Binary* ***Variables*** $\in \{0, 1\}$
- *3 Possible* ***Operators*** The **AND operator**, the **OR** operator, and the **NOT** operator

Physically

- **Constants** $\Rightarrow$ Power Supply Voltage (Logic 1)
  $\Rightarrow$ Ground Voltage (Logic 0)
- **Variables** $\Rightarrow$ Signals (**H**igh = **1**, **L**ow = **0**)
- **Operators** $\Rightarrow$ Electronic Devices (**Logic Gates**)
  1. • **AND** - Gate
  2. • **OR** - Gate
  3. • **NOT** - Gate (***Inverter***)

Logic Gates & Logic Operations

The AND Operation

- If ***X*** and ***Y*** are two *binary variables*, the result of the operation ***X*** **AND** ***Y*** is **1** *if and only if* **both** ***X*** **= 1 and** ***Y*** **= 1**, and is **0 otherwise**.
- In Boolean expressions, the **AND** operation is represented either by a "*dot*" or by the absence of an operator. Thus, ***X*** **AND** ***Y*** is written as ***X.Y*** or just ***XY***
- This is summarized in the following table (commonly called *truth table*):

Table 1 Truth Table of the AND operation

X	*Y*	*Z = X AND Y* *Z=XY*
F	F	F
F	T	F
T	F	F
T	T	T

Table 1 Truth Table of the AND operation

X	*Y*	*Z = X AND Y* *Z=XY*
0	0	0
0	1	0
1	0	0
1	1	1

- The electronic device which performs the AND operation is called the **AND gate**. Figure 2 shows the symbol of a 2-input AND gate which has two inputs (***X*** and ***Y***) and gives one output ***Z=XY***

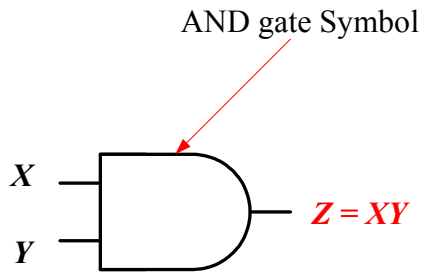


Figure 2 Two-Input AND gate

- The AND logic can be further illustrated using what is known as the Venn diagram
- AND gates may have more than 2 inputs. Figure 3 shows a 3-input AND gate.

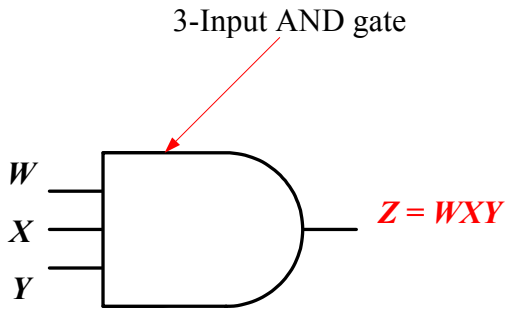


Figure 3 Three-Input AND gate

- The truth table of the output variable Z=WXY of the 3-input AND gate is given in Table 2

Table 2 Truth Table of 3-Input AND gate

W	*X*	*Y*	*Z=WXY*
0	**0**	**0**	**0**
0	**0**	**1**	**0**
0	**1**	**0**	**0**
0	**1**	**1**	**0**
1	**0**	**0**	**0**
1	**0**	**1**	**0**
1	**1**	**0**	**0**
1	**1**	**1**	**1**

Notes

- The output of an AND gate is 1 *if and only if* **ALL** its input signals are 1's, otherwise it is 0.
- A function of two input binary variables will have a truth table of 4 rows since each variable may assume any one of two possible values (0 or 1).
- A function of three input variables will have a truth table of 8 rows since each variable may assume any one of two possible values (0 or 1).
- In general, ***n*** input variables have 2^n possible combinations. Accordingly, a function of ***n*** input variables, will have a truth table of 2^n rows.

The OR Operation

- If ***X*** and ***Y*** are two *binary variables*, the result of the operation ***X* OR *Y*** is **1** *if and only if* **either *X* = 1 or *Y* = 1 or both X & Y are 1's**, but it is **0 otherwise**.
- In other words, ***X* OR *Y*** is 0 if and only if both ***X* = 0 and *Y* = 0**, but is **1 otherwise**.
- In Boolean expressions, the **OR** operation is represented by a "*plus*" sign. Thus, ***X* OR *Y*** is written as ***X*+*Y***
- This is summarized in the Table 3.

Table 3 Truth Table of the OR operation

X	*Y*	*Z = X OR Y* *Z=X+Y*
0	**0**	**0**
0	**1**	**1**
1	**0**	**1**
1	**1**	**1**

- The electronic device which performs the OR operation is called the **OR gate**. Figure 4 shows the symbol of a 2-input OR gate which has two inputs (***X*** and ***Y***) and gives one output ***Z=X+Y***

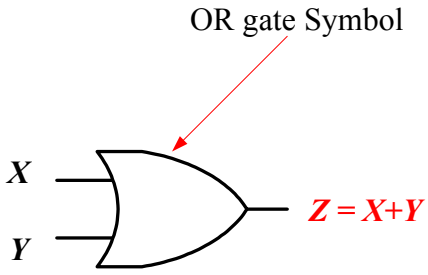


Figure 4 Two-Input OR gate

- The OR logic can be further illustrated using the Venn diagram
- OR gates may have more than 2 inputs. Figure 5 shows a 3-input OR gate.

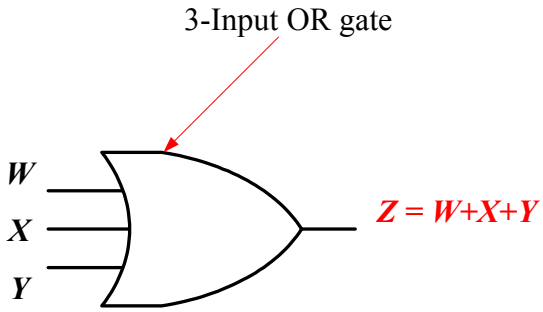


Figure 5 Three-Input OR gate

- The truth table of a 3 input OR gate Z=W+X+Y is given in **Table** 4

Table 4 Truth Table of

3-Input OR gate

W	*X*	*Y*	*Z=WXY*
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

- In general, the output of an OR gate is 1 *unless* **ALL** its input signals are 0's.

The NOT Operation

- NOT is a "*unary*" operator.
- IF ***Z*** = **NOT** ***X***, then the value of ***Z*** will always be the complement of the value of ***X***. In other words, if ***X*** = 0 then ***Z*** = 1, and if ***X*** = 1 then ***Z*** =0.
- In Boolean expressions, the **NOT** operation is represented by either a bar on top of the variable (e.g. $Z = \overline{X}$) or a prime (e.g. $Z = X'$).
- This is summarized in Table 5.

Table 5 Truth Table of the

NOT operation

X	*Z=X'*
0	1
1	0

- The electronic device which performs the NOT operation is called the **NOT gate, or simply** ***INVERTER***. Figure 5 shows the inverter symbol.

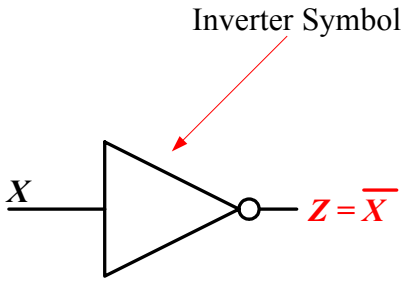


Figure 5 An Inverter

- If $Z = \overline{X}$, Z is commonly referred to as the *Complement* of X. Alternatively, we say that *Z equals X-complemented*
- The NOT operation can be further illustrated using the Venn diagram

Boolean Algebra

Logic Circuits and Boolean Expressions

- A Boolean expression (or a Boolean function) is a combination of Boolean *variables*, *AND*-operators, *OR*-operators, and *NOT* operators.
- • *Boolean Expressions* (*Functions*) are fully defined by their *truth tables*Each Boolean function (expression) can be implemented by a digital *logic circuit* which consists of logic gates.
    - Variables of the function correspond to signals in the logic circuit,
    - Operators of the function are converted into corresponding logic gates in the logic circuit.

Example

Consider the expression $F = X + (\overline{Y}.Z)$ The diagram of the logic circuit corresponding to this function is shown in Figure 6

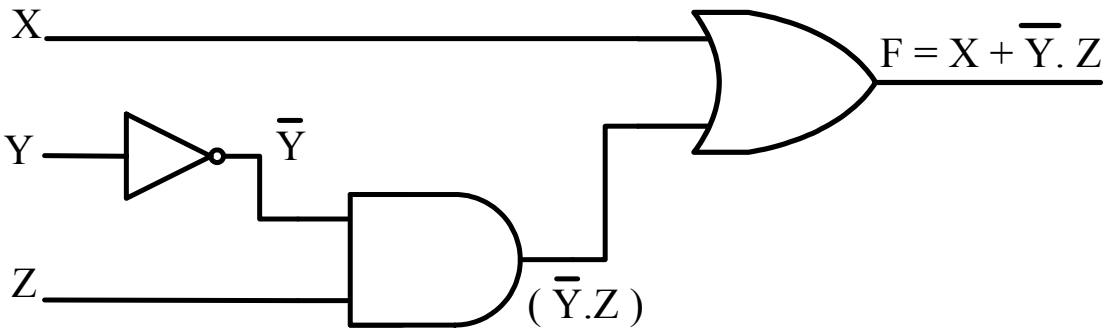


Figure 6 Logic Circuit Diagram of $F = X + (\overline{Y}.Z)$

The truth table of this function is shown in Table 6

Table .6 Truth Table of $F = X + (\overline{Y}.Z)$

X	Y	Z	Y`	Y`Z	F= X + Y`Z
0	0	0	1	0	0
0	0	1	1	1	1
0	1	0	0	0	0
0	1	1	0	0	0
1	0	0	1	0	1
1	0	1	1	1	1
1	1	0	0	0	1
1	1	1	0	0	1

- Since F is function of 3 variables (X, Y, Z), the truth table has 2^3 or 8 rows.

Basic Identities of Boolean Algebra

AND Identities

From the truth table of the AND operation, shown here for reference, we can derive some basic identities. These identities can be easily verified by showing that they are valid for both possible values of X (0 and 1).

AND Truth Table		
X	***Y***	***Z=XY***
0	**0**	**0**
0	**1**	**0**
1	**0**	**0**
1	**1**	**1**

1. $0 \cdot X = 0$

AND Truth Table		
X	***Y***	***Z=XY***
0	**0**	**0**
0	**1**	**0**
1	**0**	**0**
1	**1**	**1**

2. $1 \cdot X = X$

AND Truth Table		
X	***Y***	***Z=XY***
0	**0**	**0**
0	**1**	**0**
1	**0**	**0**
1	**1**	**1**

3. $X \cdot X = X$

AND Truth Table		
X	***Y***	***Z=XY***
0	**0**	**0**
0	**1**	**0**
1	**0**	**0**
1	**1**	**1**

4. $X.\overline{X} = 0$

AND Truth Table		
X	***Y***	***Z=XY***
0	**0**	**0**
0	**1**	**0**
1	**0**	**0**
1	**1**	**1**

OR Identities

From the truth table of the OR operation, shown here for reference, we can derive some basic identities. These identities can be easily verified by showing that they are valid for both possible values of X (0 and 1).

1. $1 + X = 1$

OR Truth Table		
X	***Y***	***Z=X+Y***
0	**0**	**0**
0	**1**	**1**
1	**0**	**1**
1	**1**	**1**

2. $0 + X = X$

OR Truth Table		
X	***Y***	***Z=X+Y***
0	**0**	**0**
0	**1**	**1**
1	**0**	**1**
1	**1**	**1**

3. $X + X = X$

OR Truth Table		
X	***Y***	***Z=X+Y***
0	**0**	**0**
0	**1**	**1**
1	**0**	**1**
1	**1**	**1**

4. $X + \overline{X} = 1$

OR Truth Table		
X	***Y***	***Z=X+Y***
0	**0**	**0**
0	**1**	**1**
1	**0**	**1**
1	**1**	**1**

Summary of the basic identity

AND Identities

1. $0 . X = 0$
2. $1 . X = X$
3. $X . X = X$
4. $X . \overline{X} = 0$

OR Identities

5. $1 + X = 1$
6. $0 + X = X$
7. $X + X = X$
8. $X + \overline{X} = 1$

Duality Principle

- Given a Boolean expression, its *dual* is obtained by replacing each 1 with a 0, each 0 with a 1, each AND (.) with an OR (+), and each OR (+) with an AND(.).
- The dual of an identity is also an identity. This is known as the duality principle.

It can be easily shown that the AND basic identities and the OR basic identities are duals as shown in Table 7

Table 7 Duality of the AND and OR Basic Identities

AND Identities		Dual Identities (OR Udentities)
0 . X = 0	**0 → 1** **. → +**	**1 + X = 1**
1 . X = X	**1 → 0** **. → +**	**0 + X = X**
X . X = X	**. → +**	**X + X = X**
X . X = 0	**0 → 1** **. → +**	**X + X = 1**

Another Important Identity

$$\left(\overline{\overline{X}}\right) = X$$

- This can be simply proven from the truth table of the NOT operation as shown.

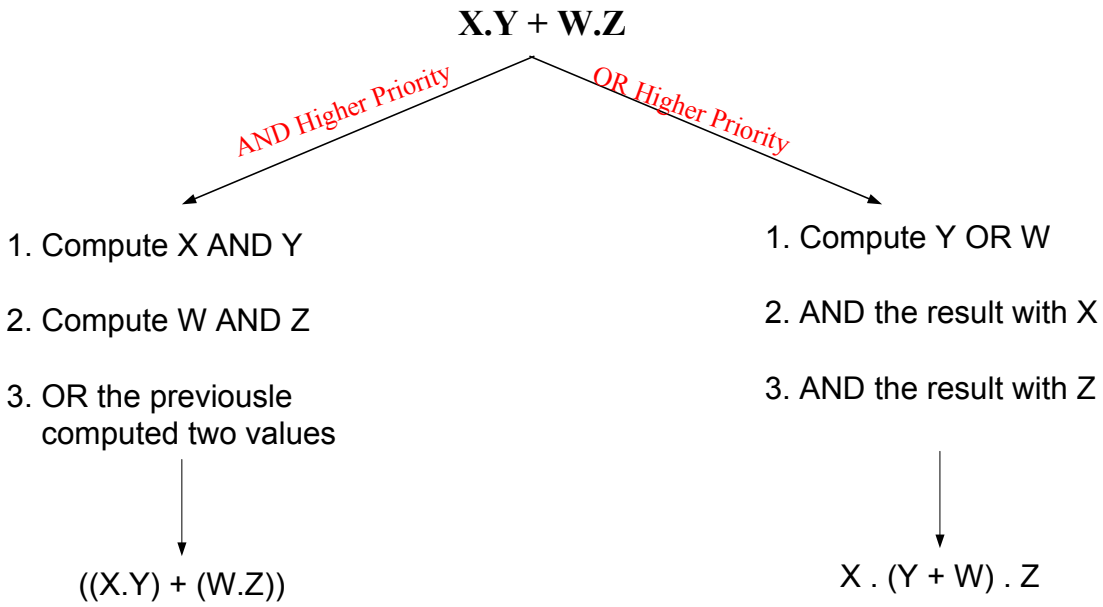
NOT operation

Truth Table

X	$\overline{X}$	$\left(\overline{\overline{X}}\right)$
0	**1**	**0**
1	**0**	**1**

Operator Precedence

Given the Boolean expression **X.Y + W.Z** the order of applying the operators will affect the final value of the expression.



For Boolean Algebra, the precedence rules for various operators are given below , in a decreasing order of priority:

1- *Parentheses* → *Highest Priority*

2- *Not* operator (*Complement*)

3- *AND* operator,

4- *OR* operator → *Lowest Priority*

Properties of Boolean Algebra

Important properties of Boolean Algebra are shown in Table

		Property	Dual Property
1	Commutative	X + Y = Y + X	X . Y = Y . X
2	Distributive	X.(Y + Z) = X.Y + X.Z	X+(Y.Z) = (X+Y).(X+Z)
3	DeMorgan	(X + Y) ` = X`.Y`	(X.Y) ` = X` + Y`
4	Extended DeMorgan	(A+B+C+ ….+Z)` = A`.B`.C`…..Z`	(A.B.C….Z)` = A`+B`+C`+….+Z`
5	Generalized DeMorgan	$[F(x_1,x_2,\ldots,x_n,0,1,+,.)]` = F(x`_1,x`_2,\ldots,x`_n,1,0,.,+)$	

Notes

- The above properties can be easily proved using truth tables.
- The only difference between the *dual* of an expression and the *complement* of that expression is that in the dual *variables* are not complemented while in the complement expression, all variables are complemented.
- Using the above properties, complex *Boolean expressions* can be *manipulated* into a *simpler forms* resulting in simpler logic circuit implementations.
- Simpler expressions are generally implemented by simpler logic circuits which are both faster and less expensive. This represents a great advantage since *cost* and *speed* are prime factors in the success and profitability of any product.

Algebraic Manipulation

- The objective here is to acquire some skills in manipulating Boolean expressions into simpler forms for more efficient implementations.
- Properties of Boolean algebra will be utilized for this purpose.

Example Prove that $X + XY = X$

Proof: $X + XY = X.(1 + Y) = X.1 = X$

Example Prove that $X + \overbrace{\quad}^{=1} = \overbrace{\quad}^{=X}$ → *This an important identity that is useful in simplifying more complex expressions*

Proof: This will be proved in two ways

(1) $X + X`Y = \overbrace{(X + X`)}^{=1}(X + Y)$

$= 1.(X + Y)$

$= X + Y$

(2) $X + X`Y = X.1 + X`Y =$

$= X.\overbrace{(1+Y)} + X`Y$

$= \overbrace{X + XY} + X`Y$

$= X + (XY + X`Y)$

$= X + Y(\overbrace{X + X`}^{=1})$

$= X + Y$

Example ``Consensus Theory``

Show that $XY + X`Z + YZ = XY + X`Z$

Proof:

LHS $= XY + X`Z + YZ$

$= XY + X`Z + YZ.1$ (1 = 1)

$= XY + X`Z + YZ.(X + X`)$

$= XY + X`Z + YZX + YZX`$

$= XY + YZX + X`Z + YZX`$

$= XY(1 + Z) + X`Z(1 + Y)$ (1 = 1, 1 = 1)

$= XY.1 + X`Z.1 = XY + X`Z =$ **LHS**

Example

Simplify the following function

$$F_1 = \overline{(A + \overline{B} + A\overline{B})}\ \ \overline{(AB + \overline{A}C + BC)}$$

Solution:

$$\mathbf{F1} = \overline{(A + \overline{B} + A\overline{B})}\ \ \overline{(AB + \overline{A}C + BC)}$$

Using De-Morgan theorem

- $\overline{(A + \overline{B} + A\overline{B})} = A` . B. (A` + B) = A` . B + A` . B = A` . B$
- $\overline{(AB + \overline{A}C + BC)} = (A` + B`).(A + C` ).(B` + C`)$

$$\mathbf{F1} = \overline{(A + \overline{B} + A\overline{B})}\ \ \overline{(AB + \overline{A}C + BC)}$$

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

Since X= X.X=X.X.X, we can rewrite the previous expression as follows

$$\mathbf{F1} = (A`.B). (A` . B). (A` . B). (A` + B).(A + C` ).(B` + C`)$$

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

$$= (A`.B + 0 ) . (0 + A` . B.C` ) . (A`.B + A`.B.C`)$$

$$= (A`.B) . (A` . B.C` ) . (A`.B)$$

$$= A` . B.C`$$

Example

Simplify the following function

i. $\mathbf{G} = \overline{\left(\overline{(A + \overline{B} + C)} . (\overline{AB} + \overline{C}\,\overline{D}) + \overline{ACD} \right)}$

Solution:

$$G = \overline{\left(\overline{(A + \overline{B} + C)} . (\overline{AB} + \overline{C}\,\overline{D}) + \overline{ACD} \right)}$$

$$= \left((A + \overline{B} + C) + \left(AB . (C + D) \right) \right) . ACD$$

$$= (A + \overline{B} + C) . ACD + \left(AB . (C + D) \right) . ACD$$

$$= (ACD + ACD\overline{B}) + (ACDB + ACDB)$$

$$= ACD + ACDB$$

$$= ACD$$

Standard & Canonical Forms

CHAPTER OBJECTIVES

- **Learn** ***Binary Logic*** **and** ***BOOLEAN Algebra*****Learn How to Map a** ***Boolean Expression*** **into Logic** ***Circuit Implementation*** **Learn How To Manipulate** ***Boolean Expressions*** **and** ***Simplify Them*****Lesson Ojectives**Learn how to derive a Boolean expression of a function defined by its truth table. The derived expressions may be in one of two possible standard forms: *The Sum of Min-terms* or the *Product of Max-Terms*.

2. Learn how to map these expressions into logic circuit implementations (2-Level Implementations).

MinTerms

- Consider a system of 3 input signals (variables) x, y, & z.
- A term which ANDs all input variables, either in the true or complement form, is called a minterm.
- Thus, the considered 3-input system has 8 minterms, namely:

$$\bar{x}\bar{y}\bar{z}, \bar{x}\bar{y}z, \bar{x}y\bar{z}, \bar{x}yz, x\bar{y}\bar{z}, x\bar{y}z, xy\bar{z} \;\&\; xyz$$

- Each minterm equals 1 at exactly one particular input combination and is equal to 0 at all other combinations
- Thus, for example, $\bar{x}\bar{y}\bar{z}$ is always equal to 0 except for the input combination ***xyz*** = **000**, where it is equal to 1.
- Accordingly, the minterm $\bar{x}\bar{y}\bar{z}$ is referred to as m_0.
- In general, minterms are designated m_i, where i corresponds the input combination at which this minterm is equal to 1.

- For the 3-input system under consideration, the number of possible input combinations is 2^3, or 8. This means that the system has a total of 8 minterms as follows:
  - $m_0 = \bar{x}\,\bar{y}\,\bar{z} = 1$ IFF $\boldsymbol{xyz} = \mathbf{000}$, *otherwise it equals 0*
  - $m_1 = \bar{x}\,\bar{y}z = 1$ IFF $\boldsymbol{xyz} = \mathbf{001}$, *otherwise it equals 0*
  - $m_2 = \bar{x}y\bar{z} = 1$ IFF $\boldsymbol{xyz} = \mathbf{010}$, *otherwise it equals 0*
  - $m_3 = \bar{x}yz = 1$ IFF $\boldsymbol{xyz} = \mathbf{011}$, *otherwise it equals 0*
  - $m_4 = x\bar{y}\bar{z} = 1$ IFF $\boldsymbol{xyz} = \mathbf{100}$, *otherwise it equals 0*
  - $m_5 = x\bar{y}z = 1$ IFF $\boldsymbol{xyz} = \mathbf{101}$, *otherwise it equals 0*
  - $m_6 = xy\bar{z} = 1$ IFF $\boldsymbol{xyz} = \mathbf{110}$, *otherwise it equals 0*
  - $m_7 = xyz = 1$ IFF $\boldsymbol{xyz} = \mathbf{111}$, *otherwise it equals 0*

In general,

- For *n*-input variables, the number of minterms = the total number of possible input combinations = 2^n.
- A minterm = 0 at all input combinations except one where the minterm = 1.

MaxTerms

- Consider a circuit of 3 input signals (variables) x, y, & z.
- A term which ORs all input variables, either in the true or complement form, is called a Maxterm.
- With 3-input variables, the system under consideration has a total of 8 Maxterms, namely:

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

- Each Maxterm equals 0 at exactly one of the 8 possible input combinations and is equal to 1 at all other combinations.
- For example, $(x+y+z)$ equals 1 at all input combinations except for the combination $\boldsymbol{xyz} = \mathbf{000}$, where it is equal to 0.
- Accordingly, the Maxterm $(x+y+z)$ is referred to as M_0.
- In general, Maxterms are designated M_i, where *i* corresponds to the input combination at which this Maxterm is equal to 0.

- For the 3-input system, the number of possible input combinations is 2^3, or 8. This means that the system has a total of 8 Maxterms as follows:
  - $M_0 = (x + y + z) = 0$ IFF $xyz = \mathbf{000}$, *otherwise it equals 1*
  - $M_1 = (x + y + \bar{z}) = 0$ IFF $xyz = \mathbf{001}$, *otherwise it equals 1*
  - $M_2 = (x + \bar{y} + z) = 0$ IFF $xyz = \mathbf{010}$, *otherwise it equals 1*
  - $M_3 = (x + \bar{y} + \bar{z}) = 0$ IFF $xyz = \mathbf{011}$, *otherwise it equals 1*
  - $M_4 = (\bar{x} + y + z) = 0$ IFF $xyz = \mathbf{100}$, *otherwise it equals 1*
  - $M_5 = (\bar{x} + y + \bar{z}) = 0$ IFF $xyz = \mathbf{101}$, *otherwise it equals 1*
  - $M_6 = (\bar{x} + \bar{y} + z) = 0$ IFF $xyz = \mathbf{110}$, *otherwise it equals 1*
  - $M_7 = (\bar{x} + \bar{y} + \bar{z}) = 0$ IFF $xyz = \mathbf{111}$, *otherwise it equals 1*

In general,

- For *n*-input variables, the number of Maxterms = the total number of possible input combinations = 2^n.
- A Maxterm = 1 at all input combinations except one where the Maxterm = 0.

Imprtant Result

Using De-Morgan's theorem, or truth tables, it can be easily shown that:

$$M_i = \overline{m_i} \qquad \forall i = 0, 1, 2, ..., (2^n - 1)$$

Expressing Functions as a *Sum of Minterms* and Product of Maxterms

Example: Consider the function *F* defined by the shown truth table

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

Now let's rewrite the table, with few added columns.

- A column *i* indicating the input combination
- Four columns of minterms m_2, m_4, m_5 and m_7
- One last column **OR-ing** the above minterms ($m_2 + m_4 + m_5 + m_7$)

i	*x*	*y*	*z*	*F*	m_2	m_4	m_5	m_7	$m_2+m_4+m_5+m_7$
0	0	0	0	0	0	0	0	0	0
1	0	0	1	0	0	0	0	0	0
2	0	1	0	1	1	0	0	0	1
3	0	1	1	0	0	0	0	0	0
4	1	0	0	1	0	1	0	0	1
5	1	0	1	1	0	0	1	0	1
6	1	1	0	0	0	0	0	0	0
7	1	1	1	1	0	0	0	1	1

- From this table, we can clearly see that $F = m_2 + m_4 + m_5 + m_7$
- This is logical since F = 1, only at input combinations *i*= 2, 4, 5 and 7
- Thus, by ORing minterm m_2 (which has a value of 1 only at input combination *i*= 2) with minterm m_4 (which has a value of 1 only at input combination *i*= 4) with minterm m_5 (which has a value of 1 only at input combination *i*= 5) with minterm m_7 (which has a value of 1 only at input combination *i*= 7) the resulting function will equal F.
- *In general*, Any function can be expressed by *OR-ing* all minterms (m_i) corresponding to input combinations (*i*) at which the function has a value of 1.
- The resulting expression is commonly referred to as the ***SUM of minterms*** and is typically expressed as **F = ∑(2, 4, 5, 7),** where ∑ indicates OR-ing of the indicated minterms. Thus, **F = ∑(2, 4, 5, 7)** = ($m_2 + m_4 + m_5 + m_7$)

Example:

- Consider the function *F* of the previous example.
- We will, first, derive the sum of minterms expression for the complement function F`.

i	*x*	*y*	*z*	*F*	F`
0	0	0	0	0	1
1	0	0	1	0	1
2	0	1	0	1	0
3	0	1	1	0	1
4	1	0	0	1	0
5	1	0	1	1	0
6	1	1	0	0	1
7	1	1	1	1	0

The truth table of F` shows that F` equals 1 at *i = 0, 1, 3 and 6, then,*

$F` = m_0 + m_1 + m_3 + m_6$, *i.e*

F` = ∑(0, 1, 3, 6), **(1)**

F = ∑(2, 4, 5, 7) **(2)**

- Obviously, the sum of minterms expression of F` contains *all* minterms that do not appear in the sum of minterms expression of F.

Using De-Morgan theorem on equation (2),

$$\overline{F} = \overline{(m_2 + m_4 + m_5 + m_7)} = \overline{m}_2.\overline{m}_4.\overline{m}_5.\overline{m}_7 = M_2.M_4.M_5.M_7$$

This form is designated as the *Product of Maxterms* and is expressed using the $\prod$ symbol, which is used to designate product in regular algebra, but is used to designate AND-ing in Boolean algebra.

Thus,

F` = $\prod$ (2, 4, 5, 7) = M_2. M_4. M_5. M_7 **(3)**

From equations (1) and (3) we get,

F` = $\sum$(0, 1, 3, 6) = $\prod$ (2, 4, 5, 7)

In general, *any function can be expressed both as a sum of minterms and as a product of maxterms*. Consider the derivation of F back from **F`** given in equation (3):

$$F = \overline{\overline{F}} = \overline{m_0 + m_1 + m_3 + m_6} = \overline{m}_0 \cdot \overline{m}_1 \cdot \overline{m}_3 \cdot \overline{m}_6 = M_0 \cdot M_1 \cdot M_3 \cdot M_6$$

F = $\sum$(2, 4, 5, 7) = $\prod$ (0, 1, 3, 6)

F` = $\prod$ (2, 4, 5, 7) = $\sum$ (0, 1, 3, 6)

Conclusions:

- Any function can be expressed both as a sum of minterms ($\sum m_i$) and as a product of maxterms. The product of maxterms expression ($\prod M_j$) expression of F contains *all* maxterms M_j ($\forall j \neq i$) that do not appear in the sum of minterms expression of F.
- The sum of minterms expression of F` contains *all* minterms that do not appear in the sum of minterms expression of F.
- This is true for all complementary functions. Thus, each of the 2^n minterms will appear either in the sum of minterms expression of F or the sum of minterms expression of $\overline{F}$ but not both.
- The product of maxterms expression of F` contains *all* maxterms that do not appear in the product of maxterms expression of F.
- This is true for all complementary functions. Thus, each of the 2^n maxterms will appear either in the product of maxterms expression of F or the product of maxterms expression of $\overline{F}$ but not both.

Example:

Given that **F (a, b, c, d) = ∑(0, 1, 2, 4, 5, 7)**, derive the product of maxterms expression of **F** and the 2 standard form expressions of **F`**.

Since the system has 4 input variables (a, b, c & d) → The number of minterms and Maxterms = 2^4= 16

F (a, b, c, d) = ∑(0, 1, 2, 4, 5, 7)

1. List all maxterms in the Product of maxterms expression

F = ∏ (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15).

F = ∏ (~~0, 1, 2,~~ 3, ~~4, 5,~~ 6, ~~7,~~ 8, 9, 10, 11, 12, 13, 14, 15).

2. Cross out maxterms corresponding to input combinations of the minterms appearing in the sum of minterms expression

F = ∏ (3, 6, 8, 9, 10, 11, 12, 13, 14, 15).

Similarly, obtain both canonical form expressions for F`

F` = ∑ (3, 6, 8, 9, 10, 11, 12, 13, 14, 15).

F` = ∏ (0, 1, 2, 4, 5, 7)

Canonical Forms:

The sum of minterms and the product of maxterms forms of Boolean expressions are known as the canonical forms (الصيغ القانونية) of a function.

Standard Forms:

- A product term is a term with ANDed literals*. Thus, AB, A'B, A'CD are all product terms.
- A minterm is a special case of a product term where all input variables appear in the product term either in the true or complement form.
- A sum term is a term with ORed literals*. Thus, (A+B), (A'+B), (A'+C+D) are all sum terms.
- A maxterm is a special case of a sum term where all input variables, either in the true or complement form, are ORed together.
- Boolean functions can generally be expressed in the form of a Sum of Products (SOP) or in the form of a Product of Sums (POS).
- The sum of minterms form is a special case of the SOP form where all product terms are minterms.
- The product of maxterms form is a special case of the POS form where all sum terms are maxterms.
- The SOP and POS forms are Standard forms for representing Boolean functions.

* *A Boolean variable in the true or complement forms*

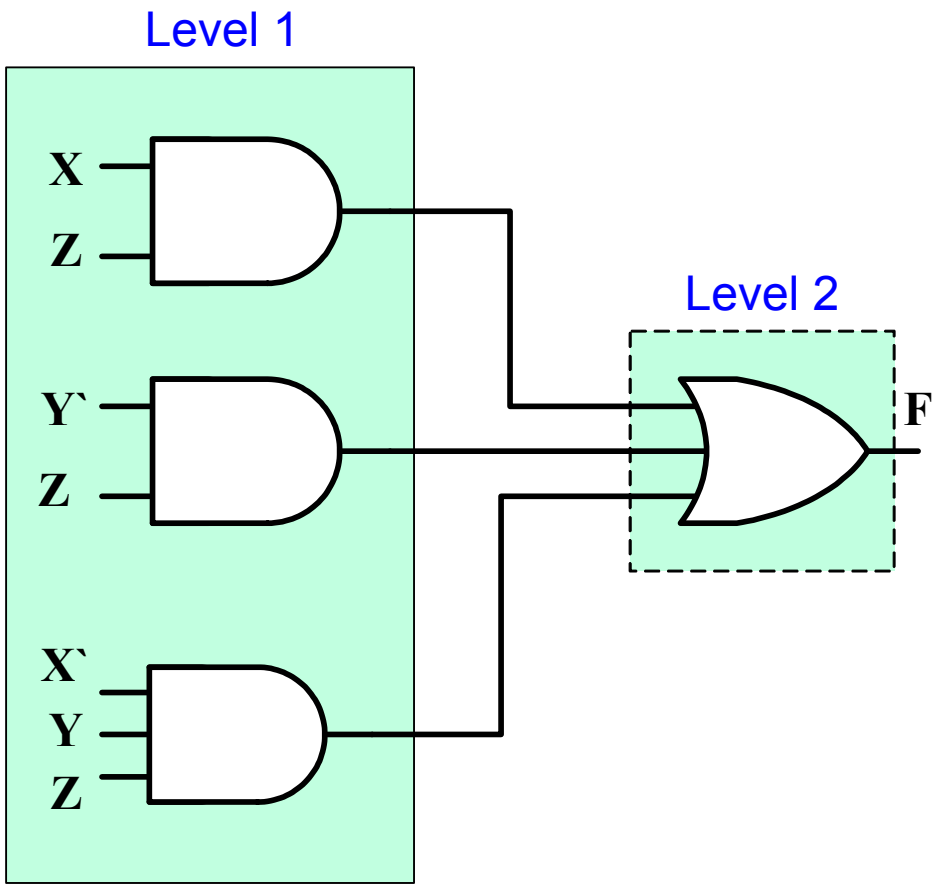
Two-Level Implementations of Standard Forms

Sum of Products Expression (SOP):

- Any SOP expression can be implemented in 2-levels of gates.
- The first level consists of a number of AND gates which equals the number of product terms in the expression. Each AND gate implements one of the product terms in the expression.
- The second level consists of a *SINGLE* OR gate whose number of inputs equals the number of product terms in the expression.

Example Implement the following SOP function

F = XZ + Y`Z + X`YZ



Two-Level Implementation (F = XZ + Y`Z + X`YZ)

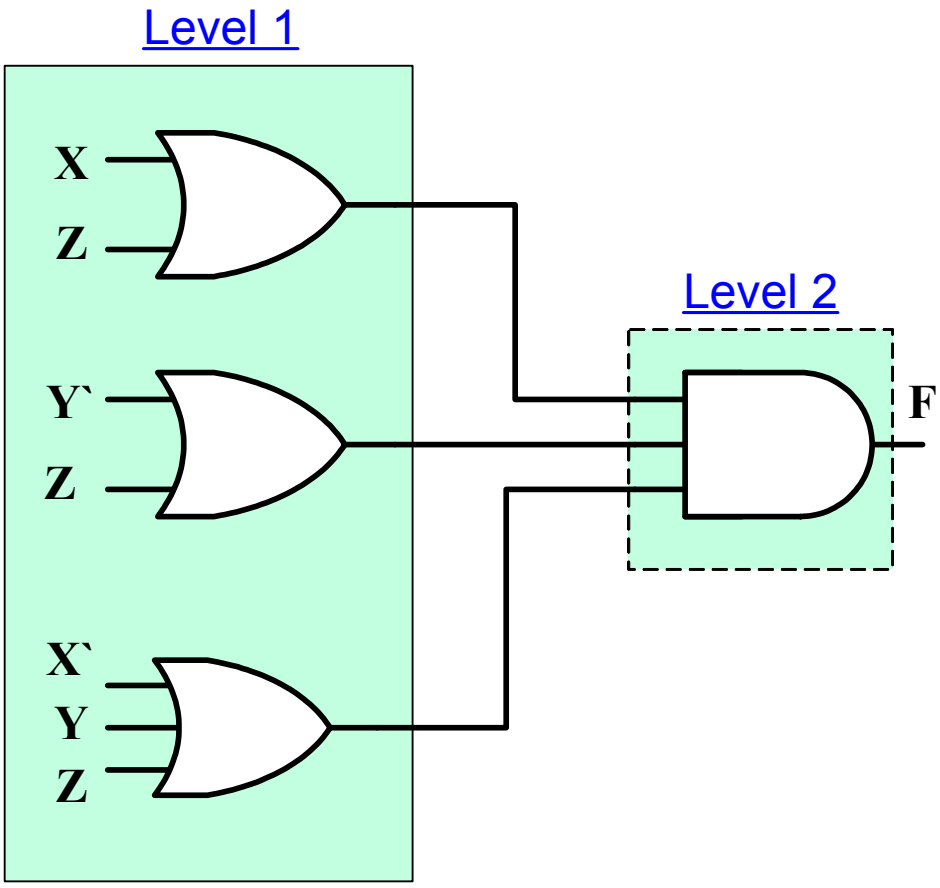
Level-1: AND-Gates ; **Level-2**: One OR-Gate

Product of Sums Expression (POS):

- Any POS expression can be implemented in 2-levels of gates
- The first level consists of a number of OR gates which equals the number of sum terms in the expression, each gate implements one of the sum terms in the expression.
- The second level consists of a *SINGLE* AND gate whose number of inputs equals the number of sum terms.

Example Implement the following SOP function

F = (X+Z)(Y`+Z)(X`+Y+Z)



Two-Level Implementation {F = (X+Z)(Y`+Z)(X`+Y+Z)}

Level-1: OR-Gates ; **Level-2**: One AND-Gate

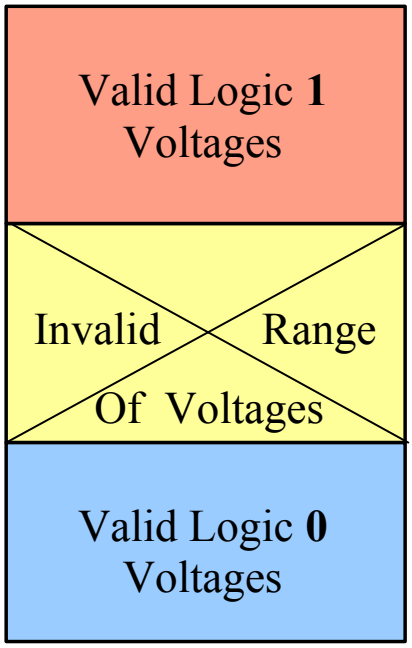
Practical Aspects Of Logic Gates

Introduction & Objectives

- Logic gates are physically implemented as Integrated Circuits (**IC**).
- Integrated circuits are implemented in several technologies.
- Two landmark IC technologies are the **TTL** and the **CMOS** technologies.
- Major physical properties of a digital IC depend on the implementation technology.
- In this lesson, the following major properties of digital IC's are described:
    1. Allowed physical range of voltages for logic 0 and logic 1,
    2. Gate propagation delay/ speed,
    3. The fanin and fanout of a gate,
    4. The use of buffers, and
    5. Tri-State drivers

Allowed Voltage Levels

- Practically, logic 0 is represented by a **certain** *RANGE* of Voltages rather than by a single voltage level.
- .In other words, if the voltage level of a signal falls in this range, the signal has a logic 0 value.
- Likewise, logic 1 is represented by a different *RANGE* of **valid** voltages.
- The range of voltages between the highest logic 0 voltage level and the lowest logic 1 voltage level is an "*Illegal Voltage Range*".



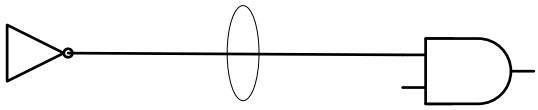
- **No signal is allowed to assume a voltage value in this range.**

Input & Output Voltage Ranges

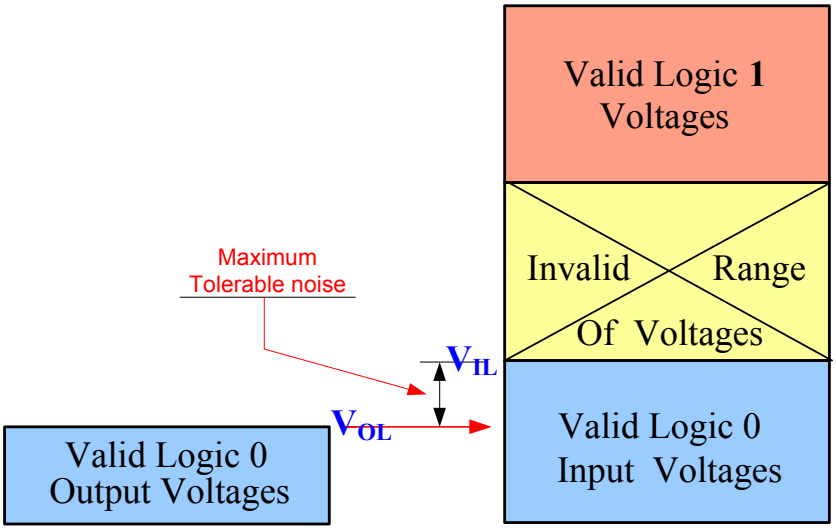
- Inputs and outputs of IC's *do not have the same allowed range of voltages* neither for logic 0 nor for logic 1.
- V_{IL} is the maximum *input* voltage that considered a *Logic 0*.
- V_{OL} is the maximum *output* voltage that considered a *Logic 0*.
- V_{OL} must be lower than V_{IL} to guard against noise disturbance.

Why is $V_{IL} > V_{OL}$?

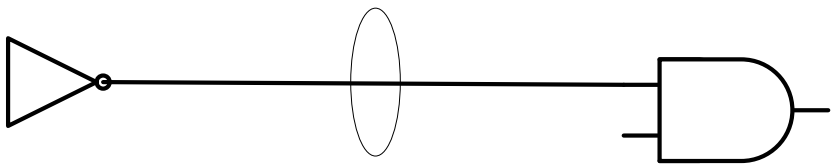
- Consider the case of connecting the output of gate **A** to the input of another gate **B**:



  - The logic 0 output of **A** must be within the range of acceptable logic 0 voltages of gate **B** inputs.
  - Voltage level at the input of **B** = Voltage level at the output of **A** + Noise Voltage
  - If the highest logic 0 output voltage of **A** (V_{OL}) is equal to the highest logic 0 input voltage of **B** (V_{IL}), then the noise signal can cause the actual voltage at the input of **B** to fall in the *invalid range* of voltages.

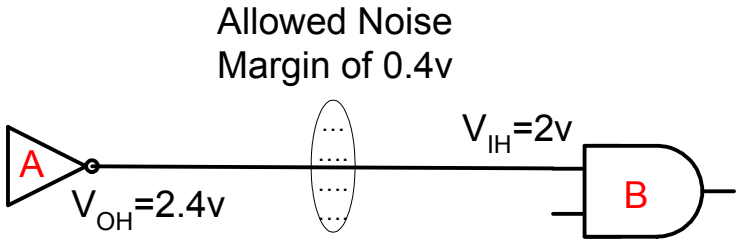


- Accordingly, V_{OL} is designed to be lower than V_{IL} to allow for some noise margin.

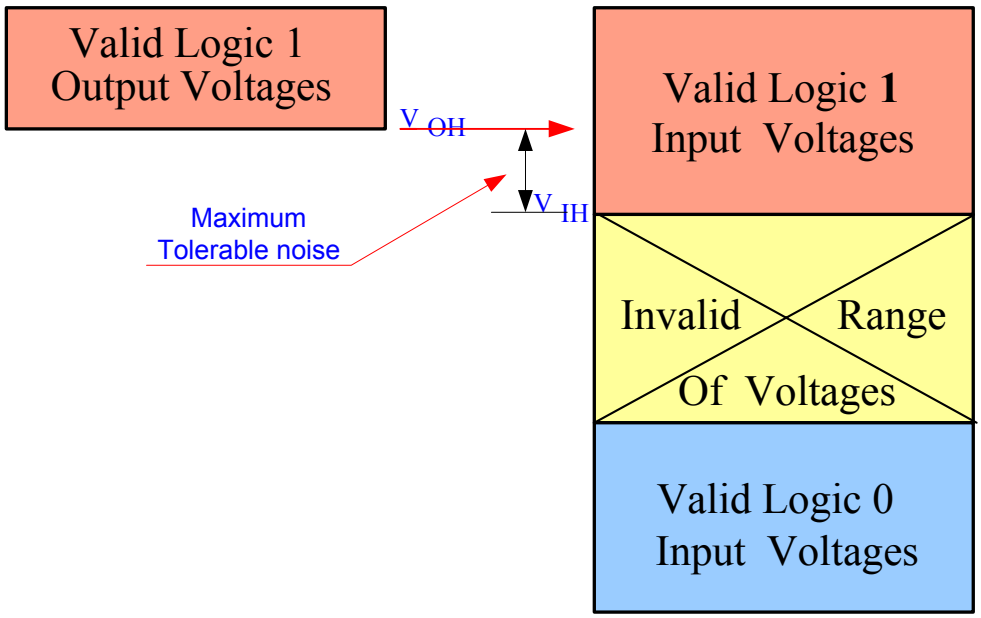


- The difference (V_{IL} - V_{OL}) is thus known as the noise margin for logic 0 (NM_0).
- V_{IH} is the minimum *input* voltage that considered a *Logic 1*.
- V_{OH} is the minimum *output* voltage that considered a *Logic 1*.
- V_{OH} must be higher than V_{IH} to guard against noise signals.

Why is $V_{OH} > V_{IH}$?



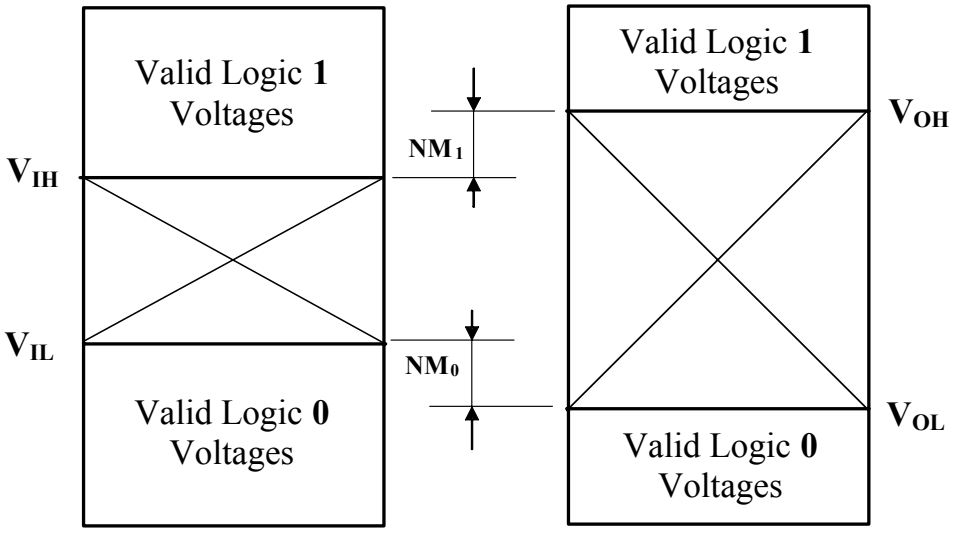
- Consider the case of connecting the output of gate **A** to the input of another gate **B**:
  - The logic 1 output of **A** must accepted as logic 1 by the input of gate **B.**
  - Thus, the logic 1 output of **A** must be *within* the range of voltages which are acceptable as logic 1 input for gate **B**.
  - If the lowest logic 1 output voltage of **A** (V_{OH}) is equal to the lowest logic 1 input voltage of **B** (V_{IH}), then noise signals can cause the actual voltage at the input of **B** to fall in the *invalid range* of input voltages.



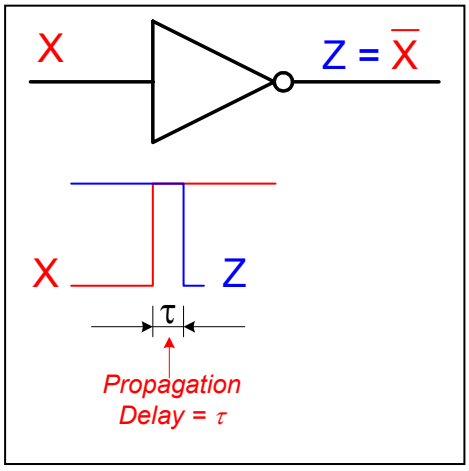
- Accordingly, V_{OH} is designed to be higher than V_{IH} to allow for some noise margin.
- The difference (V_{OH} - V_{IH}) is thus known as the noise margin for logic 1 (NM_1).

Definition

- *Noise margin* is the maximum *noise voltage that can be* added to the input signal of a digital circuit without causing an undesirable change in the circuit output..



Propagation Delay



Consider the shown inverter with input X and output Z.

- A change in the input (X) from 0 to 1 causes the inverter output (Z) to change from 1 to 0.
- The change in the output (Z), however is *not instantaneous*. Rather, it occurs slightly after the input change.
- This *delay* between an input signal change and the corresponding output signal change is what is known as the *propagation delay*.

In general,

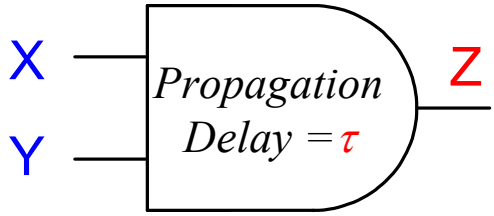
- A signal change on the input of some IC takes a finite amount of time to cause a corresponding change on the output.
- This finite delay time is known as **Propagation Delay**.
- Faster circuits are characterized by smaller propagation delays.
- Higher performance systems require higher speeds (smaller propagation delays).

Timing Diagrams

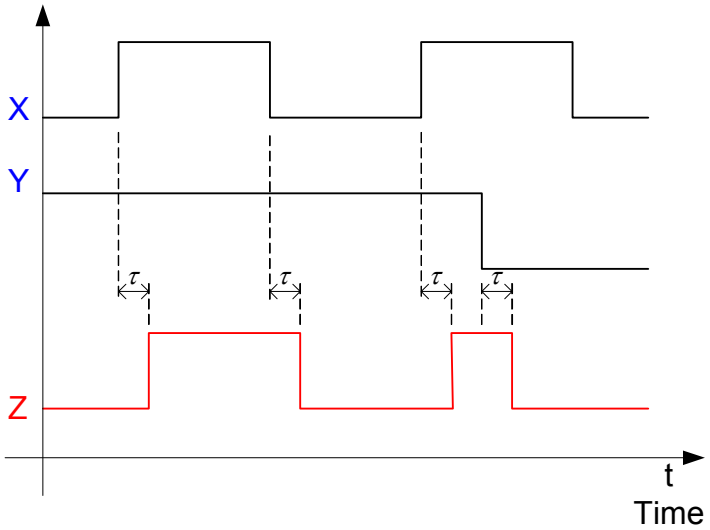
- A timing diagram shows the logic values of signals in a circuit versus time.
- A signal shape versus time is typically referred to as *Waveform*.

Example

The figure shows the timing diagram of a 2-input AND gate. The gate is assumed to have a propagation delay of τ.



- The timing diagram shown Figure illustrates the waveforms of signals X, Y, and Z.
- Note how the output **Z** is delayed from changes of the input signals **X** & **Y** by the amount of the gate **Propagation Delay** τ.

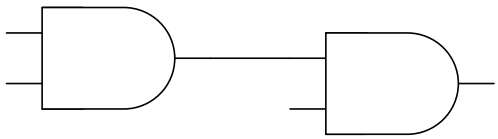


Fanin Limitations

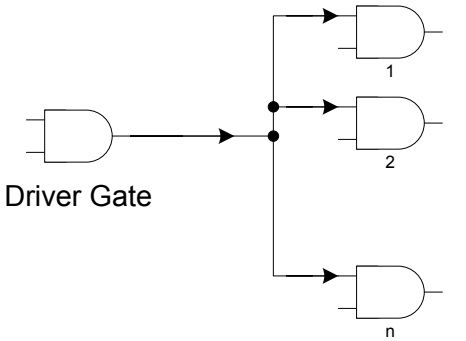
- The *fanin of a* gate is the number of inputs of this gate.
- Thus, a 4-input AND gate is said to have a *fanin* of 4.
- A physical gate cannot have a large number of inputs (*fanin*).

- •For CMOS technology, the more inputs a gate has the slower it is (larger propagation delay). For example, a 4-input AND gate is slower than a 2-input one.
- In CMOS technology, no more than 4-input gates are typically built since more than 4 inputs makes the devices too slow.
- TTL gates can have more inputs (e.g, 8 input NAND 7430).

Fanout Limitations



- If the output of some gate **A** is connected to the input of another gate **B**, gate **A** is said to be *driving* gate, while gate **B** is said to be the *load* gate.
- As the Figure shows, a driver gate may have more than one load gate.
- There is a limit to the number of gate inputs that a single output can drive.
- The *fanout of a* gate is the largest number of gate inputs this gate can drive.



- For TTL, the *fanout* limit is based on CURRENT.
  - A TTL *output* can supply a maximum current $\mathbf{I}_{OL}$ = 16 mA (milliamps)
  - A TTL *input* requires a current of $\mathbf{I}_{IL}$ =1.6mA.
  - Thus, the *fanout* for TTL is 16mA/1.6 mA = 10 loads.
- For CMOS, the limit is based on SPEED/propagation delay.
  - A CMOS input resembles a capacitive load (≈10 pf - picofarads).
  - The more inputs tied to a single output, the higher the capacitive load.
  - The HIGHER the capacitive load, the SLOWER the propagation delay.

- Typically, it is advisable to avoid loads much higher than about 8 loads.

Q. What is meant by the *DRIVE* of a gate?

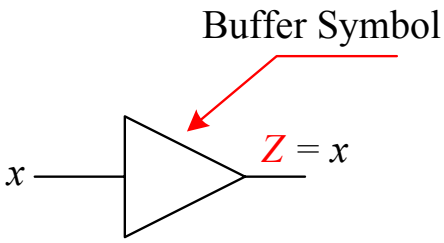
A. It is the "*CURRENT*" driving-ability of a gate. In other words, it is the amount of current the gate can deliver to its load devices.

- A gate with *high-drive* is capable of driving more load gates than another with *low-drive*.

Q. How to drive a number of load gates that is larger than the fanout of the driver gate?

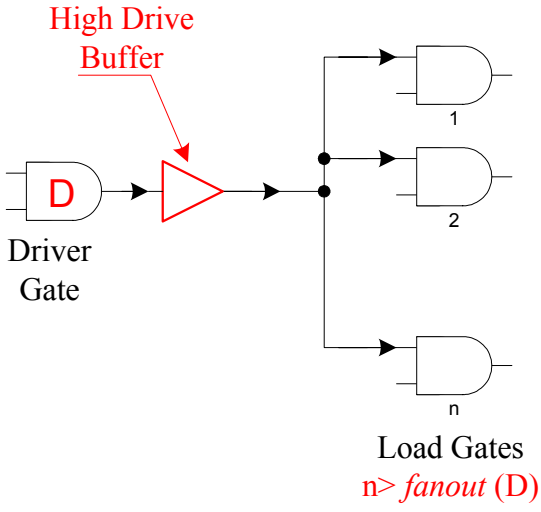
A. In this case, we can use one of two methods:

1. Use high drive buffers
2. Use multiple drivers.



Use of High-Drive Buffers:

- A buffer is a single input, single output gate where the logic value of the output equals that of the input.
- The logic symbol of the buffer is shown in the Figure.
- The buffer provides the necessary drive capability which allows driving larger loads.
- Note that the symbol of the buffer resembles the inverter symbol except that it does not have the inverting circle that the inverter symbol has.
- The figure shows how the buffer is used to drive the large load.

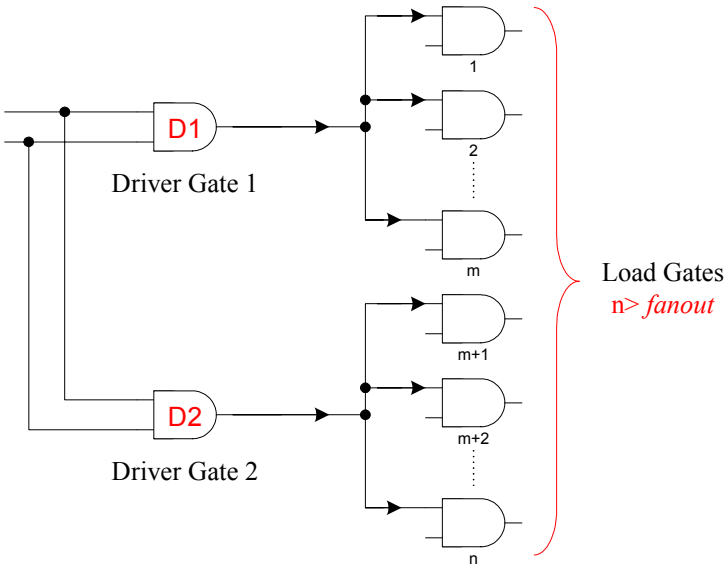


Use of Multiple Drivers:

- The Figure shows the case of 2 identical drivers driving the load gates.
- In general, the large number of load

gates is divided among more than one driver such that each of the identical drivers is driving no more than the fanout.

- The multiple driver gates (D1, D2) are of identical type and should be connected to the same input signals



Tri-State Outputs

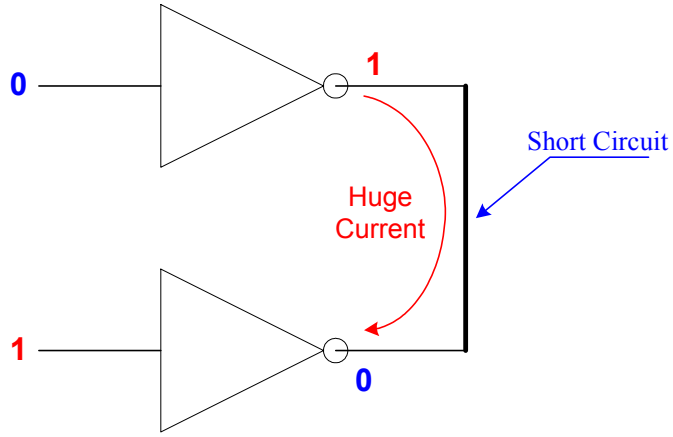
Q. Can the outputs of 2 ICs, or 2 gates, be directly connected?

A. Generally, Noooooooooo!!! This is only possible if special types of gates are used.

Q. Why cann't the outputs of 2 normal gates be directly connected?

A. Because this causes a ***short Circuit*** that results in huge current flow with a subsequent potential for damaging the circuit.

- This is obvious since one output may be at logic 1 (High voltage), while the other output may be at logic 0 (Low voltage).
- Furthermore, the common voltage level of the shorted outputs will most likely fall in the invalid range of voltage levels.



Q. What are the types of IC output pins that can be directly connected?

A. These are pins/gates with special output drivers. The two main types are:

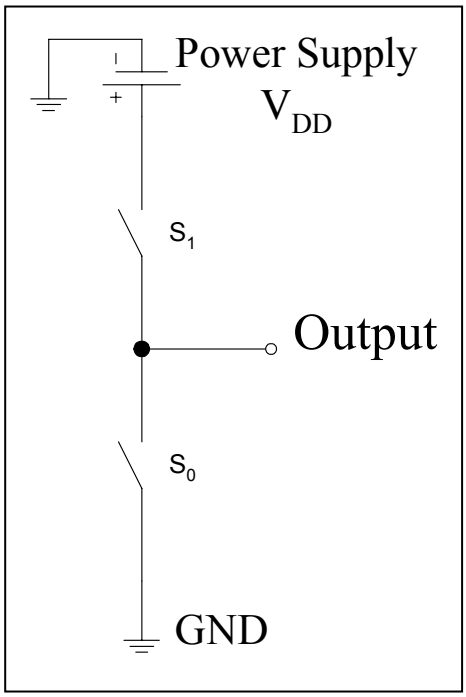
- Open-Collector outputs → this will not be discussed in this course.
- Outputs with Tri-State capability.

Gates with Tri-State Outputs

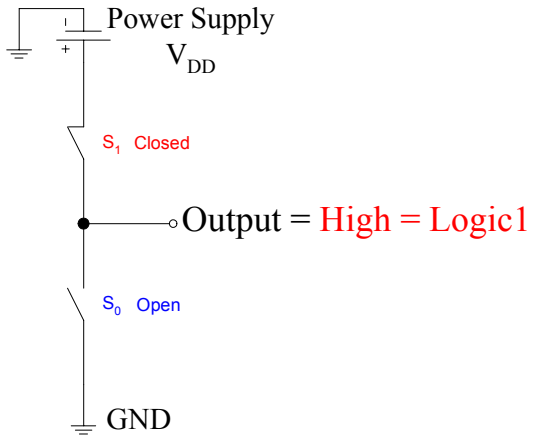
- These gates can be in one of 2 possible states:
  1. An enabled state where the output may assume one of two possible values:
     - Logic 0 value (low voltage)
     - Logic 1 value (high voltage)
  2. A disabled state where the gate output is in a the Hi-impedance (Hi-Z) state. In this case, the gate output is disconnected (open-circuit) from the wire it is driving.
- An enable input (E) is used to control the gate into either the enabled or disabled state.
- The enable input (E) may be either active high or active low.
- Any gate or IC output may be provided with tri-state capability.

Output State Illustrations

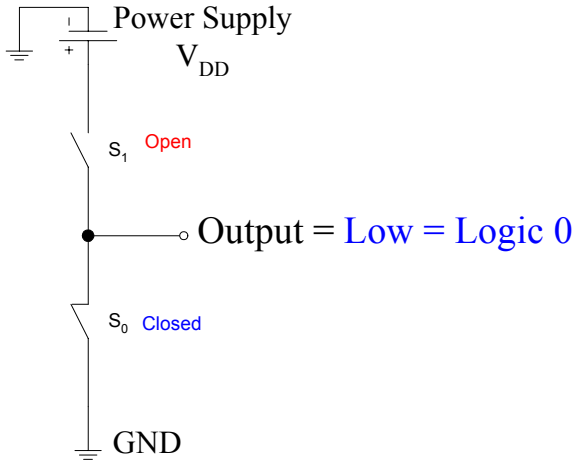
- A generalized output driver can be simply modeled using 2 switches S_1 and S_0 as shown in Figure.
- The output state is defined by the state of the 2 switches (closed -open)
- If S_1 is closed and S_0 is open, the output is high (logic 1) since it is connected to the power supply (V_{DD}).



- If S_1 is open and S_0 is closed, the output is low (logic 0) since it is connected to the ground voltage (0 volt).

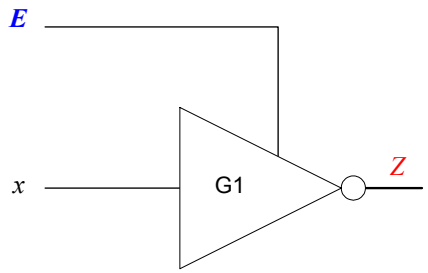


- If, however, both S_1 is and S_0 are open, then the output is neither connected to ground nor to the power supply. In this case, the output node is floating or is in the Hi-Impedance (Hi-Z) state.

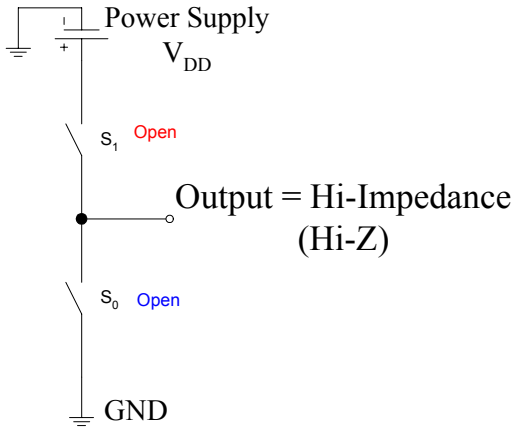


Examples

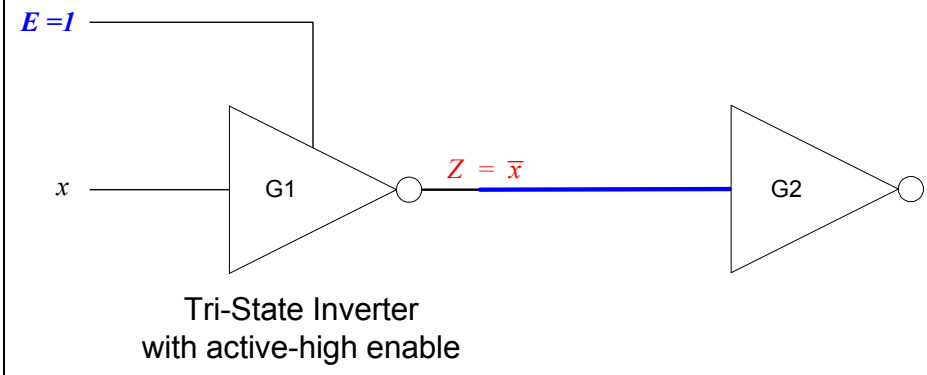
a) Tri-State Inverter with active high enable

Logic Symbol	 Tri-State Inverter with active-high enable
Truth Table	

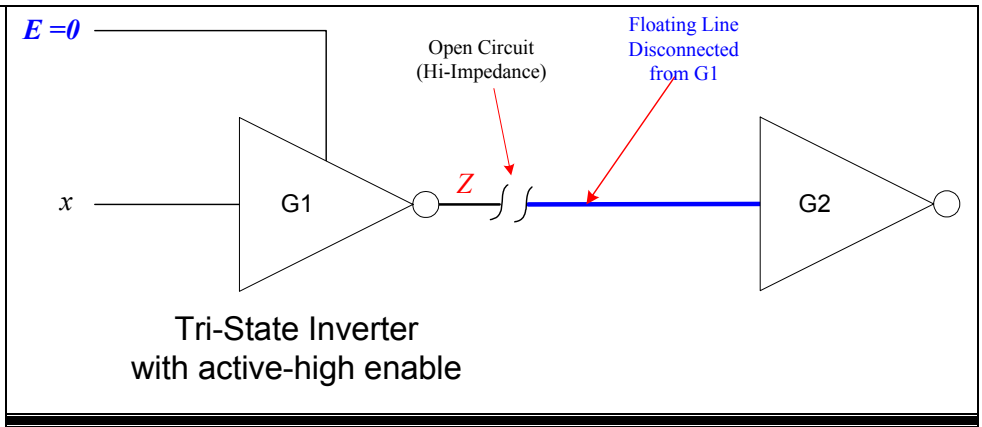
E	x	Z
1	0	1
1	1	0
0	0	Hi-Z
0	1	Hi-Z

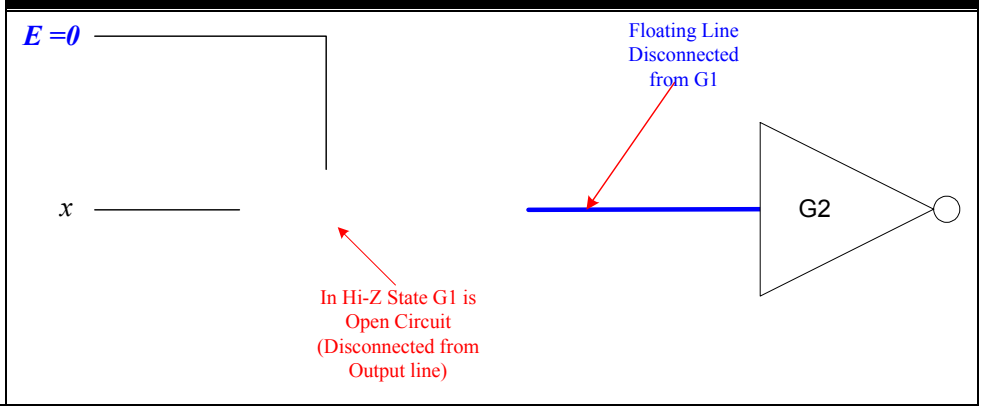


Enabled State



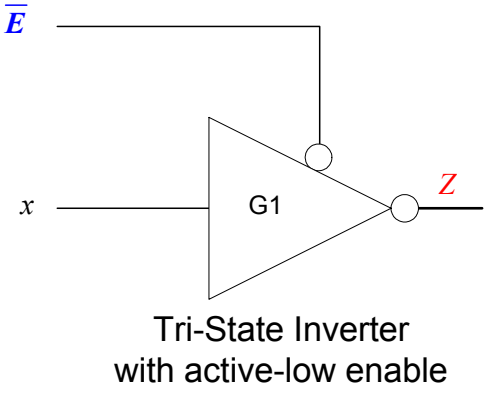
Dis-abled State





b) Tri-State Inverter with active low enable

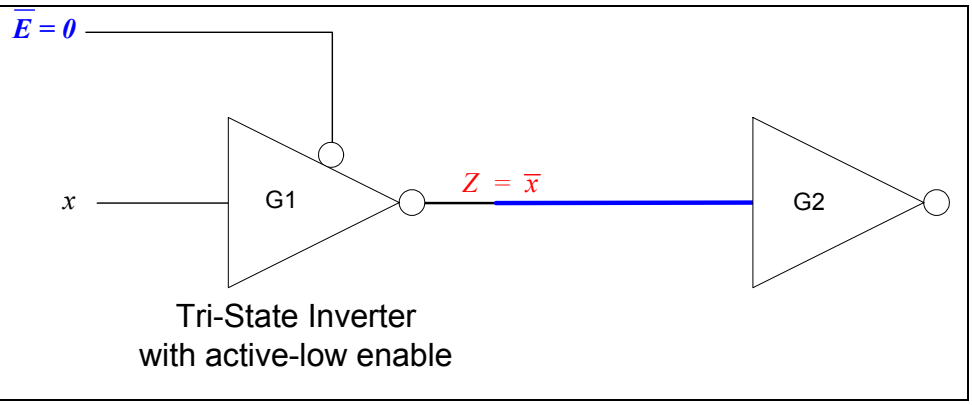
Logic Symbol



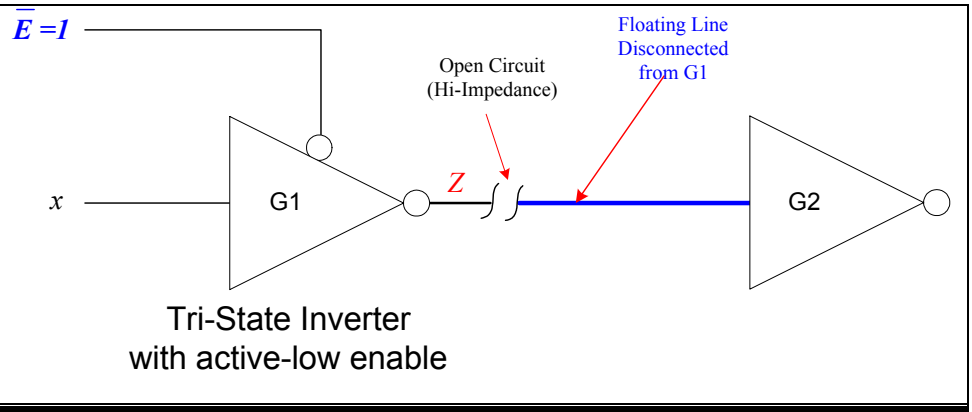
Truth Table

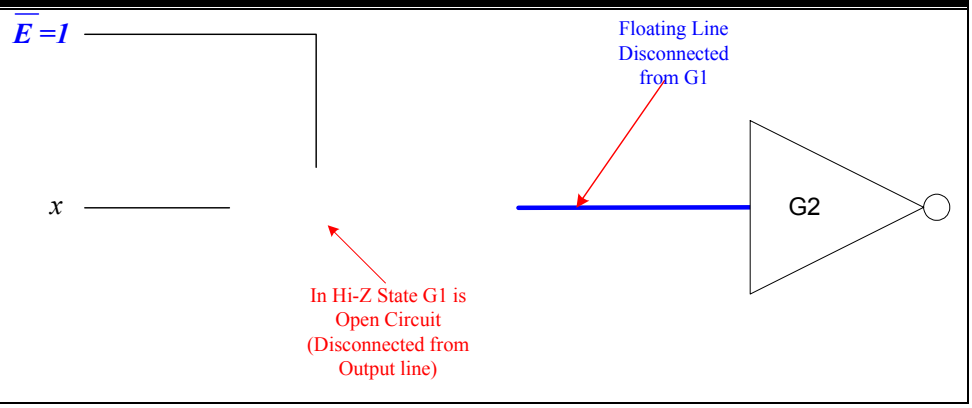
$\overline{E}$	x	Z
0	0	1
0	1	0
1	0	Hi-Z
1	1	Hi-Z

Enabled State



Dis-abled State

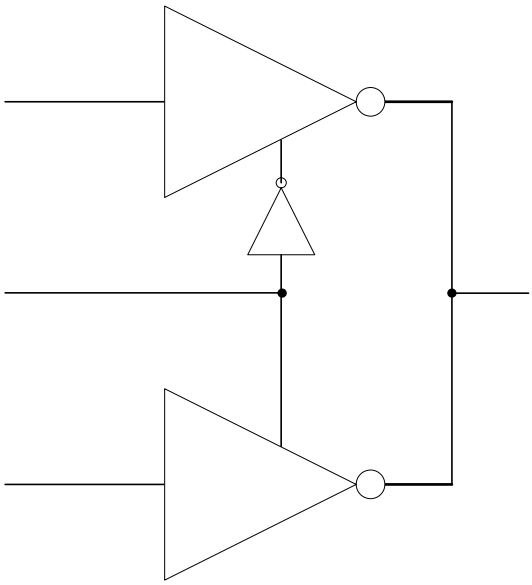




Condition for Connecting Outputs of Tri-State Gates

- Two or more tri-state *outputs* may be connected provided that *at most one* of these outputs is *enabled* while *all others are in the Hi-Z state*.
- This avoids conflict situations where one gate output is high while another is low.

Circuit Examples



- The shown circuit has tri-state inverters with active high enable inputs.
- The outputs of these 2 inverters are shorted together as a common output signal Z
- The 2 gates are NEVER enabled at the same time.
- G1 is enabled when E=1, while G2 is enabled when E=0
- The circuit performs the function: $Z = E\,\overline{x} + \overline{E}\,\overline{y}$

K-Map 1

Lesson Objectives:

Even though **Boolean expressions** can be simplified by algebraic manipulation, such an approach lacks clear regular rules for each succeeding step and it is difficult to determine whether the ***simplest expression*** has been achieved.

In contrast, Karnaugh map (K-map) method provides a straightforward procedure for simplifying Boolean functions.

K-maps of up to 4 variables are very common to use. Maps of 5 and 6 variables can be made as well, but are more cumbersome to use.

Simplified expressions produced by K-maps are always either in the **SOP** or the **POS** form.

The map provides the same information contained in a Truth Table but in a different format.

The objectives of this lesson are to learn:

1. How to build a 2, 3, or 4 variables K-map.
2. How to obtain a minimized SOP function using K-maps.

Code Distance:

Let's first define the concept of Code Distance. The distance between two binary code-words is the number of bit positions in which the two code-words have different values.

For example, the distance between the code words **1001** and **0001** is **1** while the distance between the code-words **0011** and **0100** is **3**.

This definition of code distance is commonly known as the ***Hamming distance*** between two codes.

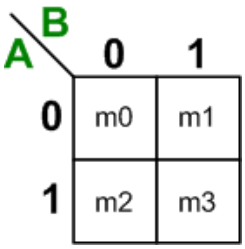
Two-Variable K-Maps:

The 2-variable map is a table of 2 rows by 2 columns. The 2 rows represent the two values of the first input variable A, while the two columns represent the two values of the second input variable B.

Thus, all entries (squares) in the first row correspond to input variable A=0, while entries (squares) of the second row correspond to A=1.

Likewise, all entries of the first column correspond to input variable B = 0, while entries of the second column correspond to B=1.

Thus, each map entry (or square) corresponds to a unique value for the input variables A and B.



For example, the top left square corresponds to input combination AB=00. In other words, this square represents minterm m_0.

Likewise, the top right square corresponds to input combination AB=01, or minterm m_1 and the bottom left square corresponds to input combination AB=10, or minterm m_2. Finally, the bottom right square corresponds to input combination AB=11, or minterm m_3.

In general, each map entry (or square) corresponds to a particular input combination (or minterm).

Since, Boolean functions of two-variables have four minterms, a 2-variable K-map can represent any 2-variable function by plugging the function value for each input combination in the corresponding square.

Definitions/Notations:

Two K-map squares are considered ***adjacent*** if the input codes they represent have a *Hamming distance of 1*.

A K-map square with a function value *of 1* will be referred to as a ***1-Square***.
A K-map square with a function value *of 0* will be referred to as a ***0-Square***.

The simplification procedure is summarized below:

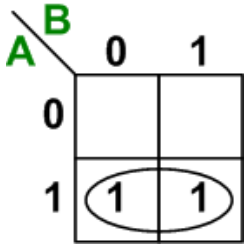
Step 1: Draw the map according to the number of input variables of the function.
Step 2: Fill "1's" in the squares for which the function is true.
Step 3: Form as big group of ***adjacent 1-squares*** as possible. There are some rules for this which you will learn with bigger maps.
Step 4: Find the common literals for each group and write the simplified expression in SOP.

Example:

Consider the given truth table of two variable function. Obtain the simplified function using K-map.

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

First draw a 2-variable K-map. The function F is true when AB' (m_2) is true and when AB (m_3) is true, so a 1 is placed inside the square that belongs to m_2 and a 1 is placed inside the square that belongs to m_3.



Since both of the 1-squares have different values for variable B but the same value for variable A, which is 1, i.e., wherever A = 1 then F = 1 thus F = A.

This simplification is justified by algebraic manipulation as follows:
$F = m_2 + m_3 = AB' + AB = A\,(B' + B) = A$

To understand how combining squares simplifies Boolean functions, the basic property possessed by the **adjacent squares** must be recognized.

In the above example, the two 1-squares are adjacent with the same value for variable A (A=1) but different values for variable **B** (one square has B=0, while the other has B=1).

This reduction is possible since both squares are adjacent and the net expression is that of the common variable (A).

Generally, this is true for any 2 codes of Hamming distance 1 (adjacent). For an *n*-variable K-map, let the codes of two ***adjacent squares*** (distance of 1) have the same value for all variables except the i^{th} variable. Thus,

Code of 1st Square: $X_1.X_2.....X_{i-1}.\boldsymbol{X_i}.X_{i+1}.....X_n$

Code of 2nd Square: $X_1.X_2.....X_{i-1}.\overline{\boldsymbol{X_i}}.X_{i+1}.....X_n$

Combining these two squares in a group will eliminate the different variable $\boldsymbol{X_i}$ and the combined expression will be

$X_1.X_2.....X_{i-1}.X_{i+1}.....X_n$

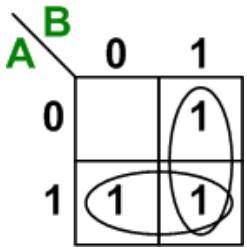
since:

$$\left(X_1.X_2.....X_{i-1}.\boldsymbol{X_i}.X_{i+1}.....X_n\right)+\left(X_1.X_2.....X_{i-1}.\overline{\boldsymbol{X_i}}.X_{i+1}.....X_n\right)$$
$$=\left(X_1.X_2.....X_{i-1}.X_{i+1}.....X_n\right)\left(\boldsymbol{X_i}+\overline{\boldsymbol{X_i}}\right)$$
$$=\left(X_1.X_2.....X_{i-1}.X_{i+1}.....X_n\right)$$

The variable in difference is dropped.

Another Example:

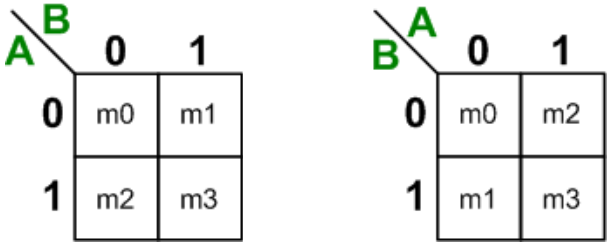
Simplify the given function using K-map method:
$F = \sum (1, 2, 3)$



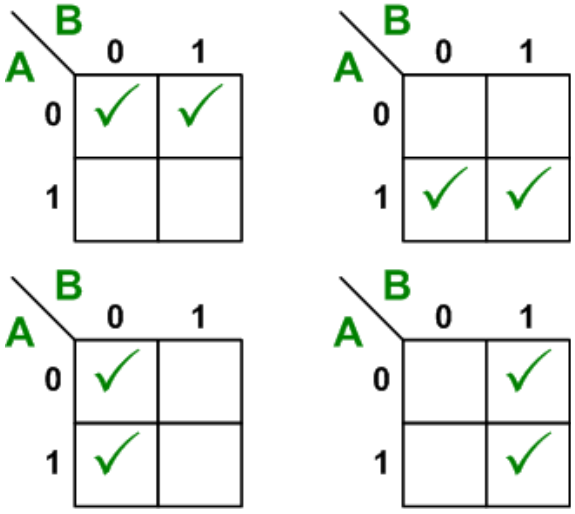
In this example:
$F = m_1 + m_2 + m_3 = m_1 + m_2 + (m_3 + m_3)$
$F = (m_1 + m_3) + (m_2 + m_3) = A + B$

Rule: A 1-square can be member of more than one group.

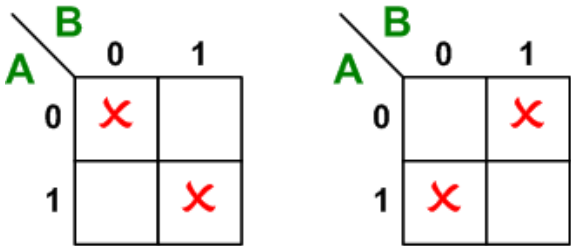
If we exchange the places of **A** and **B**, then minterm positions will also change. Thus, **m_1** and **m_2** will be exchanged as well.



In an *n*-variable map each square is ***adjacent*** to "***n***" other squares, e.g., in a 2-variable map each square is adjacent to two other squares as shown below:



Examples of non-adjacent squares are shown below:



Three-Variable K-Maps:

There are eight minterms for a Boolean function with three-variables. Hence, a **three-variable** map consists of **8 squares**.

A \ BC	00	01	11	10
0	m0	m1	m3	m2
1	m4	m5	m7	m6

All entries (squares) in the first row correspond to input variable A=0, while entries (squares) of the second row correspond to A=1.

Likewise, all entries of the first column correspond to input variable B = 0, C = 0, all entries of the second column correspond to input variable B = 0, C = 1, all entries of the third column correspond to input variable B = 1, C = 1, while entries of the fourth column correspond to B=1, C = 0.

To maintain adjacent columns physically adjacent on the map, the column coordinates do not follow the binary count sequence. This choice yields unit distance between codes of one column to the next (00 – 01—11 – 10), like **Grey Code**.

Variations of Three-Variable Map:

The figure shows variations of the three-variable map. Note that the minterm corresponding to each square can be obtained by substituting the values of variables **ABC** in order.

C \ AB	00	01	11	10
0	m0	m2	m6	m4
1	m1	m3	m7	m5

B \ AC	00	01	11	10
0	m0	m1	m5	m4
1	m2	m3	m7	m6

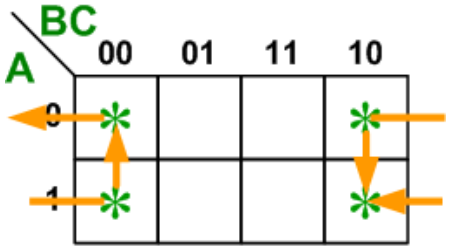
AB \ C	0	1
00	m0	m1
01	m2	m3
11	m6	m7
10	m4	m5

BC \ A	0	1
00	m0	m4
01	m1	m5
11	m3	m7
10	m2	m6

Examples: **(see authorware version)**

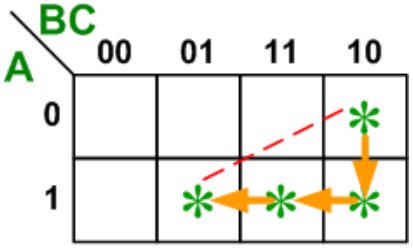
There are cases where two squares in the map are considered to be adjacent even though they do not physically touch each other.

In the figure of 3-variable map, m_0 is adjacent to m_2 and m_4 is adjacent to m_6 because the minterms differ by only one variable. This can be verified algebraically:
$m_0 + m_2 = A'B'C' + A'BC' = A'C' (B' + B) = A'C'$
$m_4 + m_6 = AB'C' + ABC' = AC' (B' + B) = AC'$



Rule: Groups may only consist of **2**, **4**, **8, 16,**… squares (always power of 2). For example, groups may not consist of 3, 6 or 12 squares.

Rule: Members of a group must have a closed loop adjacency, i.e., **L-Shaped 4** squares do not form a valid group.



Notes:
1. Each square is adjacent to 3 other squares.
2. One square is represented by a minterm (i.e. a product term containing all 3 literals).
3. A group of 2 adjacent squares is represented by a product term containing only 2 literals, i.e., 1 literal is dropped.
4. A group of 4 adjacent squares is represented by a product term containing only 1 literal, i.e., 2 literals are dropped.

Four-Variable K-Maps:

There are **16 minterms** for a Boolean function with **four-variables**. Hence, four-variable map consists of 16 squares.

AB \ CD	00	01	11	10
00	m0	m1	m3	m2
01	m4	m5	m7	m6
11	m12	m13	m15	m14
10	m8	m9	m11	m10

ⓘ **Notes:**

1. Each square is **adjacent** to **4** other squares.
2. One square is represented by a minterm (a product of all 4-literals).
3. Combining **2** squares drops **1**-literal.
4. Combining **4** squares drops **2**-literals.
5. Combining **8** squares drops **3**-literals.

Examples: **(see authorware version)**

Rule: The combination of squares that can be chosen during the simplification process in the **n-variable** map are as follows:
A group of 2^n **squares** produces a function that always equal to **logic 1**.
A group of 2^{n-1} **squares** represents a product term of **one literal**.
A group of 2^{n-2} **squares** represents a product term of **two literals** and so on.
One square represents a minterm of ***n*** **literals**.

K-Map 2

Lesson Objectives:

In this lesson you will learn:

1. The difference between prime implicants and essential prime implicants.
2. How to get a minimized POS function using a K-map.
3. How to minimize a combinational circuit that is not completely specified (has don't care conditions).
4. How to make a 5 and 6 variable K-map given a truth table or a SOP representation.

Definitions/Notations:

A product term of a function is said to be an **implicant**.

A **Prime Implicant** (**PI**) is a product term obtained by combining the maximum possible number of adjacent *1-squares* in the map.

If a minterm is covered only by one prime implicant then this prime implicant is said to be an **Essential Prime Implicant** (**EPI**).

Examples: **(see authorware version)**

POS Simplification:

Until now we have derived simplified Boolean functions from the maps in SOP form. Procedure for deriving simplified Boolean functions **POS** is slightly different. Instead of making groups of **1's**, make the groups of **0's**.

Since the simplified expression obtained by making group of 1's of the function (say F) is always in SOP form. Then the simplified function obtained by making group of 0's of the function will be the complement of the function (i.e., F') in SOP form.

Applying DeMorgan's theorem to F' (in SOP) will give F in POS form.

Examples: **(see authorware version)**

Don't Care Conditions:

In some cases, the function is not specified for certain combinations of input variables as 1 or 0.

There are two cases in which it occurs:
1. The input combination never occurs.
2. The input combination occurs but we do not care what the outputs are in response to these inputs.

In both cases, the outputs are called as **unspecified** and the functions having them are called as incompletely specified functions.

In most applications, we simply do not care what value is assumed by the function for unspecified minterms.

Unspecified minterms of a function are called as ***don't care conditions***. They provide further simplification of the function, and they are denoted by X's to distinguish them from 1's and 0's.

In choosing adjacent squares to simplify the function in a map, the don't care minterms can be assumed either 1 or 0, depending on which combination gives the simplest expression.

A don't care minterm need not be chosen at all if it does not contribute to produce a larger implicant.

Five-Variable K-Maps:

There are **32** minterms for a Boolean function with **five-variables**. Hence, Five-variable map consists of 32 squares.

It consists of **2** four-variable maps. Variable **A** distinguishes between the two maps, as indicated on the top of the diagram. The left-hand four-variable map represents the 16 squares where **A=0**, and the other four-variable map represents the squares where **A=1**.

Minterms **0** through **15** belong to the four-variable map with **A=0** and minterms **16** through **31** belong to the four-variable map with **A=1**.

DE / BC	A' 00	A' 01	A' 11	A' 10	A 00	A 01	A 11	A 10
00	m_0	m_1	m_3	m_2	m_{16}	m_{17}	m_{19}	m_{18}
01	m_4	m_5	m_7	m_6	m_{20}	m_{21}	m_{23}	m_{22}
11	m_{12}	m_{13}	m_{15}	m_{14}	m_{28}	m_{29}	m_{31}	m_{30}
10	m_8	m_9	m_{11}	m_{10}	m_{24}	m_{25}	m_{27}	m_{26}

Each four-variable map retains the previously defined adjacency when taken separately. In addition, each square in the A=0 map is adjacent to the corresponding square in the A=1 map. For example, minterm 4 is adjacent to minterm 20 and minterm 15 to 31.

The best way to visualize this new rule for adjacent squares is to consider the two half maps as being one on top of the other. Any two squares that fall one over the other are considered adjacent.

Six-Variable K-Maps:

There are 64 minterms for a Boolean function with six-variables. Hence, Six-variable map consists of 64 squares.

By following the procedure used for the five-variable map, it is possible to construct a six-variable map with 4 four-variable maps to obtain the required 64 squares.

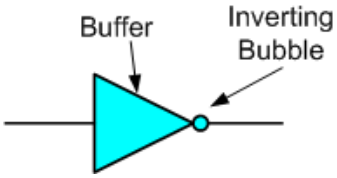
Universal Gates

Lesson Objectives:

In addition to AND, OR, and NOT gates, other logic gates like NAND and NOR are also used in the design of digital circuits.

The NOT circuit inverts the logic sense of a binary signal.

The small circle (bubble) at the output of the graphic symbol of a NOT gate is formally called a negation indicator and designates the logical complement.



The objectives of this lesson are to learn about:

1. Universal gates - NAND and NOR.
2. How to implement NOT, AND, and OR gate using NAND gates only.
3. How to implement NOT, AND, and OR gate using NOR gates only.
4. Equivalent gates.
5. Two-level digital circuit implementations using universal gates only.
6. Two-level digital circuit implementations using other gates.

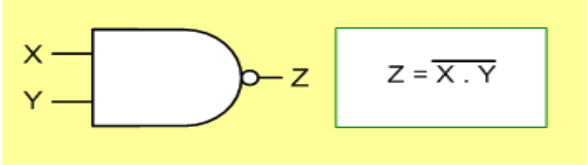
NAND Gate:

The **NAND** gate represents the complement of the AND operation. Its name is an abbreviation of **N**OT **AND**.

The graphic symbol for the NAND gate consists of an AND symbol with a bubble on the output, denoting that a complement operation is performed on the output of the AND gate.

The truth table and the graphic symbol of NAND gate is shown in the figure.

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



The truth table clearly shows that the NAND operation is the complement of the AND.

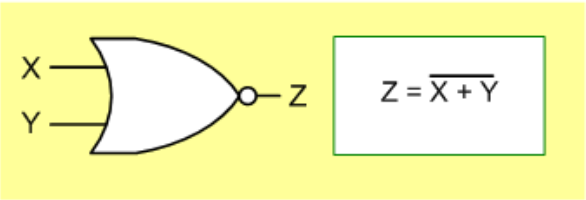
NOR Gate:

The **NOR** gate represents the complement of the OR operation. Its name is an abbreviation of **N**OT **OR**.

The graphic symbol for the NOR gate consists of an OR symbol with a bubble on the output, denoting that a complement operation is performed on the output of the OR gate.

The truth table and the graphic symbol of NOR gate is shown in the figure.

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



The truth table clearly shows that the NOR operation is the complement of the OR.

Universal Gates:

A universal gate is a gate which can implement any Boolean function without need to use any other gate type.

The NAND and NOR gates are universal gates.

In practice, this is advantageous since NAND and NOR gates are economical and easier to fabricate and are the basic gates used in all IC digital logic families.

In fact, an AND gate is typically implemented as a NAND gate followed by an inverter not the other way around!!

Likewise, an OR gate is typically implemented as a NOR gate followed by an inverter not the other way around!!

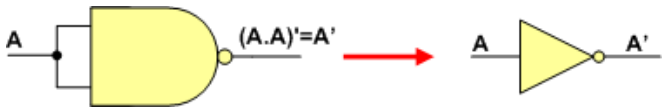
NAND Gate is a Universal Gate:

To prove that any Boolean function can be implemented using only NAND gates, we will show that the AND, OR, and NOT operations can be performed using only these gates.

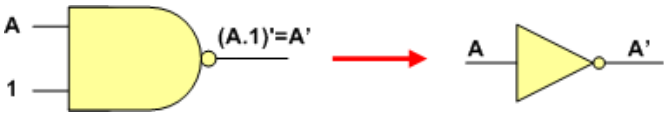
Implementing an Inverter Using only NAND Gate

The figure shows two ways in which a NAND gate can be used as **an inverter (NOT gate)**.

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

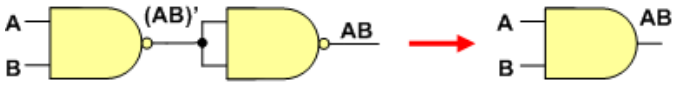


2. One NAND input pin is connected to the input signal **A** while all other input pins are connected to logic **1**. The output will be **A'**.



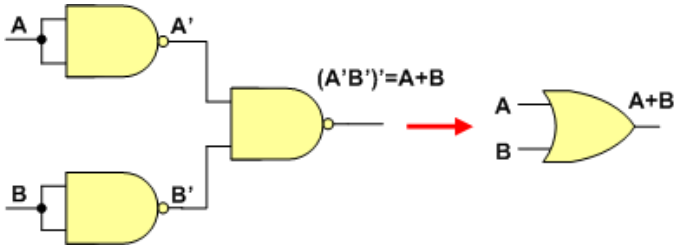
Implementing AND Using only NAND Gates

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



Implementing OR Using only NAND Gates

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



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

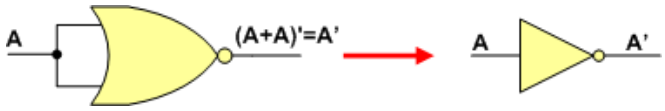
NAND Gate is a Universal Gate:

To prove that any Boolean function can be implemented using only NOR gates, we will show that the AND, OR, and NOT operations can be performed using only these gates.

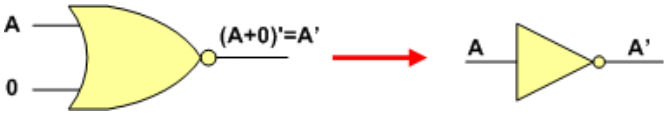
Implementing an Inverter Using only NOR Gate

The figure shows two ways in which a NOR gate can be used as **an inverter (NOT gate)**.

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



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

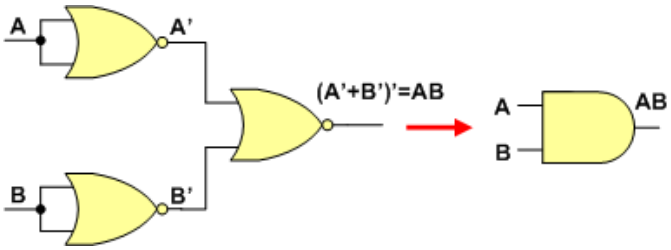


Implementing OR Using only NOR Gates

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

Implementing AND Using only NOR Gates

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

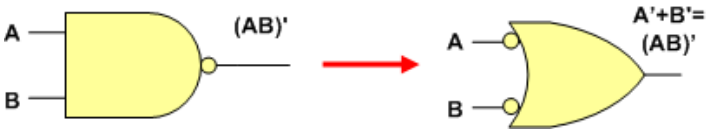


Thus, the NOR gate is a universal gate since it can implement the AND, OR and NOT functions.

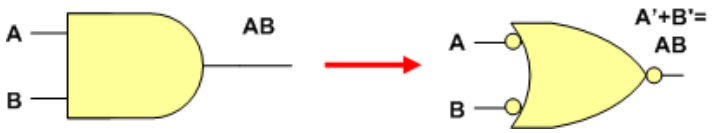
Equivalent Gates:

The shown figure summarizes important cases of gate equivalence. Note that bubbles indicate a complement operation (inverter).

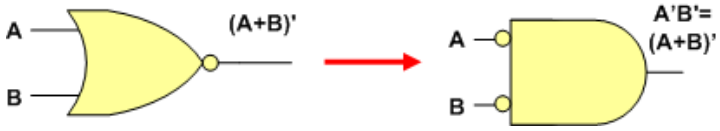
A NAND gate is equivalent to an inverted-input OR gate.



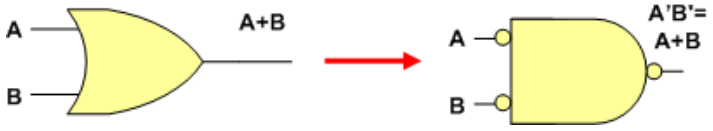
An AND gate is equivalent to an inverted-input NOR gate.



A NOR gate is equivalent to an inverted-input AND gate.



An OR gate is equivalent to an inverted-input NAND gate.



Two NOT gates in series are same as a buffer because they cancel each other as A'' = A.

Two-Level Implementations:

We have seen before that Boolean functions in either SOP or POS forms can be implemented using 2-Level implementations.

For SOP forms AND gates will be in the first level and a single OR gate will be in the second level.

For POS forms OR gates will be in the first level and a single AND gate will be in the second level.

Note that using inverters to complement input variables is not counted as a level.

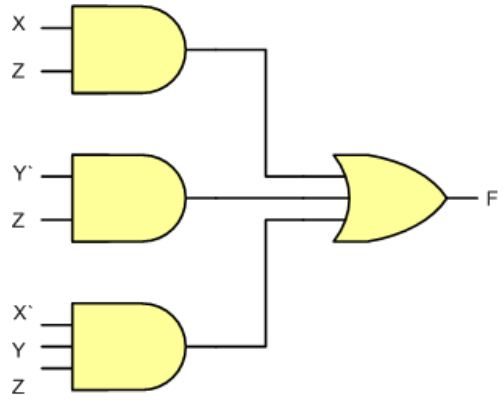
We will show that SOP forms can be implemented using only NAND gates, while POS forms can be implemented using only NOR gates.

This is best explained through examples.

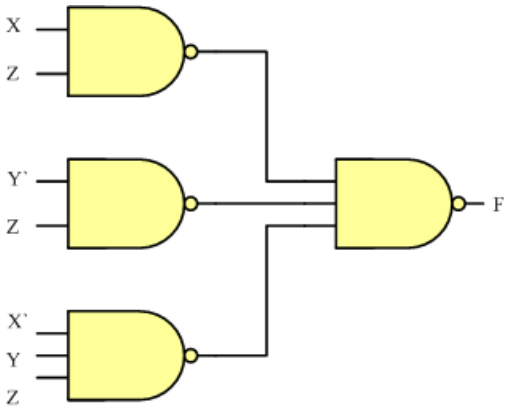
Example 1: Implement the following SOP function

$$F = XZ + Y'Z + X'YZ$$

Being an SOP expression, it is implemented in 2-levels as shown in the figure.



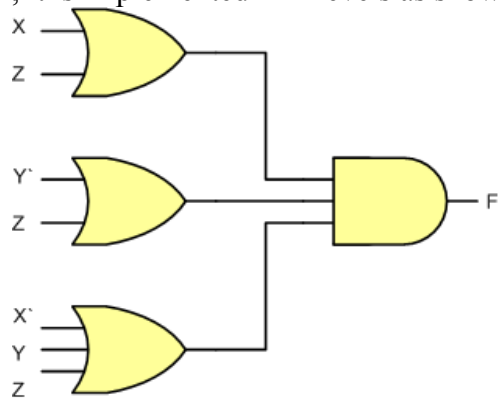
Introducing two successive inverters at the inputs of the OR gate results in the shown equivalent implementation. Since two successive inverters on the same line will not have an overall effect on the logic as it is shown before.
(see animation in authorware version)
By associating one of the inverters with the output of the first level AND gate and the other with the input of the OR gate, it is clear that this implementation is reducible to 2-level implementation where both levels are NAND gates as shown in Figure.



Example 2: Implement the following POS function

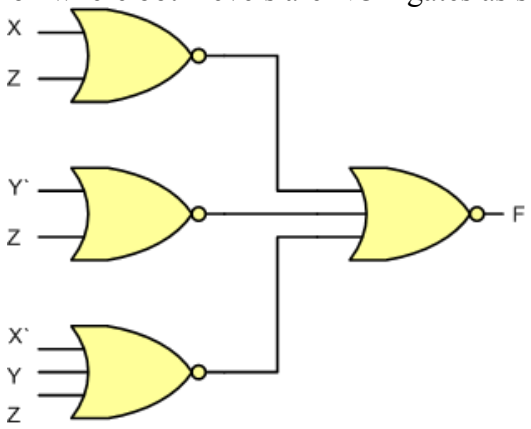
F = (X+Z) (Y'+Z) (X'+Y+Z)

Being a POS expression, it is implemented in 2-levels as shown in the figure.



Introducing two successive inverters at the inputs of the AND gate results in the shown equivalent implementation. Since two successive inverters on the same line will not have an overall effect on the logic as it is shown before.

(see animation in authorware version)
By associating one of the inverters with the output of the first level OR gates and the other with the input of the AND gate, it is clear that this implementation is reducible to 2-level implementation where both levels are NOR gates as shown in Figure.



There are some other types of 2-level combinational circuits which are

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

These are explained by examples.

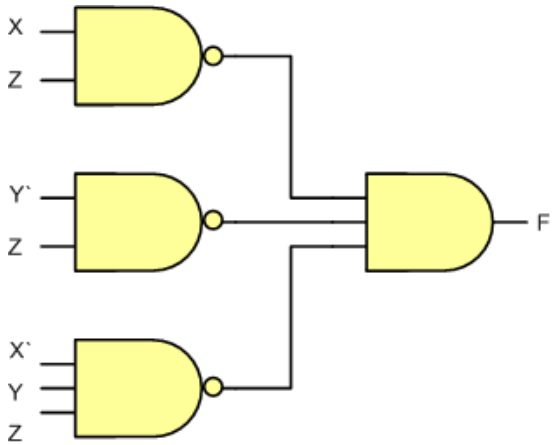
AND-NOR functions:

Example 3: Implement the following function

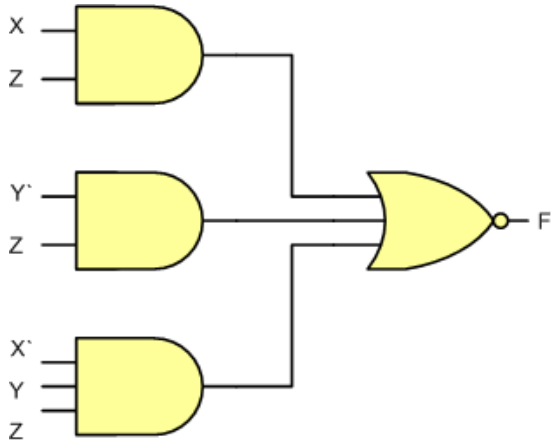
$F = \overline{XZ + \overline{Y}Z + \overline{X}YZ}$ **or**

$\overline{F} = XZ + \overline{Y}Z + \overline{X}YZ$

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



It can also be implemented using ***AND-NOR*** circuit as it is equivalent to NAND-AND circuit as shown in the figure. (see animation in authorware version)



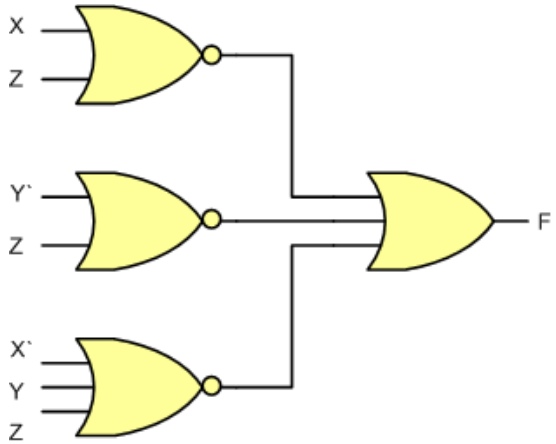
OR-NAND functions:

Example 4: Implement the following function

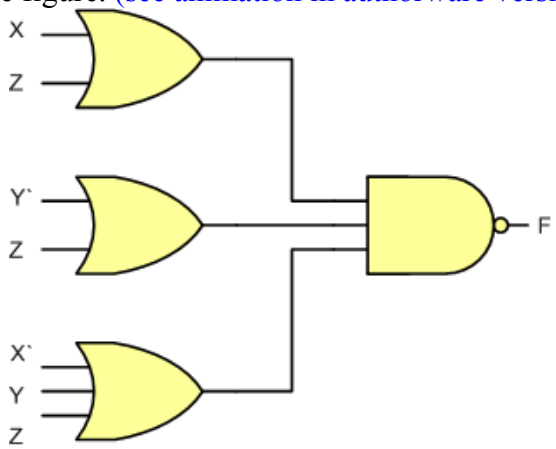
$F = \overline{(X+Z).(\overline{Y}+Z).(\overline{X}+Y+Z)}$ **or**

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

Since **F'** is in POS form, it can be implemented by using NOR-NOR circuit.
By complementing the output we can get **F**, **or** by using ***NOR-OR*** circuit as shown in the figure.



It can also be implemented using ***OR-NAND*** circuit as it is equivalent to NOR-OR circuit as shown in the figure. (see animation in authorware version)



XOR - XNOR Gates

Lesson Objectives:

In addition to AND, OR, NOT, NAND and NOR gates, exclusive-OR (XOR) and exclusive-NOR (XNOR) gates are also used in the design of digital circuits.
These have special functions and applications. These gates are particularly useful in arithmetic operations as well as error-detection and correction circuits.

XOR and XNOR gates are usually found as 2-input gates. No multiple-input XOR/XNOR gates are available since they are complex to fabricate with hardware.

The objectives of this lesson are to learn about:

1. XOR gates and XNOR gates
2. Their properties of operation and basic identities
3. Odd function and Even function
4. Parity generation and checking.

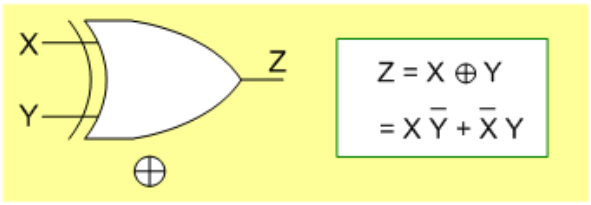
XOR Gate:

The exclusive-OR (XOR), operator uses the symbol ⊕, and it performs the following logic operation:

$X \oplus Y = X Y' + X' Y$

The graphic symbol and truth table of XOR gate is shown in the figure.

X	Y	XOR
0	0	0
0	1	1
1	0	1
1	1	0



The result is 1 only when either X is equal to 1 or Y is equal to 1, but not when both X and Y are equal to 1.

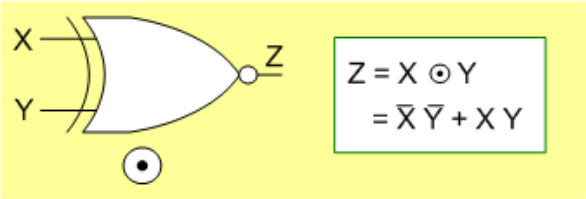
XNOR Gate:

The exclusive-NOR (XNOR), operator uses the symbol ⊙, and it performs the following logic operation

$X \odot Y = X Y + X' Y' = (X \oplus Y)'$

The graphic symbol and truth table of XNOR (Equivalence) gate is shown in the figure.

X	Y	XNOR
0	0	1
0	1	0
1	0	0
1	1	1



The result is 1 when either both X and Y are 0's or when both are 1's. That is why this gate is often referred to as the **Equivalence** gate.

The truth tables clearly show that the exclusive-NOR operation is the complement of the exclusive-OR.

This can also be shown by algebraic manipulation as follows:

$(X \oplus Y)' = (X\,Y' + X'\,Y)'$

$= (X\,Y')'\,(X'\,Y)' = (X' + Y)\,(X + Y')$

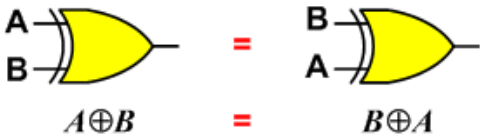
$= (XY + X'Y')$

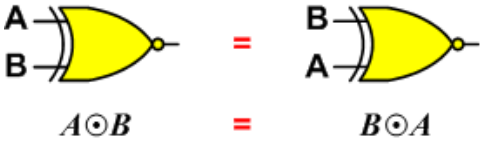
$= X \odot Y$

Properties of XOR/XNOR Operations:

1- Commutativity

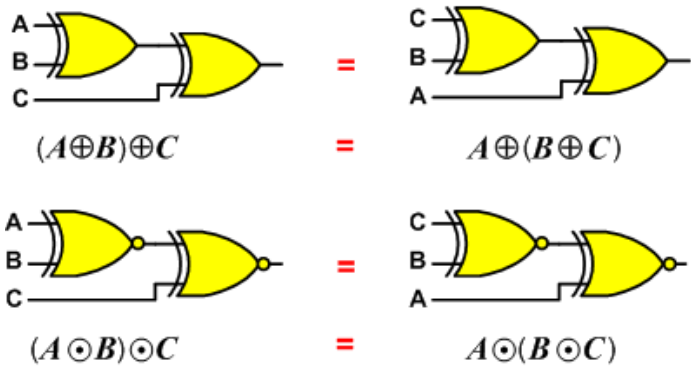
- $A \oplus B = B \oplus A$, and
- $A \odot B = B \odot A$





2- Associativity

- $A \oplus (B \oplus C) = (A \oplus B) \oplus C$, and
- $A \odot (B \odot C) = (A \odot B) \odot C$

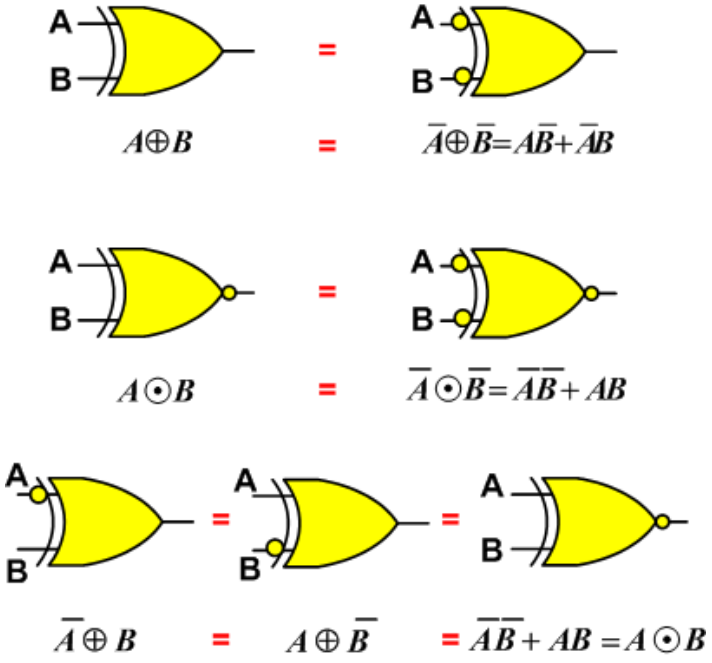


Basic Identities of XOR Operation:

Any of the following identities can be proven using either truth tables or algebraically by replacing the ⊕ operation by its equivalent Boolean expression:

- X ⊕ 0 = X
- X ⊕ 1 = X'
- X ⊕ X = 0
- X ⊕ X' = 1
- X ⊕ Y' = X' ⊕ Y = (X ⊕ Y)' = X ⊙ Y

The figure provides a graphical presentation of important XOR/XNOR rules and gate equivalence.

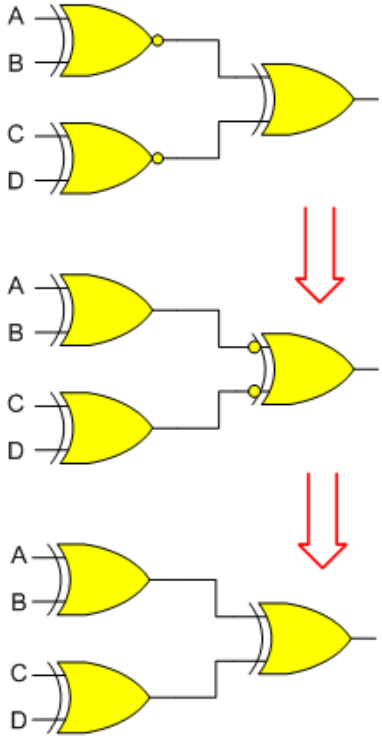


Example:

Show that **(A ⊙ B) ⊕ (C ⊙ D) = A ⊕ B ⊕ C ⊕ D**

Proving the above identity is easier done using graphical equivalence between gates as specified by the previous figure.

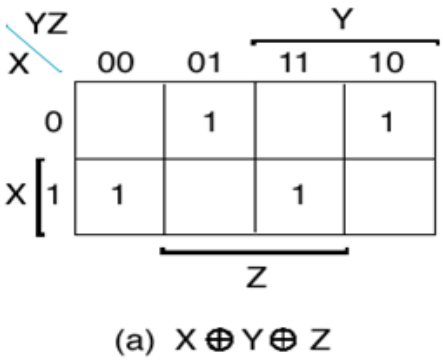
The following figure shows a step-by-step approach starting by the logic circuit corresponding to the left-hand-side of the identity and performing equivalent gate transformations till a circuit is reached that corresponds to the right-hand-side of the identity.



ODD Function:

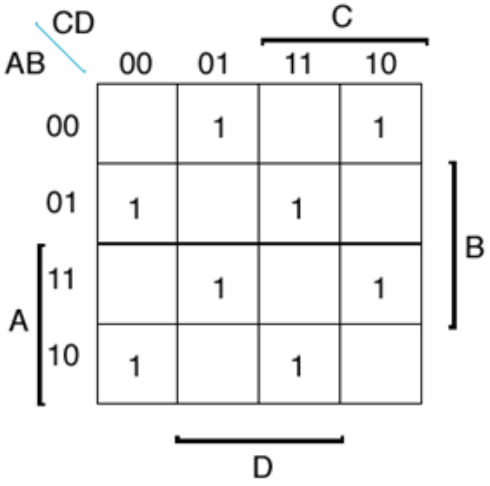
As shown in the K-map, **X ⊕ Y ⊕ Z = 1, IFF** *(if and only if)* the number of 1's in the input combination is ***odd***.

X	Y	Z	ODD
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1



(a) X ⊕ Y ⊕ Z

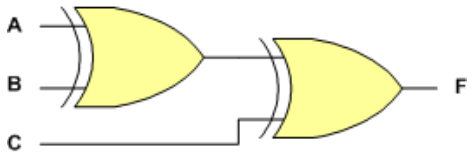
Likewise, **A ⊕ B ⊕ C ⊕ D = 1, IFF** the number of 1's in the input combination is ***odd***.



(b) A ⊕ B ⊕ C ⊕ D

In general, an exclusive-OR function of n-variables is an *odd function* which has a value of 1 **IFF** the number of 1's in the input combination is ***odd***, otherwise it has a value of 0.

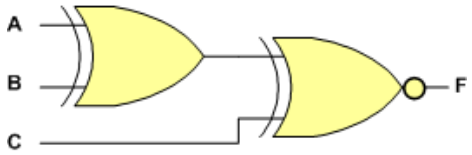
Since XOR gates are only designed with 2 inputs, the 3-input XOR function is implemented by means of two 2-input XOR gates, as shown in figure.



EVEN Function:

The complement of an odd function is an ***even function***. The even function is equal to 1 when the number of 1's in the input combination is even.

The complement of an odd function (an ***even function***) is obtained by replacing the output gate with an exclusive-NOR gate, as shown in figure.



Parity Generation and Checking:

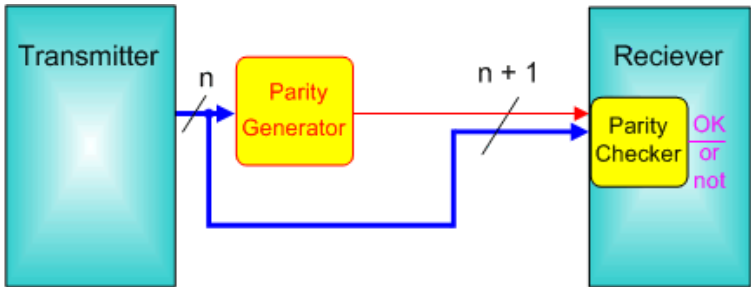
Exclusive-OR functions are very useful in systems using parity bits for error-detection.

A parity bit is used for the purpose of detecting errors during transmission of binary information.

A parity bit is an extra bit included with a binary message to make the **total number of 1's** in this message (including the parity bit) either **odd** or **even**.

The message, including the parity bit, is transmitted and then checked at the receiving end for errors. An error is detected if the checked parity does not correspond with the one transmitted.

The circuit that generates the parity bit at the transmitter side is called a *parity generator*. The circuit that checks the parity at the receiver side is called a *parity checker.*



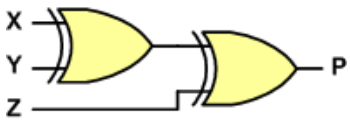
As an example, consider a 3-bit message to be transmitted together with an even parity bit. The table shows the ***truth table for the even parity generator***.

Three-Bit Message			Parity Bit
X	Y	Z	P
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

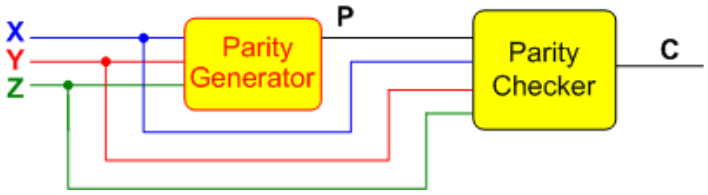
The three bits, **X, Y,** and **Z,** constitute the message and are the inputs to the *even parity generator* circuit whose output is the parity bit ***P***.

For even parity, whenever the message bits (**X, Y& Z**) have an odd number of 1's, the parity bit *P* must be 1. Otherwise, *P* must be 0.
Therefore, *P* can be expressed as a three-variable exclusive-OR function:
$\mathbf{P = X \oplus Y \oplus Z}$

The logic diagram for the even parity generator circuit is shown in the figure.



The 4 bits (**X, Y, Z & *P***) are *transmitted* to their destination, where they are applied to a *parity-checker circuit* to check for possible errors in the transmission.



Since the information was transmitted with even parity, the received four bits must have an even number of 1's.

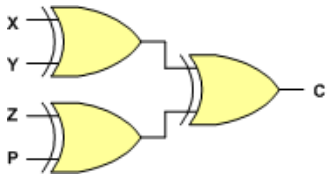
The parity checker generates an error signal (***C = 1***), whenever the received four bits have an odd number of 1's.

The table below shows the ***truth table for the even-parity checker.***

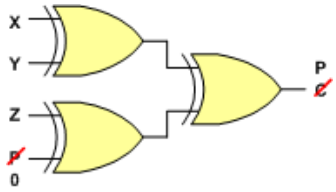
Four Bits Received				Parity Error Check
X	Y	Z	P	C
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	0
0	1	0	0	1
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	1
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	0

Obviously, the parity checker error output signal ***C*** is given by the following expression:
C = X ⊕ Y ⊕ Z ⊕ P

The logic diagram of the even-parity checker is shown in the figure.



It is worth noting that the parity generator can also be implemented with the circuit of this figure if the input *P* is connected to logic-0 and the output is marked with P. This is because $Z \oplus 0 = Z$, causing the value of *Z* to pass through the gate unchanged.



The advantage of this is that the same circuit can be used for both parity generation and checking.

Combinational Logic

Lesson Objectives

In this lesson, you will learn about

- What are combinational circuits
- Design procedure of combinational circuits
- Examples of combinational circuit design

Combinational Circuits

Logic circuit can be classified into two types. Combinational circuit, which consists of logic gates whose outputs at any time are determined by combining the values of the applied inputs using logic operations, and sequential circuits, which will be studied later.

In combinational circuits, the output at any time is a direct function of the applied external inputs (Figure 1). In other words,

$$Z = F(X)$$

That is, the outputs depend only on present inputs. A combinational circuit can be specified by a truth table.

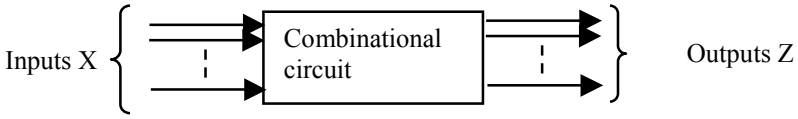


Figure 1: Combinational Circuit

Design procedure

The design of a combinational circuit starts from the specification of the problem, which leads to the truth table. Using the output values in the truth table, the logic equation for output function is found and simplified using K maps, or Algebraic manipulation or computer base tools. The equation of the output functions, the corresponding circuit is found. The process is shown in Figure

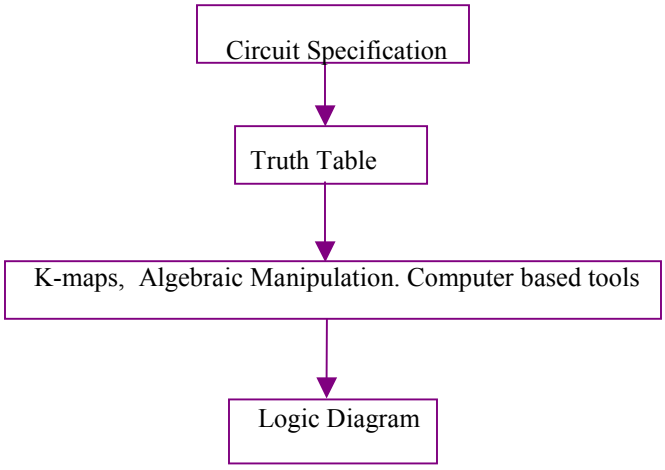


Figure 2: Design Procedure

Let us state these steps formally.

1) The first step is to find the truth table from circuit specification. This involves two sub-steps.

- The first is to determine the required number of inputs and outputs from the specification or verbal description of the problem. Then, assign a letter symbol to each input.
- Then, derive the truth table that defines the required relationship between inputs and outputs

2) Using the truth table, obtain the simplified Boolean expression for each output as a function of the input variables. The simplified equations can then be obtained using algebraic manipulation, K-maps, or computer-based tools.

3) Once the simplified equations are found, the corresponding logic diagram can be derived.

A practical design must consider constraints such as:

- Number of gates used.
- Number of gate inputs (*Fan-in*).
- Maximum number of gates an output signal can drive (*Fan-out*).
- Speed (propagation delay) requirements.

Example 1

Design a combinational circuit that has 3-bit input number and a single output (F). The output signal F is specified as follows:

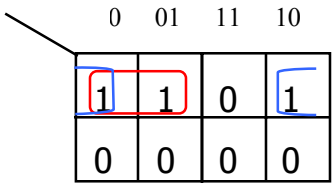
- F = 1 when the input number is less than $(3)_{10}$
- F = 0 otherwise.
- Implement F using only NAND gates

Let the three inputs be called X, Y, and Z. X is the most significant variable and Z is the least significant variable. The output F goes high, that is, the output produces logic 1 value if the input is less than 011, equivalent to a decimal value of three. This means that the output will be logic one for input combinations 000, 001, and 010. For other input combinations, which are 011 upto 111, the output is logic zero (see table 1).

Dec. #	X	Y	Z	F
0	0	0	0	1
1	0	0	1	1
2	0	1	0	1
3	0	1	1	0
4	1	0	0	0
5	1	0	1	0
6	1	1	0	0
7	1	1	1	0

Table 1: Inputs and outputs for example 1

Since SOP expressions are directly implementable as 2-Level implementation of NAND gates, we consider the 1's of the function as shown in the K-map.



$$F = X'Y' + X'Z'$$

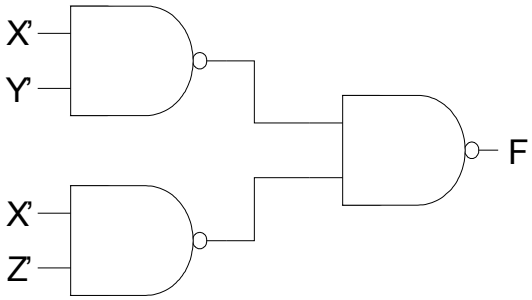


Figure 3: NAND-NAND- implementation for F = X' Y' + X' Z'

Code Converters

- Code converters are circuits which translate information from one binary code to another.
- The inputs to the circuit provide the bit combination belonging to the first code, while the outputs constitute the corresponding combination belonging to the second code.
- The combinational circuit performs the transformation from one code to another.

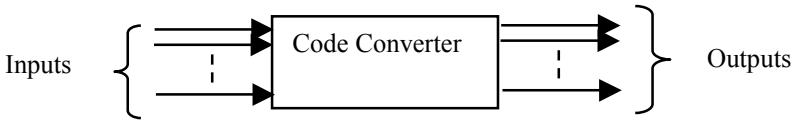


Figure 4: Code converter

Figure 4 shows the general structure of a code converter, containing the inputs, the code converter circuit, and the outputs. Consider, for example, a binary BCD to Excess-3 code converter

Example 2: BCD to excess-3 Code Converter

In this problem, the input is a BCD codeword. Since this is a 4-bit code that represents a decimal digit (0-to-9), there will be 4 input bits which will be represented by four input variables A,B,C, and D. Output is a 4-bit excess-3 code (W, X, Y ,Z)

Having defined the inputs and outputs, we proceed to build the truth table for this code converter. The truth table, lists the values of the output (that is the excess-3 code) for all possible combinations of the binary code. Note that, these codes are codes for decimal digits 0-9. In other words, even though the 4 bits of the input can represent up to 16 different combinations, ONLY 10 combinations are used to represent the 10 decimal digits.
Thus, a total of 6 input combinations are not likely to occur. Since these inputs will never occur, we use Don't cares for the corresponding output codes.

	BCD input				Ex-3 output			
Decimal #	A	B	C	D	W	X	Y	Z
0	0	0	0	0	0	0	1	1
1	0	0	0	1	0	1	0	0
2	0	0	1	0	0	1	0	1
3	0	0	1	1	0	1	1	0
4	0	1	0	0	0	1	1	1
5	0	1	1	0	1	0	0	1
6	0	1	1	1	1	0	1	0
7	1	0	0	0	1	0	1	1
8	1	0	0	1	1	1	0	0
9	0	1	1	0	1	0	0	1
10 - 15	All other inputs				X	X	X	X

Table 2: truth table for BCD to excess-3 code converter

Follow implementation procedure

As the procedure for simplification of a Boolean function suggests, we will minimize the four output functions using K-maps. Thus we will be having four K-maps, one for each output function. Each of these K-maps and the circuit are given in figure 5

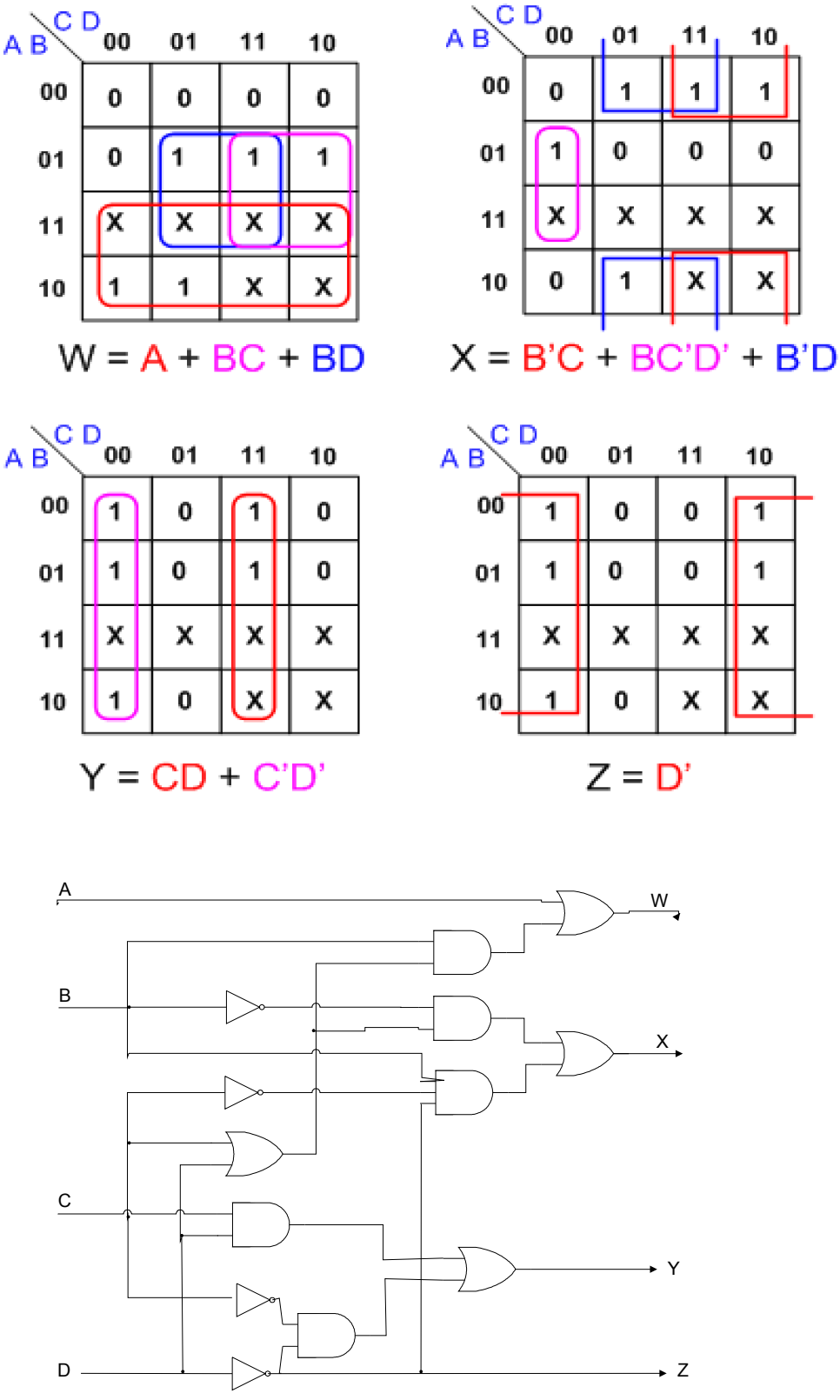


Figure 5: K-maps and circuit for example 2

Example 3: BCD to 7-segment display controller

Let's take another example. We will design a BCD to seven-segment decoder. Before proceeding, let's first understand what a 7-segment display is?

You might have noticed a digital watch, where the digits from 0 to 9 are displayed (see figure 6). These digits can be displayed using seven Light emitting diode segments (or LED's) arranged to look like digit 8 as shown in the figure. By controlling which segment is ON and which is OFF we can display illuminated patterns that correspond to the 10 decimal digits 0 to 9. For example, digit 8 can be displayed by illuminating all the segments.

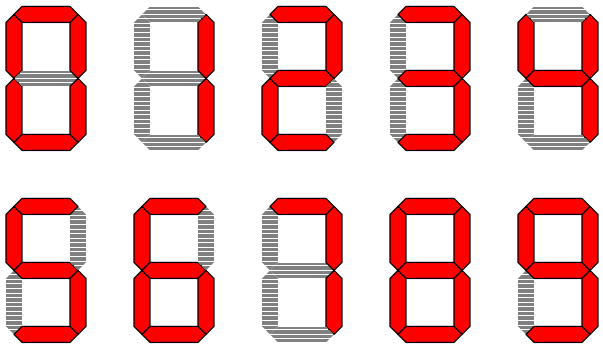


Figure 6: Numbers displayed in a digital watch

The objective is to design a circuit that will take a BCD number as input, and produces the control signals C0 to C6 which allow illuminating the corresponding segments in the 7-segment display, as shown in figure 7.

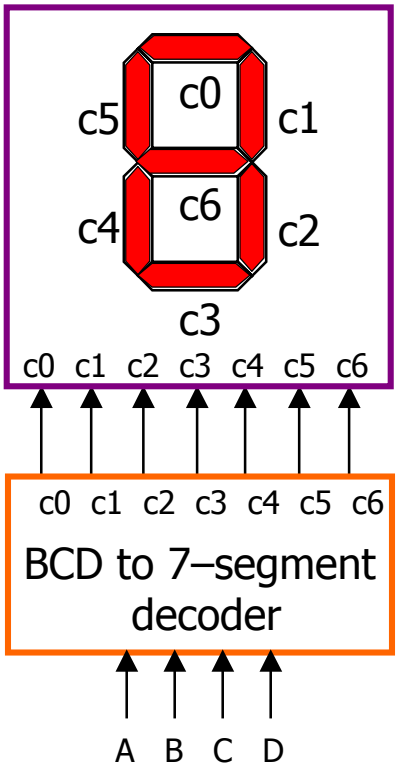


Figure 5: Design for example 3

Thus, the input is a 4-bit BCD digit A,B,C, and D; A being the most significant while D being the least significant.

The seven segments, which are actually seven output signals, are numbered C0 to C6 that control the illumination of the 7-segment display.

Each of the segment is a Light-Emitting Diode (LED) which is illuminated if current passes through it or dimmed if no current passes through it. For example, digit zero can be displayed by illuminating all the segments except segment C6. Digit 1 can be displayed by ONLY illuminating segments C1 and C2.

Having defined the format of inputs and outputs, let us find out the truth table for this circuit. In the truth table, each input BCD code and its corresponding 7-segment output is shown. The truth table assumes that a logic-1 illuminates a segment while a logic-0 turns the segment off.

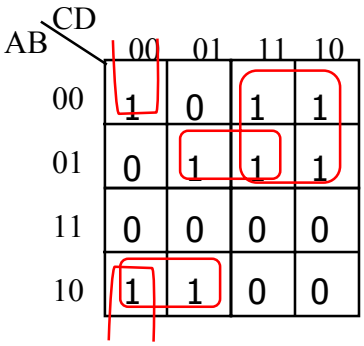
	BCD input				Outputs for 7-segments						
Decimal #	A	B	C	D	C0	C1	C2	C3	C4	C5	C6
0	0	0	0	0	1	1	1	1	1	1	0
1	0	0	0	1	0	1	1	0	0	0	0
2	0	0	1	0	1	1	0	1	1	0	1
3	0	0	1	1	1	1	1	1	0	0	1
4	0	1	0	0	0	1	1	0	0	1	1
5	0	1	0	1	1	0	1	1	0	1	1
6	0	1	1	0	1	0	1	1	1	1	1
7	0	1	1	1	1	1	1	0	0	0	0
8	1	0	0	0	1	1	1	1	1	1	1
9	1	0	0	1	1	1	1	0	0	1	1
10 -15	All other inputs				0	0	0	0	0	0	0

Table 3: Truth table for BCD to 7-segment display converter

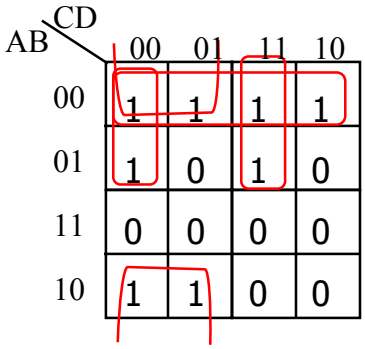
Even though the 4 bits of the input can represent up to 16 different combinations, ONLY 10 input combinations representing the 10 decimal digits are considered Valid.

We will design the controller such that the Invalid Input combinations would turn-off all segments. Thus all 7 segments are turned off for input codes beyond 1001.

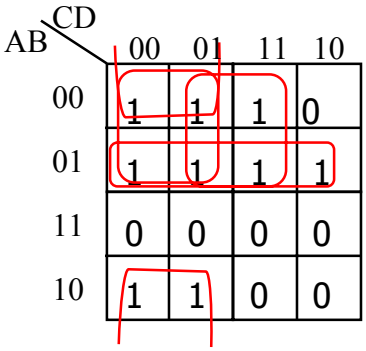
Now we are ready to build the seven K-maps, one for each output segment, as shown below



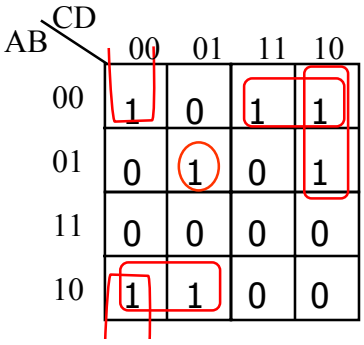
C0 = A’ C + A’ B D +B’ C’ D’ + A B’ C’



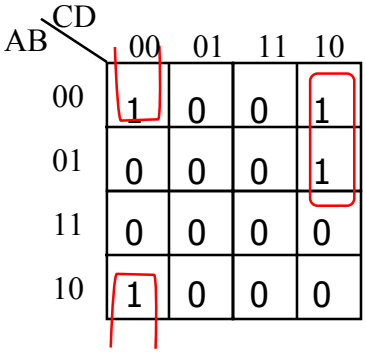
C1 = A’ B’ + B’ C’ + A’ C’ D’ + A’ C D



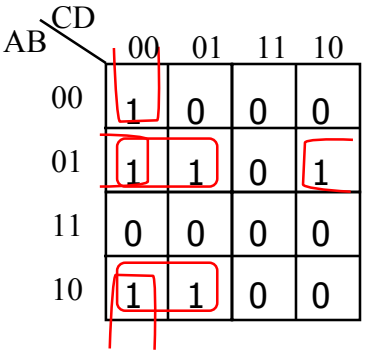
C2 = A’ B + B’ C’ + A’ C’ + A’ D



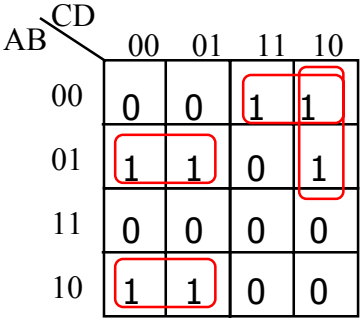
C3 = A’ C D’ + A’ B’ C + B’ C’ D’ + A B’ C’ + A’ B C’ D



C4 = A’ C D’ + B’ C’ D’



C5 = A’ B C’ + A’ C’ D’ + A’ B D’ + A B’ C’



C6 = A’ C D’ + A’ B’ C+ A’ B C’ + A B’ C’

Adders - Subtractors

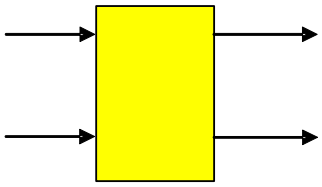
Lesson Objectives:

The objectives of this lesson are to learn about:

1. Half adder circuit.
2. Full adder circuit.
3. Binary parallel adder circuit.
4. Half subtractor circuit.
5. Full subtractor circuit.

Half Adder:

A ***half adder (HA)*** is an arithmetic circuit that is used to add two bits. The block diagram of HA is shown. It has two inputs and two outputs.



The inputs of the HA are the 2 bits to be added; the augend, and addend. The output is the result of this addition, i.e. a sum bit (S) and a carry bit (C).

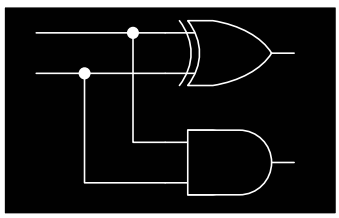
INPUTS		OUTPUTS	
X	**Y**	**C**	**S**
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

The truth table of HA is shown. The Boolean functions for the two outputs can be obtained from the truth table which are:

$$S = (\overline{X}Y + X\overline{Y}) = X \oplus Y$$

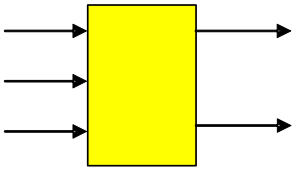
$$C = XY$$

Thus, the HA can be implemented using one XOR gate and one AND gate as shown in the Figure.

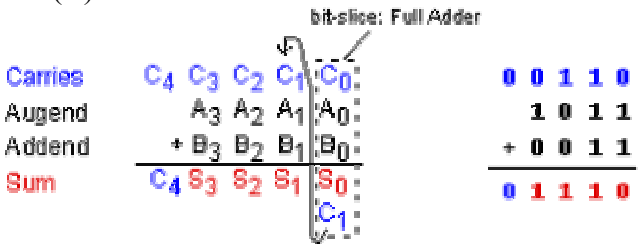


Full Adder:

A ***full adder (FA)*** is an arithmetic circuit that is used to add three bits. The block diagram of FA is shown. It has three inputs and two outputs.

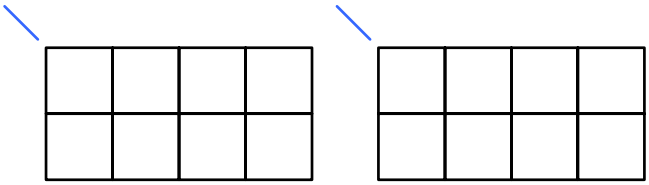


The inputs of the FA are the 3 bits to be added; the augend, addend, and carry from previous lower significant position. The output is the result of this addition, i.e. a sum bit (S) and a carry bit (C).



INPUTS			OUTPUTS	
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

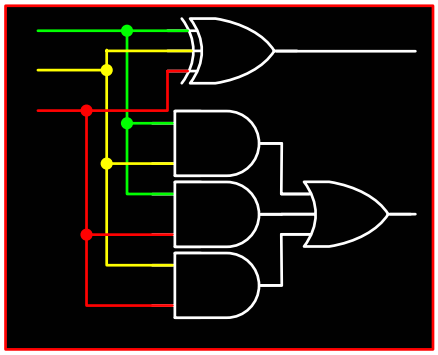
The truth table of FA is shown. The simplified Boolean functions for the two outputs can be obtained from the truth table, which are:



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

$$= X \oplus Y \oplus Z$$

$$C = XY + XZ + YZ$$



The Boolean functions for the two outputs can be **manipulated** to simplify the circuit, as shown below: **(see animation in authorware version)**

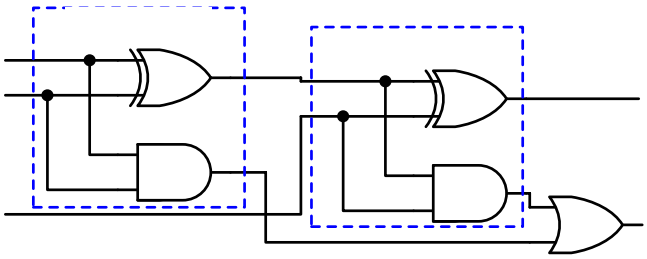
$$S = \overline{X}\overline{Y}Z + \overline{X}Y\overline{Z} + X\overline{Y}\overline{Z} + XYZ$$

$$= X \oplus Y \oplus Z$$

$$= (X \oplus Y) \oplus Z$$

$$C = XY + X\overline{Y}Z + \overline{X}YZ$$
$$= XY + Z(X\overline{Y} + \overline{X}Y)$$
$$= XY + Z(X \oplus Y)$$

Thus the full adder can be implemented using two half adders and an OR gate as shown in the Figure.

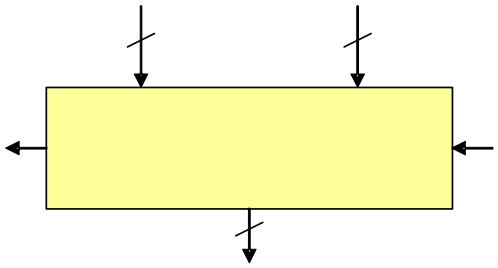


Binary Parallel Adder:

An ***n-bit adder*** is a circuit which adds two n-bits numbers, say, A and B.

In addition, an *n*-bit adder will have another single-bit input which is added to the two numbers called the carry-in (C_{in}).

The output of the *n*-bit adder is an n-bit sum (S) and a carry-out (C_{out}) bit. The block diagram of the *n*-bit adder is shown.



If all input bits of the two numbers (A & B) are applied simultaneously in parallel, the adder is termed a ***Parallel Adder.***

Consider the problem of designing a 4-bit binary parallel adder.

The total number of inputs is 9, since the two numbers have 4-bits each in addition to the C_{in} bit. Using conventional techniques for design would require a truth table of 2^9=512 rows.

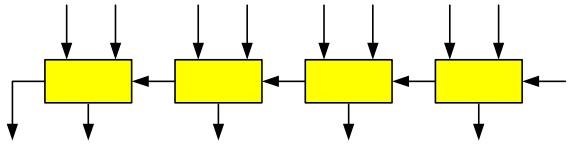
This causes the conventional design procedure to be unacceptable in this case.

Alternatively, the 4-bit binary parallel adder can be designed using 4 full adders connected in-cascade as shown in the figure.

X

Y

Z



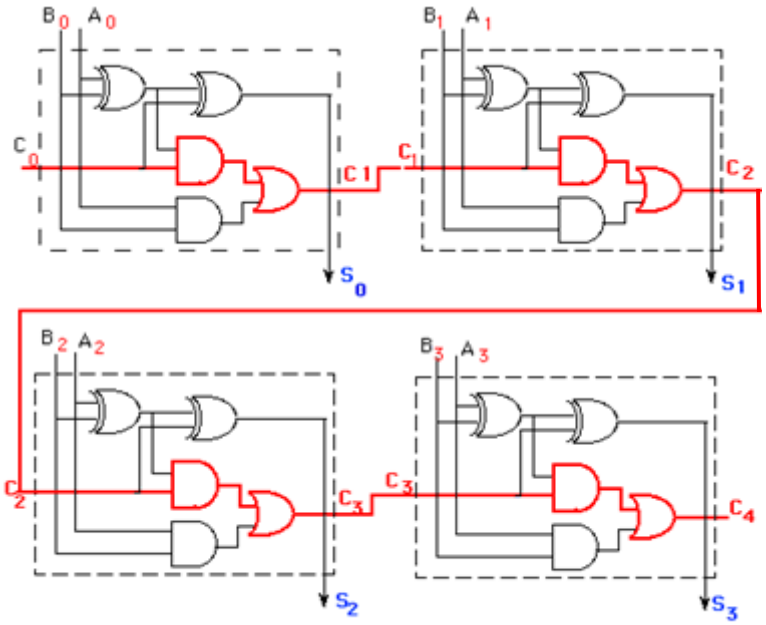
That is the *carry-out* bit of one full adder stage is used as *carry-in* input to the next stage.

In general, an n-bit binary parallel adder can be built out of *n* full adders connected in cascade.

Since a carry of 1 may appear near the least significant bit of the adder and yet propagate through many full adders to the most significant bit, just as a wave ripples outward from a pebble dropped in a pond. That is why this parallel adder is also called as ***ripple carry adder.***

The disadvantage of the ripple-carry adder is that it can get very slow when one needs to add many bits.

The propagation delay of this adder is fairly long since under worst case conditions, the carry has to propagate through all the stages as shown in the figure by **red colored path**.



This propagation delay is a limiting factor on the adder speed.

The signal from the input carry to the output carry propagates through an AND gate and OR gate, which constitute two gate levels. If there are four full adders, the output carry would have 2 x 4 = 8 levels from C_0 to C_4.

The total propagation time in this 4-bit adder would be the propagation time in one half adder (which is the first half adder) plus eight gate levels.
(see animation in authorware version)

Assuming that all the different types of gates have same propagation delay, say ***T***, the propagation delay of adder can be generalized as ***(2n + 1) T***, where ***n*** is the number of stages. In this example, ***n = 4,*** so the delay is ***(2 x 4 + 1) T = 9T***

Since all other arithematic operations are implemented by successive additions, the time consumed during addition process is very critical.

For fast applications, a better design is required. The ***carry-look-ahead adder*** solves this problem by calculating the carry signals in advance, based on the input signals.

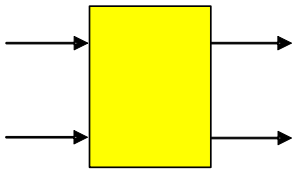
It is expalined in the next lesson.

Appendix:

Half Subtractor:

A ***half subtractor*** is an arithmetic circuit that subtracts two bits and produces their difference.

The block diagram of half subtractor is shown. The circuit has two inputs minuend (X) and subtrahend (Y) and two output bits, one is the difference bit (D) and the other is the borrow bit (B).



It performs the operation X – Y.

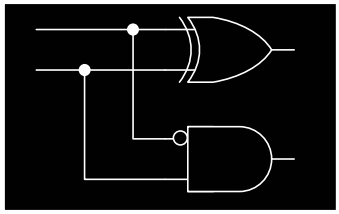
It should be noted that the weight of the output borrow bit is -2, while the weight of the output difference bit is +1.

INPUTS			OUTPUTS	
X	**Y**		**B**	**D**
0	0		0	0
0	1		1	1
1	0		0	1
1	1		0	0

The truth table of the half subtractor is shown. The Boolean functions for the two outputs can be obtained directly from the truth table as:

$$D = (\overline{X}Y + X\overline{Y}) = X \oplus Y$$

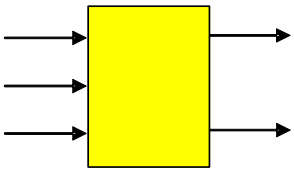
$$B = \overline{X}Y$$



Full Subtractor:

A ***full subtractor*** is a combinational circuit that performs a subtraction between two bits, taking into account that a 1 may have been borrowed by a lower significant bit.

The block diagram of full subtractor is shown. The circuit has three inputs and two outputs.



Input variables are minuend (X), subtrahend (Y), and previous borrow (Z); output variables are difference (D) and output borrow (B).
It performs the operation X – Y – Z.

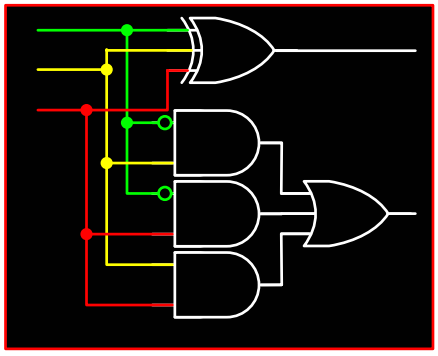
It should be noted that the weight of the output borrow bit is -2, while the weight of the output difference bit is +1.

The truth table of the full subtractor is shown.

INPUTS			OUTPUTS	
X	Y	Z	B	D
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	1	0
1	0	0	0	1
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

The simplified Boolean functions for the two outputs are:

$$D = \overline{X}\overline{Y}Z + \overline{X}Y\overline{Z} + X\overline{Y}\overline{Z} + XYZ$$
$$= X \oplus Y \oplus Z$$
$$B = \overline{X}Y + \overline{X}Z + YZ$$



Carry Look Ahead Adders

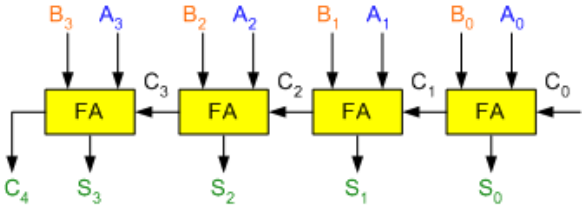
Lesson Objectives:

The objectives of this lesson are to learn about:

1. Carry Look Ahead Adder circuit.
2. Binary Parallel Adder/Subtractor circuit.
3. BCD adder circuit.
4. Binary mutiplier circuit.

Carry Look Ahead Adder:

In ***ripple carry adders***, the carry propagation time is the major speed limiting factor as seen in the previous lesson.



Most other arithmetic operations, e.g. multiplication and division are implemented using several add/subtract steps. Thus, improving the speed of addition will improve the speed of all other arithmetic operations.

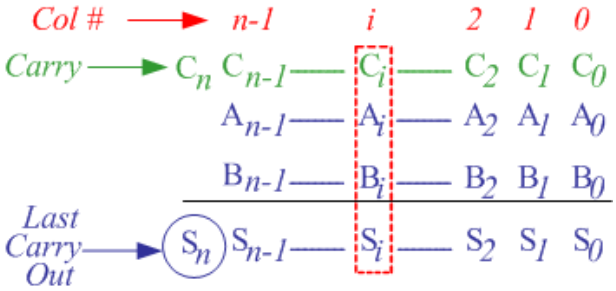
Accordingly, reducing the carry propagation delay of adders is of great importance. Different logic design approaches have been employed to overcome the carry propagation problem.

One widely used approach employs the principle of ***carry look-ahead*** solves this problem by calculating the carry signals in advance, based on the input signals.

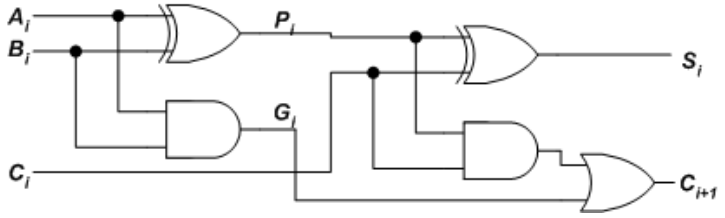
This type of adder circuit is called as ***carry look-ahead adder (CLA adder).*** It is based on the fact that a carry signal will be generated in two cases:

(1) when both bits A_i and B_i are 1, or
(2) when one of the two bits is 1 and the carry-in (carry of the previous stage) is 1.

To understand the carry propagation problem, let's consider the case of adding two *n-bit* numbers ***A*** and ***B***.



The Figure shows the full adder circuit used to add the operand bits in the i^{th} column; *namely A_i & B_i and the carry bit coming from the previous column (C_i).*



In this circuit, the 2 internal signals P_i and G_i are given by:

$$P_i = A_i \oplus B_i \text{ .....(1)}$$

$$G_i = A_i B_i \text{ .....(2)}$$

The output sum and carry can be defined as :

$$S_i = P_i \oplus C_i \text{ .....(3)}$$

$$C_{i+1} = G_i + P_i C_i \text{ .....(4)}$$

G_i is known as the ***carry Generate*** signal since a carry (C_{i+1}) is generated whenever G_i =1, regardless of the input carry (C_i).

P_i is known as the ***carry propagate*** signal since whenever P_i =1, the input carry is propagated to the output carry, i.e., C_{i+1}. = C_i (note that whenever P_i =1, G_i =0).

Computing the values of P_i and G_i only depend on the input operand bits (A_i & B_i) as clear from the Figure and equations.

Thus, these signals settle to their ***steady-state value*** after the propagation through their respective gates.

Computed values of ***all*** the P_i's are valid one XOR-gate delay after the operands A and B are made valid.

Computed values of ***all*** the G_i's are valid one AND-gate delay after the operands A and B are made valid.

The Boolean expression of the carry outputs of various stages can be written as follows:

$C_1 = G_0 + P_0C_0$
$C_2 = G_1 + P_1C_1 = G_1 + P_1 (G_0 + P_0C_0)$
$= G_1 + P_1G_0 + P_1P_0C_0$
$C_3 = G_2 + P_2C_2 = G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0C_0$
$C_4 = G_3 + P_3C_3$
$= G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0C_0$

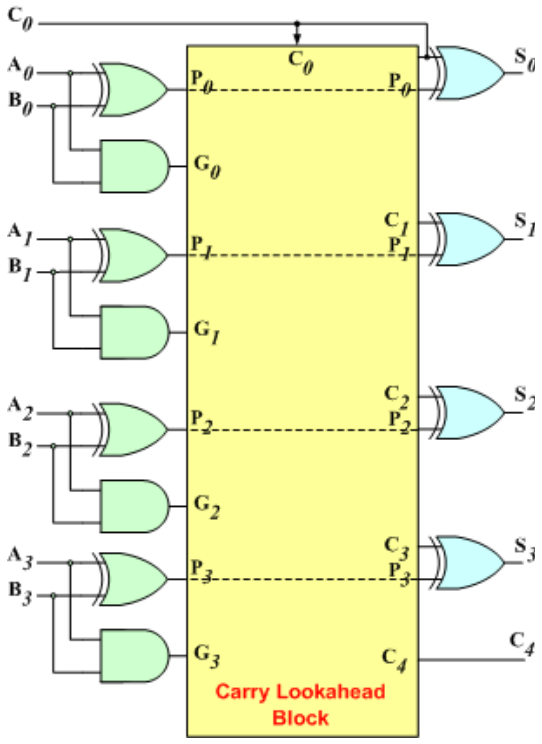
In general, the $i^{th.}$ carry output is expressed in the form $\mathbf{C_i} = F_i$ (P's, G's , $\mathbf{C_0}$).

In other words, each carry signal is expressed as a direct SOP function of $\mathbf{C_0}$ rather than its preceding carry signal.

Since the Boolean expression for each output carry is expressed in SOP form, it can be implemented in two-level circuits.

The 2-level implementation of the carry signals has a propagation delay of 2 gates, i.e., 2τ.

The 4-bit carry look-ahead (CLA) adder consists of 3 levels of logic:



First level: Generates all the P & G signals. Four sets of P & G logic (each consists of an XOR gate and an AND gate). Output signals of this level (P's & G's) will be valid after 1τ.

Second level: The Carry Look-Ahead (CLA) logic block which consists of four 2-level implementation logic circuits. It generates the carry signals (C_1, C_2, C_3, and C_4) as defined by the above expressions. Output signals of this level (C_1, C_2, C_3, and C_4) will be valid after 3τ.

Third level: Four XOR gates which generate the sum signals (S_i) ($S_i = P_i \oplus C_i$). Output signals of this level (S_0, S_1, S_2, and S_3) will be valid after 4τ.

Thus, the 4 Sum signals (S_0, S_1, S_2 & S_3) will all be valid after a total delay of 4τ compared to a delay of $(2n+1)\tau$ for Ripple Carry adders.

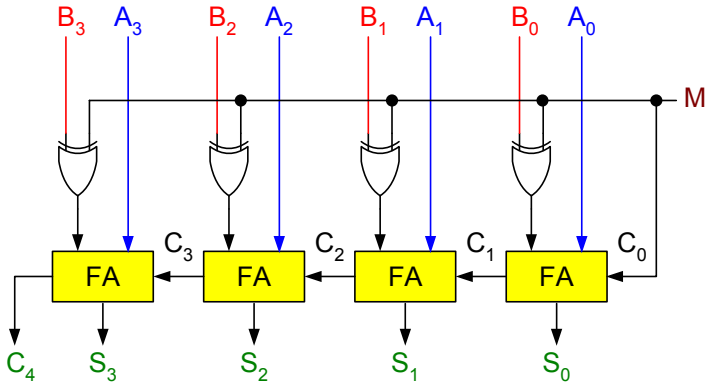
For a 4-bit adder ($n = 4$), the Ripple Carry adder delay is 9τ.

The disadvantage of the CLA adders is that the carry expressions (and hence logic) become quite complex for more than 4 bits.

Thus, CLA adders are usually implemented as 4-bit modules that are used to build larger size adders.

Binary Parallel Adder/Subtractor:

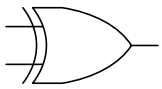
The addition and subtraction operations can be done using an Adder-Subtractor circuit. The figure shows the logic diagram of a 4-bit Adder-Subtractor circuit.



The circuit has a mode control signal M which determines if the circuit is to operate as an adder or a subtractor.

Each XOR gate receives input M and one of the inputs of B, i.e., B_i. To understand the behavior of XOR gate consider its truth table given below. If one input of XOR gate is **zero** then the output of XOR will be **same** as the second input. While if one input of XOR gate is **one** then the output of XOR will be **complement** of the second input.

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



(see animation in authorware)

So when **M = 0**, the output of XOR gate will be $B_i \oplus 0 = B_i$. If the full adders receive the value of B, and the input carry C_0 is 0, the circuit performs **A plus B**.

When **M = 1**, the output of XOR gate will be $B_i \oplus 1 = B_i'$. If the full adders receive the value of B', and the input carry C_0 is 1, the circuit performs A plus 1's complement of B plus 1, which is equal to **A minus B**.

BCD Adder:

If two BCD digits are added then their sum result will not always be in BCD.
Consider the two given examples.

Correct: Result is in BCD.

$$\begin{array}{rl} 0110 = & 6 \\ +0011 = & +3 \\ \hline 1001 = & 9 \end{array}$$

Wrong: Result is not in BCD.

$$\begin{array}{rl} 0101 = & 5 \\ +0111 = & +7 \\ \hline 1100 = & 12 \end{array}$$

In the first example, result is in BCD while in the second example it is not in BCD.

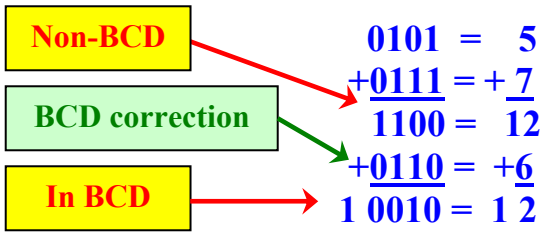
Four bits are needed to represent all BCD digits (0 – 9). But with four bits we can represent up to 16 values (0000 through 1111). The extra six values (1010 through 1111) are **not valid** BCD digits.

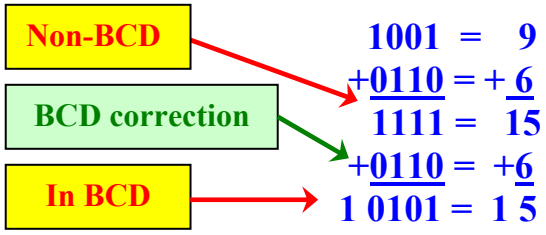
Whenever the sum result is **> 9**, it will not be in BCD and will require correction to get a valid BCD result.

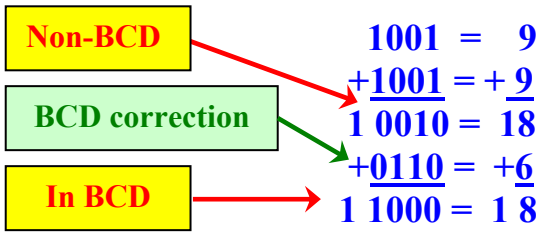
Z_3	Z_2	Z_1	Z_0		F
0	0	0	0		0
0	0	0	1		0
0	0	1	0		0
0	0	1	1		0
0	1	0	0		0
0	1	0	1		0
0	1	1	0		0
0	1	1	1		0
1	0	0	0		0
1	0	0	1		0
1	0	1	0		1
1	0	1	1		1
1	1	0	0		1
1	1	0	1		1
1	1	1	0		1
1	1	1	1		1

Correction is done through the addition of 6 to the result to skip the six invalid values as shown in the truth table by **yellow** color.

Consider the given examples of non-BCD sum result and its correction.







A BCD adder is a circuit that adds two BCD digits in parallel and produces a sum BCD digit and a carry out bit.

The maximum sum result of a *BCD input* adder can be 19. As maximum number in BCD is 9 and may be there will be a carry from previous stage also, so 9 + 9 + 1 = 19

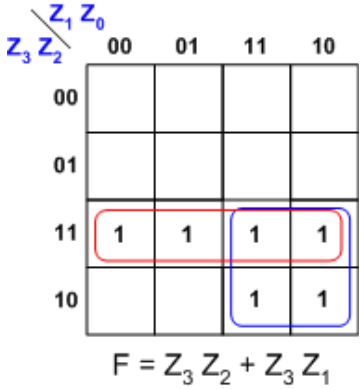
The following truth table shows all the possible sum results when two BCD digits are added.

Dec	CO	Z_3	Z_2	Z_1	Z_0		F
0	0	0	0	0	0		0
1	0	0	0	0	1		0
2	0	0	0	1	0		0
3	0	0	0	1	1		0
4	0	0	1	0	0		0
5	0	0	1	0	1		0
6	0	0	1	1	0		0
7	0	0	1	1	1		0
8	0	1	0	0	0		0
9	0	1	0	0	1		0
10	0	1	0	1	0		1
11	0	1	0	1	1		1
12	0	1	1	0	0		1
13	0	1	1	0	1		1
14	0	1	1	1	0		1
15	0	1	1	1	1		1
16	1	0	0	0	0		1
17	1	0	0	0	1		1
18	1	0	0	1	0		1
19	1	0	0	1	1		1

The logic circuit that checks the necessary BCD correction can be derived by detecting the condition where the resulting binary sum is 01010 through 10011 (decimal 10 through 19).

It can be done by considering the shown truth table, in which the function ***F*** is true when the digit is not a valid BCD digit. It can be simplified using a 5-variable K-map.

But detecting values 1010 through 1111 (decimal 10 through 15) can also be done by using a 4-variable K-map as shown in the figure.

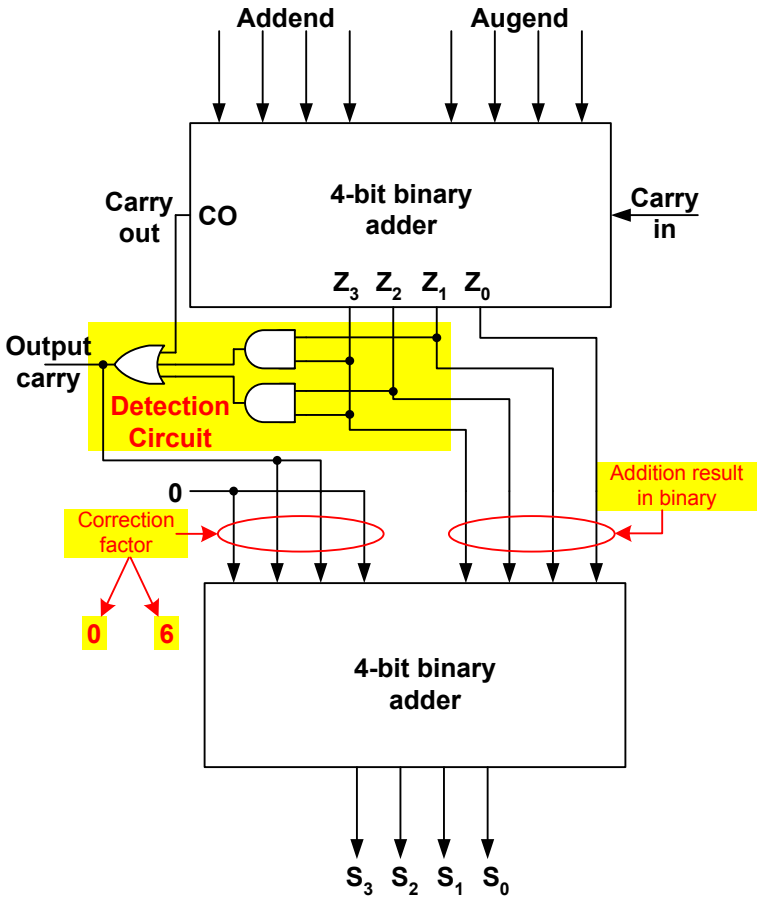


$F = Z_3 Z_2 + Z_3 Z_1$

Values greater than 1111, i.e., from 10000 through 10011 (decimal 16 through 19) can be detected by the carry out (CO) which equals 1 only for these output values. So, F = CO = 1 for these values. Hence, *F* is true when *CO* is true **OR** when **(***$Z_3 Z_2 + Z_3 Z_1$***)** is true. Thus, the correction step (adding 0110) is performed if the following function equals 1:

$$F = CO + Z_3 Z_2 + Z_3 Z_1$$

The circuit of the BCD adder will be as shown in the figure.



The two BCD digits, together with the input carry, are first added in the top 4-bit binary adder to produce the binary sum. The bottom 4-bit binary adder is used to add the correction factor to the binary result of the top binary adder.

Note:

- When the **Output carry** is equal to **zero**, the correction factor equals zero.
- When the **Output carry** is equal to **one**, the correction factor is 0110.

The output carry generated from the bottom binary adder is ignored, since it supplies information already available at the **output-carry** terminal.

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

Binary Multiplier:

Multiplication of binary numbers is performed in the same way as with decimal numbers.

The multiplicand is multiplied by each bit of the multiplier, starting from the least significant bit.

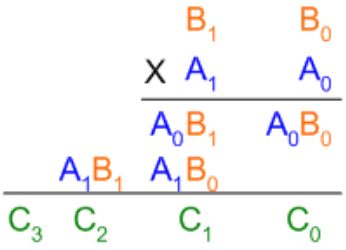
The result of each such multiplication forms a partial product. Successive partial products are shifted one bit to the left.

The product is obtained by adding these shifted partial products.

Example 1: Consider an example of multiplication of two numbers, say A and B (2 bits each), $C = A \times B$.

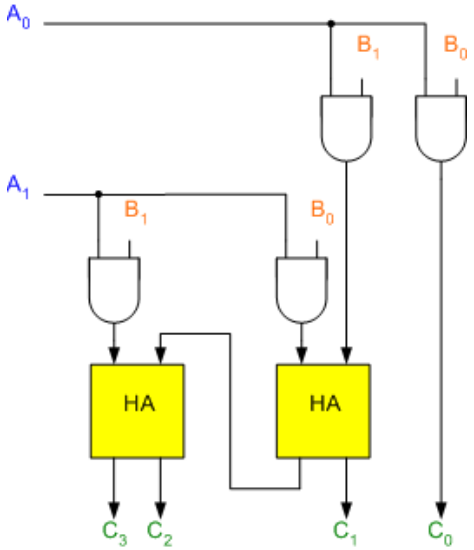
The first partial product is formed by multiplying the B_1B_0 by A_0. The multiplication of two bits such as A_0 and B_0 produces a 1 if both bits are 1; otherwise it produces a 0 like an AND operation. So the partial products can be implemented with AND gates.

The second partial product is formed by multiplying the B_1B_0 by A_1 and is shifted one position to the left.



(see animation in authorware)

The two partial products are added with two half adders (HA). Usually there are more bits in the partial products, and then it will be necessary to use FAs.

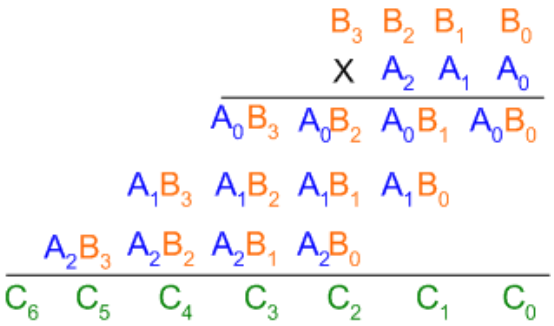


The least significant bit of the product does not have to go through an adder, since it is formed by the output of the first AND gate as shown in the Figure.

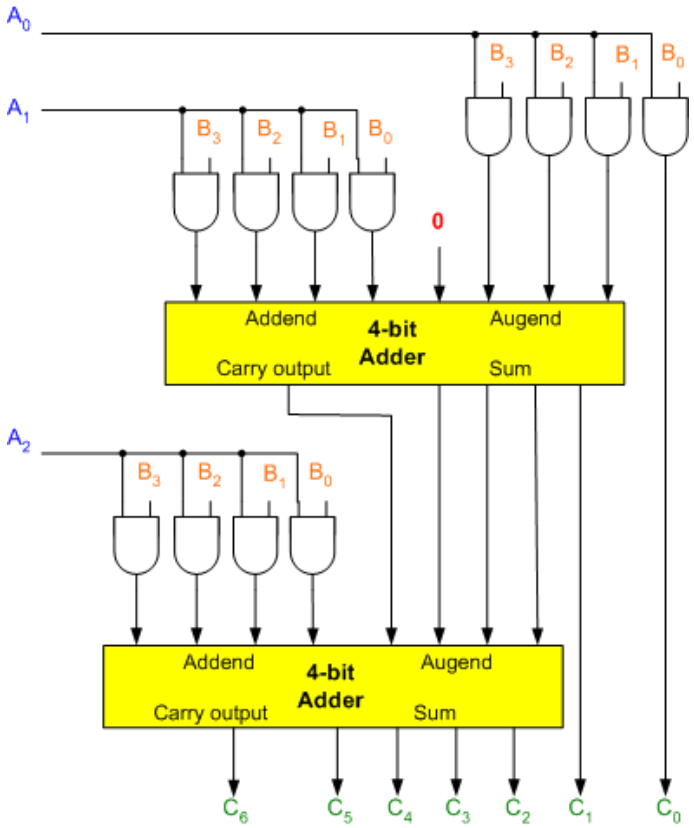
A binary multiplier with more bits can be constructed in a similar manner.

Example 2: Consider the example of multiplying two numbers, say A (3-bit number) and B (4-bit number).

Each bit of A (the multiplier) is ANDed with each bit of B (the multipcand) as shown in the Figure.



The binary output in each level of AND gates is added in parallel with the partial product of the previous level to form a new partial product. The last level produces the final product.



Since $J = 3$ and $K = 4$, 12 (J x K) AND gates and two 4-bit ((J - 1) K-bit) adders are needed to produce a product of seven ($J + K$) bits. Its circuit is shown in the Figure.

Note that **0** is applied at the most significant bit of augend of first 4-bit adder because the least significant bit of the product does not have to go through an adder.

Decoders and Encoders

Lesson Objectives

In this lesson, we will learn about

- Decoders
- Expansion of decoders
- Combinational circuit implementation with decoders
- Some examples of decoders
- Encoders
- Major limitations of encoders
- Priority encoders
- Some examples of ecnoders

Decoders

As its name indicates, a decoder is a circuit component that decodes an input code. Given a binary code of n-bits, a decoder will tell which code is this out of the 2^n possible codes (See Figure 1(a)).

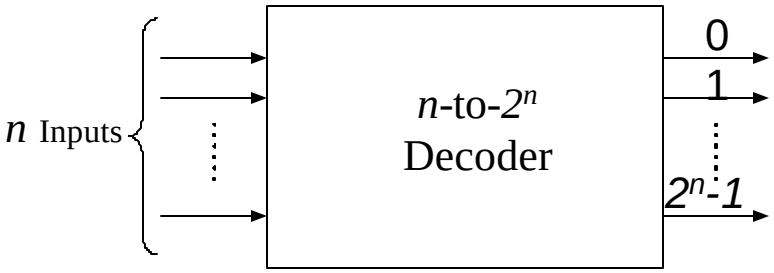


Figure 1(a): A typical decoder

Thus, a decoder has n-inputs and 2^n outputs. Each of the 2^n outputs corresponds to one of the possible 2^n input combinations.

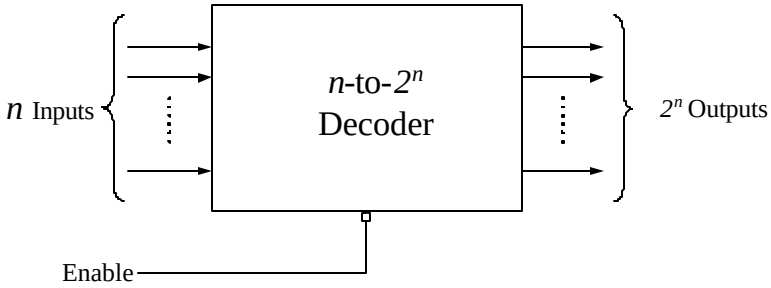


Figure 1(b): A typical decoder

Figure 1(b) shows the block diagram of a typical decoder, which has n input lines, and m output lines, where m is equal to 2^n. The decoder is called n-to-m decoder. Apart from this, there is also a single line connected to the decoder called enable line. The operations of the enable line will be discussed in the flowing text.

- In general, output i equals 1 if and only if the input binary code has a value of i.
- Thus, each output line equals 1 at only one input combination but is equal to 0 at all other combinations.
- In other words, each decoder output corresponds to a minterm of the n input variables.
- Thus, the decoder generates all of the 2^n minterms of n input variables.

Example: 2-to-4 decoders

Let us discuss the operation and combinational circuit design of a decoder by taking the specific example of a 2-to-4 decoder. It contains two inputs denoted by A_1 and A_0 and four outputs denoted by D_0, D_1, D_2, and D_3 as shown in figure 2. Also note that A_1 is the MSB while A_0 is the LSB.

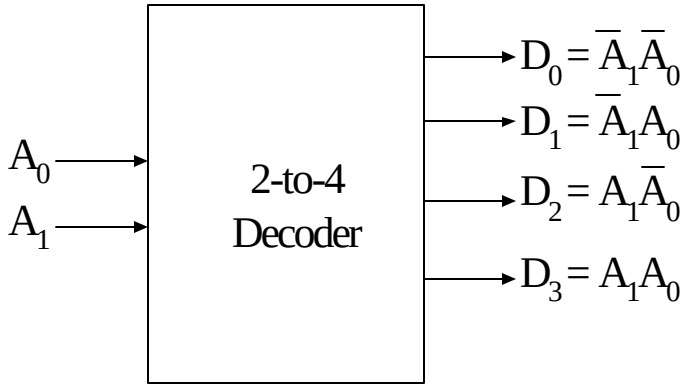


Figure 2: A 2-to-4 decoder without enable

Decimal #	Input		Output			
	A_1	A_0	D_0	D_1	D_2	D_3
0	0	0	1	0	0	0
1	0	1	0	1	0	0
2	1	0	0	0	1	0
3	1	1	0	0	0	1

Table 1: Truth table for 2-to-4 decoder

As we see in the truth table (table 1), for each input combination, one output line is activated, that is, the output line corresponding to the input combination becomes 1, while other lines remain inactive. For example, an input of 00 at the input will activate line D_0. 01 at the input will activate line D_1, and so on.

- Notice that, each output of the decoder is actually a minterm resulting from a certain combination of the inputs, that is
  - $D_0 = \overline{A_1}\ \overline{A_0}$, (minterm m_0) which corresponds to input 00
  - $D_1 = \overline{A_1}\ A_0$, (minterm m_1) which corresponds to input 01
  - $D_2 = A_1\ \overline{A_0}$, (minterm m_2) which corresponds to input 10
  - $D_3 = A_1\ A_0$, (minterm m_3) which corresponds to input 11

- This is depicted in Figures 2 where we see that each input combination will inovke the corresponding output, where each output is minterm corresponding to the input combination.

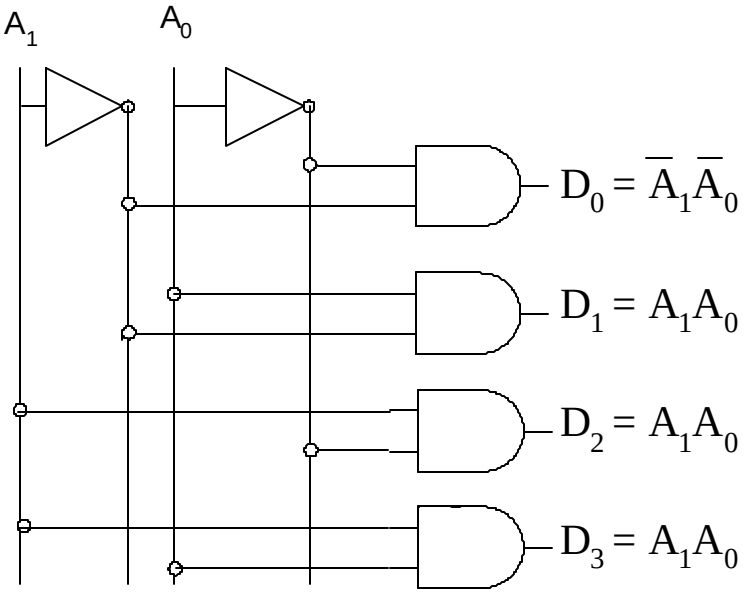


Figure 3: Implementation 2-to-4 decoder

The circuit is implemented with AND gates, as shown in figure 3. In this circuit we see that the logic equation for D_0 is $A_1' A_0'$. D_0 is $A_1' A_0$, and so on. These are in fact the minterms being implemented. Thus, each output of the decoder generates a minterm corresponding to the input combination.

The "enable" input in decoders

Generally, decoders have the "enable" input .The enable input perroms no logical operation, but is only responsible for making the decoder ACTIVE or INACTIVE.

- If the enable "E"
  - is zero, then all outputs are zero regardless of the input values.
  - is one, then the decoder performs its normal operation.

For example, consider the 2-to-4 decoder with the enable input (Figure 4). The enable input is only responsible for making the decoder active or inactive. If Enable E is zero, then all outputs of the decoder will be zeros, regardless of the values of A_1 and A_0. However, if E is 1, then the decoder will perform its normal operation, as is shown in the

truth table (table 2). In this table we see that as long as E is zero, the outputs D_0 to D_3 will remain zero, no matter whatever value you provide at the inputs A_1 A_0, depicted by two don't cares. When E becomes 1, then we see the same behavior as we saw in the case of 2-to-4 decoder discussed earlier.

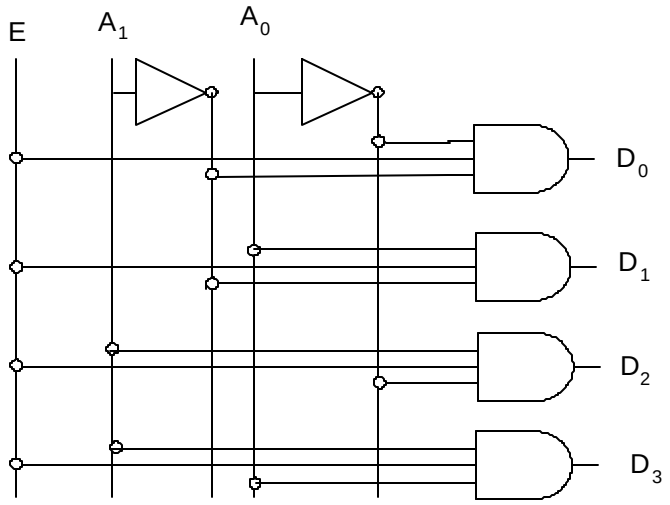


Figure 4: Implementation 2-to-4 decoder with enable

Decimal value	Enable	Inputs		Outputs			
	E	A_1	A_0	D_0	D_1	D_2	D_3
	0	X	X	0	0	0	0
0	1	0	0	1	0	0	0
1	1	0	1	0	1	0	0
2	1	1	0	0	0	1	0
3	1	1	1	0	0	0	1

Table 2: Truth table of 2-to-4 decoder with enable

Example: 3-to-8 decoders

In a three to eight decoder, there are three inputs and eight outputs, as shown in figure 5. A_0 is the least significant variable, while A_2 is the most significant variable.

The three inputs are decoded into eight outputs. That is, binary values at the input form a combination, and based on this combination, the corresponding output line is activated.

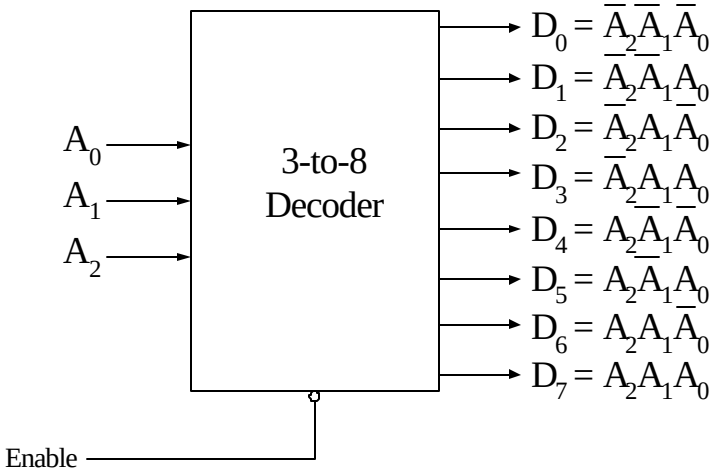


Figure 5: A 3-to-8 decoder with enable

Each output represents one minterm .

- For example, for input combination $A_2A_1A_0 = 001$, output line D_1 equals 1 while all other output lines equal 0's
- It should be noted that at any given instance of time, one and only one output line can be activated. It is also obvious from the fact that only one combination is possible at the input at a time, so the corresponding output line is activated.

Dec. Code	Inputs			Outputs							
	A_2	A_1	A_0	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7
0	0	0	0	1	0	0	0	0	0	0	0
1	0	0	1	0	1	0	0	0	0	0	0
2	0	1	0	0	0	1	0	0	0	0	0
3	0	1	1	0	0	0	1	0	0	0	0
4	1	0	0	0	0	0	0	1	0	0	0
5	1	0	1	0	0	0	0	0	1	0	0
6	1	1	0	0	0	0	0	0	0	1	0
7	1	1	1	0	0	0	0	0	0	0	1

Table 3: Truth table of 3-to-8 decoder

Since each input combination represents one minterm, the truth table (table 3) contains eight output functions, from D_0 to D_7 seven, where each function represents one and only one minterm. Thus function D_0 is $A_2' A_1' A_0'$. Similarly function D_7 is $A_2A_1A_0$. The corresponding circuit is given in Figure 6. In this figure, the three inverters provide complement of the inputs, and each one of the AND gates generates one of the minterms. It is also possible to add an Enable input to this decoder.

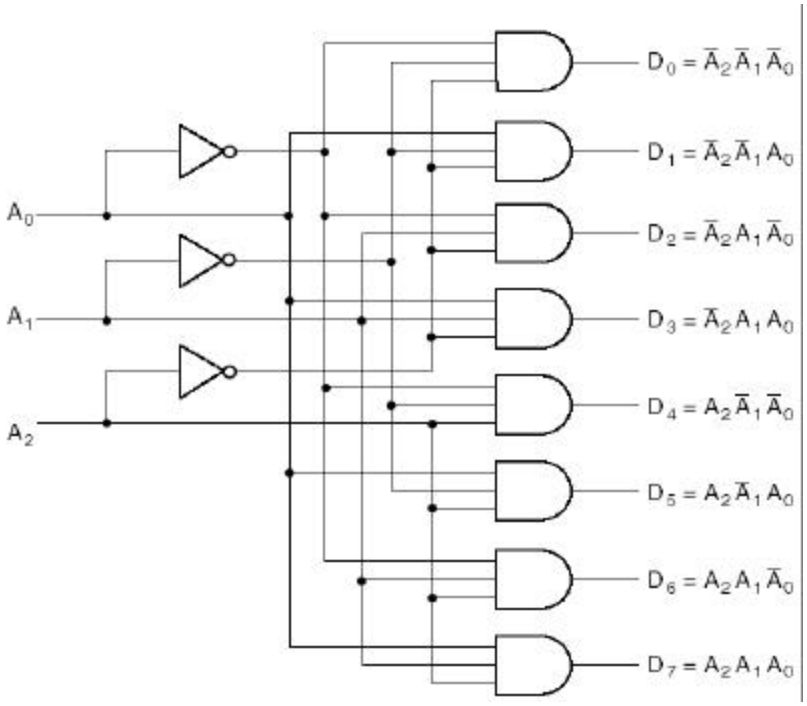


Figure 6: Implementation of a 3-to-8 decoder without enable

Decoder Expansion

- It is possible to build larger decoders using two or more smaller ones.

- For example, a 6-to-64 decoder can be designed with four 4-to-16 decoders and one 2-to-4 line decoder.

Example: Construct a 3-to-8 decoder using two 2-to-4 deocders with enable inputs.

Figure 7 shows how decoders with enable inputs can be connected to form a larger decoder. Two 2-to-4 line decoders are combined to build a 3-to-8 line decoder.

- The two least signifncat bits (i.e. A_1 and A_0) are connected to both decoders

- Most signifcant bit (A_2) is connected to the enable input of one decoder.
- The complement of most significant bit ($\overline{A_2}$) is connected to the enable of the other decoder.
- When $A_2 = 0$, upper decoder is enabled, while the lower is disabled. Thus, the outputs of the upper decoder correspond to minterms D_0 through D_3.
- When $A_2 = 1$, upper decoder is disabled, while the lower is enabled. Thus, the outputs of the lower decoder correspond to minterms D_4 through D_7.

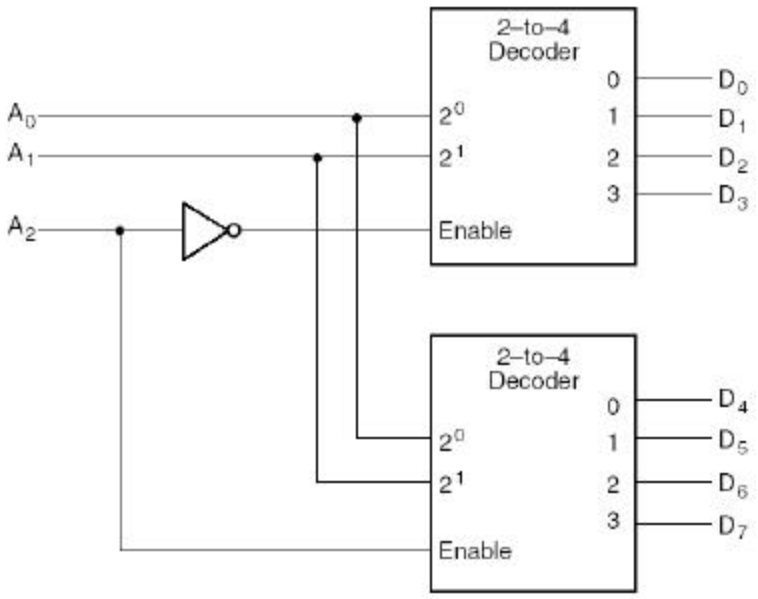


Figure 7: Implementing a 3-to-8 decoder with two 2-to-4 decoders

Decoder design with NAND gates

- Some decoders are constructed with NAND rather than AND gates.
- In this case, all decoder outputs will be 1's except the one corresponding to the input code which will be 0.

Decimal #	Input		Output			
	A_1	A_0	D_0'	D_1'	D_2'	D_3'
0	**0**	**0**	**0**	**1**	**1**	**1**
1	**0**	**1**	**1**	**0**	**1**	**1**
2	**1**	**1**	**1**	**1**	**0**	**1**
3	**1**	**1**	**1**	**1**	**1**	**0**

$$\overline{D}_0 = \overline{\overline{A}_1\overline{A}_0} \qquad \overline{D}_1 = \overline{\overline{A}_1 A_0}$$
$$\overline{D}_2 = \overline{A_1\overline{A}_0} \qquad \overline{D}_3 = \overline{A_1 A_0}$$

Table 4: Truth table of 2-to-4 decoder with NAND gates

This decoder can be constructed without enable, similar to what we have seen in the design of decoder with AND gates, without enable. The truth table and corresponding minters are given in table 4. Notice that the minters are in the complemented form.

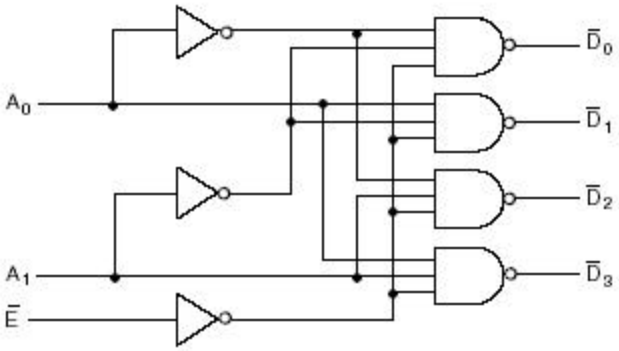


Figure 8: A 2-to-4 decoder with Enable constructed with NAND gates.

Decimal value	Enable	Inputs		Outputs			
	E'	A_1	A_0	D_0'	D_1'	D_2'	D_3'
	1	X	X	1	1	1	1
0	0	0	0	0	1	1	1
1	0	0	1	1	0	1	1
2	0	1	1	1	1	0	1
3	0	1	1	1	1	1	0

$$\overline{D}_0 = \overline{E\overline{A}_1\overline{A}_0} \qquad \overline{D}_1 = \overline{E\overline{A}_1 A_0}$$
$$\overline{D}_2 = \overline{EA_1\overline{A}_0} \qquad \overline{D}_3 = \overline{EA_1 A_0}$$

Table 5: Truth table of 2-to-4 decoder with Enable using NAND gates

A 2-to-4 line decoder with an enable input constructed with NAND gates is shown in figure 8. The circuit operates with complemented outputs and enable input E' is also complemented to match the outputs of the NAND gate decoder. The decoder is enabled when E' is equal to zero. As indicated by the truth table, only one output can be equal to zero at any given time, all other outputs being equal to one. The output with the value of zero represents the minterm selected by inputs A_1 and A_0. The circuit is disabled when E' is equal to one, regardless of the values of the other two inputs. When the circuit is disabled, none of the outputs are equal to zero, and none of the minterms are selected. The corresponding logic equations are also given in table 5.

Combinational circuit implementation using decoder

- As known, a decoder provides the 2^n minterms of *n* input variables
- Since any boolean functions can be expressed as a sum of minterms, one can use a decoder to implement any function of *n* variables.
- In this case, the decoder is used to generate the 2^n minterms and an additional OR gate is used to generate the sum of the required minterms.
- In this way, any combinational circuit with *n* inputs and *m* outputs can be

implemented using an n-to-2^n decoder in addition to m OR gates.

- Remember, that
  - The function need not be simplified since the decoder implements a function using the minterms, not product terms.

  - Any number of output functions can be implemented using a single decoder, provided that all those outputs are functions of the same input variables.

Example: Decoder Implementation of a Full Adder

Let us look at the truth table (table 6) for the given problem. We have two outputs, called S, which stands for sum, and C, which stands for carry. Both sum and carry are functions of X, Y, and Z.

Decimal value	Input			Output	
	X	Y	Z	S	C
0	0	0	0	0	0
1	0	0	1	1	0
2	0	1	0	1	0
3	0	1	1	0	1
4	1	0	0	1	0
5	1	0	1	0	1
6	1	1	0	0	1
7	1	1	1	1	1

Table 6: Truth table of the Full Adder

- The output functions S & C can be expressed in sum-of-minterms forms as follows:
  - S (X,Y,Z) = Σm (1,2,4,7)
  - C (X,Y,Z) = Σm (3,5,6,7)

Looking at the truth table and the functions in sum of minterms form, we observe that there are three inputs, X, Y, and Z that correspond to eight minterms. This implies that a 3-to-8 decoder is needed to implement this function. This implementation is given in Figure 9, where the sum S is implemented by taking minterms 1, 2, 4, and 7 and the OR gates forms the logical sum of minterm for S. Similarly, carry C is implemented by taking logical sum of minterms 3, 5, 6, and 7 from the same decoder.

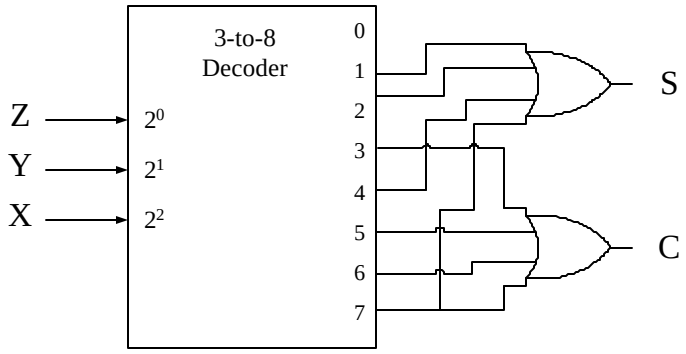


Figure 9: Decoder implementation of a Full Adder

Encoders

- An encoder performs the inverse operation of a decoder, as shown in Figure 10.
- It has 2^n inputs, and n output lines.
- Only one input can be logic 1 at any given time (active input). All other inputs must be 0's.
- Output lines generate the binary code corresponding to the active input.

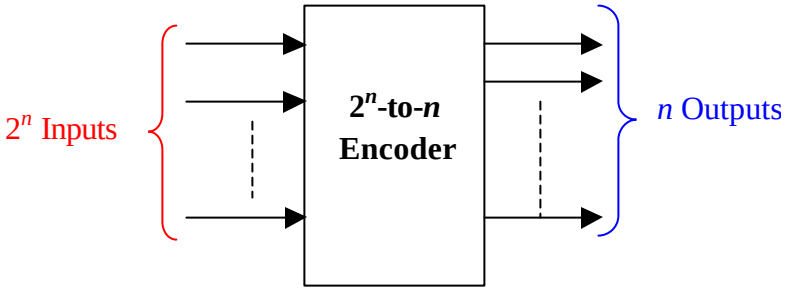


Figure 10: A typical Encoder

Example: Octal-to-binary encoder

We will use 8-to-3 encoder (Figure 11) for this problem, since we have eight inputs, one for each of the octal digits, and three outputs that generate the corresponding binary number. Thus, in the truth table, we see eight input variables on the left side of the vertical lines, and three variables on the right side of the vertical line (table 7).

Inputs								Outputs			Decimal Code
E7	E6	E5	E4	E3	E2	E1	E0	A2	A1	A0	
0	0	0	0	0	0	0	0	0	0	0	
0	0	0	0	0	0	0	1	0	0	0	0
0	0	0	0	0	0	1	0	0	0	1	1
0	0	0	0	0	1	0	0	0	1	0	2
0	0	0	0	1	0	0	0	0	1	1	3
0	0	0	1	0	0	0	0	1	0	0	4
0	0	1	0	0	0	0	0	1	0	1	5
0	1	0	0	0	0	0	0	1	1	0	6
1	0	0	0	0	0	0	0	1	1	1	7

Table 7: Truth table of Octal-to-binary encoder

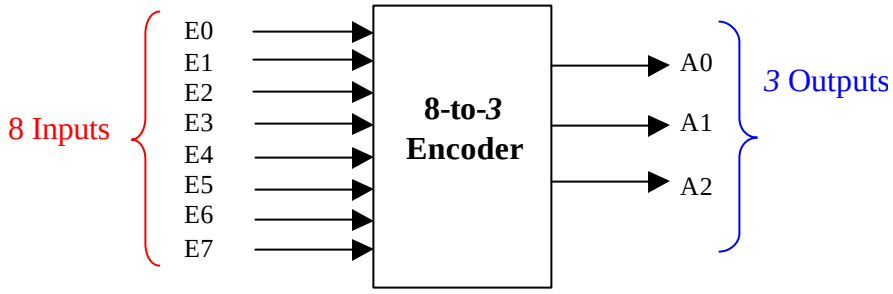


Figure 11: Octal-to-binary encoder

- Note that not all input combinations are valid.
- Valid combinations are those which have exactly one input equal to logic 1 while all other inputs are logic 0's.
- Since, the number of inputs = 8, K-maps cannot be used to derive the output Boolean expressions.
- The encoder implementation, however, can be directly derived from the truth table
  - Since $A_0 = 1$ if the input octal digit is 1 or 3 or 5 or 7, then we can write:
    $A_0 = E_1 + E_3 + E_5 + E_7$
  - Likewise, $A_1 = E_2 + E_3 + E_6 + E_7$, and similarly
  - $A_2 = E_4 + E_5 + E_6 + E_7$

- Thus, the encoder can be implemented using three 4-input OR gates.

Major Limitation of Encoders

- Exactly one input must be active at any given time.
- If the number of active inputs is less than one or more than one, the output will be incorrect.
- For example, if $E_3 = E_6 = 1$, the output of the encoder $A_2A_1A_0 = 111$, which implies incorrect output.

Two Problems to Resolve.

1. If two or more inputs are active at the same time, *what should the output be?*
2. An output of all 0's is generated in 2 cases:
   - when all inputs are 0
   - when E_0 is equal to 1.

   How can this ambiguity be resolved?

***Solution To Problem 1*:**

- Use a *Priority Encoder* which produces the output corresponding to the input with higher priority.
- Inputs are assigned priorities according to their subscript value; e.g. higher subscript inputs are assigned higher priority.
- In the previous example, if $E_3 = E_6 = 1$, the output corresponding to E_6 will be produced ($A_2A_1A_0 = 110$) since E_6 has higher priority than E_3.

***Solution To Problem 2*:**

- Provide one more output signal *V* to indicate *validity* of input data.
- *V* = 0 if none of the inputs equals 1, otherwise it is 1

Example: 4-to-2 Priority Encoders

- Sixteen input combinations
- Three output variables A1, A0, and V
- V is needed to take care of situation when all inputs are equal to zero.

Inputs				Outputs			
E3	**E2**	**E1**	**E0**	**A1**	**A0**	**V**	
0	**0**	**0**	**0**	**X**	**X**	**0**	Invalid Input
0	**0**	**0**	**1**	**0**	**0**	**1**	
0	**0**	**1**	**0**	**0**	**1**	**1**	
0	**0**	**1**	**1**	**0**	**1**	**1**	
0	**1**	**0**	**0**	**1**	**0**	**1**	
0	**1**	**0**	**1**	**1**	**0**	**1**	
0	**1**	**1**	**0**	**1**	**0**	**1**	
0	**1**	**1**	**1**	**1**	**0**	**1**	
1	**0**	**0**	**0**	**1**	**1**	**1**	
1	**0**	**0**	**1**	**1**	**1**	**1**	
1	**0**	**1**	**0**	**1**	**1**	**1**	
1	**0**	**1**	**1**	**1**	**1**	**1**	
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**	**1**	**1**	**1**	

Table 8: Truth table of 4-to-2 Priority Encoder

In the truth table (table 8), we have sixteen input combinations. In the output, we have three variables. The variable V is needed to take care of the situation where all inputs are zero. In that case V is kept at zero, regardless of the values of A_1 and A_0. This combination is highlighted green. In all other cases, V is kept at 1, because at least one of the inputs is one.

When E_0 is 1, the output combination of A_1 and A_0 is 00. This combination is highlighted blue.

Then we have two combinations highlighted yellow. In both these combinations, A_1 and A_0 are 01. This is because in both these combinations E_1 is 1, regardless of the value of E_0, and since E_1 has higher subscript, the corresponding output value is 01.

This is followed by four input combinations in pink. In these four combinations, the output A_1A_0 is 10, since E_2 is 1 in all these combinations, and E_2 has the highest

precedence compared to E_0 and E_1. Although E_0 and E_1 are also having a value of one in this set of four combinations, but they do not have the priority.

Finally we have the last eight input combinations, whose output is 11. This is because E3 is the highest priority input, and it is equal to 1. Though the other inputs with smaller subscripts, namely, E2, E1, and E0 are also having values of one in some combinations, but they do not have the priority.

The truth table can be rewritten in a more compact form using don't care conditions for inputs as shown below in table 9.

	Inputs				Outputs		
	E3	**E2**	**E1**	**E0**	**A1**	**A0**	**V**
1	**0**	**0**	**0**	**0**	**X**	**X**	**0**
2	**0**	**0**	**0**	**1**	**0**	**0**	**1**
3	**0**	**0**	**1**	**X**	**0**	**1**	**1**
4	**0**	**1**	**X**	**X**	**1**	**0**	**1**
5	**1**	**X**	**X**	**X**	**1**	**1**	**1**

Table 9: Truth table of 4-to-2 priority encoder (compact form)

- With 4 Input variables, the truth table must have 16 rows, with each row representing an input combination.
- With don't care input conditions, the number of rows can be reduced since rows with don't care inputs will actually represent more than one input combination.
- Thus, for example, row # 3 represents 2 combinations since it represents the input conditions $E_3E_2E_1E_0$=0010 and 0011.
- Likewise, row # 4 represents 4 combinations since it represents the input conditions $E_3E_2E_1E_0$=0100, 0101, 0110 and 0111.
- Similarly, row # 5 represents 8 combinations.
- Thus, the total number of input combinations represented by the 5-row truth table = 1+ 1+ 2+ 4 + 8= 16 input combinations.

Boolean Expressions for V, A_1 and A_0 and the circuit:

See next page:

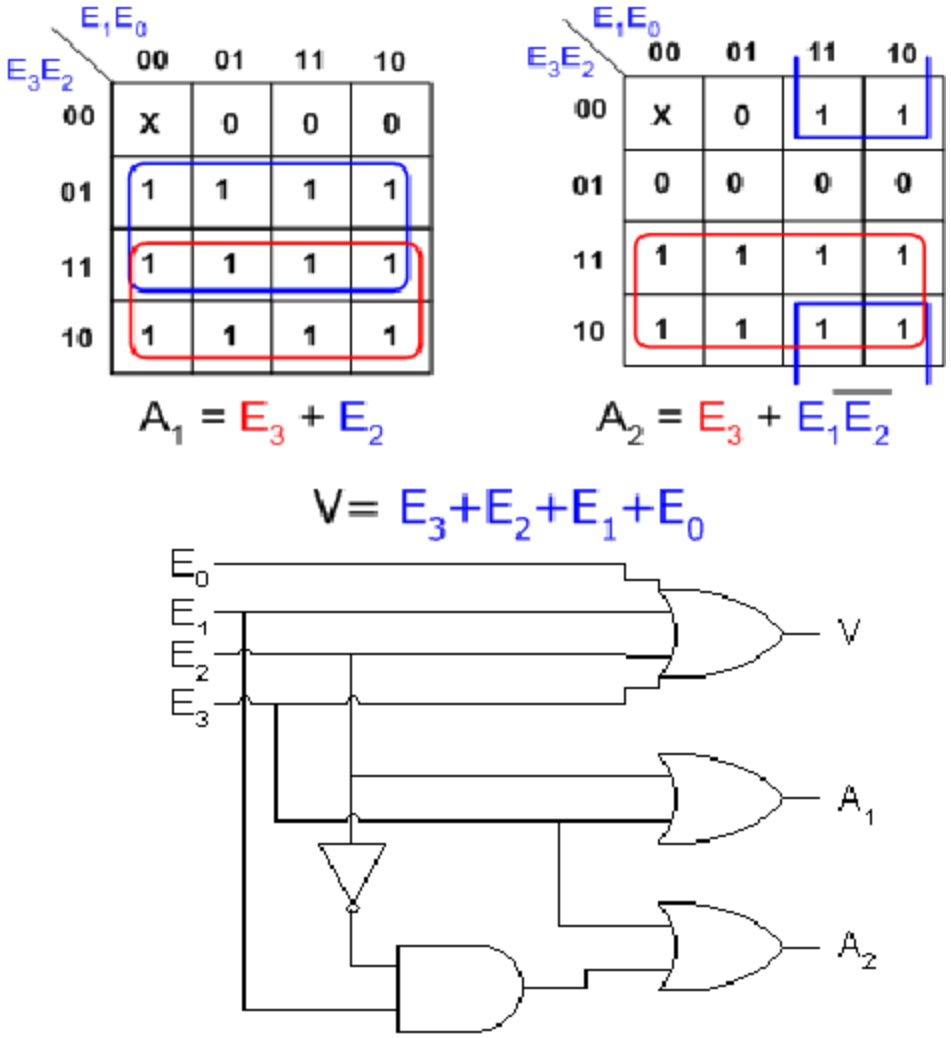


Figure 12: Equations and circuit for 4-to-2 priority encoder

Multiplexers and Demultiplexers

In this lesson, you will learn about:

1. Multiplexers
2. Combinational circuit implementation with multiplexers
3. Demultiplexers
4. Some examples

Multiplexer

A Multiplexer (see Figure 1) is a combinational circuit that selects one of the 2^n input signals (D_0, D_1, D_2, ……, D_{2^n-1}) to be passed to the single output line Y.

Q. How to select the input line (out of the possible 2^n input signals) to be passed to the output line?

A. Selection of the particular input to be passed to the output is controlled by a set of *n* input signals called "*Select Inputs*" (S_0, S_1, S_2, ……., S_{n-1}).

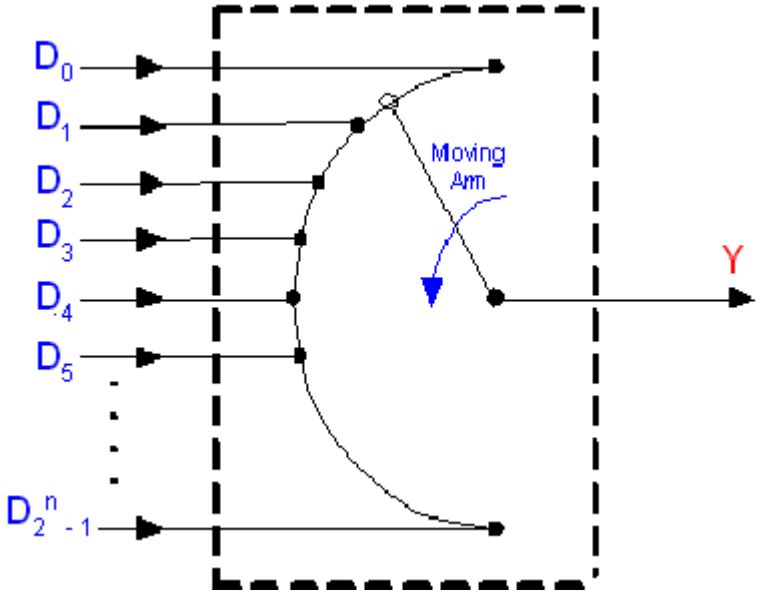


Figure 1: Multiplexer

Example 1: 2x1 Mux

A 2x1 Mux has 2 input lines (D_0 & D_1) , one select input (S), and one output line (Y). (see Figure 2)

IF S=0, then $Y= D_0$
Else (S=1) $Y= D_1$

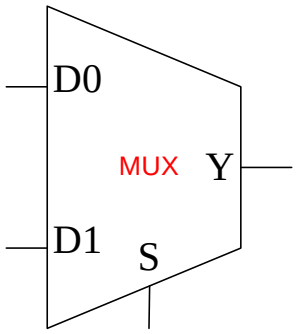


Figure 2: A 2 X 1 Multiplexer

Thus, the output signal Y can be expressed as:

$Y = \overline{S}\, D_0 + S\, D_1$

Example 2: 4x1 Mux

A 4x1 Mux has 4 input lines (D_0, D_1, D_2, D_3), two select inputs (S_0 & S_1), and one output line Y. (see Figure 3)

IF S_1S_0=00, **then** Y= D_0
IF S_1S_0=01, **then** Y= D_1
IF S_1S_0=10, **then** Y= D_2
IF S_1S_0=11, **then** Y= D_3

Thus, the output signal Y can be expressed as:

$$Y = \overline{S_1}\,\overline{S_0}\, D_0 + \overline{S_1} S_0\, D_1 + S_1 \overline{S_0}\, D_2 + S_1 S_0\, D_3$$

minterm m0 minterm m1 minterm m2 minterm m3

Obviously, the input selected to be passed to the output depends on the minterm expressions of the select inputs.

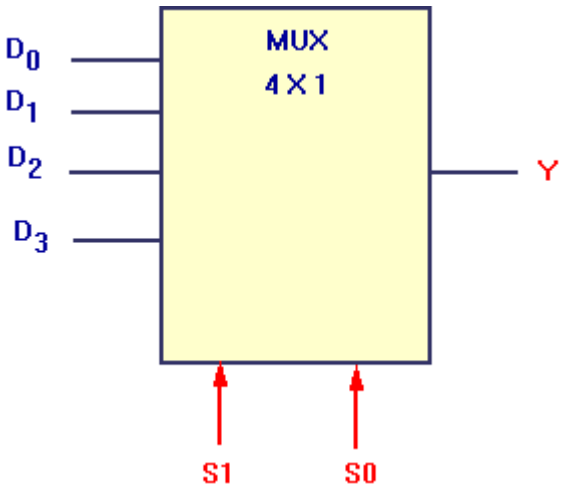


Figure 3: A 4 X 1 Multiplexer

In General,

For MUXes with n select inputs, the output Y is given by

$$\mathbf{Y = m_0D_0 + m_1D_1 + m_2D_2 + \ldots + m_{2^n-1}D_{2^n-1}}$$

Where $m_i = i^{th}$ *minterm* of the *Select Inputs*

Thus

$$Y = \sum_{i=0}^{2^n-1} m_i \, D_i$$

Example 3: Quad 2X1 Mux

Given two 4-bit numbers A and B, design a multiplexer that selects one of these 2 numbers based on some select signal S. Obviously, the output (Y) is a 4-bit number.

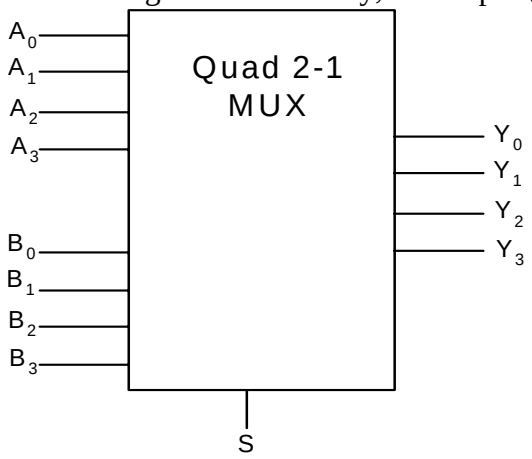


Figure 4: Quad 2 X 1 Multiplexer

The 4-bit output number Y is defined as follows:

Y = A **IF** S=0, **otherwise** Y = B

The circuit is implemented using four 2x1 Muxes, where the output of each of the Muxes gives one of the outputs (Y_i).

Combinational Circuit Implementation using Muxes

Problem Statement:

Given a function of n-variables, show how to use a MUX to implement this function. This can be accomplished in one of 2 ways:

- Using a Mux with n-select inputs
- Using a Mux with n-1 select inputs

Method 1: Using a Mux with n-select inputs

n variables need to be connected to *n* select inputs. For a MUX with *n* select inputs, the output Y is given by:

$\mathbf{Y = m_0 D_0 + m_1 D_1 + m_2 D_2 + ... + m_{2^n-1} D_{2^n-1}}$
Alternatively,

$$Y = \sum_{i=0}^{2^n-1} m_i D_i$$

Where $m_i = i^{th}$ *minterm* of the *Select Inputs*

The MUX output expression is a *SUM of minterms* expression for all minterms (m_i) which have their corresponding inputs (D_i) equal to 1.

Thus, it is possible to implement any function of *n-variables* using a MUX with *n*-select inputs by proper assignment of the input values ($D_i \in \{0 , 1\}$).
$Y(S_{n-1} \ldots S_1 S_0) = \sum(minterms)$

Example 4: Implement the function F (A, B, C) $= \sum(1, 3, 5, 6)$ (see Figure 5)
Since number of variables n = 3, this requires a Mux with 3 select inputs, i.e. an 8x1 Mux

The most significant variable A is connected to the most significant select input S_2 while the least significant variable C is connected to the least significant select input S_0 , thus:

$S_2 = A$, $S_1 = B$, and $S_0 = C$

For the MUX output expression (sum of minterms) to include minterm 1 we assign $D_1 = 1$

Likewise, to include minterms 3, 5, and 6 in the sum of minterms expression while excluding minterms 0, 2, 4, and 7, the following input (D_i) assignments are made

$D_1 = D_3 = D_5 = D_6 = 1$
$D_0 = D_2 = D_4 = D_7 = 0$

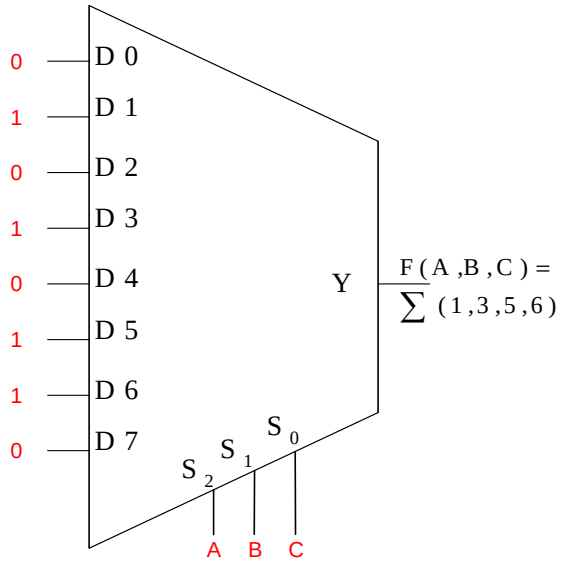


Figure 5: Implementing function with Mux with n select inputs

Method 2: Using a Mux with (n-1) select inputs

Any n-variable logic function can be implemented using a Mux with only (n-1) select inputs (e.g 4-to-1 mux to implement any 3 variable function)

This can be accomplished as follows:

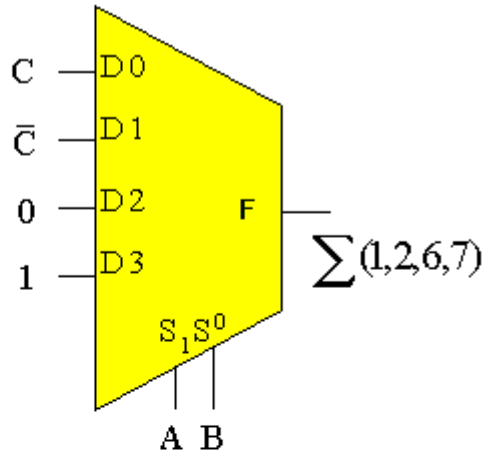
- Express function in canonical sum-of-minterms form.
- Choose n-1 variables to be connected to the mux select lines.
- Construct the truth table of the function, but grouping the n-1 select input variables together (e.g. by making the n-1 select variables as most significant inputs).

The values of D_i (mux input line) will be 0, or 1, or n^{th} variable or complement of n^{th} variable of value of function F, as will be clarified by the following example.

Example 5: Implement the function F (A, B, C) = ∑ (1, 2, 6, 7) (see figure 6)
This function can be implemented with a 4-to-1 line MUX.
A and B are applied to the select line, that is

$$A \Rightarrow S_1, B \Rightarrow S_0$$

The truth table of the function and the implementation are as shown:



	A	B	C	F	
1	0	0	0	0	F = C
	0	0	1	1	
2	0	1	0	1	F = C′
	0	1	1	0	
3	1	0	0	0	F = 0
	0	1	1	0	
4	1	1	0	1	F = 1
	1	1	1	1	

Figure 6: Implementing function with Mux with n-1 select inputs

Example 6: Consider the function F(A,B,C,D)=∑(1,3,4,11,12,13,14,15)

This function can be implemented with an 8-to-1 line MUX (see Figure 7)
A, B, and C are applied to the select inputs as follows:

$$A \Rightarrow S_2 \text{ , } B \Rightarrow S_1, C \Rightarrow S_0$$

The truth table and implementation are shown.

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

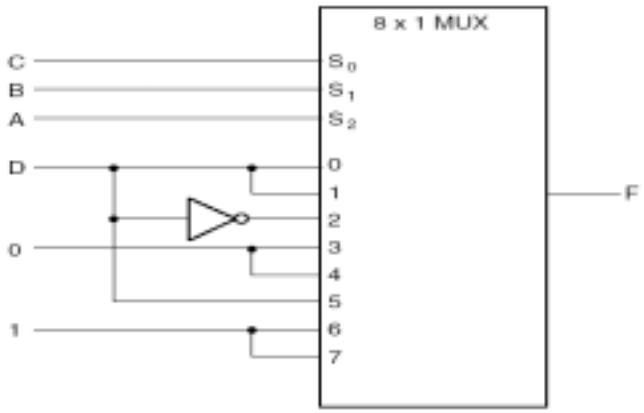


Figure 7: Implementing function of Example 6

Demultiplexer

It is a digital function that performs inverse of the multiplexing operation.
It has one input line (E) and transmits it to one of 2^n possible output lines (D_0, D_1, D_2, …, D_{2^n-1}). The selection of the specific output is controlled by the bit combination of *n* select inputs.

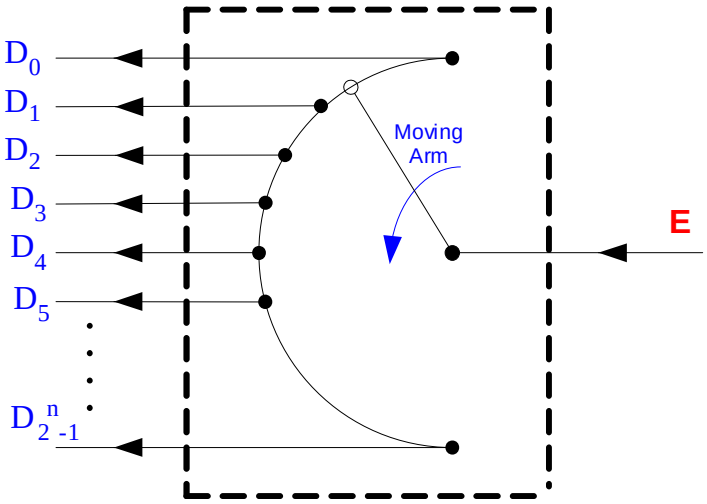


Figure 8: A demultiplexer

Example 7: A 1-to-4 line Demux

The input E is directed to one of the outputs, as specified by the two select lines S_1 and S_0.

$D_0 = E$ if $S_1S_0 = 00 \Rightarrow D_0 = S_1' S_0' E$

$D_1 = E$ if $S_1S_0 = 01 \Rightarrow D_1 = S_1' S_0 E$

$D_2 = E$ if $S_1S_0 = 10 \Rightarrow D_2 = S_1 S_0' E$

$D_3 = E$ if $S_1S_0 = 11 \Rightarrow D_3 = S_1 S_0 E$

A careful inspection of the Demux circuit shows that it is identical to a 2 to 4 decoder with enable input.

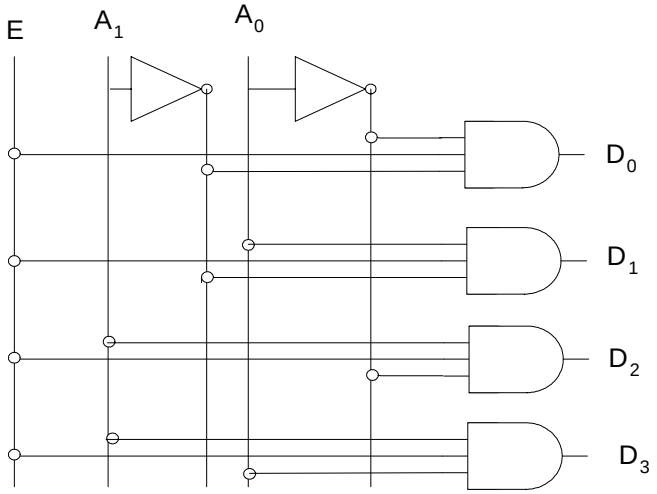


Figure 8: A 1-to-4 line demultiplexer

- For the decoder, the inputs are A_1 and A_0, and the enable is input E. (see figure 9)
- For demux, input E provides the data, while other inputs accept the selection variables.
- Although the two circuits have different applications, their logic diagrams are exactly the same.

Decimal value	Enable	Inputs		Outputs			
	E	**A_1**	**A_0**	**D_0**	**D_1**	**D_2**	**D_3**
	0	**X**	**X**	**0**	**0**	**0**	**0**
0	**1**	**0**	**0**	**1**	**0**	**0**	**0**
1	**1**	**0**	**1**	**0**	**1**	**0**	**0**
2	**1**	**1**	**0**	**0**	**0**	**1**	**0**
3	**1**	**1**	**1**	**0**	**0**	**0**	**1**

Figure 9: Table for 1-to-4 line demultiplexer

Magnitude Comparator

In this lesson you will learn about
1.Magnitude comparator
2.How to design a 4-bit comparator

Definition

A magnitude comparator is a combinational circuit that compares two numbers A & B to determine whether:

- A > B, or
- A = B, or
- A < B

Inputs

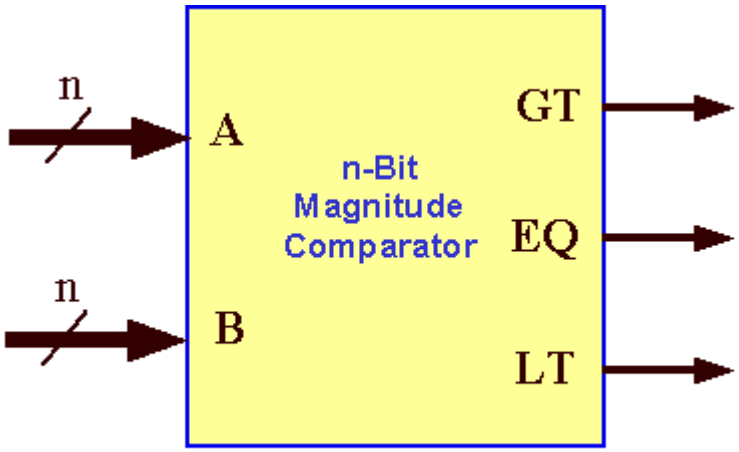
First *n*-bit number A
Second *n*-bit number B

Outputs

3 output signals (GT, EQ, LT), where:

1. GT = 1 IFF A > B
2. EQ = 1 IFF A = B
3. LT = 1 IFF A < B

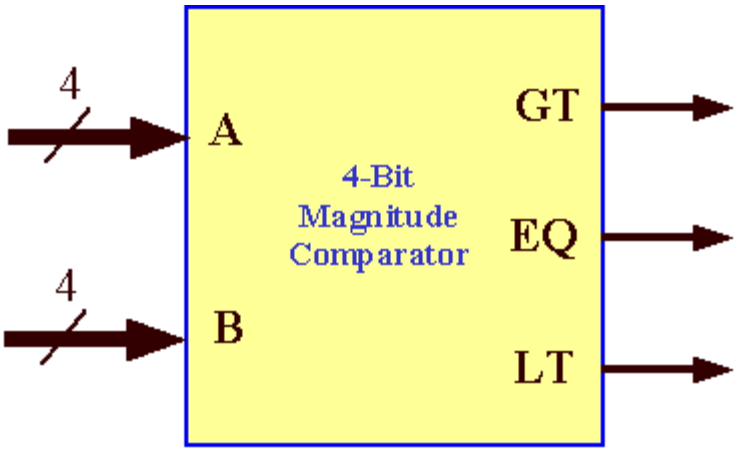
Note: *Exactly One* of these 3 outputs equals 1, while the other 2 outputs are 0`s



4-bit magnitude comparator

Inputs: 8-bits (A ⇒ 4-bits , B ⇒ 4-bits)
A and B are two 4-bit numbers

- Let A = $A_3A_2A_1A_0$, and
- Let B = $B_3B_2B_1B_0$
- Inputs have 2^8 (*256*) possible combinations
- Not easy to design using conventional techniques



The circuit possesses certain amount of regularity ⇒ *can be designed algorithmically*.

Design of the EQ output (A = B) in 4-bit magnitude comparator

Define $X_i = (A_i\ B_i) + (A_i'\ B_i')$

Thus $X_i = 1$ IFF $A_i = B_i$ $\quad \forall i = 0, 1, 2 \text{ and } 3$
$X_i = 0$ IFF $A_i \neq B_i$

Condition for A= B
EQ=1 (i.e., *A*=*B*) IFF

1. $A_3=B_3 \rightarrow (X_3 = 1)$, and
2. $A_2=B_2 \rightarrow (X_2 = 1)$, and
3. $A_1=B_1 \rightarrow (X_1 = 1)$, and
4. $A_0=B_0 \rightarrow (X_0 = 1)$.

Thus, *EQ*=1 IFF $X_3\ X_2\ X_1\ X_0 = 1$. In other words, $EQ = X_3\ X_2\ X_1\ X_0$

Design of the GT output (A > B) 4-bit magnitude comparator

If $A_3 > B_3$, then A > B (GT=1) irrespective of the relative values of the other bits of A & B. Consider, for example, A = 1000 and B = 0111 where A > B.
This can be stated as GT=1 if $\mathbf{A_3\ B_3'} = \mathbf{1}$

If $A_3 = B_3$ $(X_3 = 1)$, we compare the next significant pair of bits (A_2 & B_2).

If $A_2 > B_2$ then A > B (GT=1) irrespective of the relative values of the other bits of A & B. Consider, for example, A = 0100 and B = 0011 where A > B.
This can be stated as GT=1 if $\mathbf{X_3 A_2\ B_2'} = \mathbf{1}$

If $A_3 = B_3$ $(X_3 = 1)$ and $A_2 = B_2$ $(X_2 = 1)$, we compare the next significant pair of bits (A_1 & B_1).

If $A_1 > B_1$ then A > B (GT=1) irrespective of the relative values of the remaining bits A_0 & B_0. Consider, for example, A = 0010 and B = 0001 where A > B
This can be stated as GT=1 if $\mathbf{X_3\, X_2 A_1\, B_1'=1}$

If $A_3 = B_3$ ($X_3 = 1$) and $A_2 = B_2$ ($X_2 = 1$) and $A_1 = B_1$ ($X_1 = 1$), we compare the next pair of bits (A_0 & B_0).
If $A_0 > B_0$ then A > B (GT=1). This can be stated as GT=1 if $\mathbf{X_3X_2X_1A_0B_0'=1}$

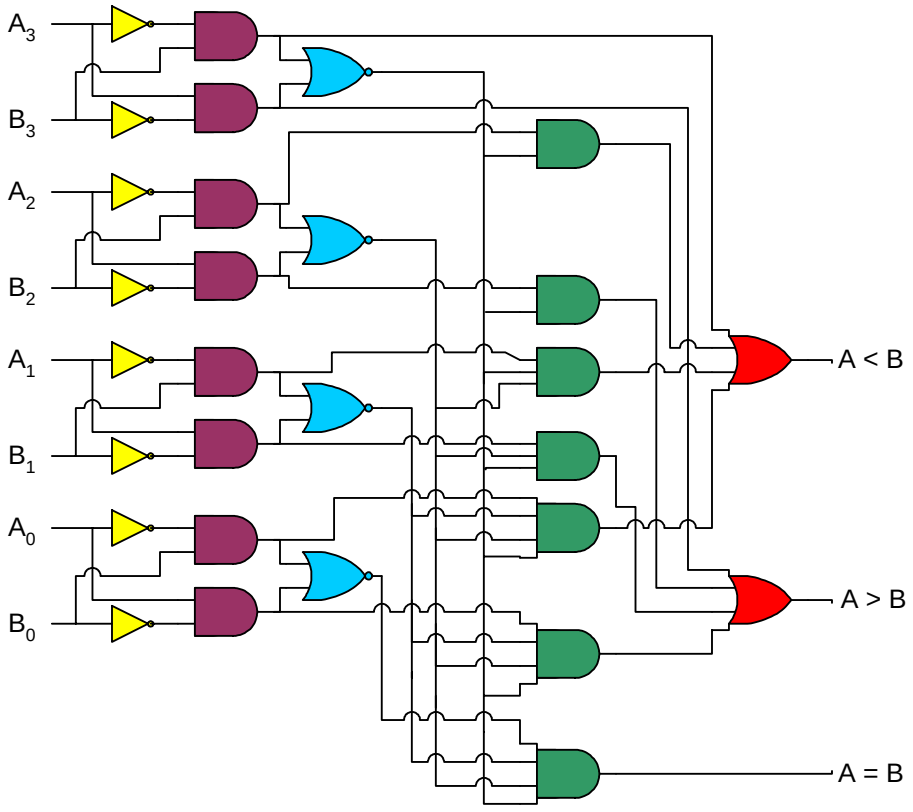
To summarize, GT =1 (A > B) IFF:

1. $\mathbf{A_3\, B_3'=1}$, or
2. $\mathbf{X_3A_2\, B_2'=1}$, or
3. $\mathbf{X_3\, X_2A_1\, B_1' = 1}$, or
4. $\mathbf{X_3X_2X_1A_0B_0' =1}$

In other words, $\mathbf{GT = A_3\, B_3' + X_3A_2\, B_2' + X_3\, X_2A_1\, B_1' + X_3X_2X_1A_0B_0'}$

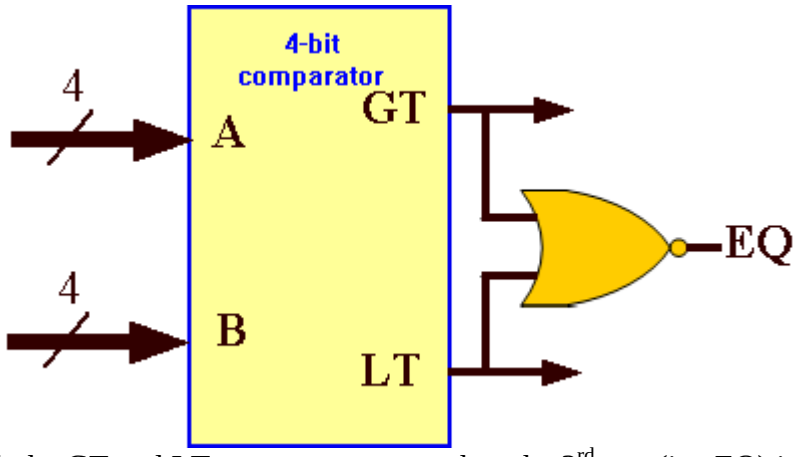
Design of the LT output (A < B) 4-bit magnitude comparator

In the same manner as above, we can derive the expression of the LT (A < B) output
$\mathbf{LT = B_3\, A_3' + X_3B_2\, A_2' + X_3\, X_2B_1\, A_1' + X_3X_2X_1B_0A_0'}$
The gate implementation of the three output variables (EQ, GT & LT) is shown in the figure below.



Modification to the Design

The hardware in the comparator can be reduced by implementing only two outputs, and the third output can be obtained using these two outputs.
For example, if we have the LT and GT outputs, then the EQ output can be obtained by using only a NOR gate, as shown in the figure below.



Thus, when both the GT and LT outputs are zeros, then the 3rd one (i.e. EQ) is a '1'

MSI Design Examples

In this lesson, you will see some design examples using MSI devices. These examples are:

- Designing a circuit that adds three 4-bit numbers.
- Design of a 4-to-16 Decoder using five 2-to-4 Decoders with enable inputs.
- Design of a circuit that takes 2 unsigned 4-bit numbers and outputs the larger of both.
- Designing a 16-bit adder using four 4-bit adders.
- Designing a 3-bit excess-3 code converter using a Decoder and an Encoder.

Designing a circuit that adds three 4-bit numbers

Recall that a 4-bit binary adder adds two binary numbers, where each number is of 4 bits. For adding three 4-bit numbers we have:

Inputs

- First *4*-bit number X = $X_3X_2X_1X_0$
- Second *4*-bit number Y = $Y_3Y_2Y_1Y_0$
- Third *4*-bit number Z = $Z_3Z_2Z_1Z_0$

Outputs

The summation of X, Y, and Z. How many output lines are exactly needed will be discussed as we proceed.

To design a circuit using MSI devices that adds three 4-bit numbers, we first have to understand how the addition is done. In this case, the addition will take place in two steps, that is, we will first add the first two numbers, and the resulting sum will be added to the third number, thus giving us the complete addition.
Apparently it seems that we will have to use two 4-bit adders, and probably some extra hardware as well. Let us analyze the steps involved in adding three 4-bit numbers.

Step 1: Addition of X and Y

A 4-bit adder is required. This addition will result in a sum and a possible carry, as follows:

$X_3X_2X_1X_0$
$Y_3Y_2Y_1Y_0$

C_4 $S_3S_2S_1S_0$

Note that the input carry C_{in} = 0 in this 4-bit adder

Step 2: Addition of S and Z

This resulting partial sum (i.e. $S_3S_2S_1S_0$) will be added to the third 4-bit number $Z_3Z_2Z_1Z_0$ by using another 4-bit adder as follows, resulting in a final sum and a possible carry:

$S_3S_2S_1S_0$
$Z_3Z_2Z_1Z_0$

D_4 $F_3F_2F_1F_0$

where $F_3F_2F_1F_0$ represents the final sum of the three inputs X, Y, and Z. Again, in this step, the input carry to this second adder will also be zero.

Notice that in Step 1, a carry C_4 was generated in bit position 4, while in Step 2, another carry D_4 was generated also in bit position 4. These two carries must be added together to generate the final Sum bits of positions 4 and 5 (F_4 and F_5).
Adding C_4 and D_4 requires a half adder. Thus, the output from this circuit will be six bits, namely F_5 F_4 $F_3F_2F_1F_0$ (See Figure 1)

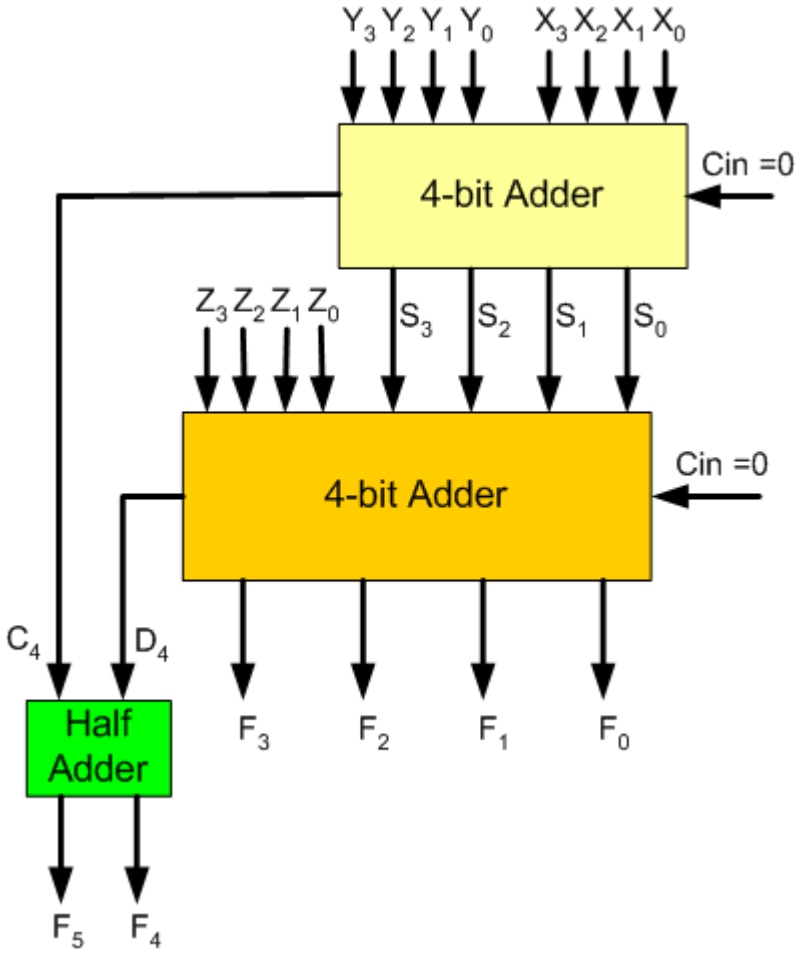


Figure 1: Circuit for adding three 4-bit numbers

Design a 4-to-16 Decoder using five 2-to-4 Decoders with enable inputs

We have seen how can we construct a bigger decoder using smaller decoders, by taking the specific example of designing a 3-to-8 decoder using two 2-to-4 decoders. Now we will design a 4-to-16 decoder using five 2-to-4 decoders.

There are a total of sixteen possible input combinations, as shown in the table (Figure 2). These sixteen combinations can be divided into four groups, each group containing four combinations. Within each group, A_3 and A_2 remain constant, while A_1 and A_0 change their values. Also, in each group, same combination is repeated for A_1 and A_0 (i.e. 00→01→10→11)

A_3	A_2	A_1	A_0
0	0	0	0
0	0	0	1
0	0	1	0
0	0	1	1
0	1	0	0
0	1	0	1
0	1	1	0
0	1	1	1
1	0	0	0
1	0	0	1
1	0	1	0
1	0	1	1
1	1	0	0
1	1	0	1
1	1	1	0
1	1	1	1

Figure 2: Combinations with 4 variables

Thus we can use a 2-to-4 decoder for each of the groups, giving us a total of four decoders (since we have sixteen outputs; each decoder would give four outputs). To each decoder, A_1 and A_0 will go as the input.

A fifth decoder will be used to select which of the four other decoders should be activated. The inputs to this fifth decoder will be A_3 and A_2. Each of the four outputs of this decoder will go to each enable of the other four decoders in the "proper order".

This means that line 0 (representing $A_3A_2 = 00$) of decoder '5' will go to the enable of decoder '1'. Line 1 (representing $A_3A_2 = 01$) of decoder '5' will go to the enable of decoder '2' and so on.

Thus a combination of A_3 and A_2 will decide which "group" (decoder) to select, while the combination of A_1 and A_0 will decide which output line of that particular decoder is to be selected.

Moreover, the enable input of decoder '5' will be connected to logic switch, which will provide logic 1 value to activate the decoder.

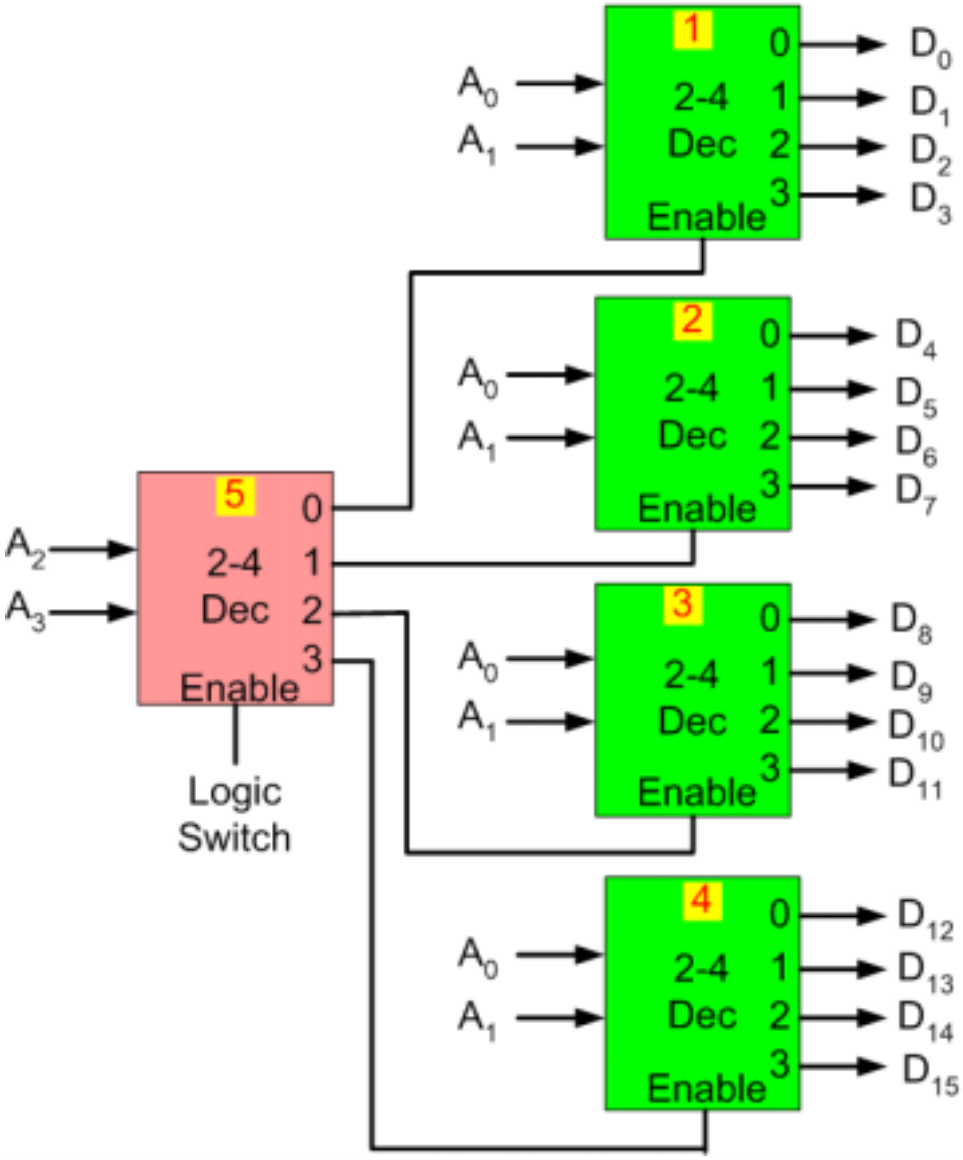


Figure 3: Constructing 4-to-16 decoder using 2-to-4 decoders

Decoder example: "Activate" line D_2. The corresponding input combination that would activate this line is 0010. Now apply 00 at input of decoder '5'. This activates line '0' connected to enable of decoder '1'. Once decoder '1' is activated, inputs at A_1A_0 = 10 activate line D_2.
Thus we get the effect of a 4-16 decoder using this design, by applying input combinations in two steps.

As another example, to "activate" the line D_{10}: The corresponding input combination is 1010. Apply 10 at the input of decoder '5'. This activates line '2' connected to enable of decoder '3'. Once decoder '3' is activated, the inputs at A_1A_0 = 10 activate line D_{10}.

Given two 4-bit unsigned numbers A and B, design a circuit which outputs the larger of the 2 numbers.

Here we will use Quad 2-1 Mux, and a 4-bit magnitude comparator. Both of these devices have been discussed earlier. The circuit is given in the figure

Since we are to select one of the two 4-bit numbers A ($A_3A_2A_1A_0$) and B ($B_3B_2B_1B_0$), it is obvious that we will need a quad 2-1 Mux.

The inputs to this Mux are the two 4-bit numbers A and B.

The select input of the Mux must be a signal which indicates the relative magnitude of the two numbers A and B. This signal may be True if A<B or if A>B.

Such signal is easily obtained from a 4-bit magnitude comparator.

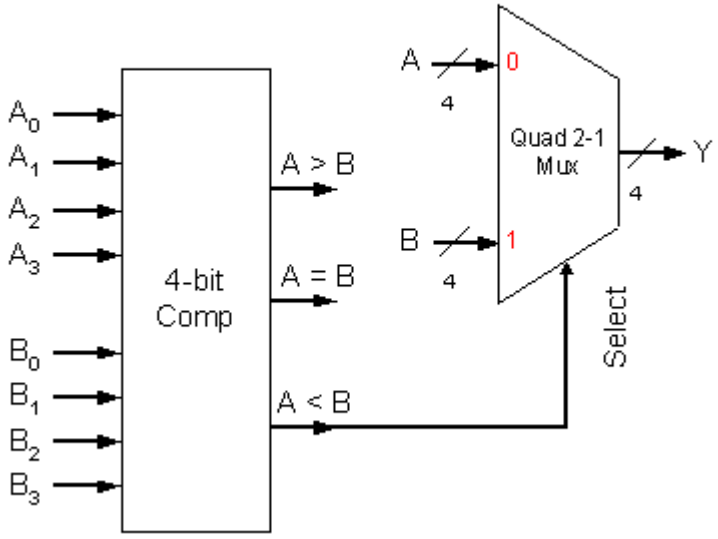


Figure 4: Circuit that outputs the larger of two numbers

By connecting the select input to the A<B output of the magnitude comparator, we must connect A to the 0 input of the Mux and B to the 1 input of the Mux . Alternatively, if we connect the select input to the A>B output of the magnitude comparator, we must connect A to the 1 input of the Mux and B the 0 input of the Mux . In either case, the Mux output will be the larger of the two numbers

Designing a 16-bit adder using four 4-bit adders

Adds two 16-bit numbers X (X_0 to X_{15}), and Y (Y_0 to Y_{15}) producing a 16-bit Sum S (S_0 to S_{15}) and a carry out C_{16} as the most significant position. Thus, four 4-bit adders are connected in cascade.

Each adder takes four bits of each input (X and Y) and generates a 4-bit sum and a carry that is fed into the next 4-bit adder as shown in Figure 5.

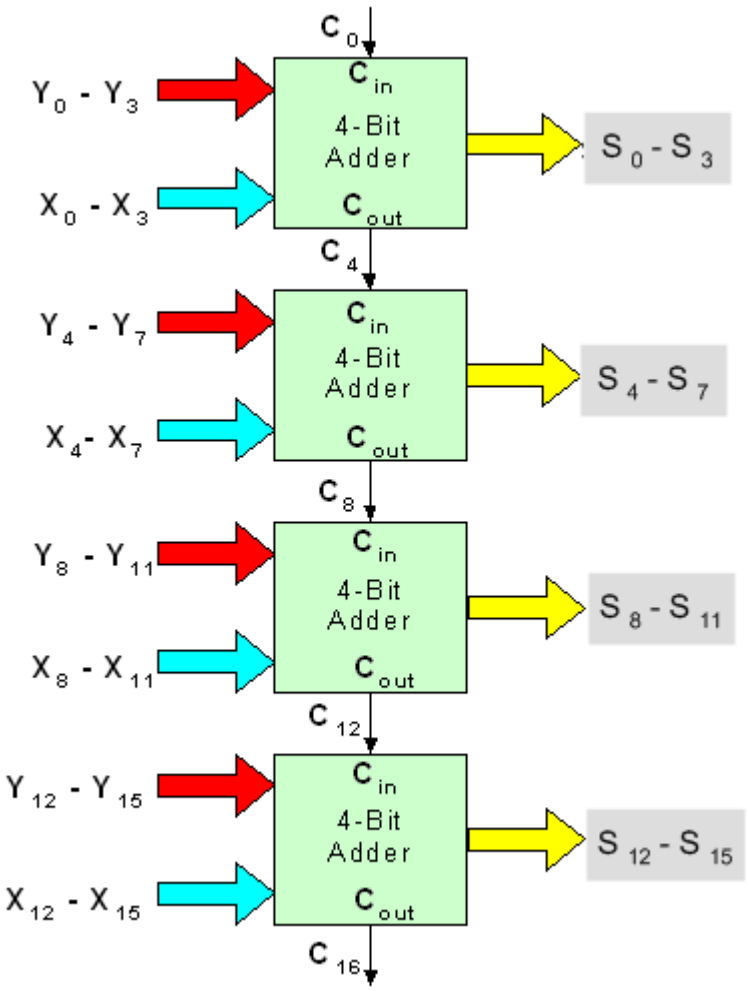


Figure 5: A 16-bit adder

Designing an Excess-3 code converter using a Decoder and an Encoder

In this example, the circuit takes a BCD number as input and generates the corresponding Ex-3 code. The truth table for this circuit is given in figure 6.
The outputs 0000, 0001, 0010, 1101, 1110, and 1111 are never generated (Why?)

To design this circuit, a 4-to-16 decoder and a 16-to-4 encoder are required. The design is given in figure 7. In this circuit, the decoder takes 4 bits as inputs, represented by variables w, x, y, and z. Based on these four bits, the corresponding minterm output is activated. This decoder output then goes to the input of encoder which is three greater than the value generated by the decoder.

The encoder then encodes the value and sends the output bits at A, B, C, and D. For example, suppose 0011 is sent as input. This will activate minterm 3 of the decoder. This

output is connected to input 6 of encoder. Thus the encoder will generate the corresponding bit combination, which is 0110.

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

Figure 6: table for BCD to Ex-3 conversion

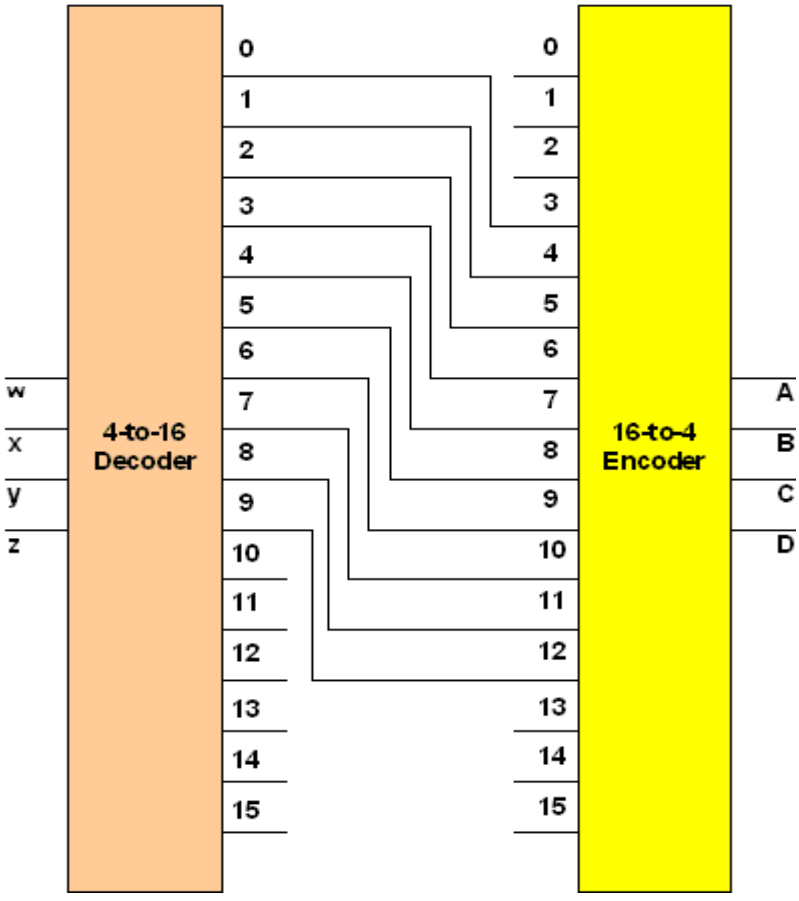


Figure 7: Circuit for BCD to Ex-3 conversion

Sequential Circuits

Objective

- In this lesson, you will learn about:

    1. Sequential Circuits, Synchronous Sequential Circuits and Memory Elements.
    2. Clocked RS, D, JK, & T latches with their analysis.
    3. Characteristic and excitation behavior of these latches.

Introduction

- This is an introductory lesson on sequential logic circuits.
- The general block diagram of a combinational circuit is shown in Figure 1.
- A Combinational logic circuit consists of input variables (X), logic gates (Combinational Circuit), and output variables (Z).
-

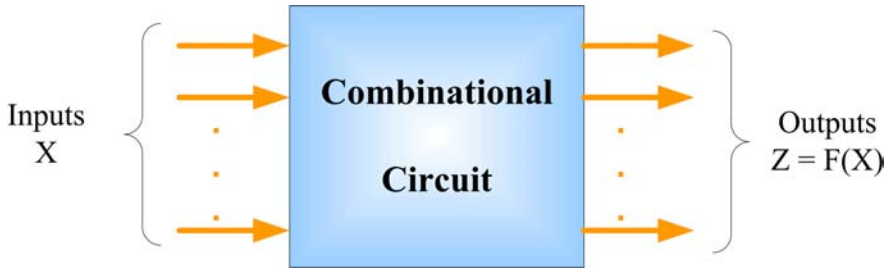


Figure 1: General Block Diagram of a Combinational Circuit

- Unlike combinational circuits, sequential circuits include **memory elements** (See Figure 2).
- The memory elements are circuits capable of storing binary information.
- The binary information stored in these memory elements at any given time defines the **state** of the sequential circuit at that time.
- The outputs, **Z**, of a sequential circuit depends both on the present inputs, **X**, and the present state **Y** (i.e., information stored in the memory elements).
- The next state of the memory elements also depends on the inputs **X** and the present state **Y**.

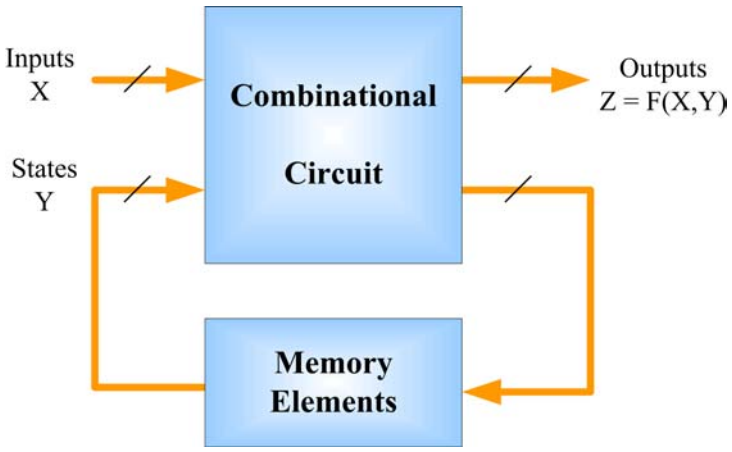


Figure 2: General Block Diagram of a Sequential Circuit

Sequential Adder

- To best understand sequential circuits, let's re-visit a known iterative circuit, a 4-bit combinational ripple carry adder (See Figure 3).

- The combinational circuit of a 4-bit ripple carry adder comprises 4-full adders. The inputs to the circuit are a single-bit *carry-in* (C_{IN}) & two 4-bit numbers A & B. This circuit produces a 4-bit sum S & a single-bit *carry-out* (C_{OUT}).

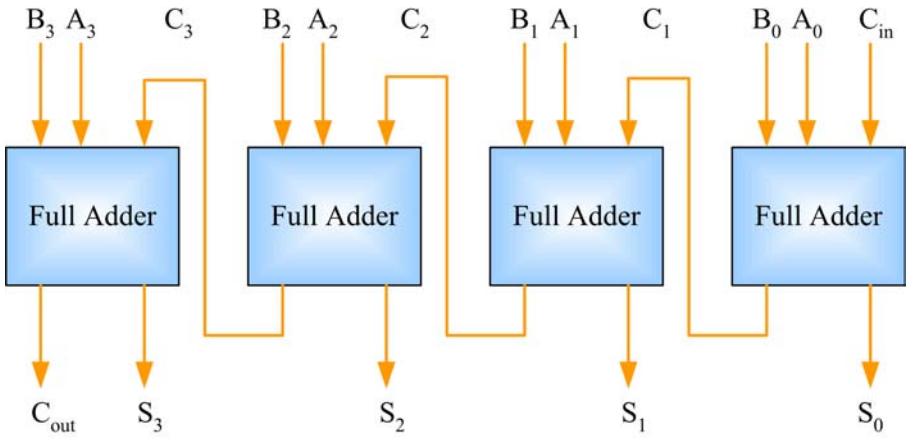


Figure 3: 4-bit Ripple-Carry Adder

- We can notice that all 4-bits of the sum are not computed at the same instance of time. The 1st stage produces the LSB of the sum, S_0, and an intermediate carry C_0 using C_{IN} and the LSB of A & B (A_0, B_0).

- The 2nd stage, using the intermediate carry C_0 along with A_1 and B_1, produces the 2nd bit of the sum, S_1. In this way, the intermediate carry propagates through the stages of the adder & each stage, on the arrival of this carry, produces its corresponding bit of final sum S.

- We observe that only one stage is *active* during the computation of the sum. Based on this observation, we can make an *n*-bit adder using only one stage

full-adder as shown in Figure 4.

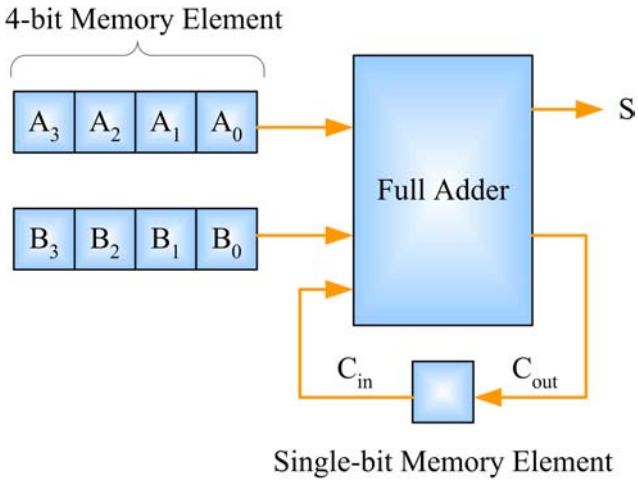


Figure 4: 4-bit Sequential Adder

- However, we need a single-bit memory element to temporarily store the value of the intermediate carry.
- Two 4-bit memory elements are used to store bit-vectors A and B while a single-bit memory element is used to store the intermediate carry.
- As we have only one full adder, it will take four instances of time to add the corresponding bits of A and B.
- We notice here that the sequential adder has one memory element, which stores the state of the circuit as carry. These states define the condition of having a carry or no carry.
- In other words, to define 2-states (0 and 1) in a sequential circuit, we require 1 memory element. In general, for an *n*-state circuit we require $\lceil \log_2 n \rceil$ memory elements.
- We also notice that to move from one state to another, we need a periodic signal, which we called the **Clock**, to synchronize the activity.

Synchronous & Asynchronous Sequential Circuits

- There are two main types of sequential circuits. Their classification depends on the timing of their signals.
- **Synchronous sequential circuits** are systems whose behaviors can be defined from the knowledge of their signals at discrete instants of time.
- While the behavior of **asynchronous sequential circuits** depends upon the order in which their input signals change at any instant of time.

- **Synchronous** sequential logic systems must employ signals that affect the memory elements only at discrete instants of time.
- To achieve this goal, a timing device called a master-clock generator is used to generate a periodic train of Clock pulses.
- These clock pulses are distributed throughout the system in such a way that memory elements are affected only with arrival of the Clock pulse.

Memory Elements

- A basic memory element, as shown in Figure 5 (a), is the **latch**.
- A latch is a circuit capable of storing one bit of information.
- The latch circuit consists of two inverters; with the output of one connected to the input of the other.
- The latch circuit has two outputs, one for the stored value (**Q**) and one for its complement (**Q'**).
- Figure 5 (b) shows the same latch circuit re-drawn to illustrate the two complementary outputs.
- The problem with the latch formed by **NOT** gates is that we can't change the stored value. For example, if the output of inverter B has logic 1, then it will be latched forever; and there is no way to change this value.

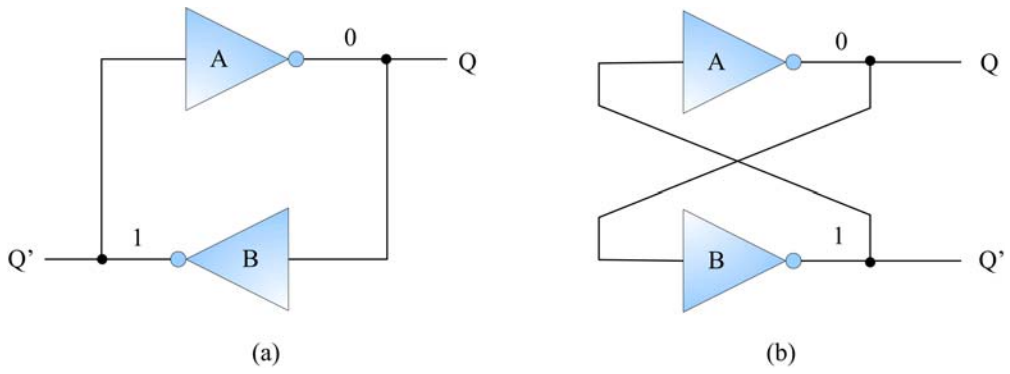


Figure 5: Simple Latch

SR Latch

- Recall that a NOT gate can alternatively be expressed using **NAND** and **NOR** gates as shown in Figure 6 (a).
- Using NOR gates, we can obtain the latch circuit shown in Figure 6 (b).
- This latch has two outputs, **Q** and **Q'**, and two inputs **S** and **R**.
- This type of latches is sometimes called a cross-coupled SR latch or simply SR latch.

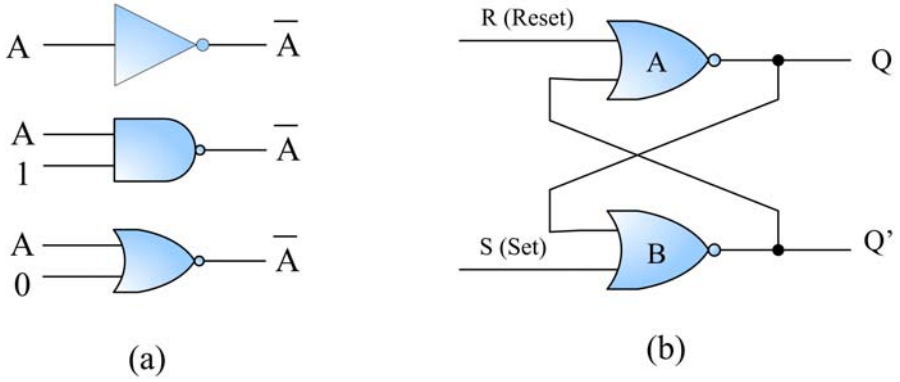


Figure 6: (a) Alternative forms of NOT gate (b) Basic SR latch with NOR gates

Table 1: Functional Table of the Basic SR Latch with NOR Gates

S	R	Q	Q'	
1	0	1	0	Set State
0	0	1	0	
0	1	0	1	Reset State
0	0	0	1	
1	1	0	0	Undefined

- The SR latch has two main states: **set** and **reset** (See Table 1).
- When output **Q=1** and **Q'=0**, the latch is said to be in the **set state**; and when **Q=0** and **Q'=1**, it is in the **reset state**.
- When the input **S=0** and **R=0**, the SR latch remains in its current state (i.e. set or reset). In this case, the values of Q and Q' are latched forever.
- When the SR latch is in the set state, we can change the state to the reset state by making R=1.
- Similarly, the state of the SR latch can be changed from reset to set by making S=1.

- If a 1 is applied to both inputs of the SR latch, both outputs go to 0.
- This produces an undefined state, because it violates the requirement that the outputs be complement of each other.
- It also results in an indeterminate next state when both inputs return to 0 simultaneously as shown in the figure.
- In normal operation, these problems are avoided by making sure that 1's are not applied to both inputs simultaneously.

SR Latch with NAND Gates

- The SR latch with two cross-coupled **NAND** gates is shown in Figure 7.
- It operates with both inputs normally at 1, unless the state of the latch has to be changed (See Table 2).
- With both inputs at 1, applying 0 to the S input causes the output Q to go to 1 (i.e. **set state**).
- In the same way, applying 0 to the R input causes the output Q to go to 0 (i.e. **reset state**).
- The condition that **undefined** for this NAND latch is when both inputs are equal to 0 at the same time, which causes both outputs Q and Q' to go to 1.

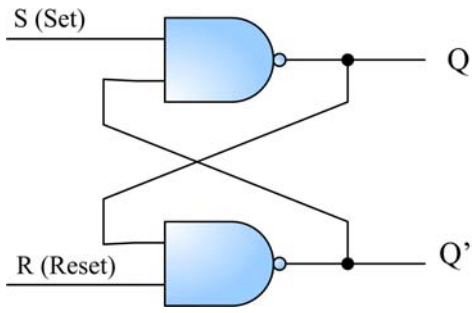


Figure 7: Basic SR LATCH with NAND Gates

Table 2: Functional Table of the Basic SR Latch with NAND Gates

S	R	Q	Q'	
0	1	1	0	Set State
1	1	1	0	
1	0	0	1	Reset State
1	1	0	1	
0	0	0	0	Undefined

Clocked SR Latch

- The operation of the basic SR latch can be modified by providing an additional control input (**clock**) that determines when the state of the latch can be changed.
- An SR latch with a control input **C** is shown in Figure 8.
- It consists of the basic SR latch with two additional AND gates.
- The control input C acts as an **enable** signal to the latch (See Table 3).
- When **C=0**, the S and R inputs have no effect on the latch, so the latch will remain in the same state regardless of the values of S and R.
- When **C=1**, the S and R inputs will have the same effect as in the basic SR latch.

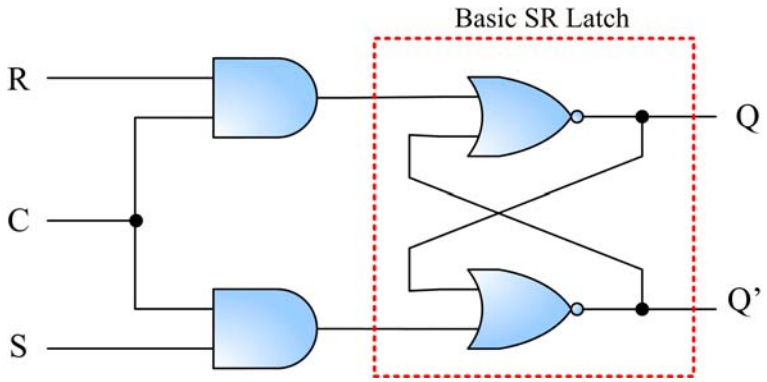


Figure 8: Clocked SR Latch

Table 3: Functional Table of Clocked SR Latch

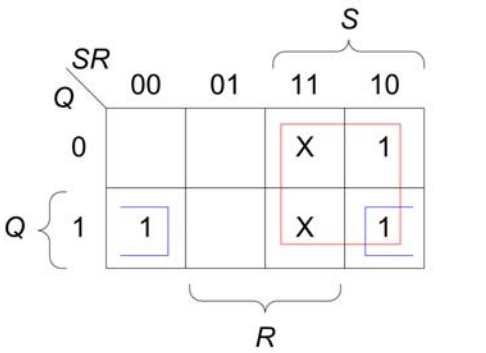
C	S	R	Next State of Q
0	X	X	No Change
1	0	0	No Change
1	0	0	Q = 0; Reset State
1	1	0	Q = 0; Set State
1	1	1	Undefined

Characteristic Table of the SR Latch

- The **characteristic** (behavior) of the sequential circuit defines its logical property by specifying the next states when the inputs and the present states are known. The characteristic of the RS latch is shown in Table 4.
- The characteristic table can also be represented algebraically using what is known as a **characteristic equation**.
- The characteristic equation is derived using the K-Map as shown in Figure 9.
- X's mark the two indeterminate states in the map in Figure 9, since their inputs are never allowed (Recall "Don't Cares").
- Note that the condition S.R = 0 must also be included as both S and R cannot simultaneously be 1.
- The characteristic equations are used in the analysis of sequential circuits.

Table 4: Characteristic Table of SR Latch

Q(t)	S	R	Q(t + 1)
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	Indeterminate
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	Indeterminate



$Q(t+1) = S + R'Q \qquad SR = 0$

Figure 9: Characteristic Equation of the SR Latch

Excitation Table of the SR Latch

- During the design process we usually know the transition from present state to next state and wish to find the latch input conditions that will cause the required transition.

- For this reason, we need a table that lists the required inputs for a given change of state. Such a table is called an *excitation table*, and it specifies the excitation behavior of the sequential circuits. These are used in the synthesis (design) of sequential circuits, which we shall see later.

- The excitation of the SR latch is given in Table 5.

Table 5: Excitaion table of the SR latch

Q(t)	Q(t+1)	S	R
0	0	0	X
0	1	1	0
1	0	0	1
1	1	X	0

Clocked D-Latch

- One way to eliminate the undesirable undefined state in the SR latch is to ensure that the inputs S and R are never equal to 1 at the same time.

- This is done in the D latch shown in Figure 10.

- This latch has only two inputs **D** (**Data**) and **C** (**Clock**). Note that D is applied directly to the set input S, and its complement is applied to the reset input R.

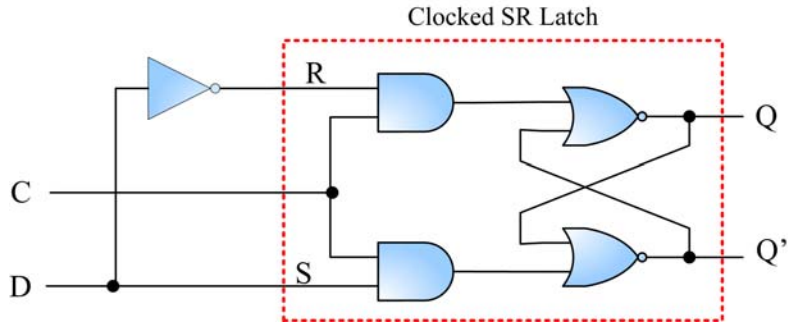


Figure 10: Clocked D Latch

- As long as the clock input **C = 0**, the SR latch has both inputs equal to 0 and it can't change its state regardless of the value of D (See Table 6).

- When C is 1, the latch is placed in the set or reset state based on the value of D.

- If **D = 1**, the **Q** output goes to **1**.
- If **D = 0**, the **Q** output goes to **0**.

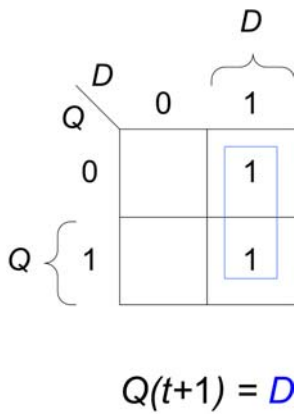
- The characteristic table and the characteristic equation of a D latch are illustrated in Table 7 and Figure 11 respectively.

Table 6: Functional Table of the D-Latch

C	D	Next State of Q
0	X	No Change
1	0	Q = 0; Reset State
1	1	Q = 1; Set State

Table 7: Characteristic Table of the D-Latch

Q(t)	D	Q(t + 1)
0	0	0
0	1	1
1	0	0
1	1	1



$Q(t+1) = D$

Figure 11: Characteristic Equation of the D-Latch

Clocked JK-Latch

- The clocked JK latch is shown in Figure 12. Note the feedback path from the outputs Q and Q' to the AND gates at the input.
- JK latch is an improvement over the SR latch in the sense that it does not have any indeterminate states.
- Inputs J and K behave like S and R of the SR latch. J and K set and clear the state of the latch, respectively.

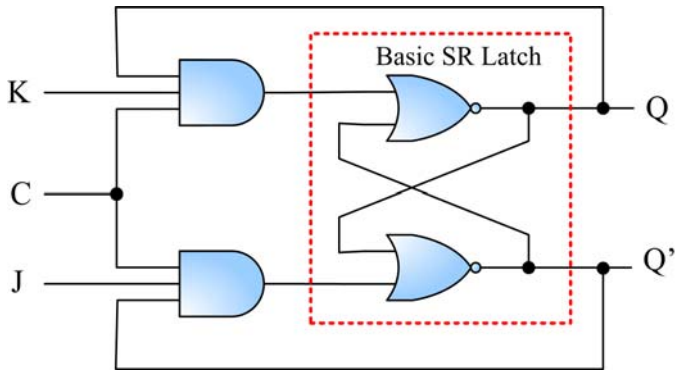


Figure 12: Clocked JK-Latch

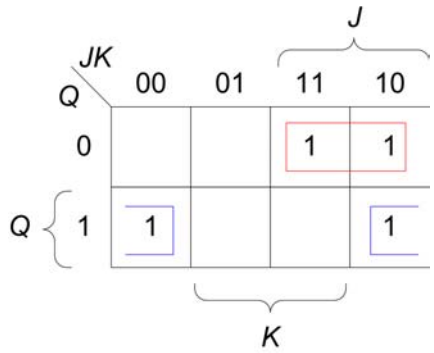
- The functional table of the clocked JK-Latch is illustrated in Table 8.
- If both J and K are made high (recall that both S and R cannot be made high at the same time) then the latch switches to its complement state, that is, if Q=1 then it switches to Q=0, and vice versa.
- Output Q is ANDed with K and C inputs so that the latch is cleared during a clock pulse only if Q was previously 1.
- Similarly, Q' is ANDed with J and C inputs so that the latch is set with a clock pulse only if Q' was previously 1.
- The JK latch behaves exactly like the SR latch, except when both J and K are 1.
- Characteristic table and characteristic equation of the JK-Latch are shown in Table 8 and Figure 13 respectively.

Table 8: Functional Table of the Clocked JK-Latch

C	J	K	Next State of Q
0	X	X	Q *(No Change)*
1	0	0	Q *(No Change)*
1	0	1	0 *(Reset State)*
1	1	0	1 *(Set State)*
1	1	1	Q' *(Complement)*

Table 9: Characteristic Table of the JK-Latch

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



$$Q(t+1) = JQ' + K'Q$$

Figure 13: Characteristic Equation of the JK-Latch

- Excitation table of the JK-Latch are illustrated in Table 9.
-
- When both states, present and the next one are to be 0, then the J input must remain at 0 and the K input can be either 0 or 1 (i.e., X).

- Similarly, when both present state and the next state are 1, the K input must remain at 0 while J input can be 0 or 1 (i.e., X).

- If the latch is to have a transition from the 0-state to 1-state, J must be equal to 1 since the J input sets the latch. However, input K may be either 0 or 1.

- Similarly, for a 1-to-0 transition, K must be set to 1 and J can be either 0 or a 1.

Table 10: Excitaion Table of the JK-Latch

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

Clocked T-Latch

- The T latch is a single-input version of the JK latch. It is obtained by tying both the inputs J and K together as shown in Figure 14. The name comes from the ability of the latch to "toggle" or change the state.

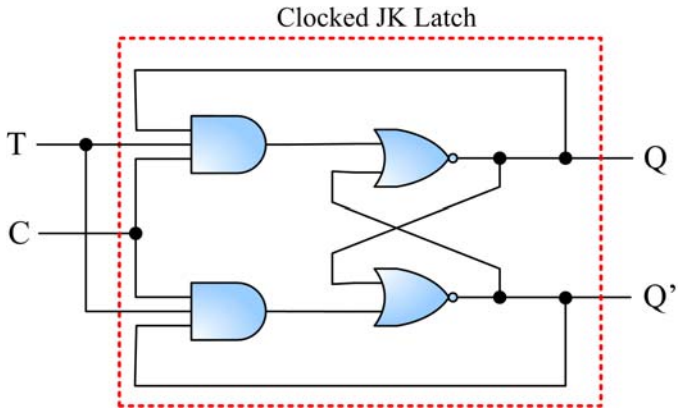


Figure 14: Clocked T-Latch

- Observe that when T=1, regardless of the present state, the latch toggles or changes to the complement state when the clock pulse occurs (See Table 11).

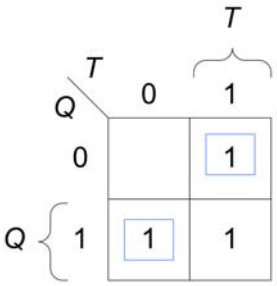
Table 11: Functional Table of the Clocked T-Latch

C	T	Next State of Q
0	X	*No Change*
1	0	*No Change*
1	1	*Q'*

- The toggling effect can be seen more clearly in the characteristic behavior of the T-Latch (See Table 12 and Figure 15). Notice that when T = 0, the state of the latch remains unchanged.

Table 12: Characteristic Table of the T-Latch

Q(t)	T	Q(t + 1)
0	0	0
0	1	1
1	0	1
1	1	0



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

Figure 15: Characteristic Equation of the T-Latch

- The excitation table of the T-Latch is illustrated in Table 12.

- Note that when the state of the latch must remain the same, the requirement is that T = 0. When the state of the latch has to be complemented, T must equal 1, as summarized in the excitation table.

Table 13: Excitation Table of the T-Latch

Q(t)	Q(t + 1)	T
0	0	0
0	1	1
1	0	1
1	1	0

Problem with the Level Triggered JK and T latches

- In JK latch, with J = 1 and K = 1 the state of the latch toggles. However, if the clock signal remains at 1 (while J = K = 1), the output will go in repeated transitions; this is an undesirable oscillating effect. And when clock goes to 0, output will be latched to an unknown state.

- To avoid this undesirable operation the clock pulse must have pulse duration, which is shorter than the propagation delay of the signal through the latch. This however is not at all acceptable since the operation of the circuit will then depend on the width of the clock pulse and/or the delay through the latch.

- For this reason, JK latches are never constructed as discussed above. The restriction on the pulse width can be eliminated with a master-slave or edge-triggered construction described in the next lesson. The same reasoning applies to the T latch.

Flip-Flops

Objectives

- The objectives of this lesson are to study:

  1. Latches versus Flip-Flops
  2. Master-Slave Flip-Flops
  3. Timing Analysis of Master-Slave Flip-Flops
  4. Different Types of Master-Slave Flip-Flops
  5. Propagation Delay

Problem with Latches

- A latch is a level sensitive device.
- Because of this the state of the latch may keep changing in circuits with feedback as long as the clock pulse remains active.
- Thus, instead of having output change once in a clock cycle, the output may change a number of times resulting in latching of unwanted input to the output.
- Due to this uncertainty, latches can not be reliably used as storage elements.

Solution to this Problem

- To overcome this problem of undesired toggling, we need to have a mechanism in which we have higher degree of control on the output of the memory element when the clock pulse changes.
- This is achieved by introducing a special clock-edge detection logic, such that the state of the memory element is switched by a momentary change in the clock pulse (i.e. an edge).
- This is effective because the clock changes only once during a clock period.
- Such a memory element is "edge-sensitive", i.e., it changes its state at the rising or falling edge of a clock.
- Edge-sensitive memory elements are called Flip-Flops.
- Figure 1 shows the standard graphic symbols for positive and negative edge triggered Flip-Flops.

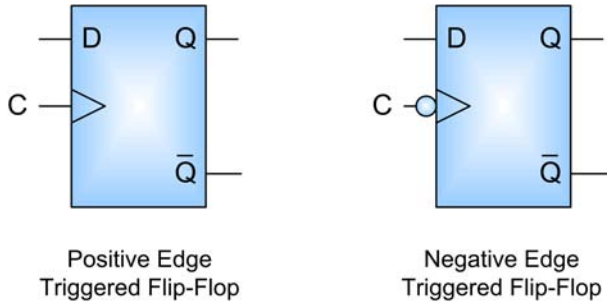


Figure 1: Graphic Symbols of Edge-Triggered Flip-Flops

Master-Slave Flip-Flops

- The simplest way to build a flip-flop is by using two latches in a 'Master-Slave' configuration as shown in Figure 2.
- In this configuration, one latch serves as the *master* receiving the external inputs and the other as a *slave,* which takes its inputs from the master.
- When the clock pulse goes high, information at S and R inputs is transmitted to master.
- The slave flip-flop however remains isolated since its control input C is 0.
- Now when the clock pulse returns to '0', the master gets disabled and blocks the external inputs to get to its outputs whereas slave gets enabled and passes the latched information to its outputs.

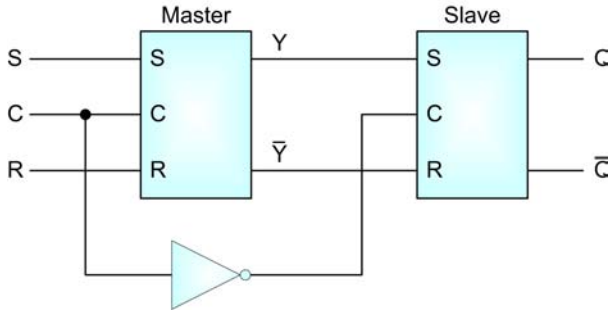


Figure 2: Block diagram of SR Master-Slave Flip-Flop

Timing Analysis of Master-Slave Flip-Flop

- Now let's view the operation of the master-slave flip-flop by analyzing its timing wave forms (See Figure 3).
- Consider a master-slave flip-flop in the clear state (i.e. Y=0 and Q=0) prior to the occurrence of a pulse.
- The inputs S=1 and R=0 are applied. So when the clock goes high, the output of the master latch will change to the set state, while the slave latch remains disabled.

- When the clock returns to 0, the master latch is disabled and the slave latch is enabled.
- Thus, the data at the slave's input when the clock was high gets latched at the slave's output.

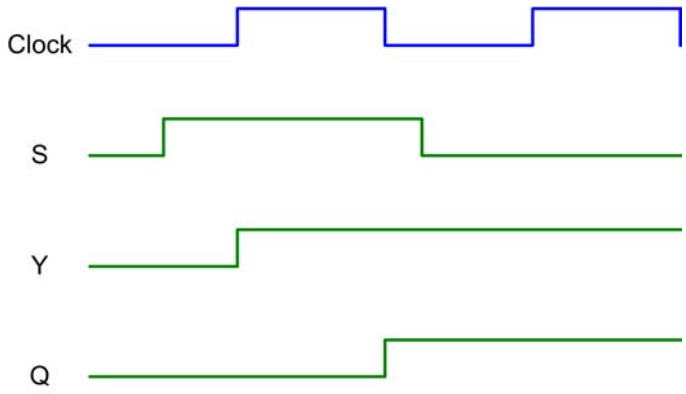


Figure 3: Timing wave form of SR Master-Slave Flip-Flop

Different Types of Master-Slave Flip-Flops

Master-Slave JK-FF

- The SR flip-flop can be modified to a JK flip-flop to eliminate the undesirable condition that leads to undefined outputs and indeterminate behavior.
- A Master-Slave JK Flip-Flop is shown in the Figure 4.
- Here, the output gets complemented when both J and K inputs are high.

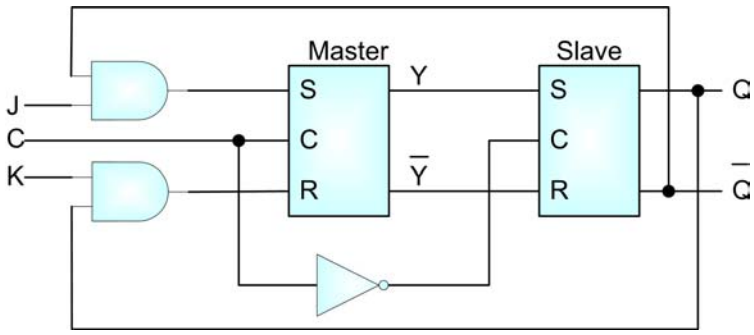


Figure 4: JK Master-Slave Flip-Flop

D-Type Positive-Edge-Triggered FF

- The logic diagram of a positive edge triggered D-type flip-flop is shown in the Figure 5.
- This flip-flop takes exactly the form of a master-slave flip-flop, with the master a D latch and the slave an SR latch. Also, an inverter is added to the clock input of the master latch.

- Because the master latch is a D latch, the flip-flop exhibits edge-triggered rather than master-slave (pulse-triggered) behavior.

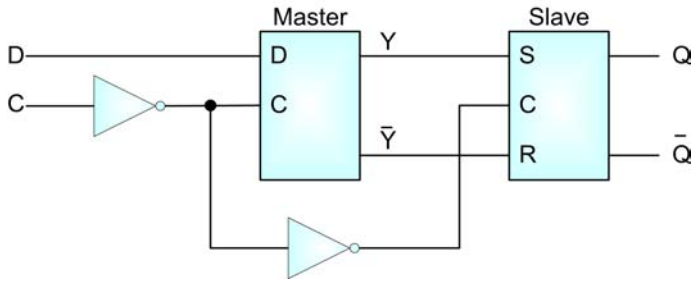


Figure 5: D-Type Positive-Edge-Triggered FF

Propagation Delay

- In digital logic, every gate has got some finite amount of delay because of which the change in the output is not instantaneous to the change in the input.
- In simple terms, the times it takes for an input to appear at the output is called the propagation delay.
- In Figure 6, $\mathbf{t_{PHL}}$, describes the time it takes for an input to cause the output to change from logic-level-high to logic-level-low.
- Similarly, $\mathbf{t_{PLH}}$, refers to the delay associated when an input change causes the output to change from logic-level-low to logic-level-high.
- The overall delay is average of these two delays.

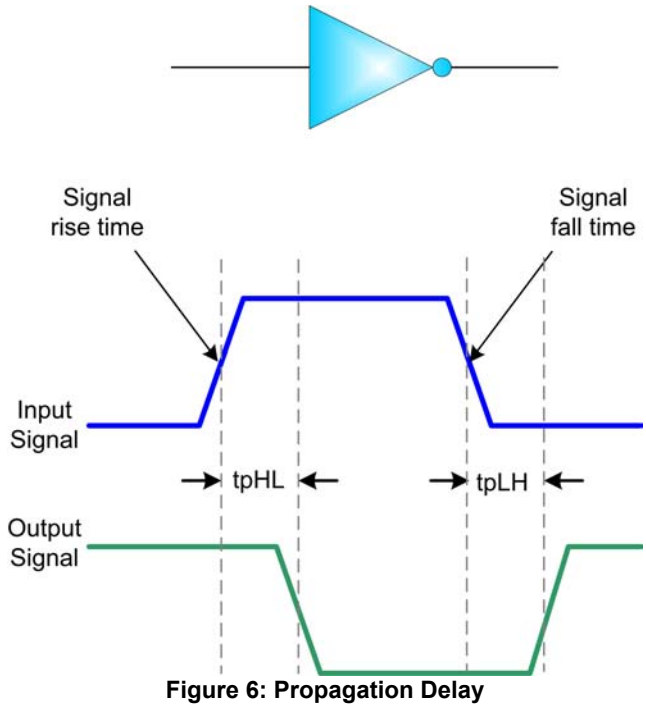


Figure 6: Propagation Delay

Setup and Hold Times

- For correct operation of logic gates we need to satisfy some timing constrains regarding application of inputs and collecting of their outputs.
- **Setup time** (T_s) refers to a constant duration for which the inputs must be held prior to the arrival of the clock transition (See Figure 7).
- Once the inputs are properly set, it must be kept for some time for their proper reading-in by the gate once the transition signal is triggered.
- **Hold time** (T_h) refers to the duration for which the inputs must not change after the arrival of the transition (See Figure 7).
- If the setup and hold times are violated, a gate may produce an unknown logic signal at its output. This condition is called as **meta-stability**.

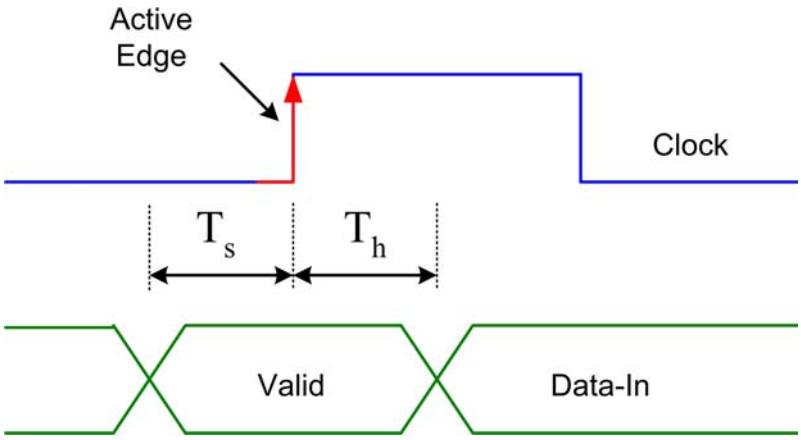


Figure 7: Setup and Hold Times

Propagation Delay

- To set or clear flip-flops asynchronously (i.e., without the use of clock and inputs) some flip-flops have direct inputs usually called **direct preset** or **direct clear**.
- These inputs are needed to bring the flip-flops to a known initial state prior to the normal clocked operation.
- A direct preset input, sets the output of a flip-flop to some known value, asynchronously, for example logic-1 or logic-0.
- A direct clear switch clears or resets all the flip-flops to logic value-0.
- Figure 8 shows the graphical symbol of a negative-edge-triggered JK-flip-flop with a direct clear.

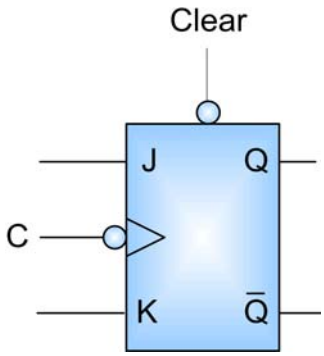


Figure 8: Negative-edge-triggered JK Flip-Flop with Asynchronous Clear

Design of Synchronous Sequential Circuits

Objectives

1. Design of synchronous sequential circuits with an example.
2. Construction of state diagrams and state tables/
3. Translation of State transition table into excitation table.
4. Logic diagram construction of a synchronous sequential circuit

Sequential Circuit Design Steps

- The design of sequential circuit starts with verbal specifications of the problem (See Figure 1).

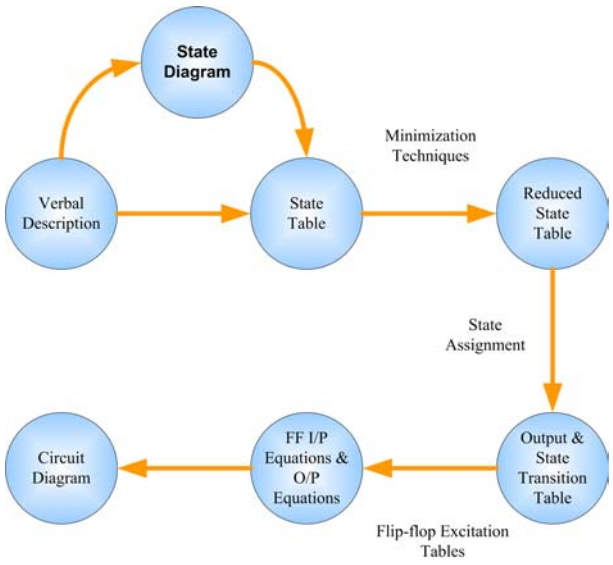


Figure 1: Sequential Circuit Design Steps

- The next step is to derive the state table of the sequential circuit. A state table represents the verbal specifications in a tabular form.
- In certain cases state table can be derived directly from verbal description of the problem.
- In other cases, it is easier to first obtain a state diagram from the verbal description and then obtain the state table from the state diagram.
- A state diagram is a graphical representation of the sequential circuit.
- In the next step, we proceed by simplifying the state table by minimizing the number of states and obtain a reduced state table.

- The states in the reduced state table are then assigned binary-codes. The resulting table is called output and state transition table.

- From the state transition table and using flip-flop's excitation tables, flip-flops input equations are derived. Furthermore, the output equations can readily be derived as well.

- Finally, the logic diagram of the sequential circuit is constructed.

- An example will be used to illustrate all these concepts.

Sequence Recognizer

- A sequence recognizer is to be designed to detect an input sequence of '1011'. The sequence recognizer outputs a '1' on the detection of this input sequence. The sequential circuit is to be designed using JK and D type flip-flops.

- A sample input/output trace for the sequence detector is shown in Table 1.

Table 1: Sample Input/Output Trace

Input	0	1	1	0	1	0	1	1	0	1	1	1	0	1	0	1	1	1	0	0
Output	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	1	0	0	0

- We will begin solving the problem by first forming a state diagram from the verbal description.

- A state diagram consists of circles (which represent the states) and directed arcs that connect the circles and represent the transitions between states.

- In a state diagram:

    1. The number of circles is equal to the number of states. Every state is given a label (or a binary encoding) written inside the corresponding circle.

    2. The number of arcs leaving any circle is 2^n, where n is the number of inputs of the sequential circuit.

    3. The label of each arc has the notation x/y, where x is the input vector that causes the state transition, and y is the value of the output during that present state.

    4. An arc may leave a state and end up in the same or any other state.

- Before we begin our design, the following should be noted.

    1. We do not have an idea about how many states the machine will have.

    2. The states are used to "remember" something about the history of past inputs. For the sequence 1011, in order to be able to produce the output value 1 when the final 1 in the sequence is received, the circuit must be in a state that "remembers" that the previous three inputs were 101.

    3. There can be more than one possible state machine with the same behavior.

Deriving the State Diagram

- Let us begin with an initial state (since a state machine must have at least one state) and denote it with ***'S0'*** as shown in Figure 2 (a).

- Two arcs leave state ***'S0'*** depending on the input (being a 0 or a 1). If the input is a 0, then we return back to the same state. If the input is a 1, then we have to remember it (recall that we are trying to detect a sequence of 1011). We remember that the last input was a one by changing the state of the machine to a new state, say ***'S1'***. This is illustrated in Figure 2 (b).

- ***'S1'*** represents a state when the last single bit of the sequence was one. Outputs for both transitions are zero, since we have not detected what we are looking for.

- Again in state ***'S1',*** we have two outgoing arcs. If the input is a 1, then we return to the same state and if the input is a 0, then we have to remember it (second number in the sequence). We can do so by transiting to a new state, say ***'S2'***. This is illustrated in Figure 2 (c).

- Note that if the input applied is '1', the next state is still ***'S1'*** and not the initial state ***'S0'***. This is because we take this input 1 as the first digit of new sequence. The output still remains 0 as we have not detected the sequence yet.

- State ***'S2'*** represents detection of '10' as the last two bits of the sequence. If now the input is a '1', we have detected the third bit in our sequence and need to remember it. We remember it by transiting to a new state, say ***'S3'*** as shown in Figure 2 (d). If the input is '0' in state ***'S2'*** then it breaks the sequence and we need to start all over again. This is achieved by transiting to initial state ***'S0'***. The outputs are still 0.

- In state ***'S3'***, we have detected input sequence '101'. Another input 1 completes our detection sequence as shown in Figure 2 (e). This is signaled by an output 1. However we transit to state ***'S1'*** instead of ***'S0'*** since this input 1 can be counted as first 1 of a new sequence. Application of input 0 to state ***'S3'*** means an input sequence of 1010. This implies the last two bits in the sequence were 10 and we transit to a state that remembers this input sequence, i.e. state ***'S2'***. Output remains as zero.

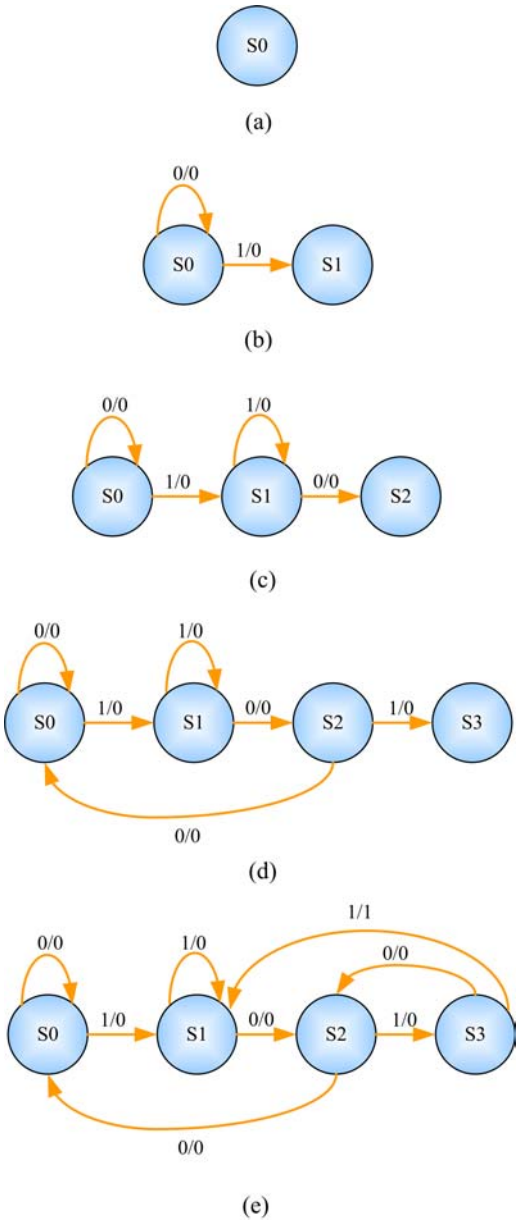


Figure 2: Deriving the State Diagram of the Sequence Recognizer

Deriving the State Table

- A state table represents time sequence of inputs, outputs, and states in a tabular form. The state table for the previous state diagram is shown in Table 2.
- The state table can also be represented in an alternate form as shown in Table 3.
- Here the present state and inputs are tabulated as inputs to the combinational circuit. For every combination of present state and input, next state column is filled from the state table.
- The number of flip-flops required is equal to $\lceil log_2(number\ of\ states) \rceil$.

- Thus, the state machine given in the figure will require two flip-flops $\lceil log_2(4) \rceil = 2$. We assign letters A and B to them.

Table 2: State Table of the Sequence Recognizer

Present State	Next State		Output	
	X=0	X=1	X=0	X=1
S0	*S0*	*S1*	0	0
S1	*S2*	*S1*	0	0
S2	*S0*	*S3*	0	0
S3	*S2*	*S1*	0	1

Table 3: Alternative Format of Table 2

Inputs of Combinational Circuit		Next State	Output
Present State	Input		
S0	0	*S0*	0
S0	1	*S1*	0
S1	0	*S2*	0
S1	1	*S1*	0
S2	0	*S0*	0
S2	1	*S3*	0
S3	0	*S2*	0
S3	1	*S1*	1

State Assignment

- The states in the constructed state diagram have been assigned symbolic names rather than binary codes.
- It is necessary to replace these symbolic names with binary codes in order to proceed with the design.
- In general, if there are ***m*** states, then the codes must contain ***n*** bits, where $\mathbf{2}^{n} \geq \boldsymbol{m}$, and each state must be assigned a unique code.
- There can be many possible assignments for our state machine. One possible assignment is show in Table 4.

Table 4: State Assignment

State	*Assignment*
S0	00
S1	01
S2	10
S3	11

- The assignment of state codes to states results in state transition table as shown.

- It is important to mention here that the binary code of the present state at a given time ***t*** represents the values stored in the flip-flops; and the next-state represents the values of the flip-flops one clock period later, at time ***t+1***.

Table 5: State Transition Table

Inputs of Combinational Circuit		Next State	Output
Present State	Input		
A B	***X***	***A B***	***Y***
0 0	0	0 0	0
0 0	1	0 1	0
0 1	0	1 0	0
0 1	1	0 1	0
1 0	0	0 0	0
1 0	1	1 1	0
1 1	0	1 0	0
1 1	1	0 1	1

General Structure of Sequence Recognizer

- The specifications required using JK and D type flip-flops.

- Referring to the general structure of sequential circuit shown in Figure 3, our synthesized circuit will look like that as shown in the figure. Observe the feedback paths.

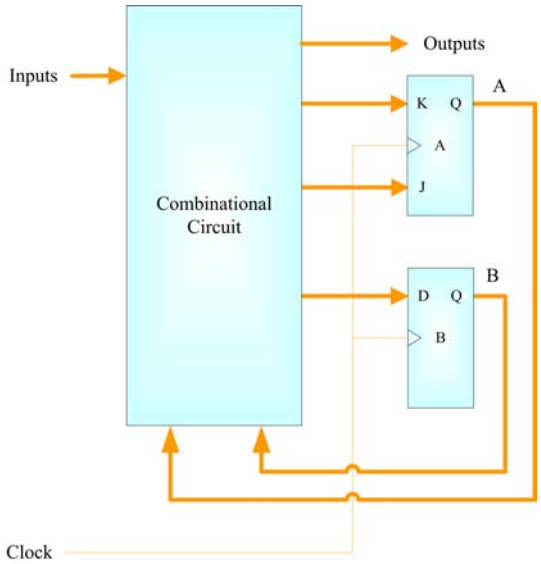


Figure 3: General Structure of the Sequenc Recognizer

- What remains to be determined is the combinational circuit which specifies the external outputs and the flip-flop inputs.

- The state transition table as shown can now be expanded to construct the excitation table for the circuit.

- Since we are designing the sequential circuit using JK and D type flip-flops, we need to correlate the required transitions in state transition table with the excitation tables of JK and D type-flip-flops.

- The functionality of the required combinational logic is encapsulated in the excitation table. Thus, the excitation table is next simplified using map or other simplification methods to yield Boolean expressions for inputs of the used flip-flops as well as the circuit outputs.

Deriving the Excitation Table

- The excitation table (See Table 6) describes the behavior of the combinational portion of sequential circuit.

Table 6: Excitation Table of the Sequence Recognizer

Present State		***Input***				***Flip-flops Inputs***		
A	***B***	***X***	***A***	***B***	***Y***	J_A	K_A	D_B
0	0	0	0	0	0	0	X	0
0	0	1	0	1	0	0	X	1
0	1	0	1	0	0	1	X	0
0	1	1	0	1	0	0	X	1
1	0	0	0	0	0	X	1	0
1	0	1	1	1	0	X	0	1
1	1	0	1	0	0	X	0	0
1	1	1	0	1	1	X	1	1

- For deriving the actual circuitry for the combinational circuit, we need to simplify the excitation table in a similar way we used to simplify truth tables for purely combinational circuits.

- Whereas in combinational circuits, our concern were only circuit outputs; in sequential circuits, the combinational circuitry is also feeding the flip-flops inputs. Thus, we need to simplify the excitation table for both outputs as well as flip-flops inputs.

- We can simplify flip-flop inputs and output using K-maps as shown in Figure 4.

- Finally the logic diagram of the sequential circuit can be made as shown in Figure 5.

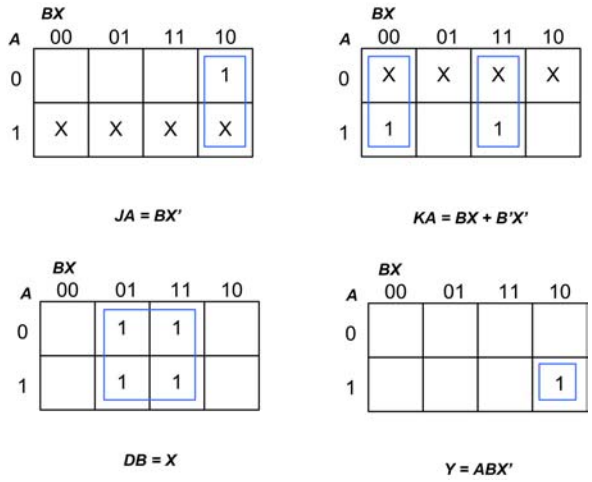


Figure 4: Input Equations of the Sequence Recognizer

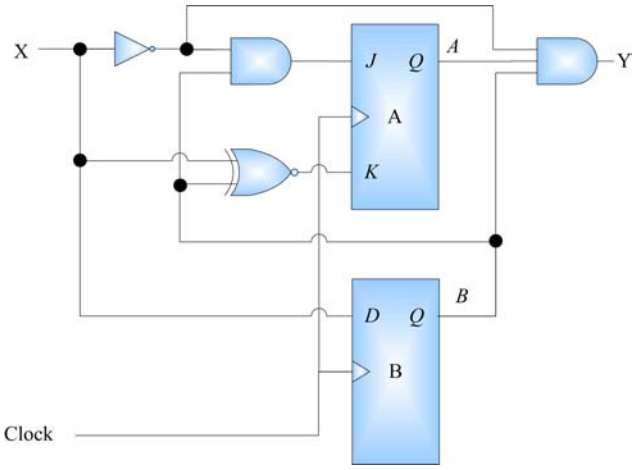


Figure 5: Circuit Diagram of the Sequence Recognizer

Analysis of Clocked Sequential Circuits

Objectives

The objectives of this lesson are as follows:

- Analysis of clocked sequential circuits with an example
- State Reduction with an example
- State assignment
- Design with unused states
- Unused state hazards

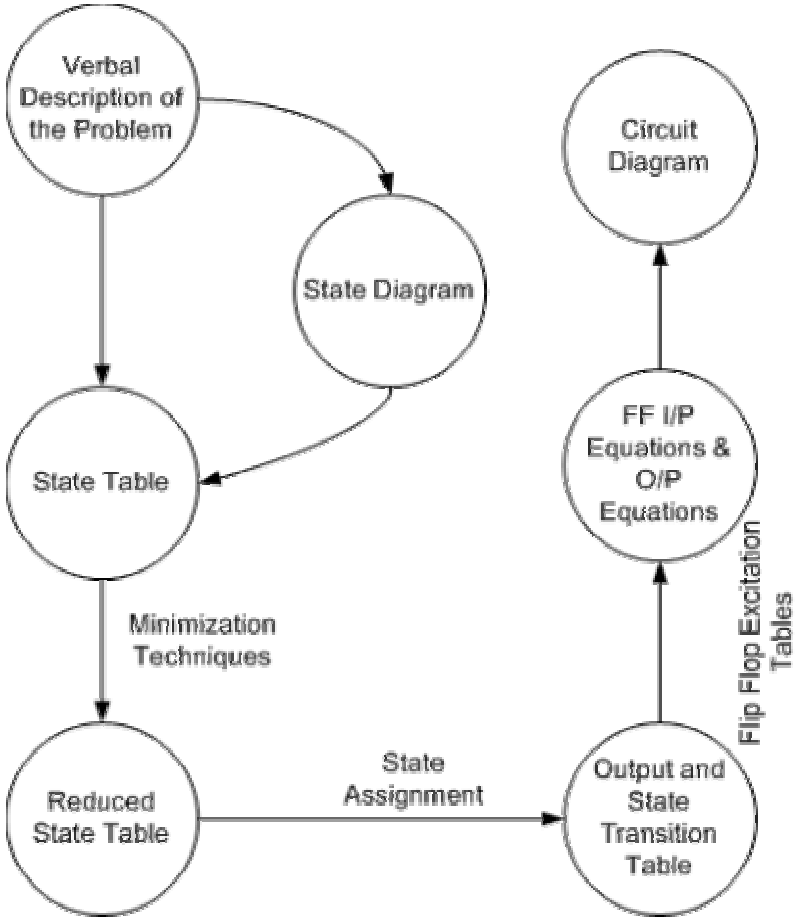


Figure 1: Sequential Circuit Design Steps

The behavior of a sequential circuit is determined from the inputs, outputs and states of its flip-flops.

Both the outputs and the next state are a function of the inputs and the present state.

Recall from previous lesson that sequential circuit design involves the flow as shown.

Analysis consists of obtaining a state-table or a state-diagram from a given sequential circuit implementation. In other words analysis closes the loop by forming state-table from a given circuit-implementation.

We will show the analysis procedure by deriving the state table of the example circuit we considered in synthesis. The circuit is shown in Figure.

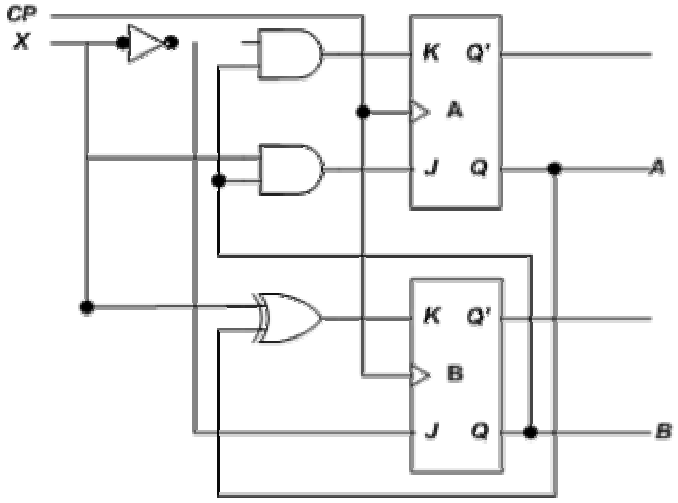


Figure 2: A Clocked sequential circuit

The circuit has

- Clock input, CP.
- One input *x*
- One output *y*
- One clocked JK flip-flop
- One clocked D flip-flop (the machine can be in maximum of 4 states)

A State table is representation of sequence of inputs, outputs, and flip-flop states in a tabular form. Two forms of state tables are shown (In this lesson, the second form will be used).

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

Figure 3: State Table: Form 1

Analysis is the generation of state table from the given sequential circuit.

The number of rows in the state table is equal to $2^{(\text{number of flip-flops+ number of inputs})}$. For the circuit under consideration, number of rows = $2^{(2+1)} = 2^{(3)} = 8$

Present state		Input	Next state		Output
A(t)	B(t)	X	A(t+1)	B(t+1)	y
0	0	0			
0	0	1			
0	1	0			
0	1	1			
1	0	0			
1	0	1			
1	1	0			
1	1	1			

Figure 4: State Table - Form 2

In the present case there are two flip-flops and one input, thus a total of 8 rows as shown in the table.

Present state		Input	Next state		Output
A(t)	B(t)	X	A(t+1)	B(t+1)	y
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	1	0	0
0	1	1	0	1	0
1	0	0	0	0	0
1	0	1	1	1	0
1	1	0	1	0	0
1	1	1	0	1	1

Figure 5: State Table

The analysis can start from any arbitrary state. Let us start deriving the state table from the initial state 00.

As a first step, the input equations to the flip-flops and to the combinational circuit must be obtained from the given logic diagram. These equations are:

$J_A = BX'$
$K_A = BX + B'X'$
$D_B = X$
$y = ABX$

The first row of the state-table is obtained as follows:

When input X = 0; and present states A = 0 and B = 0 (as in the first row);

then, using the above equations we get:

$y = 0$, $J_A = 0$, $K_A = 1$, and $D_B = 0$.

The resulting state table is exactly same from which we started our design example. Thus analysis is opposite to design and combined they act as a closed loop.

State Reduction

The problem of state reduction is to find ways of reducing the number of states in a sequential circuit without altering the input-output relationships.

In other words, to reduce the number of states, redundant states should be eliminated. A redundant state S_i is a state which is equivalent to another state S_j.

Two states are said to be equivalent if, for each member of the set of inputs, they give exactly the same output and send the circuit either to the same state or to an equivalent state.

Since 'm' flip-flops can describe a state machine of up to 2^m states, reducing the number of states may (or may not) result in a reduction in the number of flip-flops. For example, if the number of states are reduced from 8 to 5, we still need 3 flip-flops.

However, state reduction will result in more don't care states. The increased number of don't care states can help obtain a simplified circuit for the state machine.

Consider the shown state diagram.

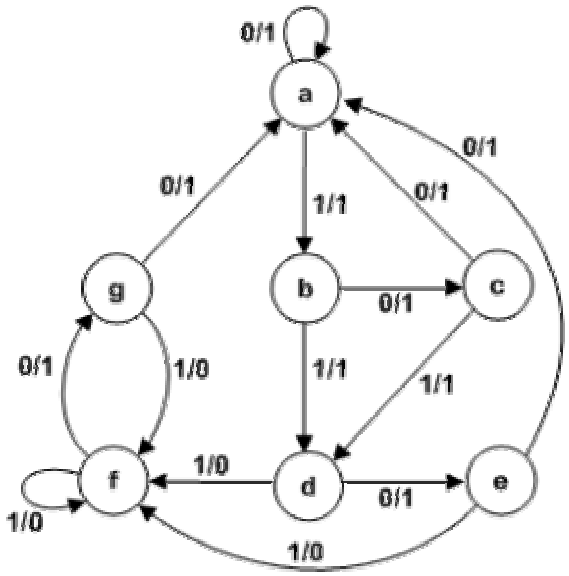


Figure 6: State Diagram

The state reduction proceeds by first tabulating the information of the state diagram into its equivalent state-table form (as shown in the table)

The problem of state reduction requires identifying equivalent states. Each N states is replaced by 1 state.
Consider the following state table.

States 'g' and 'e' produce the same outputs, i.e. '1' and '0', and take the state machine to same next-states, 'a' and 'f', on inputs '0' and '1' respectively. Thus, states 'g' and 'e' are equivalent states.

We can now remove state ‘g’ and replace it with ‘e’ as shown.

We next note that the above change has caused the states ‘d’ and ‘f’ to be equivalent. Thus in the next step, we remove state ‘f’ and replace it with ‘d’.

There are no more equivalent states remaining. The reduced state table results in the following reduced state diagram.

Present state	Next state		Output	
	$x=0$	$x=1$	$x=0$	$x=1$
a	a	b	1	1
b	c	d	1	1
c	a	d	1	1
d	e	~~f~~ d	1	0
e	a	~~f~~ d	1	0
~~f~~	~~g~~ e	f	1	0
~~g~~	a	f	1	0

Figure 7: State Table after reduction

States Assignment

When constructing a state diagram, variable names are used for states as the final number of states is not known a priori.

Once the state diagram is constructed, prior to implementation (using gates and flip-flops), we need to perform the step of 'state reduction'.

The step that follows state reduction is state assignment. In state assignment, binary patterns are assigned to state variables.

State	Assignment 1	Assignment 2	Assignment 3
a	001	000	000
b	010	010	100
c	011	011	010
d	100	101	101
e	101	111	011

Figure 8: Possible state assignments

For a given machine, there are several state assignments possible. Different state assignments may result in different combinational circuits of varying complexities.

State assignment procedures try to assign binary values to states such that the cost *(complexity)* of the combinational circuit is reduced. There are several heuristics that attempt to choose good state assignments (also known as state encoding) that try to reduce the required combinational logic complexity, and hence cost.

As mentioned earlier, for the reduced state machine obtained in the previous example, there can be a number of possible assignments. As an example, three different state assignments are shown in the table for the same machine.

We use ad-hoc state assignments in this lesson.

Design with unused states

There are occasions when a sequential circuit, implemented using m flip-flops, may not utilize all the possible 2^m states

Present state	Next state		Output	
	$x = 0$	$x = 1$	$x = 0$	$x = 1$
a	*a*	*b*	1	1
b	*c*	*d*	1	1
c	*a*	*d*	1	1
d	*e*	*d*	1	0
e	*a*	*d*	1	0

Figure 9: Reduced table with binary assignments

In the previous example of machine with 5 states, we need three flip-flops. Let us choose assignment 1, which is binary assignment for our sequential machine example (shown in the table).

The unspecified states can be used as don't-cares and will therefore help in simplifying the logic.

The excitation table of previous example is shown. There are three states, 000, 110, and 111 that are not listed in the table under present state and input.

Present state			Input	Next state			Flip-flop inputs						Output
A	*B*	*C*	*x*	*A*	*B*	*C*	*SA*	*RA*	*SB*	*RB*	*SC*	*RC*	*y*
0	0	1	0	0	0	1	0	X	0	X	X	0	1
0	0	1	1	0	0	1	0	X	1	0	0	1	1
0	1	0	0	0	1	0	0	X	X	0	1	0	1
0	1	0	1	0	1	0	1	0	0	1	0	X	1
0	1	1	0	0	1	1	0	X	0	1	X	0	1
0	1	1	1	0	1	1	1	0	0	1	0	1	1
1	0	0	0	1	0	0	X	0	0	X	1	0	1
1	0	0	1	1	0	0	X	0	0	X	0	X	0
1	0	1	0	1	0	1	0	1	0	X	X	0	1
1	0	1	1	1	0	1	X	0	0	X	0	1	0

Figure 10: Excitation Table

With the inclusion of input 1 or 0, we obtain six don't-care minterms: 0, 1, 12, 13, 14, and 15.

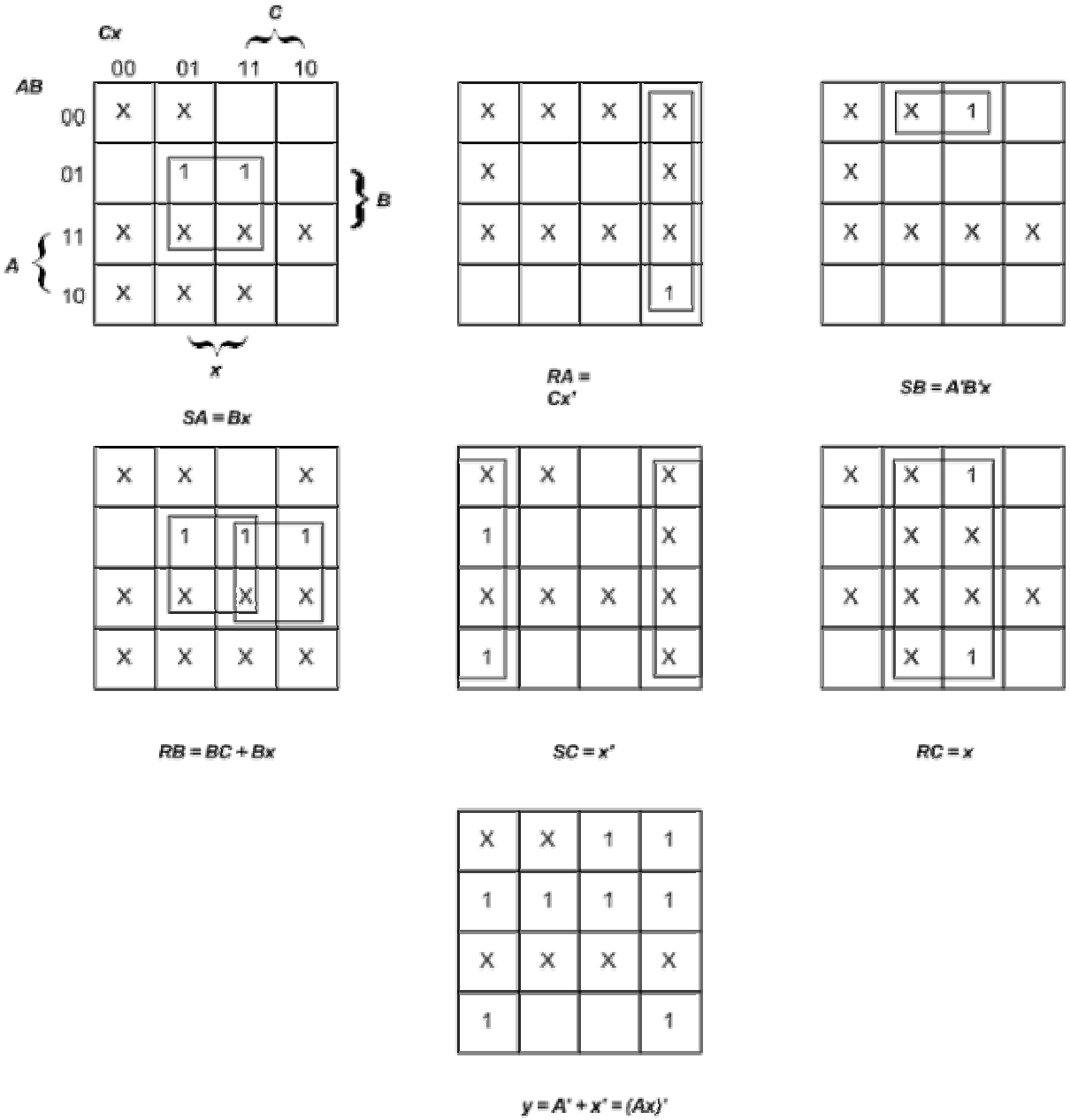


Figure 11: K-Maps

The K-maps of SA and RA is shown in the figure. Other K-Maps can be obtained similarly and the equations derived are shown in the figure.

The logic diagram thus obtained is shown in the figure.

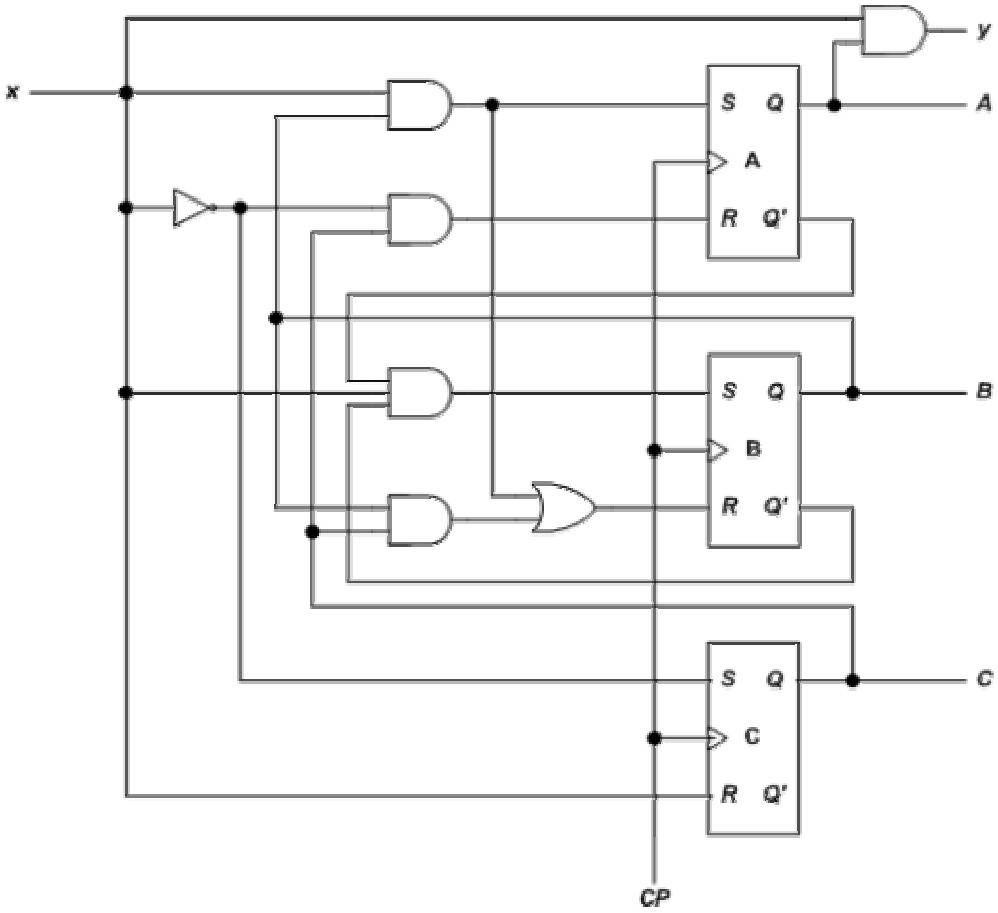


Figure 12: Logic Diagram

$$SA = Bx$$

$$RA = Cx'$$

$$SB = A'B'x$$

$$RB = BC + Bx$$

$$SC = x'$$

$$RC = x$$

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

Figure 13: Equations

Note that the design of the sequential circuit is dependent on binary codes for states. A different binary state codes set may have resulted in some different combinational circuit.

Unused States Hazard

Sequential circuits with unused states can cause the circuit to produce erroneous behavior.

This may happen when the circuit enters one of the unused states due to some reason, e.g. due to power-on, and continues cycling between the invalid states.

Thus, a circuit that is designed must be carefully analyzed to ensure that it converges to some valid state.

Consider the circuit of the previous example that employed three unused states 000, 110 and 111. We will now investigate its behavior if it enters in any of these states.

The state diagram (from previous example) is shown in the figure. We will use the state diagram to derive next state from each of the unused states and derive the state table.

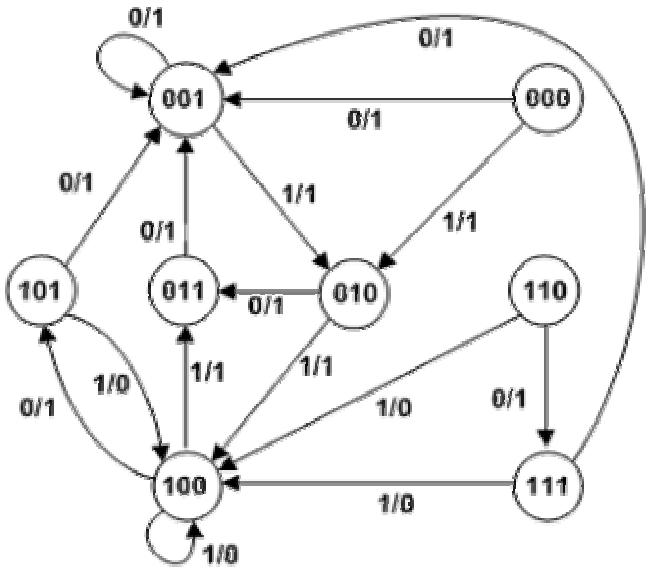


Figure 14: State Diagram

For instance, the circuit enters unused state 000.

On application of input 0, ABCx = 0000, from the equations (figure), we see that this minterm is not included in any function except for SC, i.e., the set input of flip-flop C and output y.

Thus the circuit enters the state ABC = 001 from the unused state 000 when input 0 is applied.

On the other hand, if the input applied is 1 then ABCx combination = 0001. The maps indicate that this minterm is included in the functions for SB, RC and y.

Therefore B will be set and C gets cleared.

So the circuit enters next state ABC = 010 when input 1 is applied to unused state 000.

Note that both states 001 and 010 are valid states.

Similar analysis is carried out for all other unused states and the derived state diagram is formed (shown in the figure).

We note that the circuit converges into one of the valid states if it ever finds itself in one of the invalid states 000, 110, and 111.

Such a circuit is said to be self-correcting, free from hazards due to unused states.

Mealy and Moore Type Finite State Machines

Objectives

- There are two basic ways to design clocked sequential circuits. These are using:
  1. Mealy Machine, which we have seen so far.
  2. Moore Machine.

- The objectives of this lesson are:
  1. Study Mealy and Moore machines
  2. Comparison of the two machine types
  3. Timing diagram and state machines

Mealy Machine

- In a Mealy machine, the outputs are a function of the present state and the value of the inputs as shown in Figure 1.

- Accordingly, the outputs may change asynchronously in response to any change in the inputs.

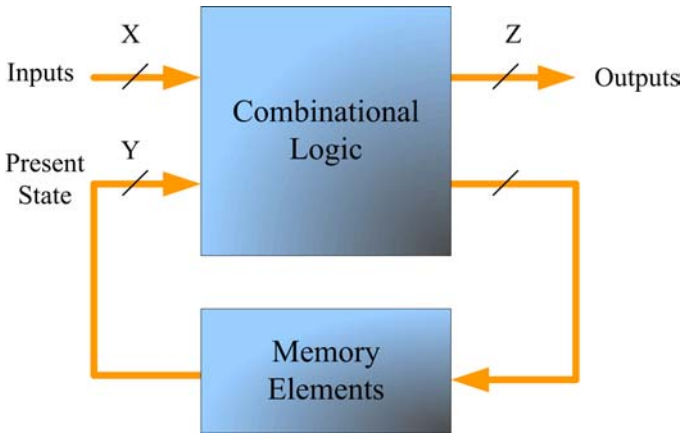


Figure 1: Mealy Type Machine

Mealy Machine

- In a Moore machine the outputs depend only on the present state as shown in Figure 2.

- A combinational logic block maps the inputs and the current state into the necessary flip-flop inputs to store the appropriate next state just like Mealy machine.

- However, the outputs are computed by a combinational logic block whose inputs are only the flip-flops state outputs.

- The outputs change synchronously with the state transition triggered by the active clock edge.

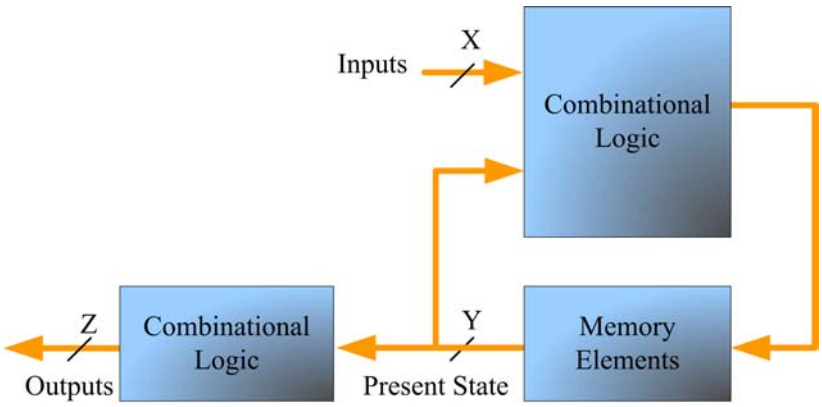


Figure 2: Moore Type Machine

Comparison of the Two Machine Types

- Consider a finite state machine that checks for a pattern of '10' and asserts logic high when it is detected.
- The state diagram representations for the Mealy and Moore machines are shown in Figure 3.
- The state diagram of the Mealy machine lists the inputs with their associated outputs on state transitions arcs.
- The value stated on the arrows for Mealy machine is of the form Zi/Xi where Zi represents input value and Xi represents output value.
- A Moore machine produces a unique output for every state irrespective of inputs.
- Accordingly the state diagram of the Moore machine associates the output with the state in the form state-notation/output-value.
- The state transition arrows of Moore machine are labeled with the input value that triggers such transition.
- Since a Mealy machine associates outputs with transitions, an output sequence can be generated in fewer states using Mealy machine as compared to Moore machine. This was illustrated in the previous example.

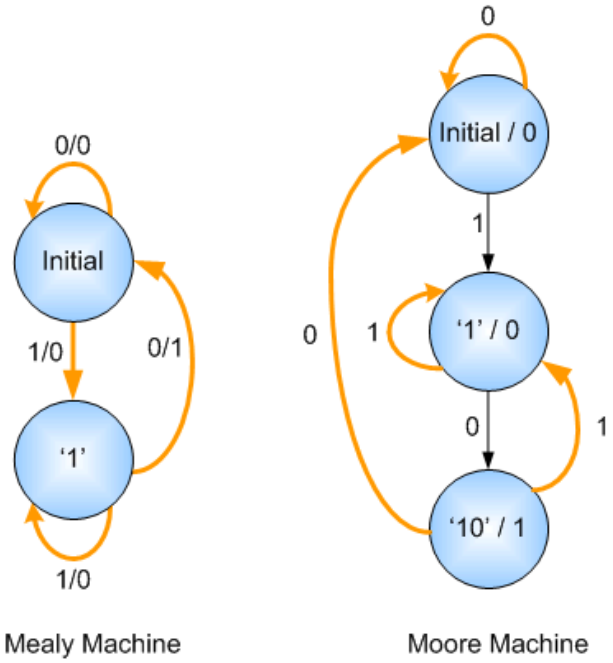


Figure 3: Mealy and Moore State Diagrams for '10' Sequence Detector

Timing Diagrams

- To analyze Mealy and Moore machine timings, consider the following problem. A state-machine outputs '1' if the input is '1' for three consecutive clocks.

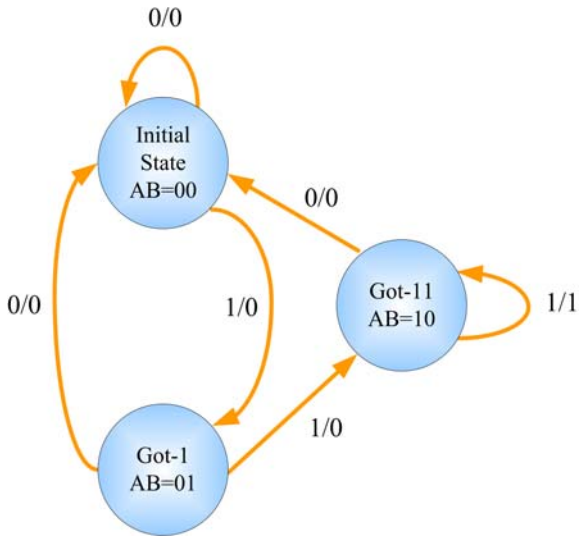


Figure 4: Mealy State Machine for '111' Sequence Detector

Mealy State Machine

- The Mealy machine state diagram is shown in Figure 4.
- Note that there is no reset condition in the state machine that employs two flip-flops. This means that the state machine can enter its unused state '11' on start up.
- To make sure that machine gets resetted to a valid state, we use a 'Reset' signal.
- The logic diagram for this state machine is shown in Figure 5. Note that negative edge triggered flip-flops are used.

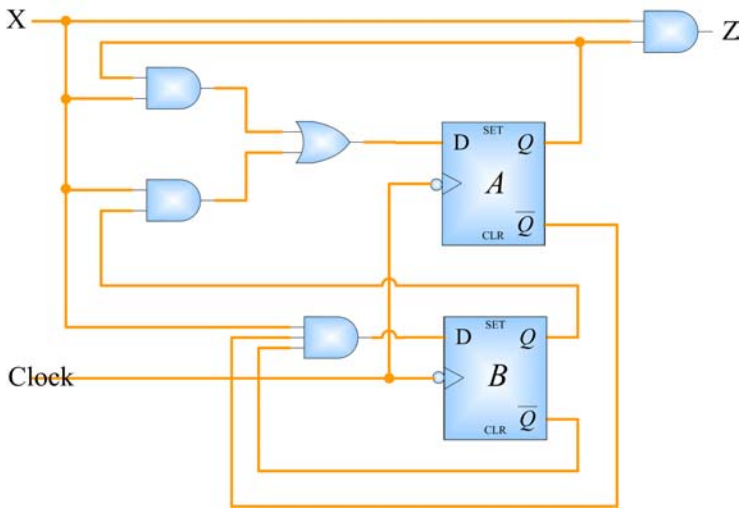


Figure 5: Mealy State Machine Circuit Implementation

- Timing Diagram for the circuit is shown in Figure 6.
- Since the output in Mealy model is a combination of present state and input values, an unsynchronized input with triggering clock may result in invalid output, as in the present case.
- Consider the present case where input 'x' remains high for sometime after state 'AB = 10' is reached. This results in 'False Output', also known as 'Output Glitch'.

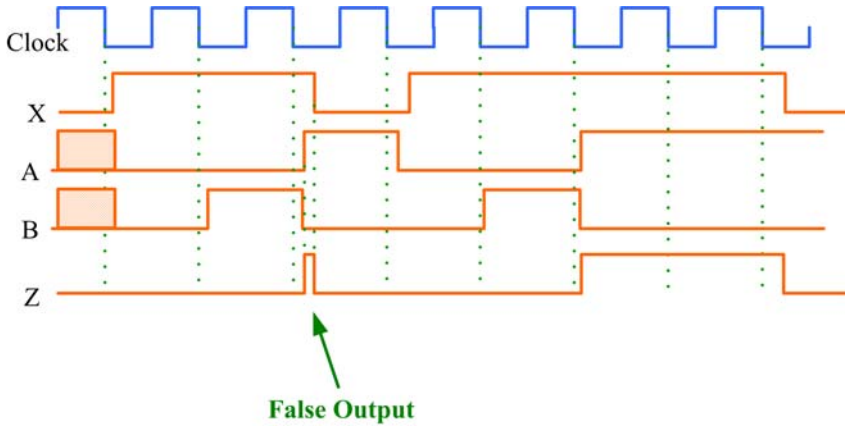


Figure 6: Timing Diagram for Mealy Model Sequence Detector

Moore State Machine

- The Moore machine state diagram for '111' sequence detector is shown in Figure 7.
- The state diagram is converted into its equivalent state table (See Table 1).
- The states are next encoded with binary values and we achieve a state transition table (See Table 2).

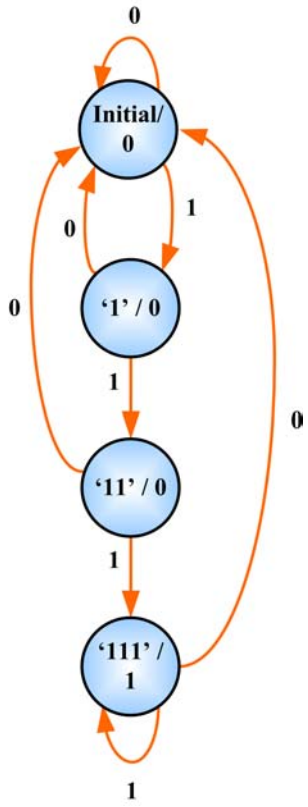


Figure 7: Moore Machine State Diagram

Table 1: State Table

Present State	Next State		Output Z
	x = 0	x = 1	
Initial	*Initial*	*Got*-1	0
Got-1	*Initial*	*Got*-11	0
Got-11	*Initial*	*Got*-111	0
Got-111	*Initial*	*Got*-111	1

Table 2: State Transition Table and Output Table

Present State	Next State		Output Z
	x = 0	x = 1	
Initial	*Initial*	*Got*-1	0
Got-1	*Initial*	*Got*-11	0
Got-11	*Initial*	*Got*-111	0
Got-111	*Initial*	*Got*-111	1

- We will use JK and D flip-flops for the Moore circuit implementation. The excitation tables for JK and D flip-flops (Table 3 & 4) are referenced to tabulate excitation table (See Table 5).

Table 3: Excitation Table for JK flip-flop

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

Table 4: Excitation Table for D flip-flop

Q(t)	Q(t+1)	D
0	0	0
0	1	1
1	0	0
1	1	1

Table 5: Excitation Table for the Moore Implementation

Inputs of Comb.Circuits			Next State		Outputs of Comb.Circuit			Output
Present State		Input			Flip-flop Inputs			
A	**B**	**X**	**A**	**B**	J_A	K_A	D_B	**Z**
0	0	0	0	0	0	X	0	0
0	0	1	0	1	0	X	1	0
0	1	0	0	0	0	X	0	0
0	1	1	1	0	1	X	0	0
1	0	0	0	0	X	1	0	0
1	0	1	1	1	X	0	1	0
1	1	0	0	0	X	1	0	1
1	1	1	1	1	X	0	1	1

- Simplifying Table 5 using maps, we get the following equations:
  - $J_A = X.B$
  - $K_A = X'$
  - $D_B = X(A + B)$
  - $Z = A \,.\, B$

- Note that the output is a function of present state values only.

- The circuit diagram for Moore machine circuit implementation is shown in Figure 8.

- The timing diagram for Moore machine model is also shown in Figure 9.

- There is no false output in a Moore model, since the output depends only on the state of the flop flops, which are synchronized with clock. The outputs remain valid throughout the logic state in Moore model.

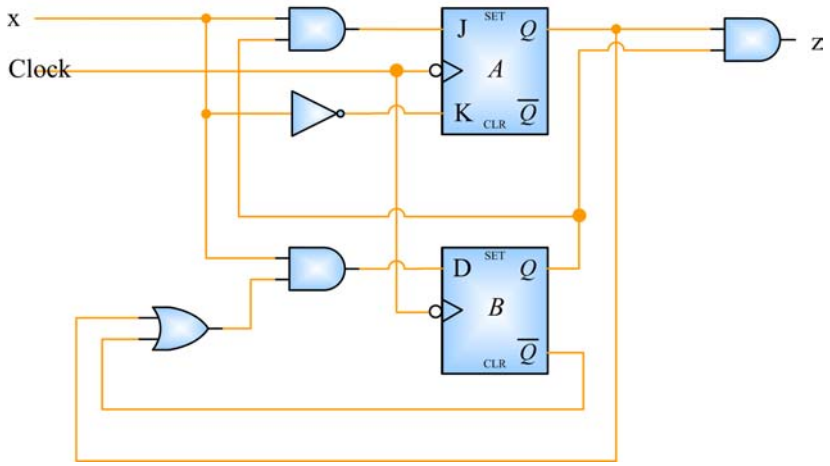


Figure 8: Moore Machine Circuit Implementation for Sequence Detector.

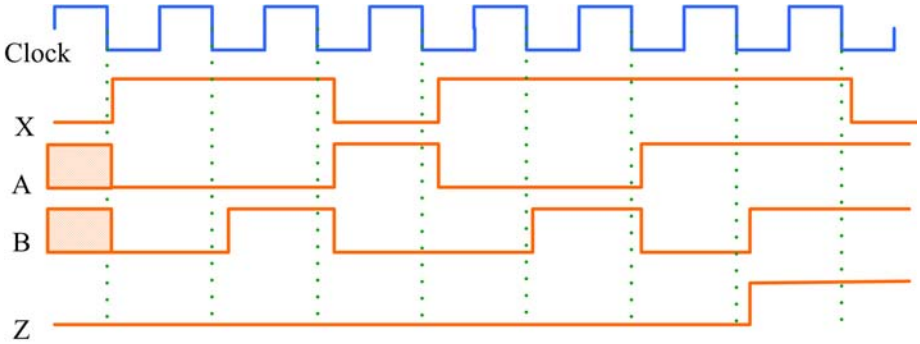


Figure 9: Timing Diagram for Moore Model Sequence Detector.

Registers

In this lesson, you will learn about

- Registers
- Registers with Parallel load
- Shift registers
- Shift registers with Parallel Load
- Bi-directional Shift Registers

Register

A register is a circuit capable of storing data. In general, an n-bit register consists of n FFs, together with some other logic that allows simple processing of the stored data.

All FFs are triggered *synchronously* by the same clock signal. In other words, new data are latched into all FFs at the same time.

Figure 1 shows a 4-bit register constructed with four D-type FFs. In this figure we have:

- Inputs D_0 to D_3
- Clock
- Clear
- Outputs Q_0 to Q_3

The Clock input is common to all the four D FFs. It triggers all FFs on the rising edge of each clock pulse, and the binary data available at the four D inputs are latched into the register.

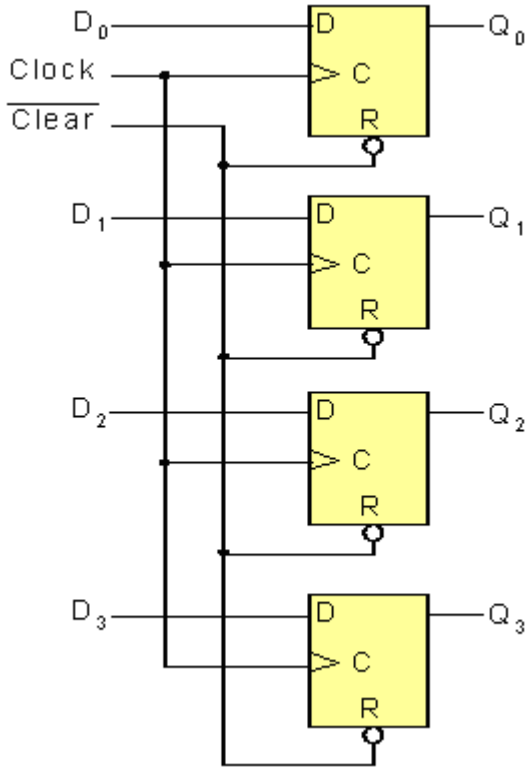


Figure 1: 4-bit register with D flip-flops

- The Clear/ input is an *active-low asynchronous* input which clears the register content to all 0's when asserted low, independent of the clock pulse.
- During normal operation, Clear/ must be maintained at logic 1.
- The transfer of new information into a register is referred to as Loading
- The term Parallel Loading is used if all the input bits are transferred into the register simultaneously, with the common clock pulse.

In most digital systems, a master clock generator supplies clock pulses to all parts of the system, just as the heart that supplies a constant beat to all parts in the human system. Because of this fact, the input values in the register are loaded when a clock pulse arrives. This implies that, whenever a clock pulse arrives, it would load the register with new values, thus overwriting the previously stored register data.

Because of this, a problem arises:
Problem: *What if the contents of the register are to be left unchanged?*
A Solution: The Clock may be prevented from reaching the clock input of the FFs of the register.
⇒ A separate control signal is used
Another Solution: Inputs D_0 to D_3 may be prevented from changing their values.
⇒ A control signal is needed for this

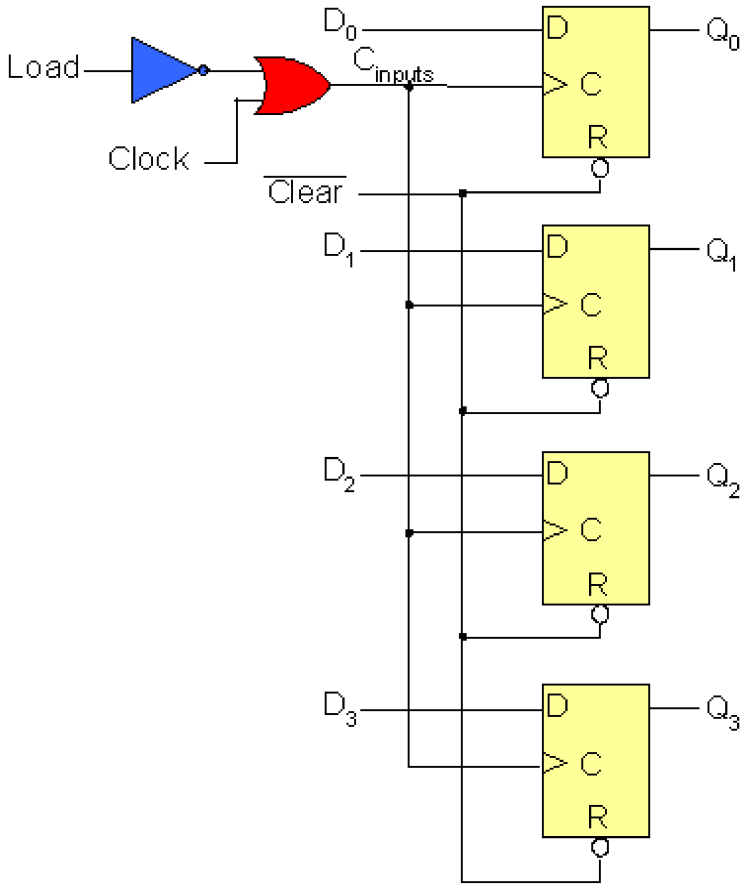


Figure 2: Clock gating

This control can be provided by implementing the following function:

$$C_{inputs} = Load' + Clock$$

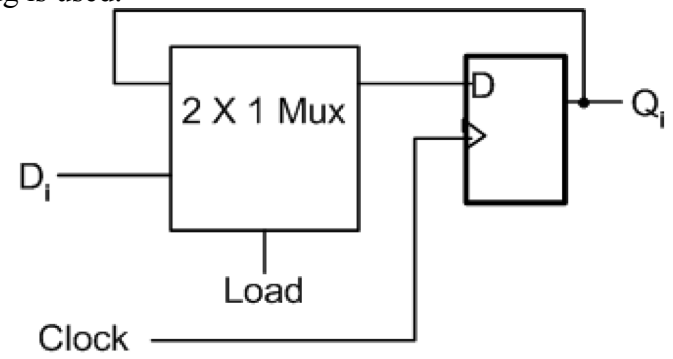
When Load = 0 $\Rightarrow$ C_{inputs} = 1, causing no positive transitions to occur on C_{inputs}. Thus contents of the register remain unchanged.

When Load = 1 $\Rightarrow$ C_{inputs} = Clock, thus the register is clocked normally.

The above phenomenon is known as **Clock gating** (Figure 2).
Problem: Different stages of the register will be gated at different time. This may cause loading of wrong information (known as **clock skew**).
Solution: Clock gating should be avoided.

Use register with *Parallel Load*. (Figure 3)

- No clock gating is used.



$$D = \overline{Load}.Q_i + Load.D_i$$

Figure 3: One stage of Register with Parallel Load

When Load = 1, the data on the input D_i is transferred into the D flip-flop with the next positive transition of clock pulse.
When Load = 0, the data input is blocked, and output Q_i gets a path to the D input of the flip-flop.

Why do we need feedback connection from output to input of D-FF?

- Because D-FF does not have a "no change" input condition. Having the feedback will cause the next state of the FF (D input) to be equal to present state of the FF, i.e. ***no change in state***.

Note that there is no *Clock gating*. Load determines whether to accept new information in the next clock pulse or not.

A 4-bit register with parallel load is shown.

Q: Can clocked latches be used instead of FFs to implement parallel-load registers?

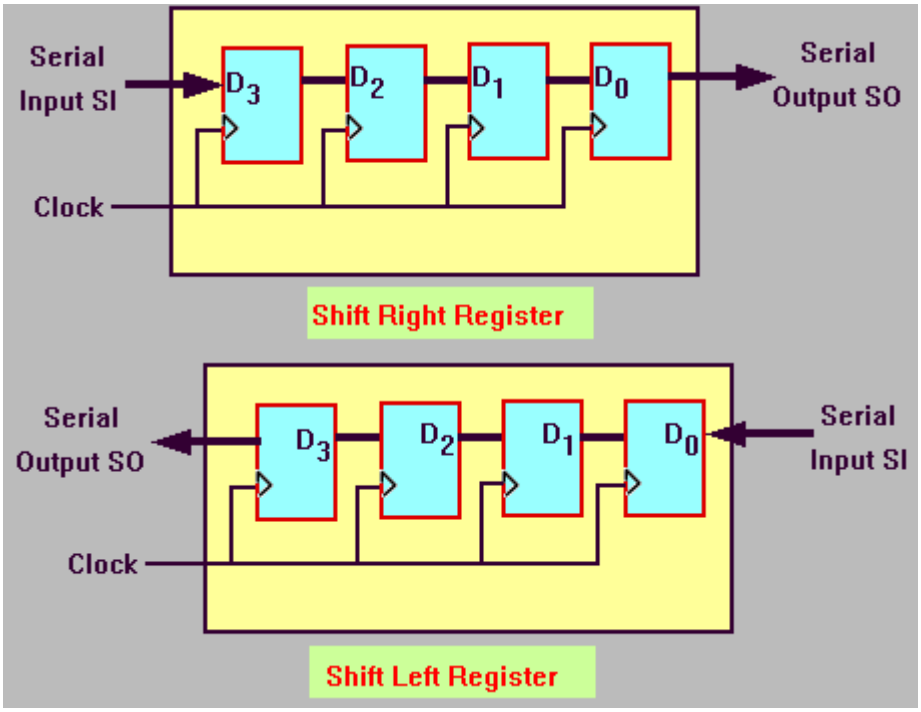


Figure 4: Shift Registers

A shift register (Figure 4) is capable of transferring data from each FF to the next in one or the other direction. Each clock pulse causes data shift from one FF to its immediate neighbor.

The configuration consists of a chain of FFs in cascade, with the output of one FF connected to the input of the next one.

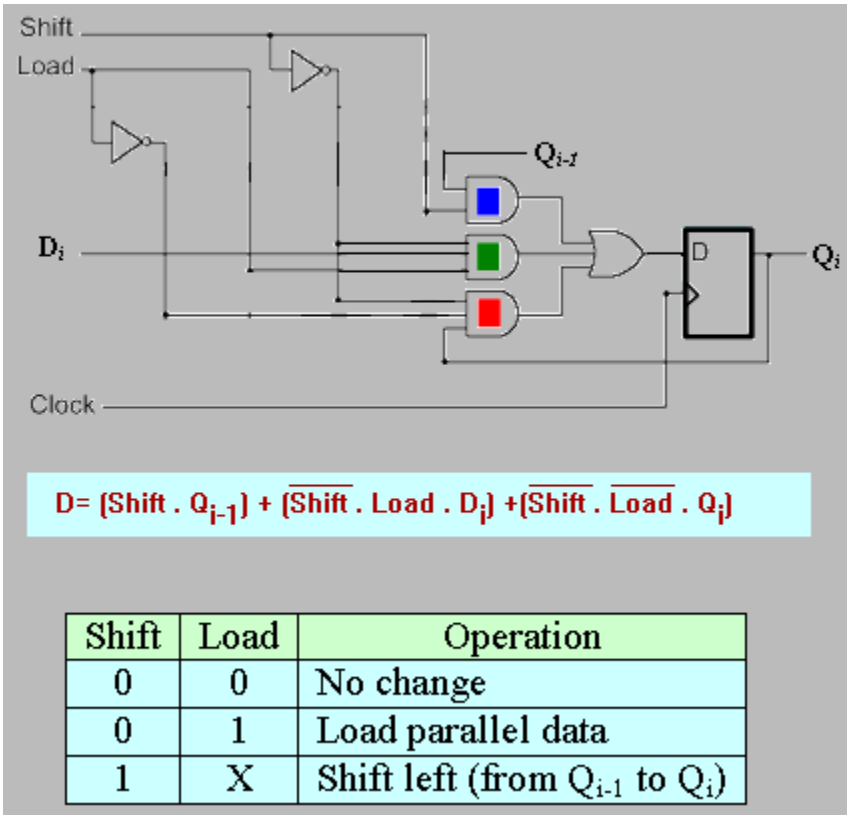
All FFs receive a common pulse, which activates the shift operation from each stage to the next.

The serial input SI is the input to the first (leftmost) FF of the chain.

The serial output SO is the output of the last (rightmost) FF of the chain.

The register discussed above is a "shift right" (MS to LS shifting) register. There is also a "shift left" (LS to MS shifting) register. (Figure 4)

Q: Can clocked latches be used instead of FFs to implement shift registers?



Shift	Load	Operation
0	0	No change
0	1	Load parallel data
1	X	Shift left (from Q_{i-1} to Q_i)

Figure 5: One stage of a shift register with parallel load

A shift register with parallel load capability can be used to input the data bits in parallel into the shift register and then take the data out in a serial fashion by applying the shift operation.

Such a register can act as a parallel-to-serial converter, where data can be loaded in parallel and shifted *out* serially (bit-by-bit)

It can also act as a serial-to-parallel converter, where data can be shifted *in* serially (bit-by-bit) and the output made available in parallel after shifting is complete.

Figure 5 shows a typical stage of a shift register with parallel load. There are two control signals: Shift and Load. A table showing the operation of the register with respect to the Shift and Load inputs is also shown in the figure.

If Shift = 0 and Load = 0, red AND gate is enabled, causing the output of the flip-flop to feed back to its D input.
➢A positive transition in the clock loads this input value into the FF⇒ **No Change** state.

Load condition: If Shift = 0 and Load = 1, the green AND gate is enabled. This causes the D_i input to propagate to its input of the D flip-flop.
➢A positive transition of the clock pulse transfers the input data into the FFs;

Shift condition: If Shift = 1 while Load = 0 or 1, the blue AND gate is enabled, while the

other two AND gates are disabled.

- A positive transition of the clock pulse causes the shift operation. That is, the blue AND gate takes the input from output Q_{i-1} of previous flip-flop. This is true for all stages except for the first stage, where, instead, serial input SI is provided

Bidirectional Shift Register (BDSR)

Design a 3-bit shift register which has 4 operating modes. The operating modes are defined by the status of two select lines S_1 and S_0. The given table specifies the values of S_1 and S_0 and its corresponding operating mode.

Mode Control		Register Operation
S1	S0	
0	0	No change
0	1	Shift down
1	0	Shift up
1	1	Parallel load

Table 1: Modes of BDSR

The design uses 3 stages, where each stage consists of a single multiplexer and a single D-FF.

The output of each MUX is connected to the input of corresponding D FF.

The MUX select inputs are connected to S_1 and S_0 to pass the proper signal to the D-FF input depending on the mode of operation.

When $S_1\ S_0 = 00$, input 0 of the MUX is selected. This forms a path from the output of the FF into its own input, which causes the same value to be loaded in the D FF when a clock pulse is applied. This results in the NO CHANGE operation.

When $S_1\ S_0 = 01$, input 1 of the MUX is selected. This forms a path from the lower significant to higher significant bit, resulting in the SHIFT LEFT (i.e. LSB to MSB) operation.

- The serial input SI is transferred into the rightmost bit in this case.

When $S_1\ S_0 = 10$, input 2 of the MUX is selected. This forms a path from the higher significant bit to lower significant bit, resulting in SHIFT RIGHT (i.e. MSB to LSB) operation.

- The serial input SI is transferred into the leftmost bit (i.e. MSB) in this case.

Finally, when $S_1\ S_0 = 11$, input 3 of the MUX is selected. On this input, the binary information on the parallel input line D_i is transferred into the FF, resulting in PARALLEL LOAD operation.

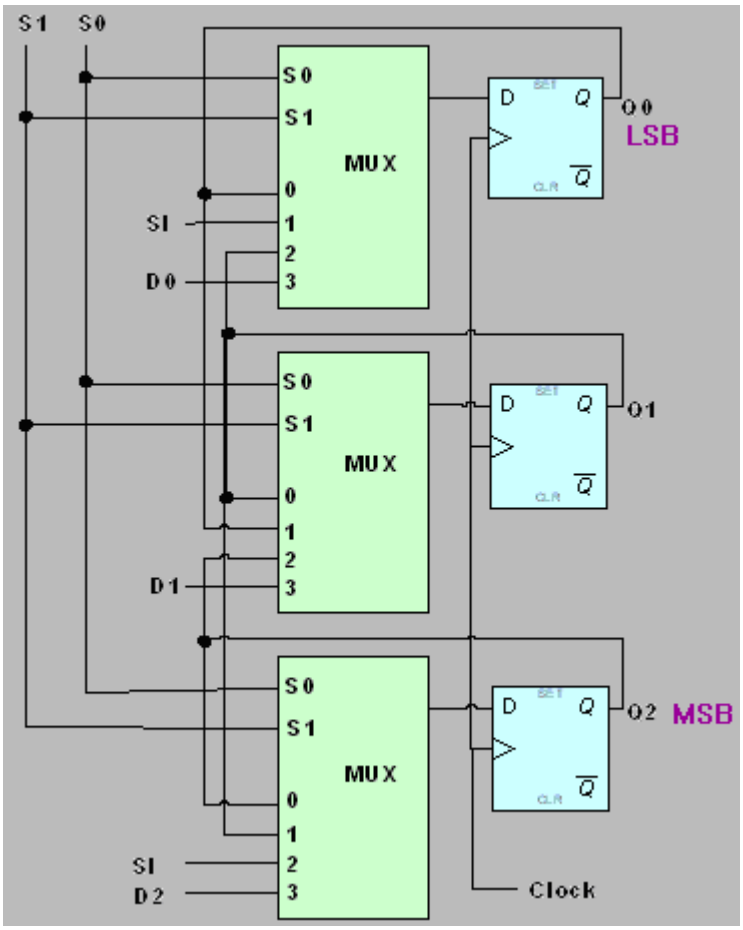


Figure 6: Bi-directional shift register

Counters

In this lesson, the operation and design of Synchronous Binary Counters will be studied.

Synchronous Binary Counters (SBC)

Description and Operation

In its simplest form, a *synchronous binary counter* (**SBC**) receives a train of clock *pulses* as input and outputs the *pulse count* (Q_{n-1} …. Q_2 Q_1 Q_0).

An example is a 3-bit counter that counts from 000 upto 111. Each counter consists of a number of FFs. (Figure 1)

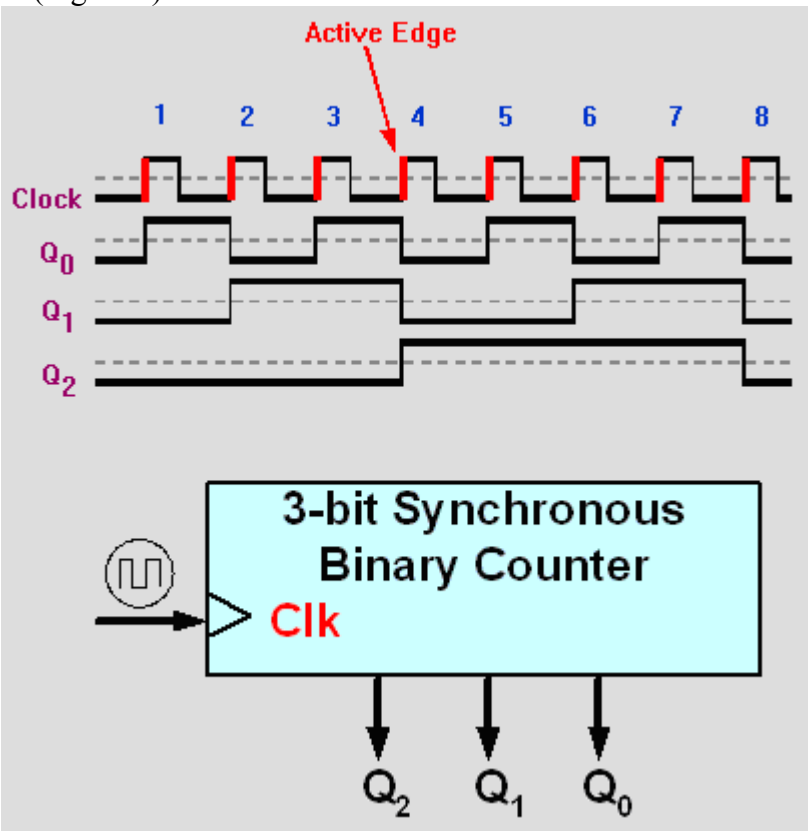


Figure 1: 3-bit SBC

In *synchronous* counters, all FFs are triggered by the same input clock.

An *n*-bit counter has *n*-FFs with 2^n distinct *states*, where each state corresponds to a particular *count*.

Accordingly, the possible *counts* of an *n*-bit counter are 0 to (2^n-1). Moreover an n-bit counter has *n* output bits (Q_{n-1} …. Q_2 Q_1 Q_0).

After reaching the maximum count of (2^n-1), the following clock pulse resets the count back to 0.

Thus, a 3-bit counter counts from 0 to 7 and back to 0. In other words, the output count actually equals (*Total # of input pulses* **Modulo** 2^n).

Accordingly, it is common to identify counters by the modulus 2^n. For example, a 4-bit counter provides a modulo 16 count, a 3-bit counter is a modulo 8 counter, etc.

Referring to the 3-bit counter mentioned earlier, each stage of the counter divides the frequency by 2, where the last stage divides the frequency by 2^n, *n* being the number of bits. (Figure 2)

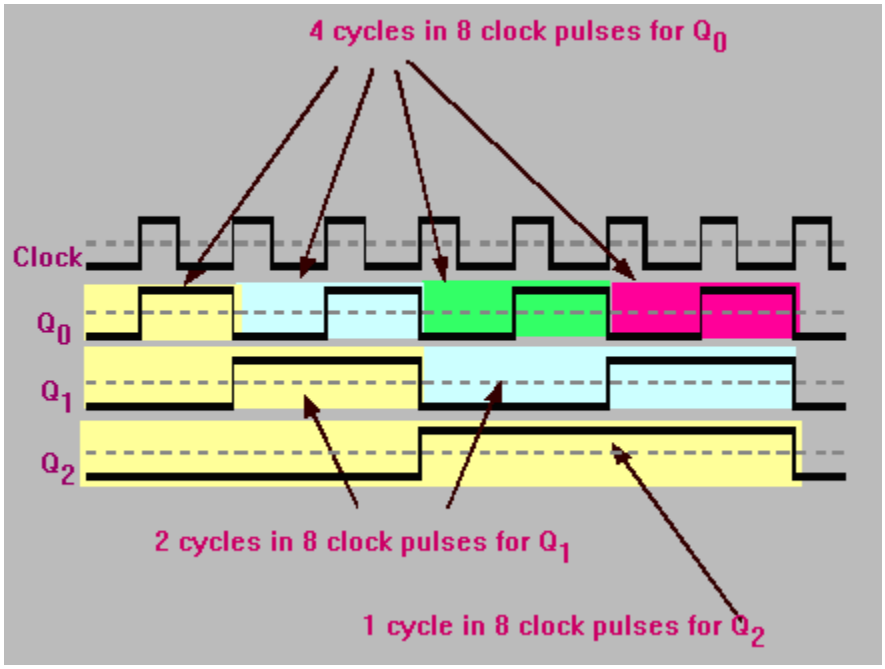


Figure 2: 3-bit SBC

Thus, if the frequency (i.e. no. of cycles/ sec) of clock is F, then the frequency of output waveform of Q_0 is F/2, Q_1 is F/4, and so on. In general, for n-bit counter, we have $F/2^n$.

Design of Binary Counters (SBC)

Design procedure is the same as for other synchronous circuits.

A counter may operate without an external input (except for the clock pulses!)

In this case, the output of the counter is taken from the outputs of the flip-flops without any additional outputs from gates.
Thus, there are no columns for the input and outputs in the state table; we only see the current state and next state…

Example Design a 4-bit SBC using JK flip-flops.

The counter has 4 FFs with a total of 16 states, (0000 to 1111) → 4 state variables Q3 Q2 Q1 Q0 are required.

Present State				Next State				Flip-Flop Inputs							
Q_3	Q_2	Q_1	Q_0	Q_3	Q_2	Q_1	Q_0	J_{Q3}	K_{Q3}	J_{Q2}	K_{Q2}	J_{Q1}	K_{Q1}	J_{Q0}	K_{Q0}
0	0	0	0	0	0	0	1	0	X	0	X	0	X	1	X
0	0	0	1	0	0	1	0	0	X	0	X	1	X	X	1
0	0	1	0	0	0	1	1	0	X	0	X	X	0	1	X
0	0	1	1	0	1	0	0	0	X	1	X	X	1	X	1
0	1	0	0	0	1	0	1	0	X	X	0	0	X	1	X
0	1	0	1	0	1	1	0	0	X	X	0	1	X	X	1
0	1	1	0	0	1	1	1	0	X	X	0	X	0	1	X
0	1	1	1	1	0	0	0	1	X	X	1	X	1	X	1
1	0	0	0	1	0	0	1	X	0	0	X	0	X	1	X
1	0	0	1	1	0	1	0	X	0	0	X	1	X	X	1
1	0	1	0	1	0	1	1	X	0	0	X	X	0	1	X
1	0	1	1	1	1	0	0	X	0	1	X	X	1	X	1
1	1	0	0	1	1	0	1	X	0	X	0	0	X	1	X
1	1	0	1	1	1	1	0	X	0	X	0	1	X	X	1
1	1	1	0	1	1	1	1	X	0	X	0	X	0	1	X
1	1	1	1	0	0	0	0	X	1	X	1	X	1	X	1

Figure 3: State table for the example

Notice that the next state equals the present state plus one.

To design this circuit, we derive the flip-flop input equations from the state transition table. Recall that to find J & K values, we have to use:

- The present state,
- The next state, and
- The JK flip-flop excitation table.

When the *count* reaches 1111, it resets back to 0000, and the count cycle is repeated.

Once the J and K values are obtained, the next step is to find out the simplified input equations by using K-maps, as shown in figure 4.

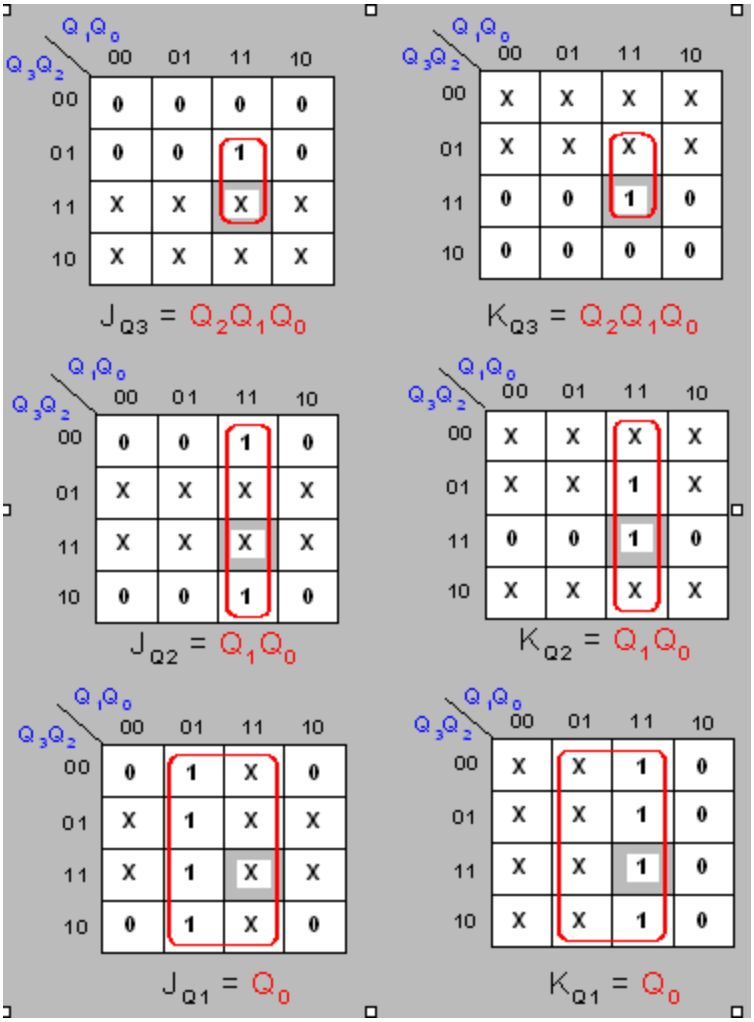


Figure 4: K-maps for the example

Notice that the maps for J_{Q0} and K_{Q0} are not drawn because the values in the table for these two variables either contain 1's or X's. This will result in $J_{Q0} = K_{Q0} = 1$

Note that the Boolean equation for J input is the same as that of the K input for all the FFs $\Rightarrow$ Can use T-FFs instead of JK-FFs.

Count Enable Control

In many applications, controlling the counting operation is necessary $\Rightarrow$ a count-enable (E_n) is required.

If E_n= 1 **then** counting of incoming clock pulses is enabled **Else if** (E_n =0), no incoming clock pulse is counted.

To accommodate the enable control, two approaches are possible.

1. Controlling the clock input of the counter
2. Controlling FF excitation inputs (JK, T, D, etc.).

Clock Control

Here, instead of applying the system clock to the counter directly, the clock is first ANDed with the E_n signal.

Even though this approach is simple, it is not recommended to use particularly with configurable logic, e.g. FPGA's.

FF Input Control (Figure 5)

In this case, the E_n =0 causes the FF inputs to assume the no change value (SR=00, JK=00, T=0, or D_i=Q_i).

To include E_n, analyze the stage when $J_{Q1} = K_{Q1} = Q_0$, and then include E_n. Accordingly, the FF input equations of the previous 4-bit counter example will be modified as follows:

$J_{Q0} = K_{Q0} = 1.\ E_N = E_n$

$J_{Q1} = K_{Q1} = Q_0.\ E_n$

$J_{Q2} = K_{Q2} = Q_1.Q_0.\ E_n$

$J_{Q3} = K_{Q3} = Q_2.Q_1.Q_0.\ E_n$

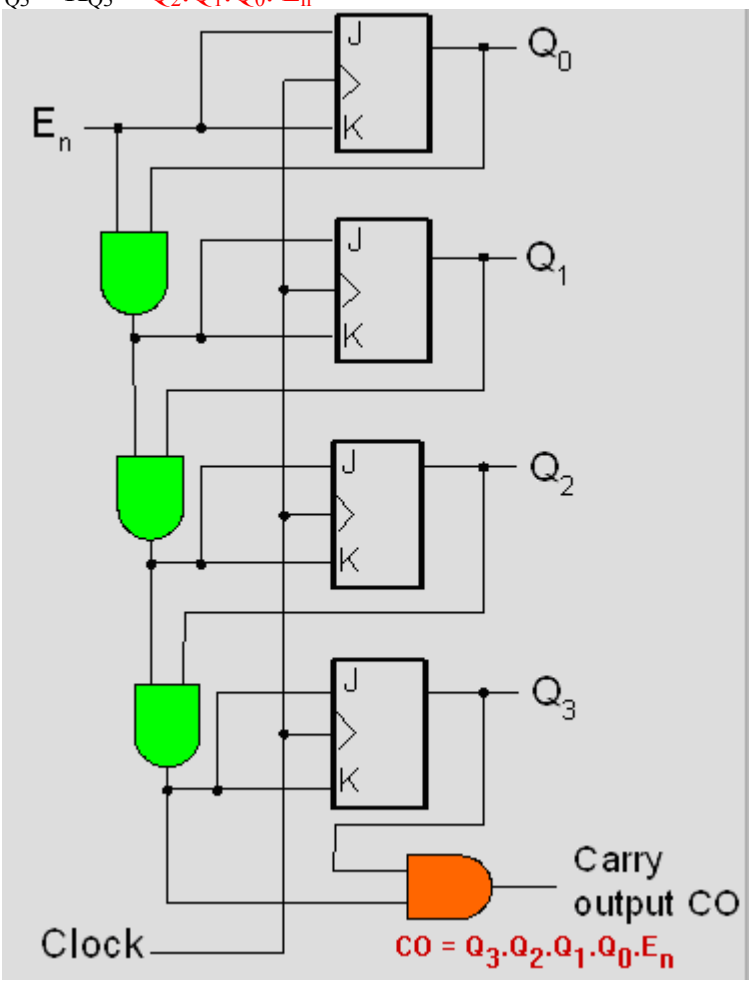


Figure 5: FF input control in counter

Thus, when E_n = 0, all J and K inputs are equal to zero, and the flip flops remain in the same state, even in the presence of clock pulses

When E_n = 1, the input equations are the same as equations of the previous example.

A carry output signal (CO) is generated when the counting cycle is complete, as seen in the timing diagram.

The CO can be used to allow cascading of two counters while using the same clock for both counters. In that case, the CO from the first counter becomes the E_n for the second counter. For example, two modulo-16 counters can be cascaded to form a modulo-256 counter.

Up-Down Binary Counters

In addition to counting up, a SBC can be made to count down as well.

A control input, S is required to control the direction of count.

IF S= 1, the counter counts up, otherwise it counts down.

FF Input Control

Design a Modulo-8 up-down counter with control input S, such that if S= 1, the counter counts up, otherwise it counts down. Show how to provide a count enable input and a carry-out (CO) output. (See figures 6 & 7)

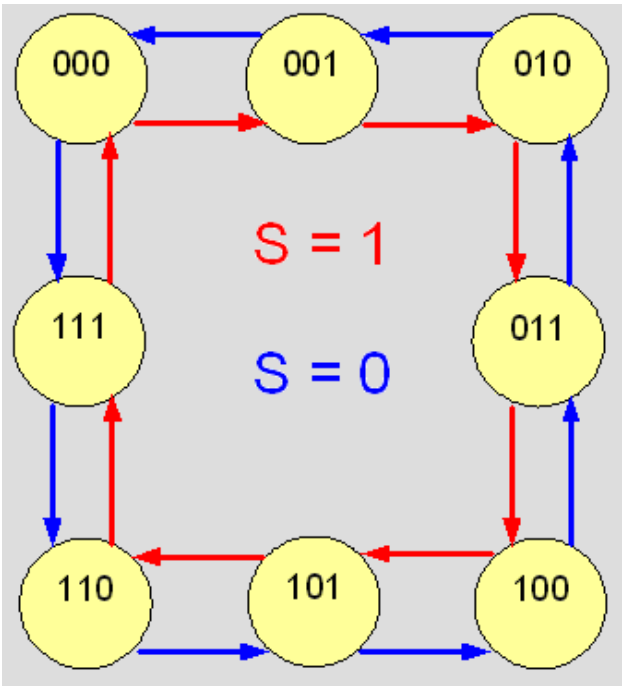


Figure 6: State diagram for FF input control example

Present State				Next State			Flip-Flop Inputs		
Q_2	Q_1	Q_0	S	Q_2	Q_1	Q_0	T_2	T_1	T_0
0	0	0	0	1	1	1	1	1	1
0	0	0	1	0	0	1	0	0	1
0	0	1	0	0	0	0	0	0	1
0	0	1	1	0	1	0	0	1	1
0	1	0	0	0	0	1	0	1	1
0	1	0	1	0	1	1	0	0	1
0	1	1	0	0	1	0	0	0	1
0	1	1	1	1	0	0	1	1	1
1	0	0	0	0	1	1	1	1	1
1	0	0	1	1	0	1	0	0	1
1	0	1	0	1	0	0	0	0	1
1	0	1	1	1	1	0	0	1	1
1	1	0	0	1	0	1	0	1	1
1	1	0	1	1	1	1	0	0	1
1	1	1	0	1	1	0	0	0	1
1	1	1	1	0	0	0	1	1	1

Figure 7: State table for FF input control example

The equations are (see figure 8)

$T_0 = 1$

$T_1 = Q_0 . S + Q'_0 . S'$

$T_2 = Q_1.Q_0 . S + Q'_1 . Q'_0 . S'$

The carry outputs for the next stage are: (see figure 8)

$C_{up} = Q_2.Q_1.Q_0$ for upward counting.

$C_{down} = Q'_2.Q'_1.Q'_0$ for downward counting.

The equations with E_n are (see figure 9)

$T_0 = E_n . 1$

$T_1 = Q_0 . S . E_n + Q'_0 . S' . E_n$

$T_2 = Q_1.Q_0 . S . E_n + Q'_1 . Q'_0 . S' . E_n$

The carry outputs for the next stage, with E_n are (see figure 9):

$C_{up} = Q_2.Q_1.Q_0 . E_n$ for counting up.

$C_{down} = Q'_2.Q'_1.Q'_0 . E_n$ for counting down.

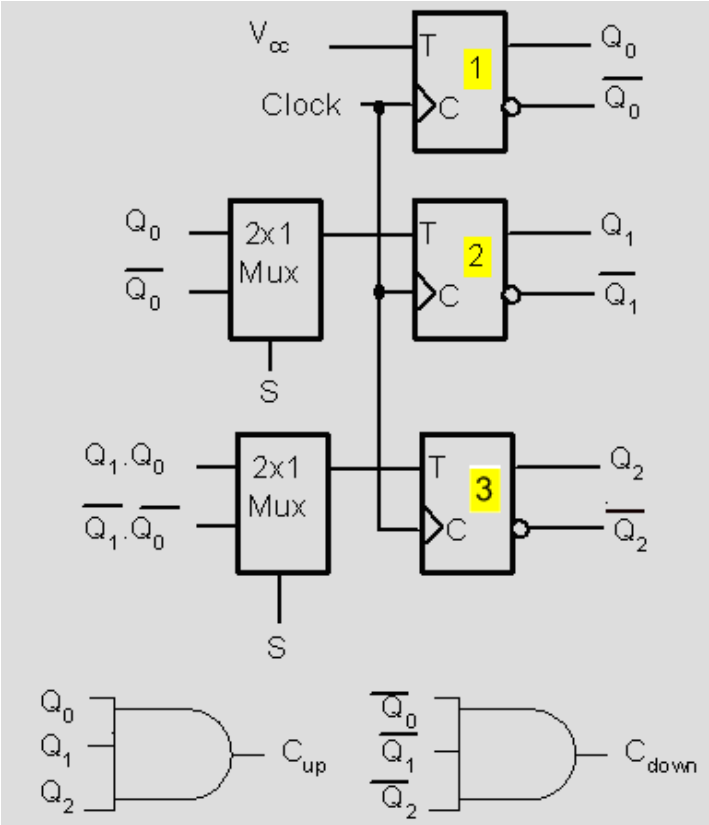


Figure 8: Circuit of up-down counter

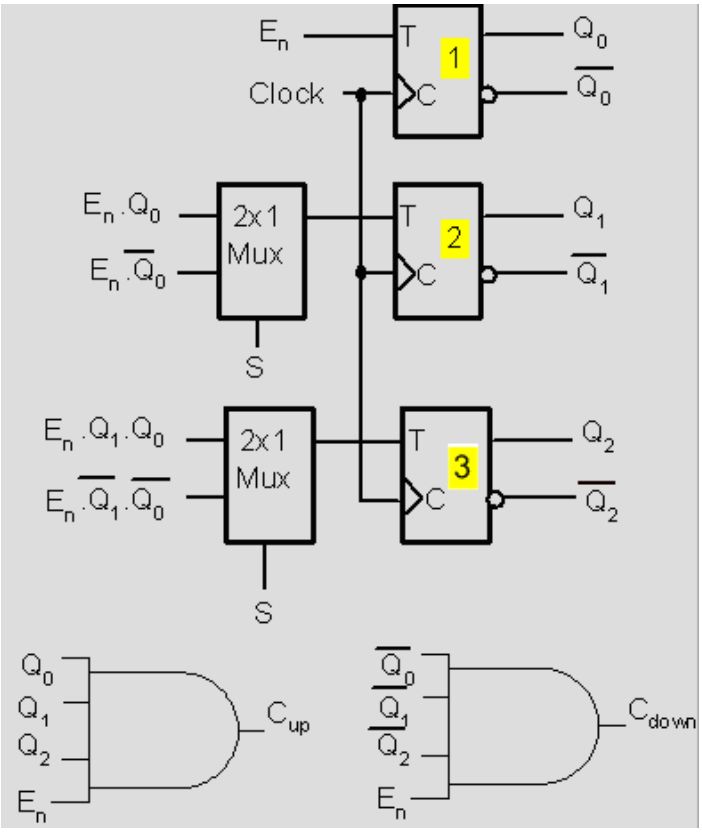


Figure 9: Circuit of up-down counter with En

More on Counters

In this lesson, you will learn

- some important counter control inputs:
  - Parallel Load (Ld)
  - Synchronous Clear
  - Asynchronous Clear
- use of available counters to build counters of different count

Counter Control

We have seen how to include a count-enable control input to enable/disable counting in the counter.

Now we show how to include important counter control inputs; namely:

- Parallel Load (Ld)
- Clear (Synchronous/Asynchronous)

The block diagram of a 4-bit counter with the above capabilities is shown in Figure 1.

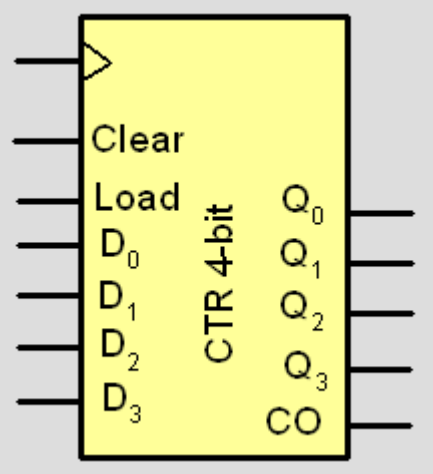


Figure 1: A 4-bit counter

Let us now discuss the design of the counter. We will start with a typical stage of basic counter, and will add the control signals to this stage in a step-wise approach. A positive edge-triggered counter will be assumed.

Figure 2 shows a stage of the basic counter, where we see that the J and K inputs of flip-flop at stage 1 are connected to an AND gate with Q_0 and Count as inputs.

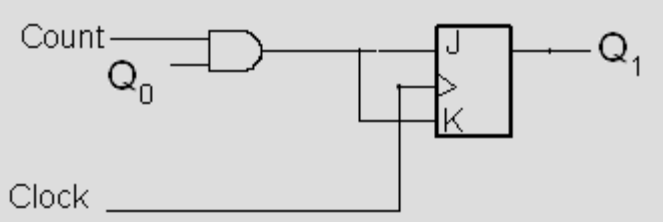


Figure 2: a stage of basic counter

Now let's add the Load control input to this stage. Thus the operation would become as shown in Table 1.

Count	Ld	Clk	Operation	Mode
0	0	x	$Q_i \leftarrow Q_i$ $\forall i = 0, n-1$	Count disable
x	1	↑	$Q_i \leftarrow D_i$ $\forall i = 0, n-1$	Parallel Load
1	0	↑	$Q_i \leftarrow Q_i + 1$ $\forall i = 0, n-1$	Count enable

Table 1: Operation of counter with Load signal added

Based on the above table, the stage will be modified as shown in Figure 3.

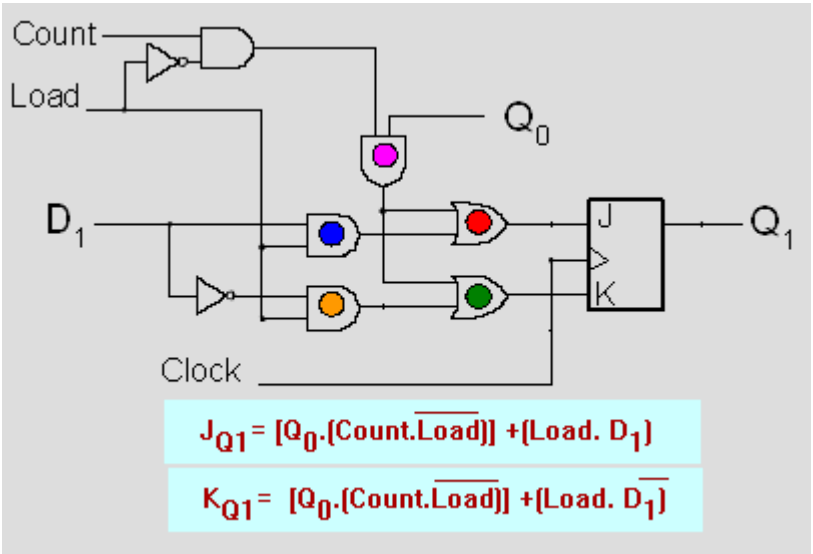


Figure 3: Modified stage of a counter

In this figure, when Count = 0 and Load = 0, J and K inputs in the flip flop will be equal to 0 and 0 ⇒ NO CHANGE state is achieved. Notice that AND gate marked with pink will also be generating an output of 0.

When Count = 0 or 1 (i.e. don't care) and Load = 1, parallel load operation will take place. The blue AND gate will propagate D while orange will propagate D' to J and K inputs respectively.

When Count = 1 and Load = 0, the circuit will operate as counter because JK =11. It is the same behavior with respect to the tradition counter.

Now, the control signal Clear can be added to the stage. We will assume *asynchronous* Clear in the design.

Count	Ld	Clr	Clk	Operation	Mode
x	x	1	x	$Q_i \leftarrow 0$ $\forall i = 0, n\text{-}1$	Asynchronous Clear
x	1	0	↑	$Q_i \leftarrow D_i$ $\forall i = 0, n\text{-}1$	Parallel Load
1	0	0	↑	$Q_i \leftarrow Q_i + 1$ $\forall i = 0, n\text{-}1$	Count enable
0	0	0	x	$Q_i \leftarrow Q_i$ $\forall i = 0, n\text{-}1$	Count disable

Table 2: Counter Operation with Count, Load, and Clear signals

The stage will be modified as given in Figure 4.

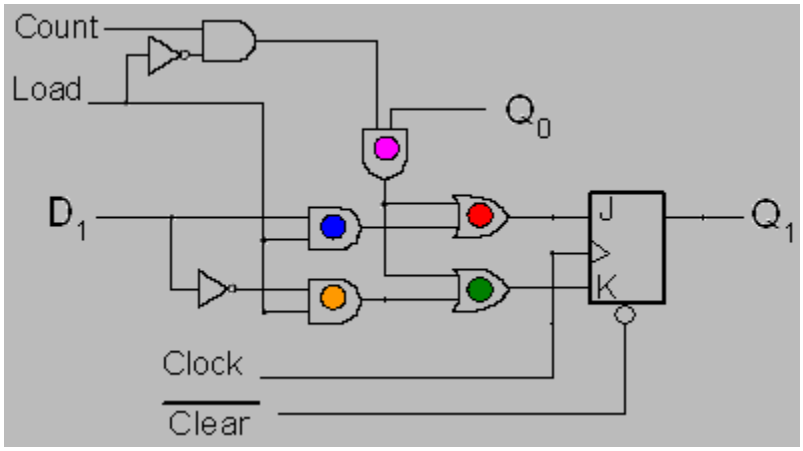


Figure 4: Addition of Clear signal to a counter stage

Designing counters with available counters

Binary counters with parallel load can be used to design different modulo-*n* counters. For example, the 4-bit parallel load counter discussed in this lesson can be used to design any counter of modulo *n* where $2 \leq n \leq 16$.

Design of decade counter

The binary counter with parallel load can be converted into a synchronous DECADE counter (without load input) by connecting an external AND gate to it, as shown in Figure 5.

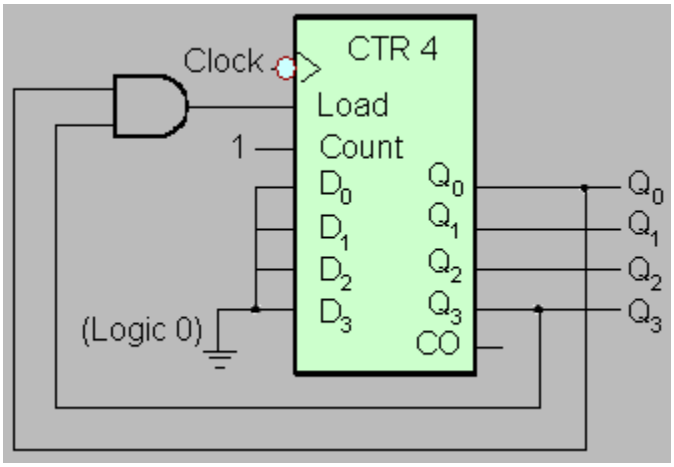


Figure 5: Decade counter

Connect all the D inputs to Ground (Logic 0). Make Count = 1. This will make the circuit always operating in the counter mode.

The 2-input AND gate connected to the Load input of the counter takes Q_0 and Q_3 as its inputs. As long as the Load input (connected to the output of the AND gate) is 0, the counter is incremented by one with each clock pulse.

When the output count reaches 9 (1001), the output of AND gate will equal 1. This puts the counter in the *load mode* (Load =1). Thus, *the next clock pulse* will load the data on the D-inputs (0000) into the counter instead of incrementing the count.

Thus the counter counts from 0000 (decimal 0) to 1001 (decimal 9) then goes back to 0000 and so on. → *Modulo 10 counting*

Design of a counter that counts from 3 to 12.

The counter discussed above can be made to count from 0011 (decimal 3) to 1100 (decimal 12). Only small modifications are required, which are (see Figure 6):

- Connect the D inputs to 0011 (i.e. $D_3D_2D_1D_0 = 0011$). This will make the circuit start counting from 3 whenever 12 has been counted.
- Connect the AND gate to Q_3 and Q_2. Thus, whenever the circuit reaches 12 (1100), $Q_3 = Q_2 = 1$, which will make the output of the AND gate equal to 1, making Load active, *so in the next clock transition* the counter does not count, but is loaded from its four inputs, with a value of 0011.

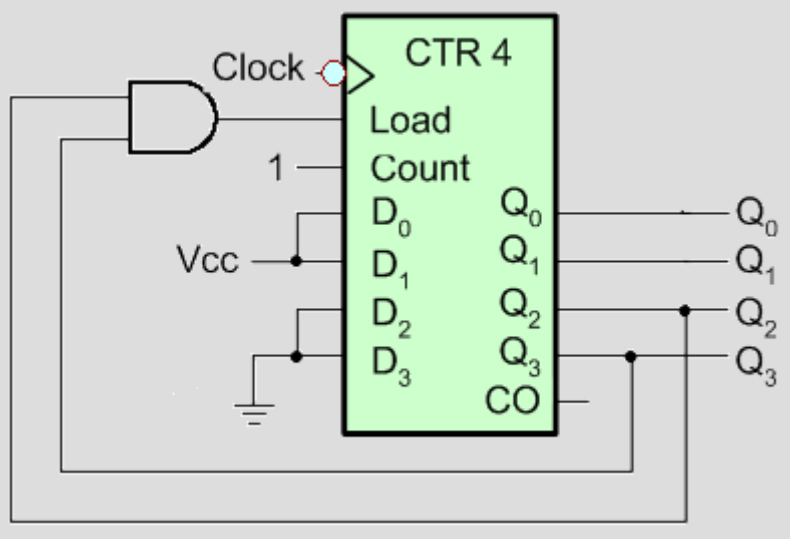


Figure 6: Counter for 3 to 12 counting

Thus the counter counts from 0011 (decimal 3) to 1100 (decimal 12), and back to 0011. This is also a mod-10 counter, since it also counts ten numbers.

Some counters may have the "Clear" control input. With this capability, the counter can be "cleared" at any time. The "Clear" signal can also be classified into two types: Synchronous and Asynchronous.

The Synchronous Clear case

The Synchronous Clear input is activated *in synchronization* with the clock pulse.

To explain this behavior, consider a **MOD-6** counter that counts from 000 (decimal 0) to 101 (decimal 5). The circuit is shown in the figure.

In this circuit, once the count of 5 (101) is detected by the AND gate, the counter is cleared on the next clock pulse. Thus, the counter counts from 0 to 5 back to 0.

Assuming negative edge-triggered FFs, the timing diagram of this counter is shown in Figure 7. Notice the delayed transitions of the counter outputs (Q's) after the negative clock edge due to gate propagation delays.

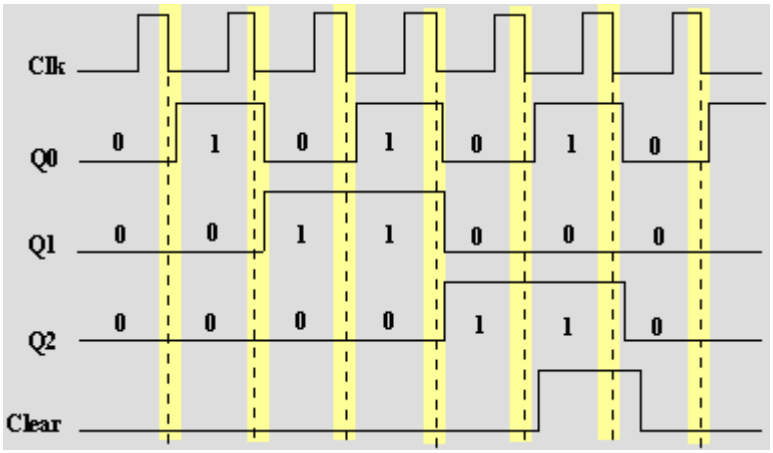


Figure 7: Timing diagram – Synchronous Clear

The timing diagram clarifies the case of $Q_2Q_1Q_0 = 101$ where the Clear input becomes 1 causing the counter to clear on the next negative clock edge. The point to notice here is that the effect of change in "Clear" is not immediately applied, but becomes effective in the following clock pulse, because the "Clear" input is "Synchronous", i.e. *it only takes effect at the next active clock edge*.

The Asynchronous Clear case

If we use asynchronous clear rather than synchronous clear, as soon as the count $Q_2Q_1Q_0$ reaches 101, "Clear"is activated and the FFs are cleared immediately without waiting for the next active clock edge.

This causes the count $Q_2Q_1Q_0 = 101$ to switch to 000 after a small delay. In other words the count 101 does not last for a full clock period as other counts, but rather will appear for a very short duration as a narrow pulse (glitch) as shown in Figure 8. Thus, it would appear that the counter counts from 0 to 4, that is, from 000 to 100.

This happens because "Clear" is "Asynchronous". It does not wait for the clock pulse to come, and does the "clearing" operation immediately.

As a result, the output values become $Q_2Q_1Q_0 = 101$ for a very short duration of time, almost negligible, and then the contents become $Q_2Q_1Q_0 = 000$ within the same clock period.

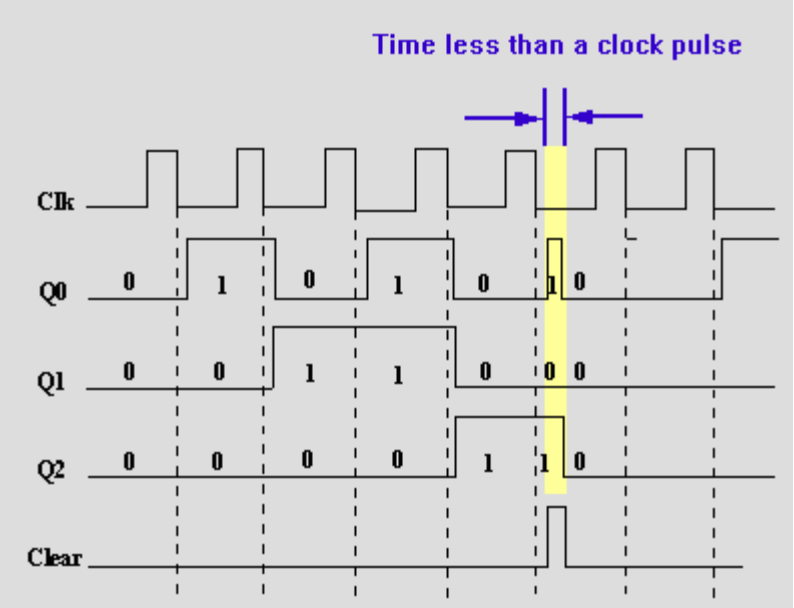


Figure 8: Timing diagram – Asynchronous Clear

To have a MOD6 counter designed using the asynchronous clear, we should detect a count of 6 (instead of 5) and use that to clear the counter asynchronously.
In this case, once the count reaches $Q_2Q_1Q_0 = 110$, "Clear" is activated, you will not be able to observe $Q_2Q_1Q_0 = 110$, because it will be for very short duration. This is shown in Figure 9.

It would seem to you that after $Q_2Q_1Q_0 = 101$, the next state is $Q_2Q_1Q_0 = 000$.

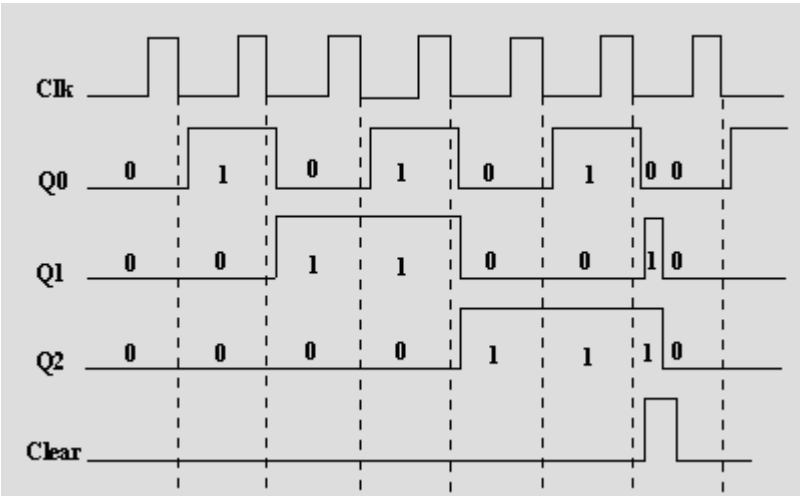


Figure 9: Timing diagram

Ripple and Arbitrary Counters

In this lesson, you will learn about:

- Ripple Counters
- Counters with arbitrary count sequence

Design of ripple Counters

Two types of counters are identifiable:

- *Synchronous* counters, which have been discussed earlier, and
- *Ripple* counters.

In ripple counters, flip-flop output transitions serve as a source for triggering other flip-flops.

In other words, clock inputs of the flip-flops are triggered by output transitions of other flip-flops, rather than a common clock signal.

Typically, T flip-flops are used to build ripple counters since they are capable of complementing their content (See Figure 1).

The signal with the pulses to be counted, i.e."*Pulse*", is connected to the clock input of the flip-flop that holds the **LSB** (FF # 1).

The output of each FF is connected to the clock input of the next flip-flop in sequence.

The flip-flops are negative edge triggered (bubbled clock inputs).

T=1 for all FFs (J = K= 1). This means that each flip-flop complements its value if C input goes through a negative transition (1 → 0).

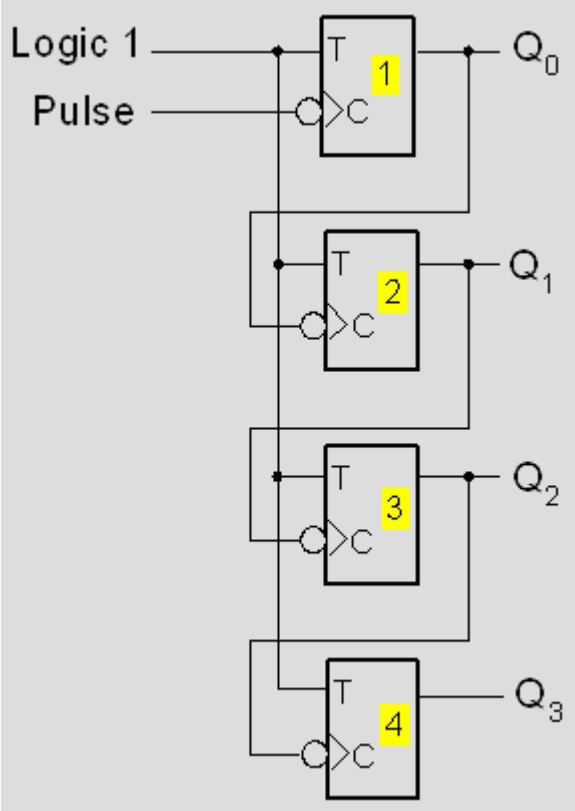


Figure 1: A ripple counter

The previous ripple up-counter can be converted into a down-counter in one of two ways:

- Replace the negative-edge triggered FFs by positive-edge triggered FFs, or
- Instead of connecting C input of FF Q_i to the output of the preceding FF (Q_{i-1}) connect it to the complement output of that FF (Q'_{i-1}).

Advantages of Ripple Counters:

- simple hardware and design.

Disadvantages of Ripple Counters:

- They are asynchronous circuits, and can be unreliable and delay dependent, if more logic is added.
- Large ripple counters are slow circuits due to the length of time required for the ripple to occur.

Counters with Arbitrary Count Sequence:

Design a counter that follows the count sequence: 0, 1, 2, 4, 5, 6. This counter can be designed with any flip-flop, but let's use the JK flip-flop.

Notice that we have two "*unused*" states (3 and 7), which have to be dealt with (see Figure 2). These will be marked by don't cares in the state table (Refer to the design of sequential circuits with unused states discussed earlier). The state diagram of this counter is shown in Figure 2.

In this figure, the unused states can go to any of the valid states, and the circuit can continue to count correctly. One possibility is to take state 7 (111) to 0 (000) and state 3 (011) to 4 (100).

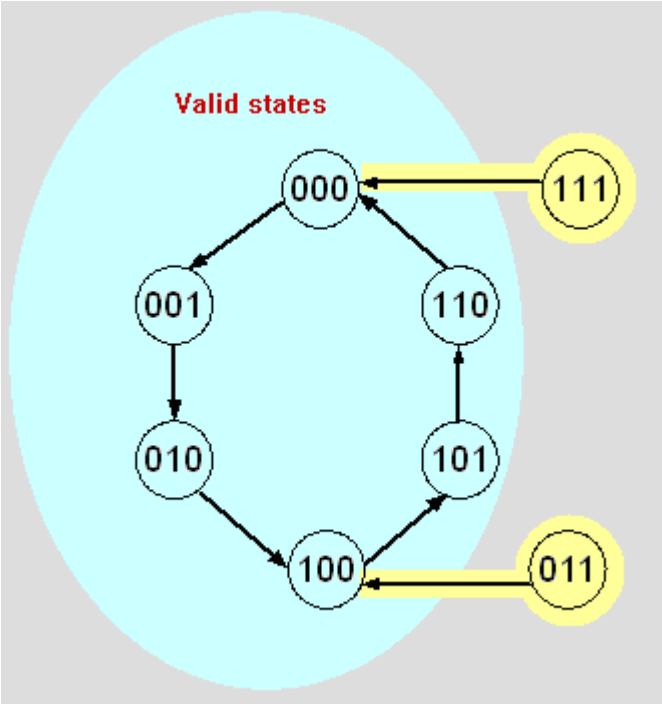


Figure 2: State diagram for arbitrary counting sequence

The design approach is similar to that of synchronous circuits. The state transition table is built as shown in Figure 3 and the equations for all J and K inputs are derived. Notice that we have used don't care for the unused state (although we could have used 100 as the next state for 011, and 000 as the next state of 111).

Unused states

Present State			Next State			Flip-Flop Inputs					
A	B	C	A	B	C	J_A	K_A	J_B	K_B	J_C	K_C
0	0	0	0	0	1	0	X	0	X	1	X
0	0	1	0	1	0	0	X	1	X	X	1
0	1	0	1	0	0	1	X	X	1	0	X
0	1	1	X	X	X	X	X	X	X	X	X
1	0	0	1	0	1	X	0	0	X	1	X
1	0	1	1	1	0	X	0	1	X	X	1
1	1	0	1	1	1	X	0	X	0	1	X
1	1	1	X	X	X	X	X	X	X	X	X

Figure 3: State table for arbitrary counting sequence

The computed J and K input equations are as follows:

$J_A = B$ $K_A = B$

$J_B = C$ $K_B = 1$

$J_C = B'$ $K_C = 1$

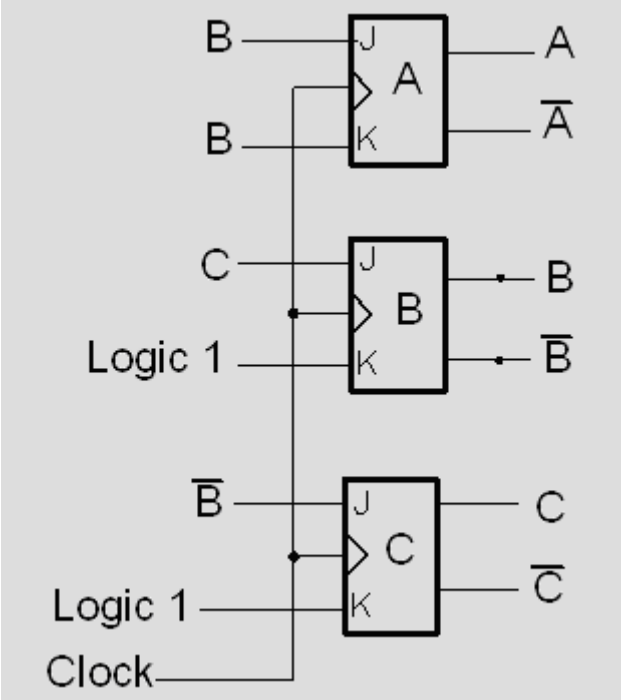


Figure 4: Circuit for arbitrary counting sequence

Semiconductor Memories: RAMs and ROMs

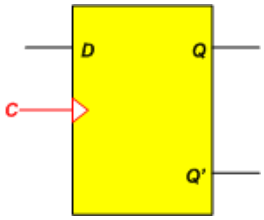
Lesson Objectives:

In this lesson you will be introduced to:

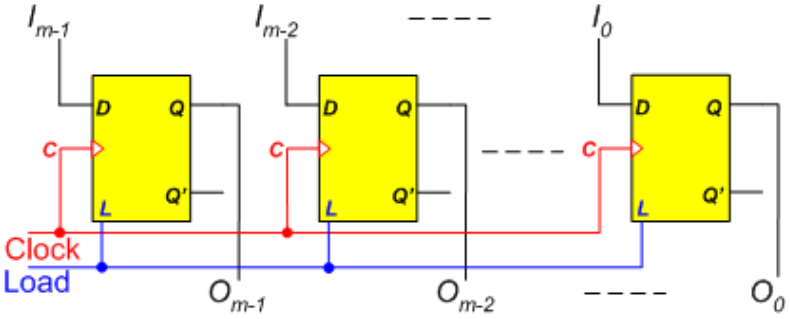
- Different memory devices like, ***RAM, ROM, PROM, EPROM, EEPROM,*** etc.
- Different terms like: ***read, write, access time, nibble, byte, bus, word, word length, address, volatile, non-volatile etc.***
- How to implement combinational and sequential circuits using ROM.

Introduction:

The smallest unit of information a digital system can store is a ***bit***, which can be stored in a ***flip-flop*** or a ***1-bit register***.



To store *m* bits of data, an ***m-bit register*** with parallel load capability may be used. Data available on the ***m***-bit input lines (I_0 to I_{m-1}) may be stored/*written* into this register under control of the clock by asserting the "*Load*" control input. The stored ***m*** bits of data may be *read* from the register outputs (O_0 to O_{m-1}).

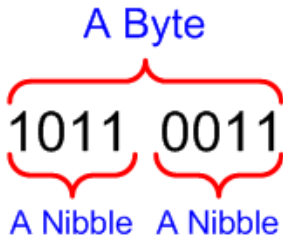


The ***m*** bits of data stored in a register make up a ***word***. It is simply a number of bits operated upon or considered by the hardware as a group. The number of bits in the word, ***m,*** is called ***word length***.

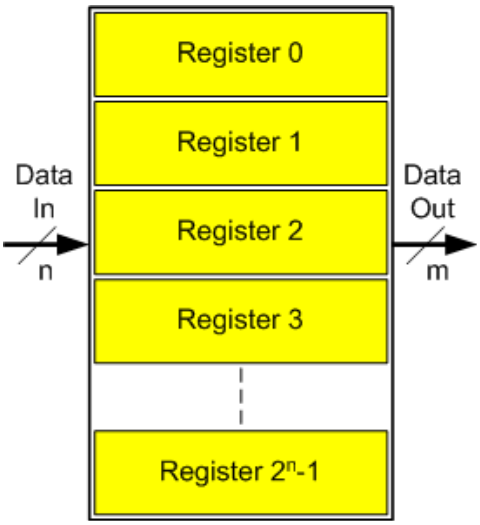
The *m* inputs of the register are provided through an *m*-bit input data *bus* and *m* outputs by an *m*-bit output data *bus*.

A ***bus*** is a number of signal lines, grouped together because of similarity of function, which connect two or more systems or subsystems.

A unit of 8-bits of information is referred to as a ***byte***, while 4-bits of information is referred to as a ***nibble***.



A ***memory*** device can be looked at as consisting of a number of equally sized registers sharing a *common set of inputs*, and a *common set of outputs*, as shown in the Figure.



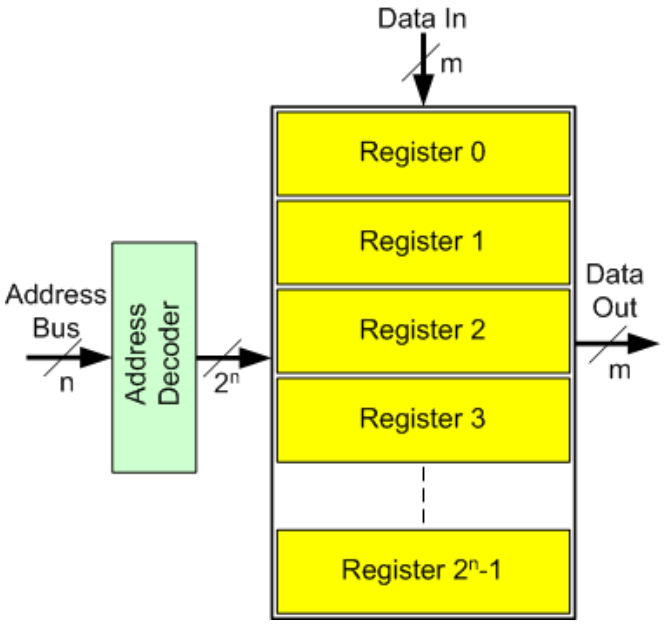
Storing data in a memory register is referred to as a memory ***write*** operation and looking up the contents of a memory register is referred to as a memory ***read*** operation.

In case of a write operation, the input data need to be written into one *particular* register in the memory device.

Since the input data lines are common to all registers of the memory device, only the selected register should have its ***load*** control signal asserted while the other registers should not.

If the number of registers is 2^n, n lines will be required to select the register to be written into. The n-lines are used as an input to a decoder where the decoder's 2^n outputs may be used as the ***load*** control inputs to the 2^n registers.

The load control signal of a particular register is asserted by a unique combination of the n-select lines. This unique combination is considered as the ***address*** for that particular register.

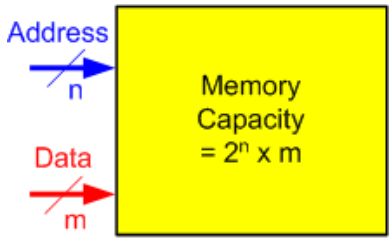


Thus, a memory device can be thought of as a collection of **addressable** registers.

A read or a write operation into the memory device has to specify the address of the particular register to be read or written into.

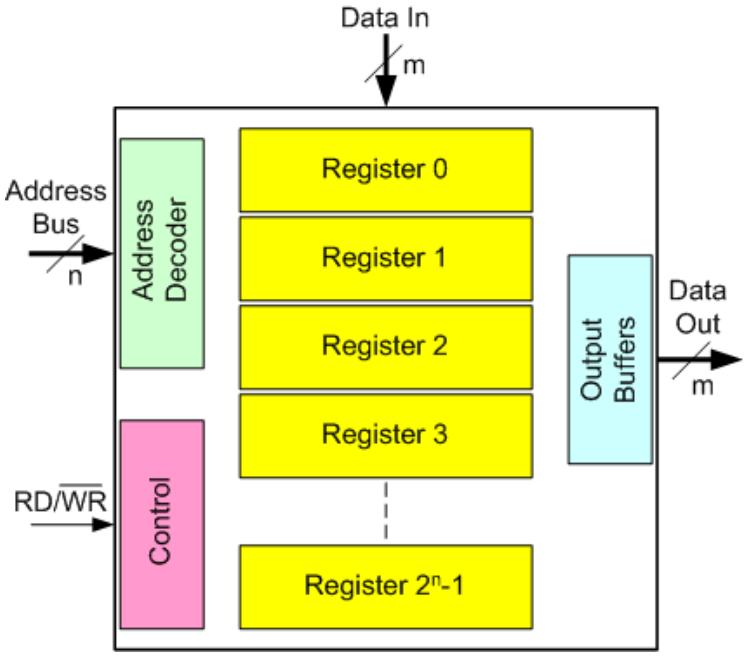
The ***capacity*** of the memory is specified in terms of the number of bits or the number of words available in this memory device.

For a memory device with ***n***-bit address lines and word (register) size of ***m***-bits, the memory has 2^n words (storage locations/registers) each having ***m*** bits for a total capacity of 2^n **x** ***m*** bits.



For example, if $n = 10$ and $m = 8$, the memory is a **"1024 x 8"** bit memory. Alternatively, it is said that the memory has 1K bytes.

A block diagram of the memory device is shown in the figure. The address inputs are decoded by **address decoder** to select one, and only one, of the memory words (registers), either for reading or writing.



The $RD/\overline{WR}$ line is a control signal that determines the type of operation to be performed; a read operation or a write operation.

$RD/\overline{WR}=1$ indicates a read operation, while $RD/\overline{WR}=0$ indicates a write operation.

To **read** the memory contents stored in a particular word, the address of this word is applied, and logic 1 is applied to the $RD/\overline{WR}$ line that enables the output buffers of the memory.

To **write** at a location, the address of the location to be written is provided at the address inputs, data is provided at the data inputs, and logic 0 is applied to the $RD/\overline{WR}$ line.

There is a time delay between the application of an address and the appearance of contents at the output, this is called the memory ***access time***. This depends both on the technology and on the structure used to implement the memory.

Random Access Memory (RAM):

For the shown above memory structure, the access time is independent of the sequence in which addresses are applied.

Such a memory is called ***random access memory (RAM).*** Thus, the contents of any one location can be accessed in essentially the same time as can the contents of any other location chosen at random.

RAMs are ***volatile*** memories that will only retain the stored data as long as power is **ON** but will lose this data when power is turned **OFF**.

RAMs are classified into two main categories: Static RAM ***(SRAM)*** and Dynamic RAM ***(DRAM).*** These will be studied in greater details in future courses.

Read Only Memory (ROM):

Read Only Memory (ROM) is memory whose stored data can only be read but cannot be re-written (altered).

It is a device in which "permanent" binary information has been stored.

ROMs are ***nonvolatile*** where stored data are not lost even when power is turned **OFF**.

The Figure shows a block diagram of a ROM.

Like RAMS, a ROM has ***n*** *address* inputs and ***m*** outputs. This corresponds to 2^n memory words each of *m* storage bits for a total capacity of 2^n **x** ***m*** bits.

Unlike RAMs, ROMs do not have data input lines, because they do not have a write operation.

ROMs are common to use in storing system-level programs that should be available at all times.

The most common example is the PC system BIOS (Basic Input Output System), which is stored in a ROM called the *system BIOS ROM*.

Several classes of ROMs are in common use. These may be categorized according to their fabrication technologies that influence the way data are introduced into the ROM. The process of storing the desired data into the ROM is referred to as *ROM programming*.

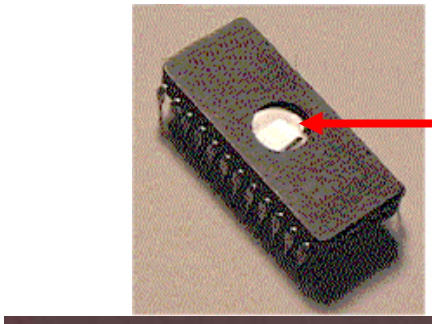
Types of ROMs:

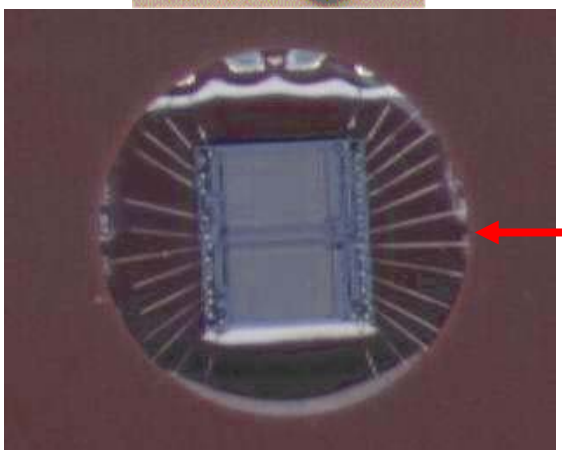
Following are the different types of ROMs.

1. Programming is done by the manufacturer during the last fabrication steps according to the truth table provided by the customer. This type is known as mask programmable ROMs or simply ***ROM***. Data stored this way can never be altered.

2. ROM is provided with fuses to allow users to introduce the desired data by electrically blowing some of these fuses. This type is referred to as a *programmable ROM*, or ***PROM***. Fuse blowing is irreversible and, once programmed the ROM stored pattern cannot be altered.

3. The ROM uses *erasable floating-gate* memory cells that allow erasure of the stored data by Ultra-Violet light. In this type, programming is performed *electrically* by the user using special hardware programmers. Data, thus stored, can later be erased globally (all memory bits = 1) by exposing the memory array to UV-light. This ROM type is referred to as UV-*erasable, programmable ROM*, or simply ***EPROM***. The EPROM IC package is provided with a quartz window to allow UV-light penetration to the memory array.





4. When special electrically erasable memory cells are used, the ROM can be electrically erased at the byte level. Thus individual bytes may be addressed and programmed or erased as desired. This type is referred to as *electrically erasable, programmable ROM*, or ***EEPROM*** or ***E^2PROM***. The ***E^2PROM*** technology is an expensive low-capacity technology and is thus not used for high density or low-cost applications.

5. The most recent ROM technology is the *flash* technology that combines the low-cost and high-density advantages of the UV-EPROM technology and the flexibility of electrical erase of ***E^2PROM*** technology. This technology is electrically erasable but the erasure is performed either globally (the full array) or partially on complete sub-arrays (sectors).

Combinational Circuit Implementation Using ROM:

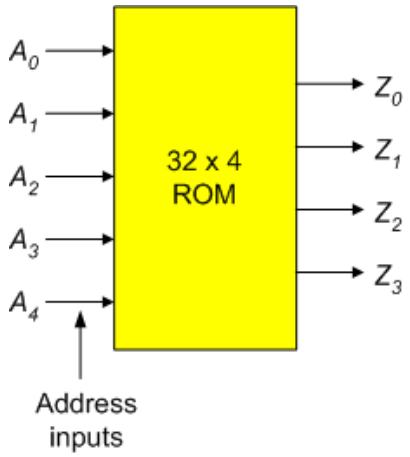
ROM devices can be used to implement complex combinational circuits directly from truth tables without need for minimization.

For an ***n***-input, ***m***-output combinational circuit, a 2^n **x** ***m*** ROM is needed (2^n words each of ***m*** storage bits). The designer needs only to specify a ROM table that gives the information stored in each of the 2^n words.

When a combinational circuit is implemented using a ROM, the function may either be expressed in the sum of minterms form, or using a truth table.

As an example, the ROM shown in the figure may be considered as a combinational circuit with four outputs, each a function of the five input variables.

Outputs $Z_0 - Z_3$ can be expressed as sum of minterms as follows:
$Z_0 (A_4, A_3, A_2, A_1, A_0) = \sum m\ (2, 3, 18, 21, 31)$
$Z_1 (A_4, A_3, A_2, A_1, A_0) = \sum m\ (0, 1, 17, 25, 31)$
$Z_2 (A_4, A_3, A_2, A_1, A_0) = \sum m\ (1, 6, 11, 29, 30)$
$Z_3 (A_4, A_3, A_2, A_1, A_0) = \sum m\ (7, 8, 16, 28, 29)$



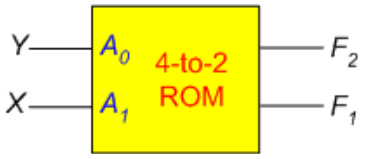
Example 1:

Consider a combinational circuit which is specified by the following two functions:
F_1 (X, Y) = $\sum$m (1, 2, 3)
F_2 (X, Y) = $\sum$m (0, 2)

The truth table for this circuit is as shown.

X	Y	F_1	F_2
0	0	0	1
0	1	1	0
1	0	1	1
1	1	1	0

In this example, the ROM that implements the two combinational functions must have two address inputs and two outputs. Thus, its size must be 4 x 2 (since 2^n **x** ***m*** is the size of ROM).



The ROM table for this example is as shown.

ROM Address		Stored Information	
A_1	A_0	F_1	F_2
0	0	0	1
0	1	1	0
1	0	1	1
1	1	1	0

Example 2:

Design a combinational circuit using a ROM. The circuit accepts a 3-bit number and generates an output binary number that is equal to the square of the input number.

The first step is to derive the truth table for the combinational circuit as shown. Three inputs and six outputs are needed to accommodate all possible numbers.

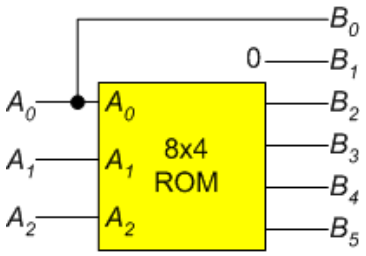
Inputs			Outputs						
A_2	A_1	A_0	B_5	B_4	B_3	B_2	B_1	B_0	Decimal
0	0	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	1	1
0	1	0	0	0	0	1	0	0	4
0	1	1	0	0	1	0	0	1	9
1	0	0	0	1	0	0	0	0	16
1	0	1	0	1	1	0	0	1	25
1	1	0	1	0	0	1	0	0	36
1	1	1	1	0	0	0	0	1	49

By observation, we note that output B_0 is always equal to input A_0, and output B_1 is always 0. Thus, there is no need to store B_0 and B_1 in the ROM. We actually need to only store values of the four outputs (B_5 through B_2) in the ROM.

The table shown specifies all the information that needs to be stored in the ROM, and figure shows the required connections of the combinational circuit. The output B_1 is connected to logic 0 and output B_0 is connected to A_0 always to get $B_1 = 0$ and $B_0 = A_0$.

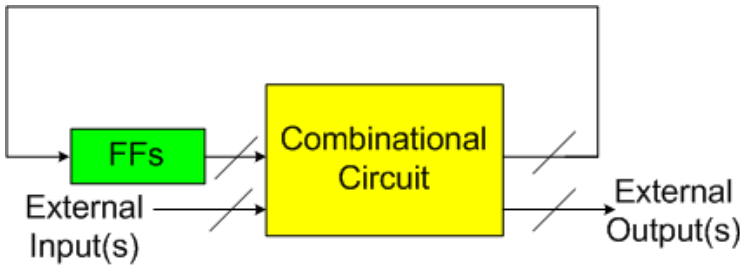
ROM Address			Stored Information			
A_2	A_1	A_0	B_5	B_4	B_3	B_2
0	0	0	0	0	0	0
0	0	1	0	0	0	0
0	1	0	0	0	0	1
0	1	1	0	0	1	0
1	0	0	0	1	0	0
1	0	1	0	1	1	0
1	1	0	1	0	0	1
1	1	1	1	0	0	0

The minimum size ROM needed must have three inputs and four outputs, for a total of 8 x 4 = 32 bits.

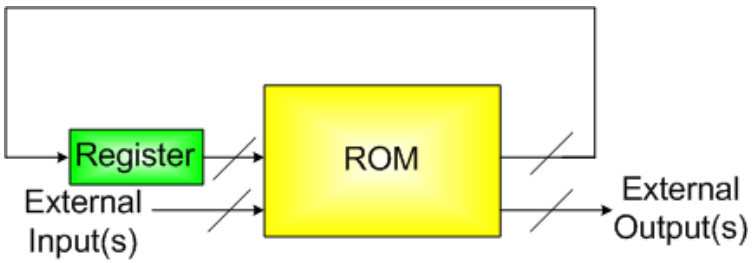


Synchronous Sequential Circuit Implementation Using ROM:

The block diagram of a sequential circuit is shown in the figure.



Since ROM can implement combinational logic, so this part can be replaced by a ROM and Flip-Flops can be replaced by a register as shown in the figure.



Example 3:

Design a sequential circuit whose state transition table is given, using a ROM and a register.

Present State		Input	Next State		Output
Q_2	Q_1	X	Q_2^+	Q_1^+	Y
0	0	0	0	0	0
0	0	1	0	1	0
1	0	0	0	1	0
1	0	1	0	0	1
0	1	0	1	0	0
0	1	1	0	1	0
1	1	0	1	1	0
1	1	1	0	0	1

The next-state and output information are obtained from the table as:

$Q_1^+ = \sum m\ (1, 2, 5, 6)$

$Q_2^+ = \sum m\ (4, 6)$

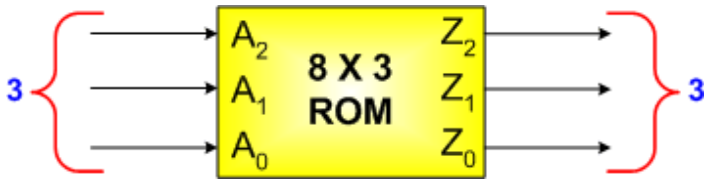
$Y\ (Q_2, Q_1, X) = \sum m\ (3, 7)$

The ROM can be used to implement the combinational circuit and register will provide the flip-flops.

The **number of address inputs** to the ROM is equal to the number of flip-flops plus the number of external inputs.

The **number of outputs** of the ROM is equal to the number of flip-flops plus the number of external outputs.

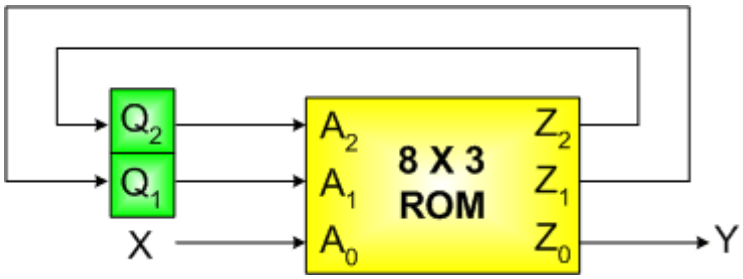
In this example, 3 inputs and 3 outputs of the ROM are required; so its size must be 8 x 3.



The ROM table is identical to the state transition table with Present State and Inputs specifying the **address** of ROM and Next State and Outputs specifying the ROM **outputs (stored information)**. It is shown below:

ROM Address			Stored Information		
A_2	A_1	A_0	Z_2	Z_1	Z_0
0	0	0	0	0	0
0	0	1	0	1	0
1	0	0	0	1	0
1	0	1	0	0	1
0	1	0	1	0	0
0	1	1	0	1	0
1	1	0	1	1	0
1	1	1	0	0	1

The next state values must be connected from the ROM outputs to the register inputs as shown in the figure below.



Programmable Logic Devices (PLDs)

Lesson Objectives:

In this lesson you will be introduced to some types of Programmable Logic Devices (PLDs):

- PROM, PAL, PLA, CPLDs, FPGAs, etc.
- How to implement digital circuits using PLAs and PALs.

Introduction:

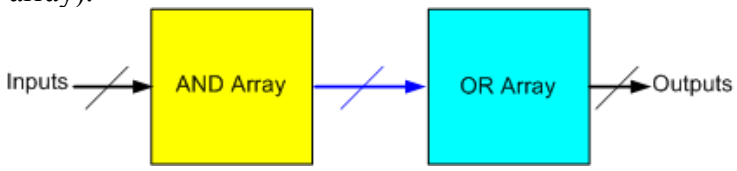
An IC that contains large numbers of gates, flip-flops, etc. that can be *configured by the user* to perform different functions is called a ***Programmable Logic Device (PLD)***.

The internal logic gates and/or connections of PLDs can be changed/configured by a programming process.

One of the simplest programming technologies is to use fuses. In the original state of the device, all the fuses are intact.

Programming the device involves blowing those fuses along the paths that must be removed in order to obtain the particular configuration of the desired logic function.

PLDs are typically built with an *array* of AND gates (AND-array) and an *array* of OR gates (OR-array).



Advantages of PLDs:

Problems of using standard ICs:

Problems of using standard ICs in logic design are that they require hundreds or thousands of these ICs, considerable amount of circuit board space, a great deal of time and cost in inserting, soldering, and testing. Also require keeping a significant inventory of ICs.

Advantages of using PLDs:

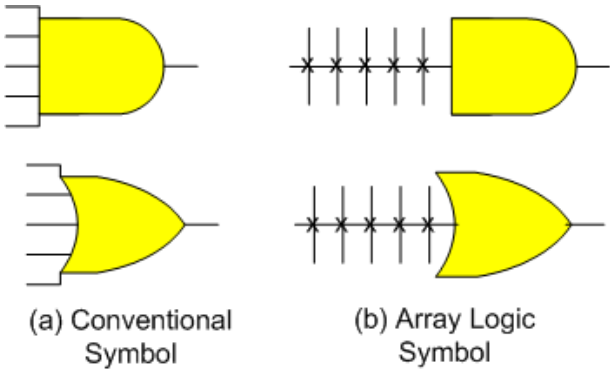
Advantages of using PLDs are less board space, faster, lower power requirements (i.e., smaller power supplies), less costly assembly processes, higher reliability (fewer ICs and circuit connections means easier troubleshooting), and availability of design software.

There are three fundamental types of standard PLDs: ***PROM, PAL,*** and ***PLA***.

A fourth type of PLD, which is discussed later, is the ***Complex Programmable Logic Device (CPLD),*** e.g., ***Field Programmable Gate Array (FPGA).***
A typical PLD may have hundreds to millions of gates.

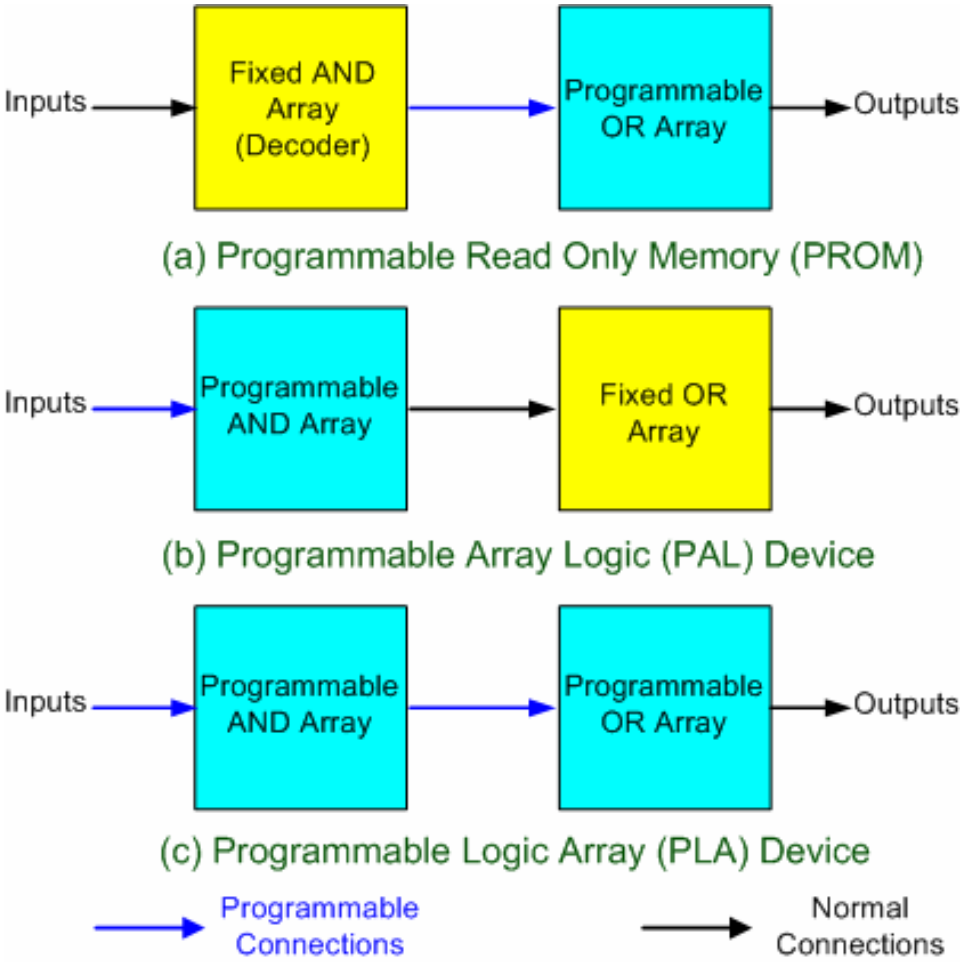
In order to show the internal logic diagram for such technologies in a concise form, it is necessary to have special symbols for array logic.

Figure shows the conventional and array logic symbols for a multiple input AND and a multiple input OR gate.



Three Fundamental Types of PLDs:

The three fundamental types of PLDs differ in the placement of programmable connections in the AND-OR arrays. Figure shows the locations of the programmable connections for the three types.



- The ***PROM (Programmable Read Only Memory)*** has a fixed AND array (constructed as a decoder) and programmable connections for the output OR gates array. The PROM implements Boolean functions in sum-of-minterms form.

- The ***PAL (Programmable Array Logic)*** device has a programmable AND array and fixed connections for the OR array.

- The ***PLA (Programmable Logic Array)*** has programmable connections for both AND and OR arrays. So it is the most flexible type of PLD.

The ROM (Read Only Memory) or PROM (Programmable Read Only Memory):
The input lines to the AND array are hard-wired and the output lines to the OR array are programmable.

Each AND gate generates one of the possible AND products (i.e., minterms).

In the previous lesson, you have learnt how to implement a digital circuit using ROM.

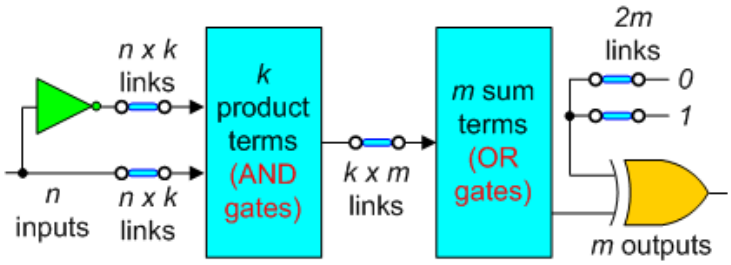
The PLA (Programmable Logic Array):
In PLAs, instead of using a decoder as in PROMs, a number (***k***) of AND gates is used where $k < 2^n$, (***n*** is the number of inputs).

Each of the AND gates can be programmed to generate a product term of the input variables and does not generate all the minterms as in the ROM.

The AND and OR gates inside the PLA are initially fabricated with the links (fuses) among them.

The specific Boolean functions are implemented in sum of products form by opening appropriate links and leaving the desired connections.

A block diagram of the PLA is shown in the figure. It consists of ***n*** inputs, ***m*** outputs, and ***k*** product terms.



The product terms constitute a group of ***k*** AND gates each of *2n* inputs.

Links are inserted between all ***n*** inputs and their complement values to each of the AND gates.

Links are also provided between the outputs of the AND gates and the inputs of the OR gates.

Since PLA has ***m***-outputs, the number of OR gates is ***m***.

The output of each OR gate goes to an XOR gate, where the other input has two sets of links, one connected to logic 0 and other to logic 1. It allows the output function to be generated either in the **true** form or in the **complement** form.

The output is inverted when the XOR input is connected to 1 (since $X \oplus 1 = X'$). The output does not change when the XOR input is connected to 0 (since $X \oplus 0 = X$).

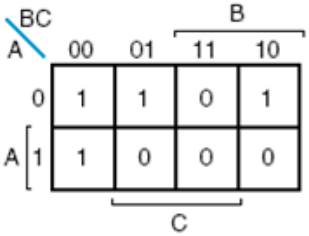
Thus, the total number of programmable links is ***2n*** x ***k*** + ***k*** x ***m*** + ***2m***.

The size of the PLA is specified by the number of inputs ***(n)***, the number of product terms ***(k)***, and the number of outputs ***(m)***, (the number of sum terms is equal to the number of outputs).

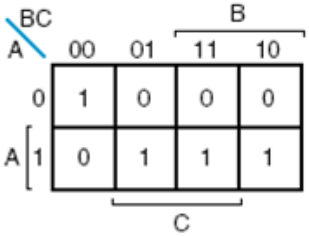
Example:
Implement the combinational circuit having the shown truth table, using PLA.

A	*B*	*C*	F_1	F_2
0	0	0	1	1
0	0	1	1	0
0	1	0	1	0
0	1	1	0	0
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	0	1

Each product term in the expression requires an AND gate. To minimize the cost, it is necessary to simplify the function to a minimum number of product terms.



$F_1 = \overline{A}\,\overline{B} + \overline{A}\,\overline{C} + \overline{B}\,\overline{C}$
$\overline{F}_1 = AB + AC + BC$



$F_2 = AB + AC + \overline{A}\,\overline{B}\,\overline{C}$
$\overline{F}_2 = \overline{A}\,\overline{C} + \overline{A}\,B + A\,\overline{B}\,\overline{C}$

Designing using a PLA, a careful investigation must be taken in order to reduce the distinct product terms. Both the true and complement forms of each function should be simplified to see which one can be expressed with fewer product terms and which one provides product terms that are common to other functions.

The combination that gives a minimum number of product terms is:
$F_1' = AB + AC + BC$ *or* $F_1 = (AB + AC + BC)'$
$F_2 = AB + AC + A'B'C'$

This gives only 4 distinct product terms: *AB, AC, BC,* and *A'B'C'*.

So the PLA table will be as follows:

PLA programming table

	Product term	Inputs A B C	Outputs (C) F_1	Outputs (T) F_2
AB	1	1 1 –	1	1
AC	2	1 – 1	1	1
BC	3	– 1 1	1	–
$\overline{A}\overline{B}\overline{C}$	4	0 0 0	–	1

For each product term, the inputs are marked with 1, 0, or – (dash). If a variable in the product term appears in its normal form (unprimed), the corresponding input variable is marked with a 1.

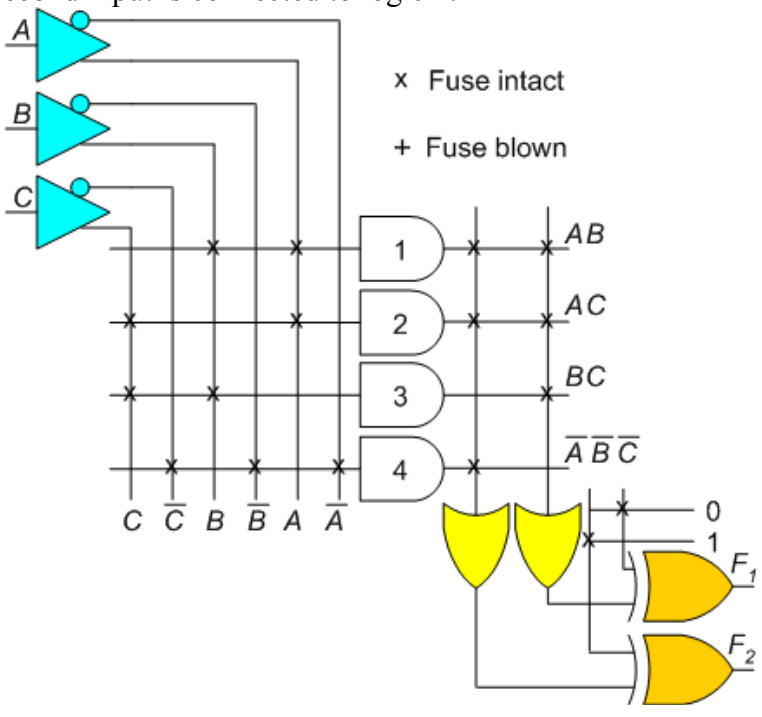
A 1 in the **Inputs** column specifies a path from the corresponding input to the input of the AND gate that forms the product term.

A 0 in the **Inputs** column specifies a path from the corresponding complemented input to the input of the AND gate. A dash specifies no connection.

The appropriate fuses are blown and the ones left intact form the desired paths. It is assumed that the open terminals in the AND gate behave like a 1 input.

In the Outputs column, a **T (true)** specifies that the other input of the corresponding XOR gate can be connected to 0, and a **C (complement)** specifies a connection to 1.

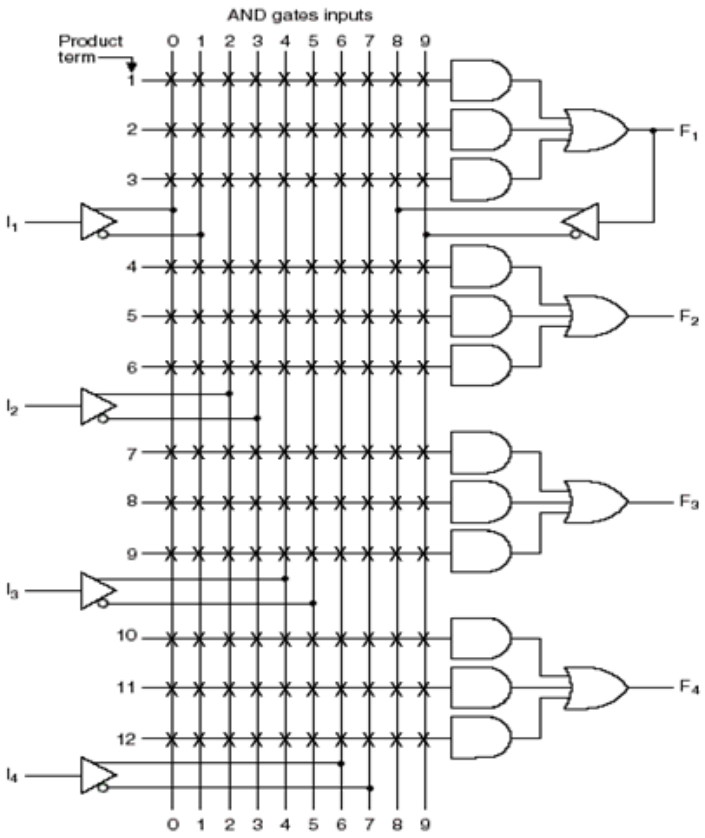
Note that output $\boldsymbol{F_1}$ is the normal (or true) output even though a **C** (for complement) is marked over it. This is because $\boldsymbol{F_1'}$ is generated with AND-OR circuit prior to the output XOR. The output XOR complements the function F_1' to produce the true F_1 output as its second input is connected to logic 1.



The PAL (Programmable Array Logic):

The PAL device is a PLD with a fixed OR array and a programmable AND array.

As only AND gates are programmable, the PAL device is easier to program but it is not as flexible as the PLA.



The device shown in the figure has 4 inputs and 4 outputs. Each input has a buffer-inverter gate, and each output is generated by a fixed OR gate.

The device has 4 sections, each composed of a 3-wide AND-OR array, meaning that there are 3 programmable AND gates in each section.

Each AND gate has 10 programmable input connections indicating by 10 vertical lines intersecting each horizontal line. The horizontal line symbolizes the multiple input configuration of an AND gate.

One of the outputs F_1 is connected to a buffer-inverter gate and is fed back into the inputs of the AND gates through programmed connections.
(see animation in authorware version)

Designing using a PAL device, the Boolean functions must be simplified to fit into each section.

The number of product terms in each section is fixed and if the number of terms in the function is too large, it may be necessary to use two or more sections to implement one Boolean function.

Example:
Implement the following Boolean functions using the PAL device as shown above:

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

Simplifying the 4 functions to a minimum number of terms results in the following Boolean functions:

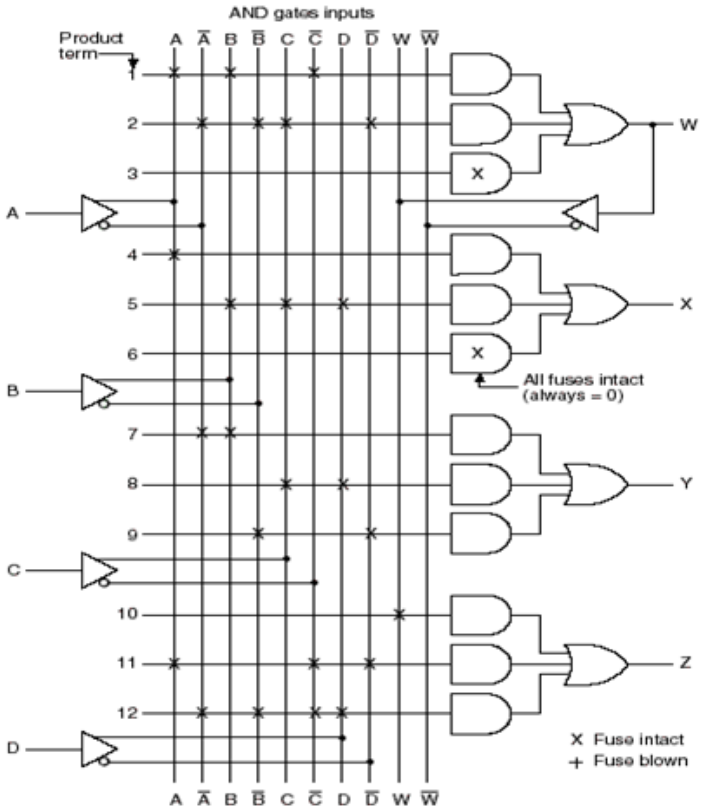
$W = ABC' + A'B'CD'$
$X = A + BCD$
$Y = A'B + CD + B'D'$
$Z = ABC' + A'B'CD + AC'D' + A'B'C'D$
$= W + AC'D' + A'B'C'D$

Note that the function for ***Z*** has four product terms. The logical sum of two of these terms is equal to ***W***. Thus, by using ***W***, it is possible to reduce the number of terms for ***Z*** from four to three, so that the function can fit into the given PAL device.

The PAL programming table is similar to the table used for the PLA, except that only the inputs of the AND gates need to be programmed.

Product term	AND Inputs A	B	C	D	W	Outputs
1	1	1	0	—	—	$W = AB\overline{C}$
2	0	0	1	0	—	$+\overline{A}\,\overline{B}CD$
3	—	—	—	—	—	
4	1	—	—	—	—	$X = A$
5	—	1	1	1	—	$+BCD$
6	—	—	—	—	—	
7	0	1	—	—	—	$Y = \overline{A}B$
8	—	—	1	1	—	$+CD$
9	—	0	—	0	—	$+\overline{B}\,\overline{D}$
10	—	—	—	—	1	$Z = W$
11	1	—	0	0	—	$+A\overline{C}\,\overline{D}$
12	0	0	0	1	—	$+\overline{A}\,\overline{B}\,\overline{C}D$

The figure shows the connection map for the PAL device, as specified in the programming table.
(see animation in authorware version)

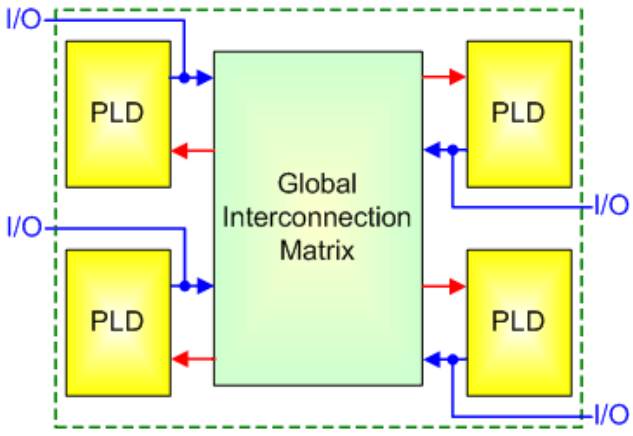


Since both W and X have two product terms, third AND gate is not used. If all the inputs to this AND gate left intact, then its output will always be 0, because it receives both the true and complement of each input variable i.e., AA' =0

Complex Programmable Logic Devices (CPLDs):

A CPLD contains a bunch of PLD blocks whose inputs and outputs are connected together by a global interconnection matrix.

Thus a CPLD has two levels of programmability: each PLD block can be programmed, and then the interconnections between the PLDs can be programmed.



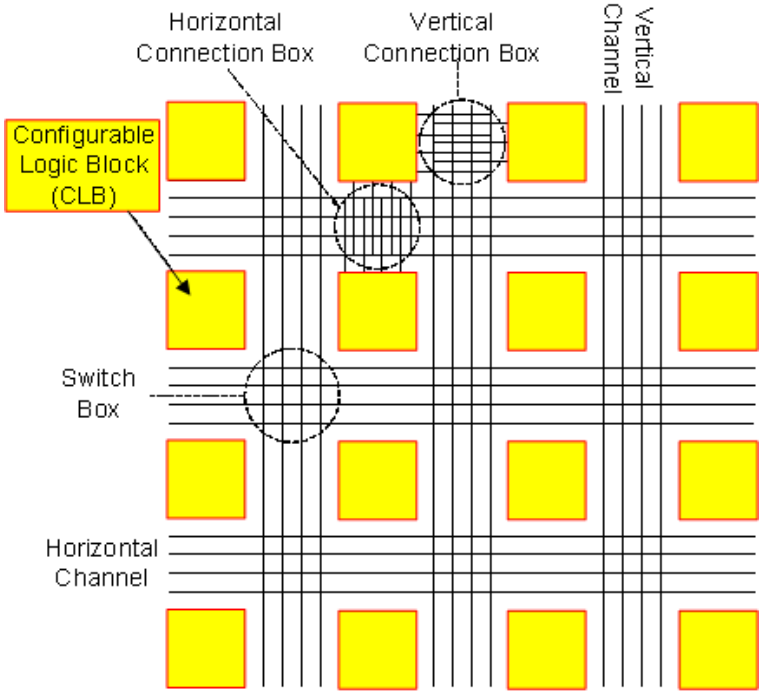
Field Programmable Gate Arrays (FPGAs):

The FPGA consists of 3 main structures:

1. Programmable logic structure,
2. Programmable routing structure, and
3. Programmable Input/Output (I/O).

1. Programmable logic structure

The programmable logic structure FPGA consists of a 2-dimensional array of configurable logic blocks (CLBs).



Each CLB can be configured (programmed) to implement *any* Boolean function of its input variables. Typically CLBs have between 4-6 input variables. Functions of larger number of variables are implemented using more than one CLB.

In addition, each CLB typically contains 1 or 2 FFs to allow implementation of sequential logic.

Large designs are partitioned and mapped to a number of CLBs with each CLB configured (programmed) to perform a particular function.

These CLBs are then connected together to fully implement the target design. Connecting the CLBs is done using the FPGA programmable routing structure.

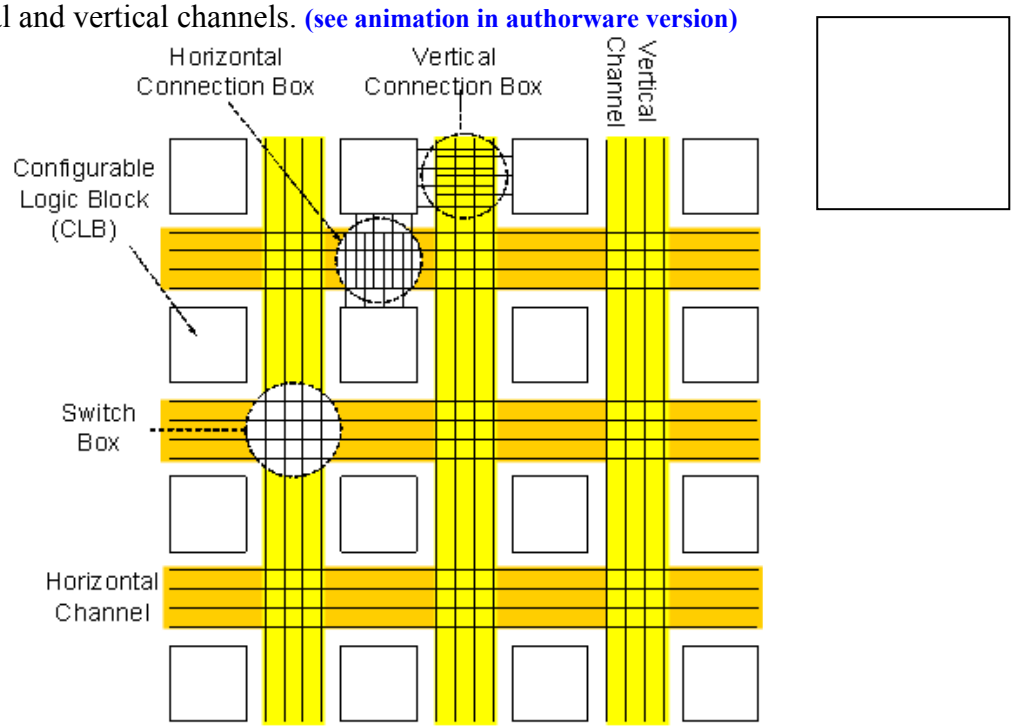
2. Programmable routing structure

To allow for flexible interconnection of CLBs, FPGAs have 3 *programmable* routing resources:

1. Vertical and horizontal routing channels which consist of different length wires that can be connected together if needed. These channel run vertically and horizontally between columns and rows of CLBs as shown in the Figure.

2. Connection boxes, which are a set of programmable links that can connect input and output pins of the CLBs to wires of the vertical or the horizontal routing channels.

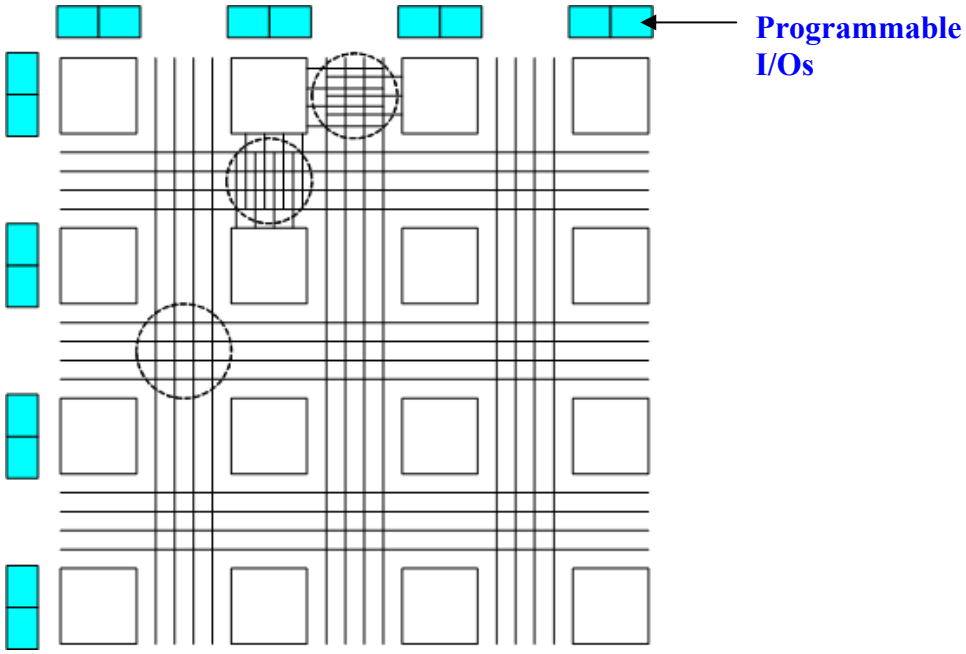
3. Switch boxes, located at the intersection of the vertical and horizontal channels. These are a set of programmable links that can connect wire segments in the horizontal and vertical channels. **(see animation in authorware version)**



3. Programmable I/O

These are mainly buffers that can be configured either as input buffers, output buffers or input/output buffers.

They allow the pins of the FPGA chip to function either as input pins, output pins or input/output pins.



eCOE200 Lessons Errata

Error Location	Error Description	Correction
Example 3-a on **P[1]19** of **U1-L5**,	8^4 -6770 = 1000000000 -6770	8^4 -6770=10000 -6770
P15 of **U1-L6**, 2's complement Representation of **-37**	11011010	11011011
Example 4- P18 of **U2-L1**, after the downward red lines	F= (A`.B+ 0) . (0 + A' B C`). (A`. B + A'.B.C`)	F= (A`.B+ 0) . (0 + A' B C`). (A`.B.B` +A`.B.C`) = (A`.B+ 0) . (0 + A' B C`). (A`.0 +A`.B.C`) = (A'B). (A'BC') . (0 + A'BC') = A'BC'
Example 2- P5 of **U2-L5**	K-Map representation of the function has 1 in in M1 , But it is located.	The 1 must be relocated in M2 and the grouping of 0's done accordingly
Quiz on P6 of **U2-L7, Question 2 of 3**	The given answer is **False**	Correct answer is **TRUE**

PS: These are the known errors so far. If you come across any other error please send for a more comprehensive list

[1] P → Page, U → Unit, L→ Lesson. Thus P19 of U1 L5 means Page 19 in lesson 5 of unit 1.