Digital Logic Design

Version 4.0 printed on September 2012
First published on August 2006

Background and Acknowledgements

This material has been developed for the first course in Digital Logic Design. The content is derived from the author's educational, technical and management experiences, in-addition to teaching experience. Many other sources, including the following specific sources, have also informed by the content and format of the following material:

- Katz, R. Contemporary Logic Design. (2005) Pearson.
- Wakerly, I. Digital Design. (2006) Prentice Hall.
- Sandige, R. Digital Design Essentials. (2002) Prentice Hall.
- Nilsson, J. Electrical Circuits. (2004) Pearson.

I would like to give special thanks to my students and colleagues for their valued contributions in making this material a more effective learning tool.

I invite the reader to forward any corrections, additional topics, examples and problems to me for future

Thanks,
Izad Khormaee
www.EngrCS.com

Contents

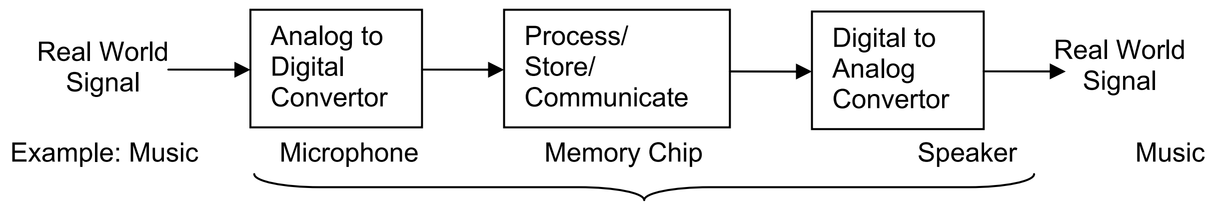
Chapter 1. Number Systems, Number Representations, and Codes

1.1. Key concepts and Overview

- Digital vs. Analog
- Digital Design Overview (from Transistor to Super Computer)
- Design Methodologies
- Number systems (Binary, Octal, Decimal, Hexadecimal)
- Base Conversions
- Signed Binary Number Conventions
- Binary Arithmetic
- Binary Code
- DC Electrical Circuit Fundamentals
- Additional Resources
- Problems

1.2. Digital vs. Analog

Natural forces and signals are all analog (or continuous) which means we hear, see and change items in a continuous manner. On the other hand, our digital technology (also called non-continuous or 2-value discrete) more effectively allows us to process and communicate more effectively. This leads us to design systems that fit the following block diagram architecture:



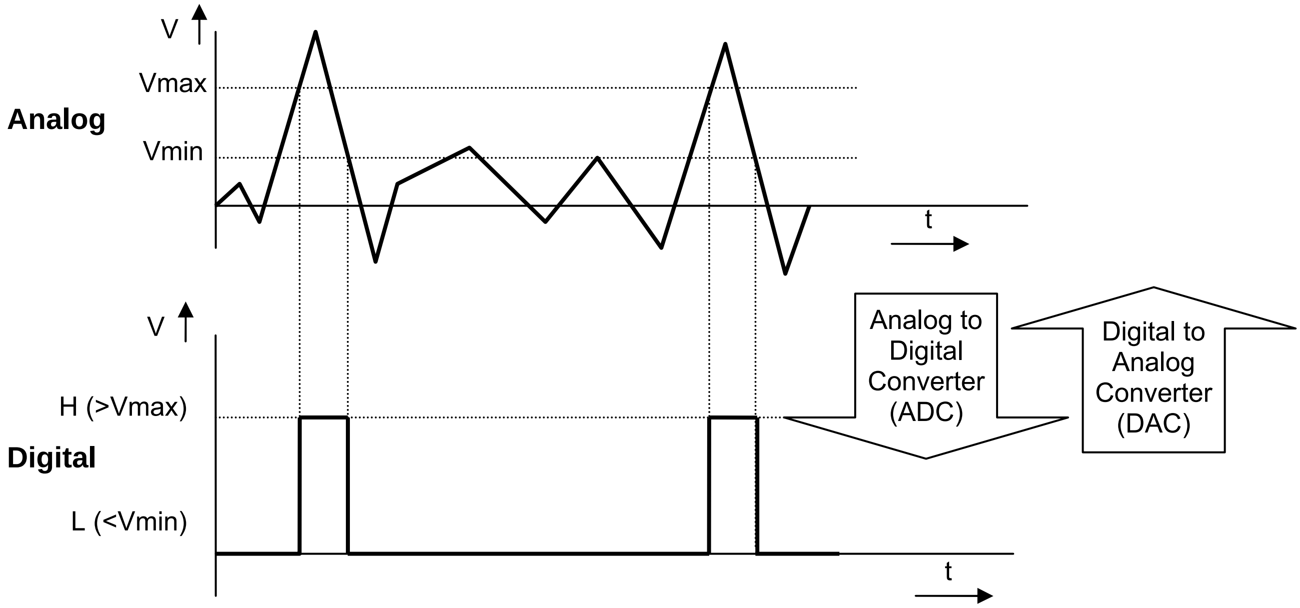
- Why convert analog data to digital data?
  - We have the information we need (on-off, timing)
    - Above a certain level is on, high, 1-state or true.
    - Below a certain level is off, low, 0-state or False.

      Note: We have introduced a discontinuity when a signal goes from 1 to 0 or 0 to 1. This means we cannot say what the exact value is at the time of transition.

  - Reduces complexity of signals and the solutions to work with the signal.
    - To deal with a digital signal we need to deal with binary algebra.
    - To deal with an analog signal we need to deal with calculus to approximate.

  - Positive vs. Negative logic
    - *Positive Logic Convention* (Default → easier for humans to understand)
      - H, (V > Vmax) is 1-state or True
      - L, (V < Vmin) is 0-state or False
    - *Negative Logic Convention* (1 is L and 0 is H)
      - H, (V > Vmax) is 0-state or False
      - L, (V < Vmin) is 1-state or True

❖ Example of analog and digital representations of human Heart Beat:



❖ Based on the definition of a digital (2-valued) system, what are some examples where a digital system could apply? What are the variables and on/off or high/low states?

❖ Example: Describe the input and output of a traffic intersection in digital form.
Solution:

Car's presence at an intersection:
(Car Present→Magnitude is 1, No Car Present→Magnitude is 0)

Status of Traffic Lights:
(Red-on → 1, Red-off → 0)

Extension: draw a typical Intersection and label the output and input in digital form.

1.3. Digital Design Overview (from Transistor to Super Computer)

- All digital systems from the smallest to largest run on a 2-valued system (also called Binary system). So a mechanism is needed to represent the two values. This is typically accomplished with a switch that can be on or off. In the early days, mechanical switches were used, followed by vacuum tubes as switches. Today we use transistors that can be configured to approximate the switch on and off modes. Transistors are fast, inexpensive and small.

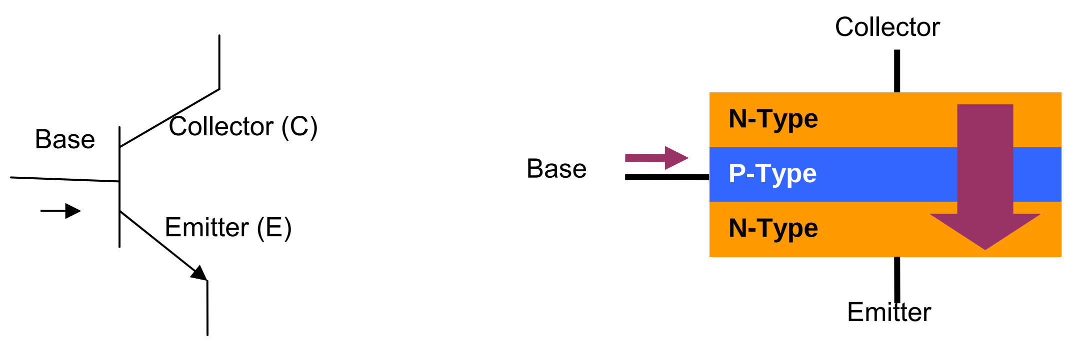
- Transistor overview (the invention that makes today’s automation possible)
  - The transistor, invented by three scientists at the Bell Laboratories in 1947, rapidly replaced the vacuum tube as an electronic signal regulator.

  - Transistors are the basic elements in integrated circuits (ICs). An IC consists of a very large number of transistors interconnected with circuitry and packaged into a single silicon microchip or "chip." A typical processor chip has many millions of transistors.

  - A transistor is developed based on semiconductor material characteristic. Semiconductor material used basically as a switch as shown below:

<u>NPN Transistor Example</u>

”A small current at the base causes the CE connection to change from open to a short”



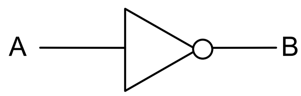
  - Semiconductor material is given special properties by a chemical process called doping. The doping results in a material that either adds extra electrons to the material (which is then called N-type for the extra negative charge carriers) or creates "holes" in the material's crystal structure (which is then called P-type because it results in more positive charge carriers).

    Today's computers use circuitry made with complementary metal oxide semiconductor (CMOS) technology. CMOS uses two complementary transistors per gate (one with N-type material; the other with P-type material). When one transistor is maintaining a logic state, it requires almost no power when not switching.

- Semiconductor Integration scaling

  - Small-Scale Integration, SSI (Basic gates: OR, NOR, NOT, AND)

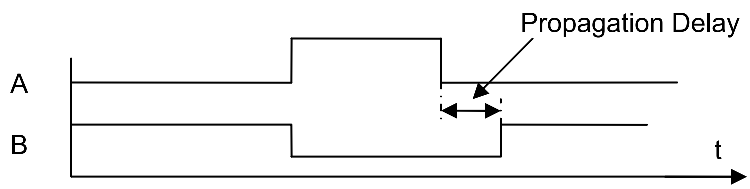
    - Example: Inverter (NOT) is a common SSI element used in Digital Design (Vendors provide usage information and specifications in the form of a data sheet)



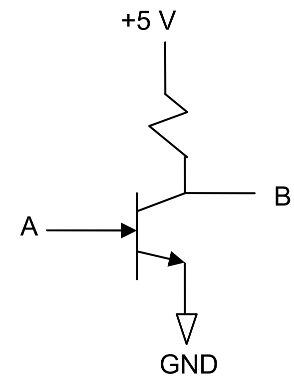
1. Block Diagram

Input A	Output B
0	1
1	0

2. Truth Table



3. Timing Diagram



4. Equivalent Circuit

- ➢ Medium Scale Integration, MSI
  - ▪ PAL--Programmable Array Logic, GAL--Generic Array Logic, EPROM--Erasable Programmable Read Only Memory, ADDER, COUNTER)
    - 1,000s to 100,000s of gates.
    - Typically, the vendor provides information in the form of a data sheet

- ➢ Large Scale Integration (LSI)
    - 100,000s to Millions of gates
    - Typically implements complex functionality
    - Processors such as special function controllers and interface chips

- ➢ Very Large Scale Integration (VLSI)
    - Millions to Billions of gates
    - Typically includes extensive functionality
    - Processors such as Intel's Pentium are examples of VLSI.

❖ Design / Analysis tools
  - We will be using manual processes for most of this text to design/analyze digital circuits in order to gain in-depth understanding of logic design.
  - The final section of this text is dedicated to the use of Hardware Description Language (HDL) to automate design, simulation, implementation, and analysis and verification process.

1.4. Design Methodologies

Digital design depends on the type of problem, the work already completed, the strategic direction of the organization and the skills/resources available to the project team. Having said that, in general, there are three approaches available to the designers under traditional Hierarchical-Oriented Designs:

- Top-down Design Methodology
  Start with larger block of design and then work out the detail of each block.

- Bottom-up Design Methodology
  Start with components and figure out how to interconnect them to design the system.

- Middle-out Design Methodology
  A combination of the bottom-up and top-down. Most designs are done this way: start with the top-down design, then modify the design to take advantage of the available components (based on cost, availability, and reliability).

Another way of thinking about the problem of design that has a strong following in the software development community and is being used in the hardware community under the module design concept is Object-Oriented Design (OOD).

Designers commonly agree that there are four main properties or benefits associated with object-oriented design:

- Encapsulation
  As the name implies, the internals of the design are hidden from the user and only the interface definition (input/output) are available to the user. Users benefit since they have a limited amount of information to learn. Designers benefit since they are able to upgrade the module without involving the user as long as the new interface is a superset of an existing interface.

- Inheritance
  This simply means that an object may be built on the features available in the base object property. Of course, the benefit is that the designers only have to work on the additional feature and simply reuse the existing functionality.

- Polymorphism
  OOD allows the designer to create objects that behave differently based on the attributes of input.

- Composition (One object can be built using many others.)
  A new object may be developed based on the composition of multiple existing objects.

Hopefully, at this point you are thinking “why wouldn’t everyone use OOD?” The main drawback of OOD is the high level of planning required for each module, and discipline needed to follow the four properties in design.

1.5. Number Systems (Decimal, Binary, Octal, Hexadecimal)

We have learned and use the decimal numbering system simply because humans are born with ten fingers! The decimal system has served us well. But with digital systems, we need a 2-value system (binary). We could attribute this to the fact that computers only have open or closed switches (or one finger, if you prefer).

This means, we have to learn the binary system in addition to the decimal system. We also will discuss the octal and hexadecimal systems because conversion to/from binary is easy and numbers in these systems are easier to read than binary numbers for humans.

- Decimal Number (base or radix 10)
  - Humans use the decimal numbering system as a default, so when you see a number 56 your assumption is that its base or radix is 10 or $(56)_{10}$ which is “56 base 10”.
  - Each digit is weighted based on its position in the sequence (power of 10) from the Least Significant Digit (LSD, power of 0) to the Most Significant Digit (MSD, highest power).
  - Each digit must be less than 10 (0 to 9)

    For example $(2375.46)_{10}$ is evaluated as:

	MSD						LSD
Digit notation	d_3	d_2	d_1	d_0	.	d_{-1}	d_{-2}
Digit	**2**	**3**	**7**	**5**	.	**4**	**6**
Value	10^3	10^2	10^1	10^0		10^{-1}	10^{-2}
Results=Value*Digit	**2000**	**300**	**70**	**5**		**0.4**	**0.06**

$$(2375.46)_{10} = 2x10^3 + 3x10^2 + 7x10^1 + 5x10^0 + 4x10^{-1} + 6\ x10^{-2}$$
$$= 2000 + 300 + 70 + 5 + 0.4 + 0.06$$

Note: The general term for decimal point is “radix point”.

- Binary Number (base or radix 2)
  - Digital and computer technology is based on the binary number system, since the foundation is based on a transistor, which only has two states: on or off.
  - Each digit of the number is called a bit or which is a short for **bi**nary digi**t**s
    - An 8-bit group is referred to as a Byte
    - An 4-bit group is referred to as a nibble
  - Each bit is weighted based on its position in the sequence (powers of 2) from the Least Significant Bit (LSB) to the Most Significant Bit (MSB).
  - Each bit must be less than 2 which means it has to be either 0 or 1.

    For example $(1010.11)_2$ is evaluated as:

	MSB			LSB		
Digit notation	b_3	b_2	b_1	b_0	b_{-1}	b_{-2}
Digit	**1**	**0**	**1**	**0**	**1**	**1**
Value	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}
Results=Value*Digit	**8**	**0**	**2**	**0**	**0.5**	**0.25**

$(1010.11)_2 = 8 + 0 + 2 + 0 + 0.5 + 0.25 = (10.75)_{10}$
Note: The general term for decimal point is radix point

- In binary, the count starts at 0 (called 0-referencing), where in decimal, the count typically starts with 1 (called 1-referencing)

❖ Octal (base 8) and Hexadecimal (base 16)
These number systems are used by humans as a representation of long strings of bits since they are:

- Easier to read and write, for example $(347)_8$ is easier to read and write than $(011100111)_2$.

- Easy to convert (Groups of 3 or 4)

- Today, the most common way is to use Hex to write the binary equivalent; two hexadecimal digits make a Byte (groups of 8-bit), which are basic blocks of data in Computers.

❖ Question: The hexadecimal system is base 16, so the digits range in value from 0 to 15. How do you represent Hexadecimal digits above 9?

Use A for 10, B for 11, C for 12, D for 13, E for 14 and F for 15. So $(CAB)_{16}$ or $(CAB)_{HEX}$ is a valid hexadecimal number.

❖ Computer memory is typically organized in 8-bit groups or bytes. Why groups of 8?

1.6. Base Conversions

- Decimal to Binary Conversion – “Subtract the weight method”
  - Steps:
    1. (1) Find the largest power of 2 (2^n) that can be subtracted out of the decimal number
    2. (2) Take the result and subtract (2^{n-1}) from it
       - If the result is not negative then that bit is one
       - If the result is negative, then that bit is zero and the result equals the result from step 1
    3. (3) Repeat step 2 until the result is exactly 0

  - Example: convert $(49)_{10}$ to a binary number

	49	17	1	1	1	1	
2^n →	-32	-16	-8	-4	-2	-1	
	----	----	----	----	----	----	
Results →	17 >0	1	-7	-3	-1	0	When =0, done
Binary # →	(1	1	0	0	0	$1)_2$	

- Binary to Decimal Conversion – “Add the weight method”
  - Step:
    1. (1) Simply multiply each bit with its weight and add to get the decimal number

  - Example: Convert $(110001)_2$ to a decimal number
    $(110001)_2 = (1 * 2^5 + 1 * 2^4 + 0 * 2^3 + 0 * 2^2 + 0 * 2^1 + 1 * 2^0)_{10} = (49)_{10}$

- Binary ↔ Octal Conversion - “Group of 3 method”
  - Step:
    1. (1) Each three bits in binary (right to left) equals one octal digit in the same direction)

  - Example - Convert $(10110111)_2$ to an Octal number.

“0” is added to make a group of 3

$(\underbrace{010}\quad \underbrace{110}\quad \underbrace{111})_2$

$(\quad 2 \quad 6 \quad 7 \quad)_8$

  - Reverse the process to convert from Octal to Binary

- Binary ↔ Hexadecimal Conversion - “Group of 4 method”
  - Steps:
    1. (1) Each four bits in binary (right to left) equals one hex digit in the same direction)

  - Example:- Convert $(110110111)_2$ to a hexadecimal number

$(\underbrace{0001}\quad \underbrace{1011}\quad \underbrace{0111})_2$

$(\quad 1 \quad B \quad 7 \quad)_{16}$

  - Reverse the process to convert from hexadecimal to binary

- ❖ Any base to Decimal Conversion - "Polynomial Function Method"
  - ➢ The most general number in any base is the real number and the general rule is as follows:

    $(\text{Real Number})_r = (d_j \ldots d_1 d_0.\ d_{-1} d_{-2} \ldots)_r = (d_j r^j + \ldots + d_1 r^1 + d_0 r^0 + d_{-1} r^{-1} + d_{-2} r^{-2} + \ldots)_{10}$

    - Example – The most common conversion is Hex integer to decimal base. For this example, convert $(1CAB)_{16}$ to decimal:

      $(1CAB)_{16} = (1{*}16^3 + 12{*}16^2 + 10{*}16^1 + 11{*}16^0) = (7339)_{10}$

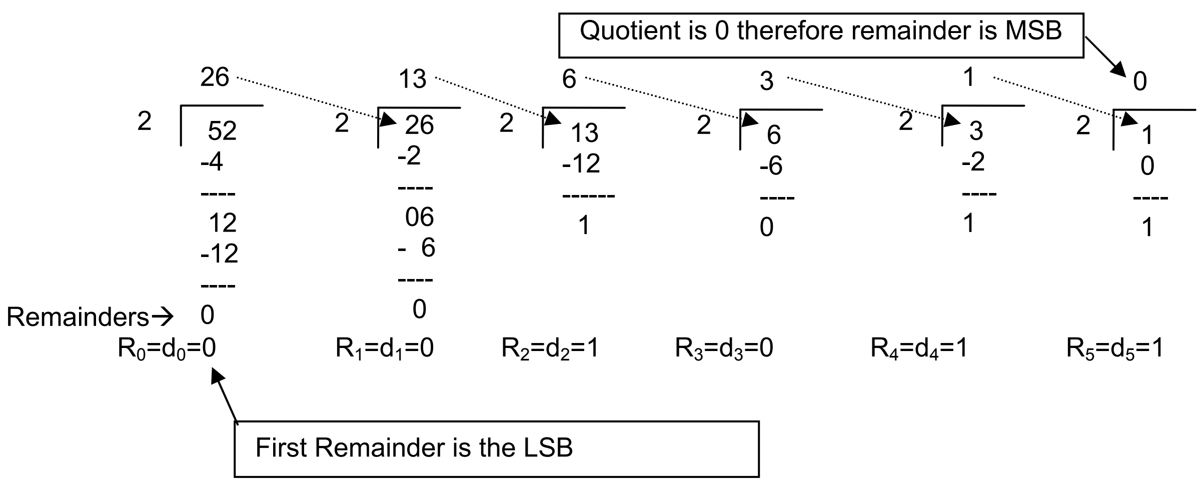
    - Example - Although not common, let's do an example of converting a real binary number to decimal so Convert $(11010)_2$ to decimal.

      $(11010.11)_2 = (\ 1{*}2^4 + \ 1{*}2^3 + 0{*}2^2 + 1{*}2^1 + 0{*}2^0 + 1{*}2^{-1} + 1{*}2^{-2}) = (26.75)_{10}$

- ❖ Integer Decimal Conversion to any Base – "Repeated Radix Division Method"
  - ➢ The solution is based on the fact that
    $(\text{integer number})_{10} = d_n r^n + \ldots + d_2 r^2 + \ d_1 r^1 + \ d_0 r^0 \ = (d_n \ldots d_2 d_1 d_0)_r$

  - ➢ Steps:
    (1) If $(\text{integer number})_{10}$ is divided by r, the remainder is d_0 (Least Significant Digit, LSD)
    (2) If the quotient from step 1 is divided by r the remainder is the next digit
    (3) Repeat step 2 until the quotient is zero were the remainder is the d_n (Most Significant digit, MSD)

    - Example: Convert $(52)_{10}$ to binary (radix, r = 2)



Therefore $(52)_{10} = (110100)_2$

- ❖ Decimal Fraction Conversion to any Base – "Repeated Radix Multiplication Method"
  - ➢ Solution is based on the fact
    $(\text{decimal fraction})_{10} = d_{-1}\, r^{--1} + d_{-2}\, r^{--2} + \ldots \ = (.d_n \ldots d_2 d_1 d_0)_r$
    $r{*}(\text{decimal fraction})_{10} = d_{-1} \ + d_{-2} r^{-1} + \ldots \ = (.d_{-1} d_{-2} d_{-3} \ldots\ )_r$

  - ➢ Steps:
    (1) Multiply $(\text{fraction})_{10}$ by r, the non-fractional part is the first digit

(2) Continue step 2 until fraction is 0

- Example: Convert $(.125)_{10}$ to binary (r=2)

Solution:

.125	.25	.5
x 2	x 2	x 2
-----	-----	-----
0.25	0.5	1.0
↑	↑	↑
d_{-1}	d_{-2}	d_{-3}

Fraction is 0 which means d_{-3} is the Least Significant Digit
Non-fraction portion is 1 which means d_{-3} is 1.

Non-fraction portion is 0 which means d_{-1} ,the Most Significant Digit, is 0.

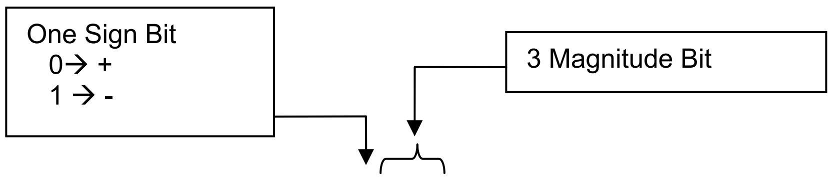
Therefore $(.125)_{10} = (.001)_2$

Note: Some numbers may not be fully convertible, so you have to decide the number of decimal points you need to convert. For example $(1/12)_{10}$ does not fully convert to binary number.

1.7. Signed Binary Number Conventions

- Signed Binary Number Representations (3 methods)

  - Signed Magnitude (SM)
    - Easiest for people to read (Not used by computers)
    - Here is an example of Signed Magnitude number with 4-bit word size



$(+5)_{10} = (0\ 1\ 0\ 1)_{2SM}$
$(-5)_{10} = (1\ 1\ 0\ 1)_{2SM}$

  - Binary SM numbers for n-bit word ranges from $+(2^{n-1} – 1)$ to $-(2^{n-1} – 1)$
    Note: there are two values for zero (Sign-bit = 1 and Sign-bit=0)

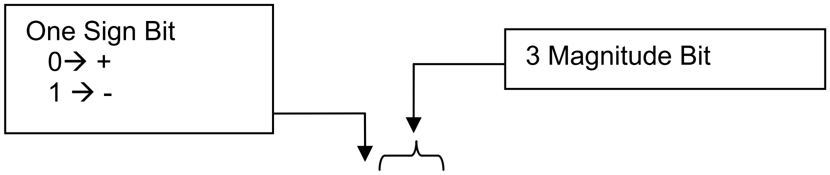
  - Example of complete list of binary SM numbers for a 4-bit word.

Binary SM Number (n=4)				**Decimal Number**
d3	**d2**	**d1**	**d0**	
0	1	1	1	$+ 7 = +(2^{4-1} - 1)$
0	1	1	0	+ 6
0	1	0	1	+ 5
0	1	0	0	+ 4
0	0	1	1	+ 3
0	0	1	0	+ 2
0	0	0	1	+ 1
0	0	0	0	+ 0
1	0	0	0	- 0
1	0	0	1	- 1
1	0	1	0	- 2
1	0	1	1	- 3
1	1	0	0	- 4
1	1	0	1	- 5
1	1	1	0	- 6
1	1	1	1	$- 7 = -(2^{4-1} - 1)$

- Diminished Radix Complement (DRC) or 1's complement
  - Some computer systems use this information because it is easier to convert.

  - To obtain a negative DRC or 1's complement:
    - Write a positive number with MSB set to 0 (positive sign)
    - Negate (Invert) every bit including sign bit to obtain the negative number.

➢ Here is an example of 4-bit word size:



$(+5)_{10} = (0\ 1\ 0\ 1\ )_{2DRC}$
$(-5)_{10} = (1\ 0\ 1\ 0)_{2DRC}$

➢ DRC numbers for n-bit word ranges from $+(2^{n-1} – 1)$ to $–(2^{n-1} – 1)$
- Note that there are two values for zero (Sign-bit = 1 and Sign-bit=0)

➢ Example of Binary DRC or 1's Complement Numbers for a 4-bit word

Binary SM Number (n=4)				**Decimal Number**
d3	**D2**	**d1**	**d0**	
0	1	1	1	$+ 7 = +(2^{4-1} -1)$
0	1	1	0	+ 6
0	1	0	1	+ 5
0	1	0	0	+ 4
0	0	1	1	+ 3
0	0	1	0	+ 2
0	0	0	1	+ 1
0	0	0	0	+ 0
1	1	1	1	- 0
1	1	1	0	- 1
1	1	0	1	- 2
1	1	0	0	- 3
1	0	1	1	- 4
1	0	1	0	- 5
1	0	0	1	- 6
1	0	0	0	$- 7 = -(2^{4-1} -1)$

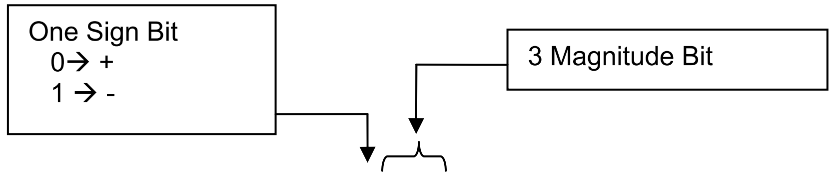
❖ Radix Complement (RC) or 2's complement

➢ Majority of Digital Systems use RC since it simplifies the binary arithmetic operation.

➢ To obtain a negative RC or 2's complement:
- Write a positive number with the MSB set to 0 (positive sign)
- Negate (Invert) every bit including sign bit
- Add a 1 to the result to obtain the negative number

Note: Taking the 2's complement of the result will return the original positive number.

➢ Below is an example of 4-bit number of SM to RC:



$(+5)_{10} = (0\ 1\ 0\ 1\ )_{2RC}$

$(\ -5)_{10} = -(0\ 1\ 0\ 1)_{2RC}$

Bit-invert →		1 0 1 0
Add 1	→	+ 1

2's Compl. →		$(1\ 0\ 1\ 1)_{2RC}$

- RC numbers for n-bit word range from $+(2^{n-1} – 1)$ to $–(2^{n-1})$ with the following two characteristics:
  - The range is not symmetrical, there is one more negative number than there are positive numbers.
  - There is only one pattern for zero (-0 and +0 have the same pattern)

- Example of Binary RC or 2's Complement Numbers for a 4-bit word

Binary SM Number (n=4)				Decimal Number
d3	**d2**	**d1**	**d0**	
0	1	1	1	$+ 7 = +(2^{4-1} -1)$
0	1	1	0	+ 6
0	1	0	1	+ 5
0	1	0	0	+ 4
0	0	1	1	+ 3
0	0	1	0	+ 2
0	0	0	1	+ 1
0	0	0	0	+ 0
0	0	0	0	- 0
1	1	1	1	- 1
1	1	1	0	- 2
1	1	0	1	- 3
1	1	0	0	- 4
1	0	1	1	- 5
1	0	1	0	- 6
1	0	0	1	- 7
1	0	0	0	$-8 == -2^{4-1}$

- Quick Inspection Method → Finding 2's complement
  - Working from the LSB of the number to be complemented toward the MSB (right to left), rewrite each bit up to and including the first "1" encountered, then complement each bit thereafter
  - Example:

	MSB LSB
Old Number:	(1 0 1 1 0 1 0)
2's Complement:	(0 1 0 0 1 1 0)

**Note: 2's complement gets you back to the original number.

1.8. Binary Arithmetic

All of today's computer systems use RC numbers (2's complement) for binary arithmetic operations. The reset of this section provides description of Binary Arithmetic using RC numbers.

- Addition of Signed Binary Numbers
  When adding RC numbers, simply add then ignore the left-most carry.

```
+7    →        0 1 1 1
+(-2) →        1 1 1 0
               ---------
               0 1 0 1    "Ignore the left-most carry, and the result is +5"
```

  Notes:
  - The left-most bit is a sign bit and there are three magnitude bits.
  - As long as we know results fits within the 1 sign-bit and n magnitude bits, this process works. Otherwise we need to consider the overflow.

- Addition of Unsigned Binary Numbers
  Unsigned addition Signed works exactly the same way as singed addition, allowing us to use the same circuitry.

```
+7    →        0 1 1 1
+3 →           0 0 1 1
               ---------
               1 0 1 0    "Result is +10. If there is a carry beyond the available bits, then an
                           an overflow has occurred.
```

- Overflow
  - An overflow occurs when the addition of two numbers results in a number larger than can be expressed with the available number of bits.

    - Example – performing the operation, 8+9=17; in a 4-bit word system, results in an overflow since 4 bits can only store 0 to 15. The result will show as a 1, which is 16 less than the correct result.

  - Detecting overflow
    - Unsigned number addition
      If the addition has a carry beyond the available bits then an overflow has occurred.

    - Signed (RC, 2's complement) number addition
      - If the operands have different signs, then overflow cannot occur, since one number is being subtracted from the other.
      - If the operands have the same sign and the result has a different sign, then an overflow has occurred.
        A quick way to identify an overflow situation is when the carry into the sign-bit position and the carry out of sign-bit position are different. Example

```
-5      →  1 0 1 1
+(-6)   →  1 0 1 0
           ----------
           0 1 0 1   Result is +5 which is wrong
                       since -5 +(-6) = -11
          1  0 1 0     →  Carry bits
```

The sign is 0 (+) where the two operands had sign 1 (-), indicating an overflow.

The carries in and out of sign-bit are different, indicating an overflow.

- Subtraction (indirect method)
  If you write subtraction as addition with a negative number, then the previous method can be used.

  - For example:
    {2 – 6} can be done by performing {2 + (-6)}

1.9. Binary Codes

Binary codes are used to translate human symbols to one and zeros. The most important of the symbols is the alphabet used for human communications. So every key and character has to have a unique binary code. The minimum number of bits required to uniquely identify all the keys on the keyboard must meet the following condition:

$$2^{\text{Number of Bits}} \geq \text{Number of keys}$$

❖ ASCII Code

Initially, IBM's scheme of representing alphanumeric and control characters for computers was the most commonly used coding method. The coding scheme was referred to as the Extended Binary-Coded Decimal Interchange Code (EBCDIC). Its dominance was driven by IBM's near-monopoly position in the computer industry until the early 1980's.

The majority of other manufacturers were looking for a non-proprietary coding, leading to the American Standard Code for Information Interchange (ASCII) coding. ASCII was adopted by the majority of vendors and very quickly overtook EBCDIC as the most commonly used coding scheme.

ASCII code is used to represent alphanumeric and control characters with 8 bits. The ASCII code table is shown below:

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.asciitable.com

In early 1990, the need for a code that was capable of representing Asian languages with large number of characters became an important competitive question. Up to that point, the language of computer interfacing was English and to lesser extend other western languages that have less than 256 characters. The ASCII code could represent them using its 8-bit word with 256 unique codes. But this is not true for a number of Asian languages.

In order to meet the need for the larger Asian languages character set and maintain compatibility with ASCII code, Unicode was introduced. Unicode use 16 bits, so it is capable of representing as many as 2^{16} or 65,536 unique symbols. The majority of today's computers use Unicode which is also referred to as the double byte code.

- ❖ Other Binary Codes

  - ➢ Binary coded decimal (BCD)
    BCD assigns 4 binary bits to each binary digit. The only draw back is that only 0 to 9 are used, and the other 6 combinations from 10 to 15 are not used.

<u>Binary Coded Decimal</u>
0000 → 0
0001 → 1
0010 → 2
0011 → 3
0100 → 4
0101 → 5
0110 → 6
0111 → 7
1000 → 8
1001 → 9

  - ➢ Reflective Gray Code (RGC)
    RGC is a binary number system organized so that consecutive codes in the sequence only require one bit change as shown below:

<u>2-bit Reflective Gray Code</u>
00
01
11
10

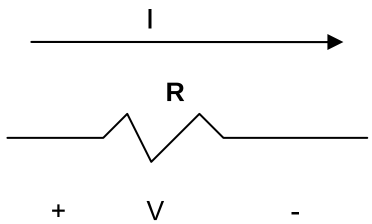
3-bit Reflective Gray Code
000
001
011
010
110
111
101
100

1.10. DC Electrical Circuit Fundamentals

The basic components used here are resistors and power supplies. The power supply provides energy to the circuit and the simplest power supply is a battery. Resistors are material that limit the amount of current follow.

This section discusses the generation and sensing of logic “1” which is typically equal to 3.5 and 5 volts. On the other hand, logic “0” is typically less than 0.7 volts.

- Electrical Resistance
  - Resistor Symbol



  - Resistance is the capacity of a material to impede the flow of current (electric charge). The most common use of resistors is to limit current flow.

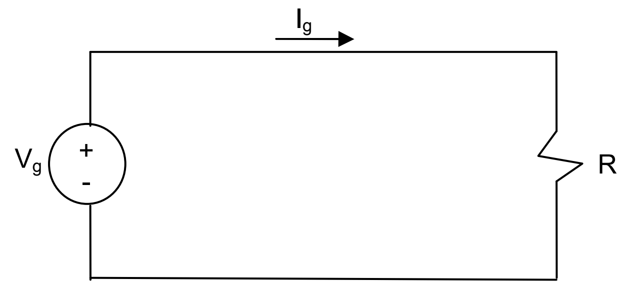
  - The flow of current through a resistor will convert electrical energy to thermal energy.
    In some applications, this property is desirable and in other application it is undesirable. Here are examples of each:

    - Undesirable: transmission line, digital devices
    - Desirable: heater, toaster. oven, stove top

  - Resistance, R, is a basic ideal element so it is defined in term of current, I, and Voltage .
    Ohms law: $V = I * R$ where:
    - R value is in Ohms or Ω
    - V is in volts
    - I is in Amperes

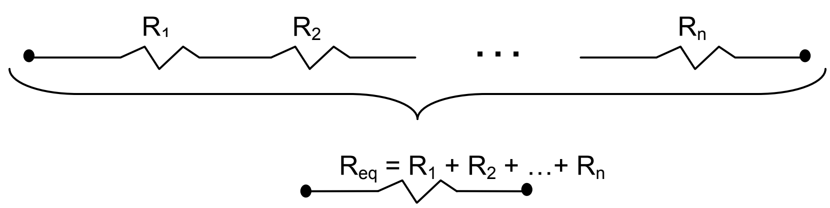
- Power
  Power is measured in Watts and can be calculated using the following equations.
  $P= V^2/R = I^2*R$ where
  V is the voltage and is in volts
  I is the current in Amps
  R is the resistance in Ohms

- Example
  Find the value of the resistance ,R, and the power consumed by the resistor if Vg = 1 kV and Ig = 5 mA.

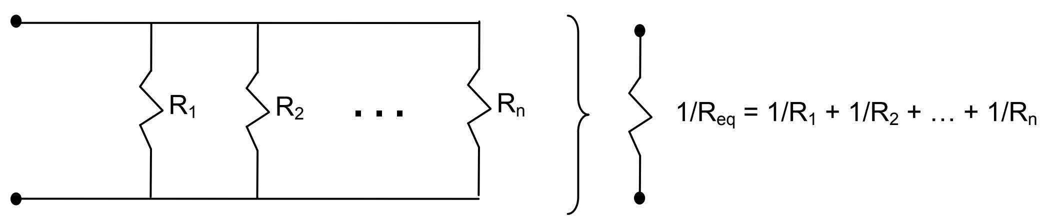


- Solution
  - R= Vg/Ig = 1000/0.005 = 200 kΩ
    $P_r = I_g^2 * R = (.005)^2 * (200{,}000) = 5$ W

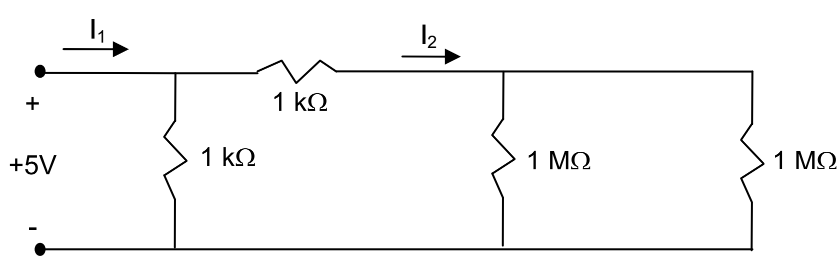
➢ Circuit simplification by combining resistors
- Resistors in Series
  Resistors in series can be replaced by an equivalent resistor that is the sum of all the resistors in series.



- Resistors in Parallel
  Resistors in parallel can be replaced by an equivalent resistor as shown below:

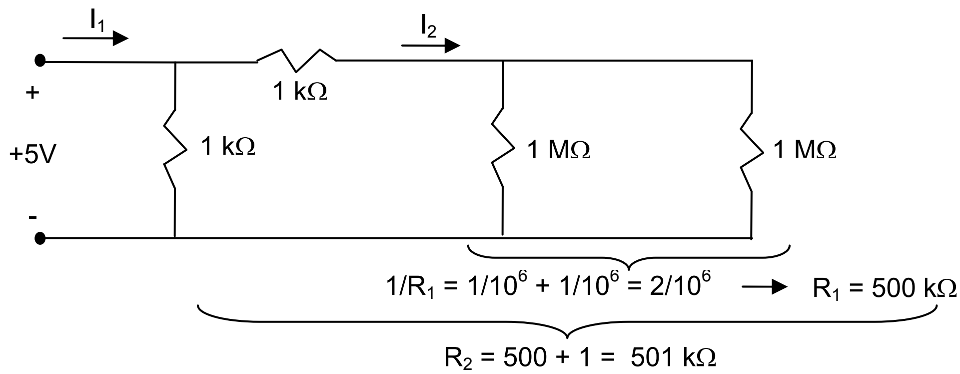


➢ Example
Find values of I_1 and I_2 for the following circuit:

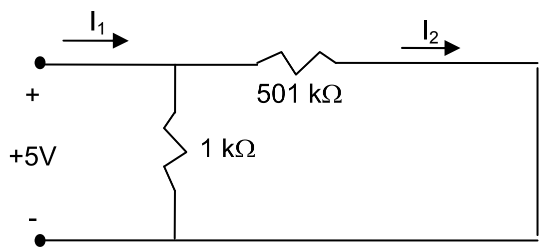


Solution:
1) Simplify the circuit by combining the two 1 MΩ parallel resistors with the 1 kΩ resistor that is in

series with them.



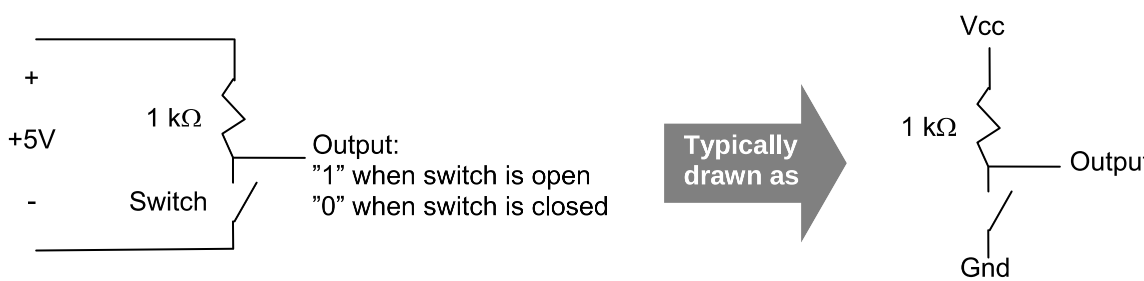
2) Redraw the circuit and apply Ohms law (V=I*R) to find currents.



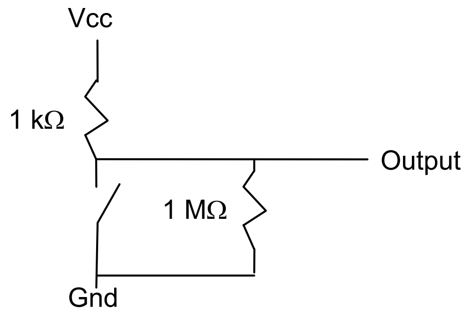
I_1 = V / R = 5 / 1000 = 0.005 A = 5 mA
I_2 = V / R = 5 / 501000 = 0.00001 A = 10 uA

Note: The amount of current through each resistor is inversely proportional to the size of the resistors.

- Using a switch to create logic 1 “+5 v” and logic 0 “Ground or 0 v”

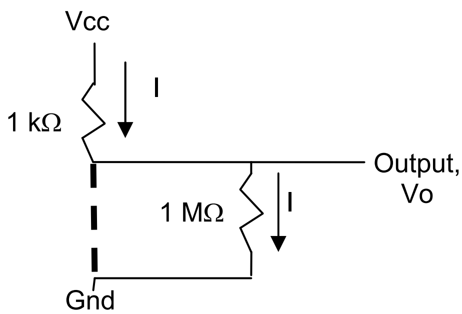


- What is the voltage at the output (vo) when the switch is Opened and closed in the following circuit:



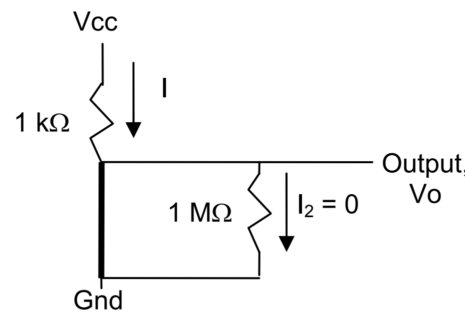
Solution:

<u>Switch Open</u>



I = V/R = 5 / (1001 x 10^3)
I ≈ 5 x 10^{-6} A
Vo = 0 + 10^6x5 x 10^{-6} = 5v

<u>Switch Closed</u>



Vo=0
"The output is directly Connected to Ground or 0 v"

1.11. Additional Resources

- Wakerly, I. Digital Design. (2006) Prentice Hall
  Chapter 1 & 2. “introduction” & “Number Systems and Code”

1.12. Problems

1. Convert the following binary numbers to equivalent hexadecimal numbers.
   a) $(10001010)_2$
   b) $(11110000)_2$
   c) $(10000001)_2$
   d) $(101001111)_2$

2. Convert the following binary numbers to equivalent octal numbers.
   a) $(010100)_2$
   b) $(111101100)_2$
   c) $(100110011)_2$
   d) $(011110101111)_2$
   e) $(010001110000011)_2$

3. Convert the following octal and hex numbers to binary numbers.
   a) $(3451)_8$
   b) $(65473)_8$
   c) $(563451)_8$
   d) $(7657.1100)_8$
   e) $(BDE)_{16}$

4. Convert the following numbers to their decimal equivalents using polynomial functions.
   a) $(27431)_8$
   b) $(476620)_8$
   c) $(1234.567)_8$
   d) $(11011110)_8$
   e) $(FFFCC)_{16}$
   f) $(123430)_{16}$
   g) $(E2B4.5E7)_{16}$
   h) $(11011110)_{16}$

5. Calculate the binary, octal and decimal equivalents of hexadecimal number $(01ECEB)_{16}$.

6. Obtain the decimal values for the following binary numbers expressed in a 2's complement representation (in each case the most significant bit is the sign bit).

   a. $(0110110)_{RC}$
   b. $(01011010111)_{RC}$
   c. $(101100101.1010)_{RC}$

7. Carry out the following arithmetic addition operations in both decimal and RC representation. Use a word size of 8 bits for the numbers expressed in RC representation.

   a. 6 +(-3)
   b. 95 + 27
   c. 101 + (-46)
   d. 39 + (-17)

8. Calculate the result (+235 – 281) using 16-bit Binary RC representation. Show the result as a Binary RC representation.

9. Determine the minimum number of bits that are required to represent all the characters on a keyboard that has the following number of keys.

a. 9
b. 16
c. 22
d. 36
e. 104

10. Determine the minimum number of bits required to uniquely represent each art work in a gallery with as many as 2520 art pieces.

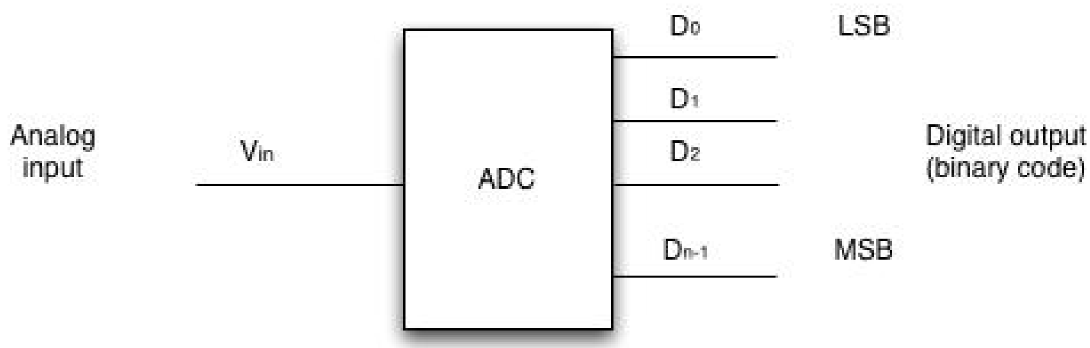
11. Look up and record the ASCII character for the following binary bit patterns.

a. 1111001
b. 1010110
c. 0000011
d. 1011001
e. 1100110

12. A block diagram for an analog-to-digital converter or ADC is shown below. This device converts an analog signal (a voltage) supplied to its input into a binary code at its output. The resolution of each bit, also called the resolution of the LSB (the voltage value), the full-scale range of the input voltage V_{in}, and the number of output bits *n* of the ADC are related as follows:

$$\text{Resolution of LSB} = V_{in} / 2^n$$

Given the following requirements, find the minimum number of output bits for each ADC.



a. Resolution of LSB <= 0.25 volts and a full-scale voltage range of 5 volts
b. Resolution of LSB <= 70 millivolts and a full-scale voltage range of 10 volts
c. Resolution of LSB <= 8 millivolts and a full-scale voltage range of 12 volts
d. Resolution of LSB <= 3/4 millivolts and a full-scale voltage range of 12 volts

13. What are the four scales of semiconductor integrations?

14. What are the design methodologies? Provide a brief explanation of each.

15. Write down the ASCII codes (in Dec.) for the characters in the string “Hello, Hope you are doing well!” excluding the quotation marks.

16. Write down the ASCII codes (in Hex.) for the characters in the string “This is a wonderful Course!” excluding the quotation marks.

Chapter 2. Boolean Algebra, Functions, and Minimization

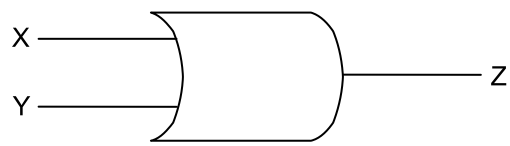
2.1. Key concepts and Overview

- Logic Gates
- Huntington's First Set of Postulates
- Principle of Duality
- Boolean Functions
- Boolean Algebra Theorems
- Canonical or Standard Forms (Min-term and Max-term)
- Function Minimization
- Karnaugh Maps (K-Maps)
- Special Case: Don't Care Terms
- Exclusive OR Properties and Applications
- Additional Resources
- Problems

2.2. Logic Gates

- The rest of the class relies on two-valued Boolean Algebra, i.e. B = {0, 1}
  - We use variables X, Y, Z, A, B, … and constants 0 and 1
    ***Note "0" and "1" are also called identity elements*

  - Binary operators
    - "+" called "OR"
      - "OR" symbol → Z=X+Y

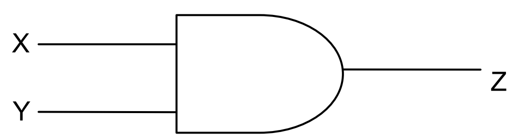


      - "OR" truth table

X	**Y**	**X+Y**
0	0	0
0	1	1
1	0	1
1	1	1

      - Review the 74LS32 Data sheet on the website

    - "." called "AND"
      - AND symbol → Z=X.Y



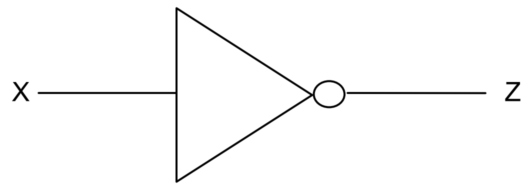
      - "AND" truth table

X	**Y**	**X.Y**
0	0	0
0	1	0
1	0	0
1	1	1

      - Review the 74LS08 Data sheet on the website

    - "¯" Called "NOT, Inversion, negation"

- “NOT” symbol → Z= $\overline{X}$ =X’ (Also called the complement)



- “NOT” truth table

X	Z=X’
0	1
1	0

- Review the 74LS04 Data sheet on the website

➢ Order of Operation Precedence (Same as decimal arithmetic)
Highest to lowest order of Precedence for Binary Operator: “=”, “()”, “ $\overline{\ }$ ”, “.”, “+”
Note:
  - Parentheses are used to force the operation order sequence much like decimal Algebra.
  - The equal sign “=” is same as decimal algebra for assignment.

➢ An expression is a combination of variables and binary operators Z+ X•Y + X

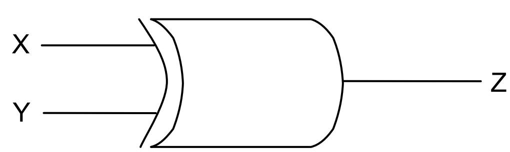
➢ The number of Literal is the total occurrences of all variables in an expression.
For example f(x,y,z) = x+ y.x.z + x’.y’.z is said to have 7 literals. The number of literals typically used as a measure of implementation complexity.

❖ Additional standard logic gates:

➢ NOR is an OR gate with the output negated
Review the 74LS02 Data sheet on the website

➢ NAND is an AND gate with the output negated
Review the 74LS00 Data sheet on the website

➢ XOR (also called “Module 2 add” or “exclusive or”) → $X \oplus Y = \overline{X}.Y + X.\overline{Y}$
  - “XOR” Symbol



  - “XOR” Truth Table
Review the 74LS86 Data sheet on the website

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

- When using three variables, the operation is performed on two at a time as shown below: $X \oplus Y \oplus Z = (X \oplus Y) \oplus Z$
- "XOR" is commonly used to check if there is an odd or even number of "1"s. This check is called "parity". Odd parity is when there are odd numbers of "1"s and even parity is when there are an even number of "1"s. Parity check is used for single bit error detection.

➢ XNOR (also called "equality coincidence" or "exclusive nor") → $\overline{X \oplus Y} = \overline{X}.\overline{Y} + X.Y$.

- "XNOR" Truth Table

X	**Y**	$\overline{X \oplus Y}$
0	0	1
0	1	0
1	0	0
1	1	1

- When using three variables, operation is perform on two at a time as shown below: $\overline{X \oplus Y \oplus Z} = \overline{(X \oplus Y) \oplus Z}$
- XNOR is not commonly available as a standard stand alone chip.

2.3. Huntington's First Set of Postulates

Postulates, Axioms or propositions are self-evident mathematical statements that are stated without proof. The following Postulates will be used to develop Boolean Algebra.

- P1a: If X and Y are in B, then X+Y is in B
- P1b: If X and Y are in B, then X•Y is in B

- P2a: There is an element 0 such that X+0 = X for every variable X.
- P2b: There is an element 1 such that X•1=X for every variable X.

- P3a: X+Y = Y+X (Commutative with respect to +)
- P3b: X•Y = Y•X (Commutative with respect to .)

- P4a: X+Y•Z = (X+Y)•(X+Z) (+ is distributive over .)

- P4b: X•(Y+Z)=X•Y+X•Z (▪ is distributive over +)

- P5: For every variable X, there is a variable $\overline{X}$ such that
  - a: $X + \overline{X} = 1$
  - b: $X \bullet \overline{X} = 0$

- P6: There are at least two distinct elements in B.

2.4. Principle of Duality

- Dual of an expression is obtained by:
  - Interchanging "0" and "1"
  - Interchanging "." and "+"

- $(Exp)^D$ represent dual of (Exp) → examples:
  - $(X+0)^D = X.1$
  - $(X + Y.Z)^D = X.(Y+Z)$ "**Note: it is not equal to X.Y + Z

- You may recognize that Huntington's first set of Postulates are true for duals (a and b of each postulate)

2.5. Boolean Functions

- Pure form
  - X.Y.Z is called the product of terms when literals are ANDed together
  - X+Y+Z is called the sum of terms when literals are ORed together

- Mixed forms
  - (X+Y).(Z+Y+X) → this form is called the product of sums form (POS form)
  - X.Y + Y.Z → this form is called the sum of products form (SOP form)

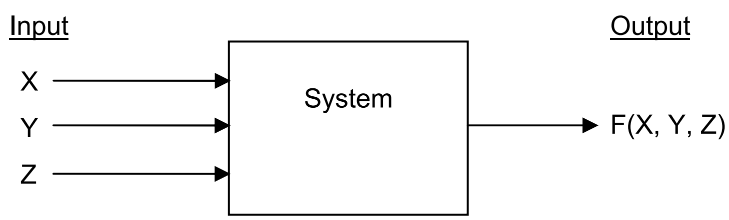
- Truth table for a function

  - Steps
    - Identify all possible combination of "1s" and "0s". For an "n" variable function, there will 2^n rows in the table counting from 0 to $2^{(n-1)}$.
    - Evaluate the output function value for each set of input variable values.

  - Example

    Draw a system diagram and generate a truth table for the function, F(X, Y, Z) = X.Y + Y.Z + Z'.Y

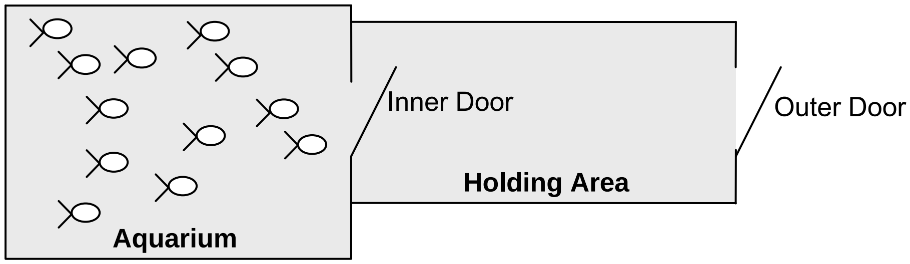
<u>System Diagram</u>



<u>Truth Table</u>

X	Y	Z	F(X,Y,Z)
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

- Example – Basics of Digital Logic
  Develop a system diagram and truth table for two gates that ensure only one fish leaves the Outer Door at any one time. Below is the physical diagram of the system:



- Holding Area has a sensor with 2 bit output
  (00: empty, 01:one fish, 10 & 11: more than one fish)
- Each door has one input where logic “1” opens and logic “0” closes the door.

2.6. Boolean Algebra Theorems

- Purpose of Theorems
  The Theorems & Huntington's Postulates are key in our ability to reduce the number of literal (variables) used in a function and therefore reduce the number of gates required to implement a given function. Sometimes they are used to simply rearrange the expression so it is easier to implement.

  - Example: $(X+Y)\bullet(X+\overline{Y})=X$
    It is clear that right-hand-side requires fewer gates to implement compared to the left hand side.

- Two methods available for proving theorems
  - Prove through Boolean Algebra
    Use the Huntington's postulates or theorems already proved to show that both sides of theorem are the same.

  - Prove through Truth Tables
    Show that for all possible values on the left hand-side is equal to the right-hand side of the equation. This method works well for a small number of variables.

- Theorems and proofs
  - Theorem 1 " Double Complementation or Double Negation Theorem"
    a) $\overline{\overline{X}} = X$

  - Theorem 2 "Idempotency Theorem"
    a) $X+X = X$
    b) $X\bullet X = X$

  - Theorem 3 "Identity Element Theorem"
    a) $X + 1 = 1$
    b) $X\bullet 0 = 0$

  - Theorem 4 "Absorption Theorem"
    a) $X + X\bullet Y=X$
    b) $X\bullet(X+Y) = X$

  - Theorem 5 "Associative Theorem"
    a) $X + (Y+Z) = (X + Y) + Z$
    b) $X\bullet(Y\bullet Z) = (X\bullet Y)\bullet Z$

  - Theorem 6 "Adjacency Theorem"
    a) $X\bullet Y + X\bullet \overline{Y} = X$
    b) $(X + Y)\bullet(X + \overline{Y}) = X$

  - Theorem 7 "Consensus Theorem"
    a) $X\bullet Y + \overline{X}\bullet Z + Y\bullet Z = X\bullet Y + \overline{X}\bullet Z$
    b) $(X + Y)\bullet(\overline{X} + Z)\bullet(Y + Z) = (X + Y)\bullet(\overline{X} + Z)$

  - Theorem 8 "Simplification Theorem"
    a) $X + \overline{X}\bullet Y = X + Y$
    b) $X\bullet(\overline{X} + Y) = X\bullet Y$

- Theorem 9 “DeMorgan’s Theorem (2-Variable form)”
  - a) $\overline{X+Y} = \overline{X}.\overline{Y}$
  - b) $\overline{X \bullet Y} = \overline{X} + \overline{Y}$

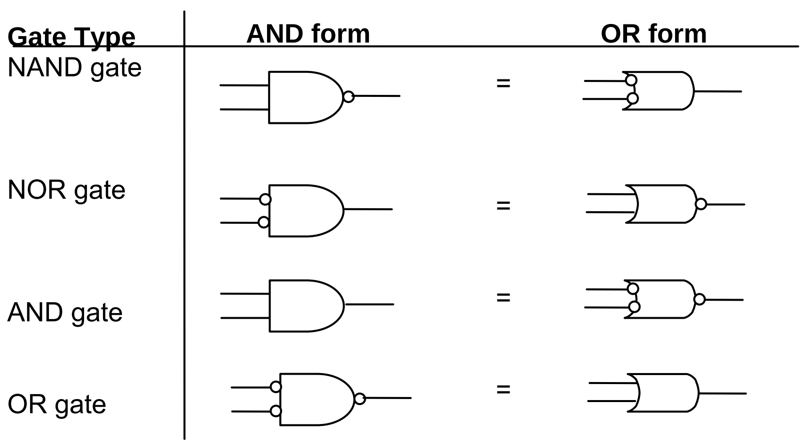
- Theorem 10 “DeMorgan’s Theorem (General form)”
  - a) $\overline{X_1 + X_2 + \ldots + X_n} = \overline{X_1} \bullet \overline{X_2} \bullet \ldots \bullet \overline{X_n}$
  - b) $\overline{X_1 \bullet X_2 \bullet \ldots \bullet X_n} = \overline{X_1} + .\overline{X_2}. + \ldots + \overline{X_n}$

❖ Example of two type of Proofs (Truth Table and Algebraic)

- Prove Theorem 8, “ Simplification Theorem”.
  Hint: Use truth table

- Prove Theorem 10, “DeMorgan’s Theorem (General form)”.
  Hint: Apply Theorem 9.

❖ Utilizing Demorgan’s Theorem NAND and NOR gates may be represented using the other’s base signal as shown below (“not” circle indicates complement):

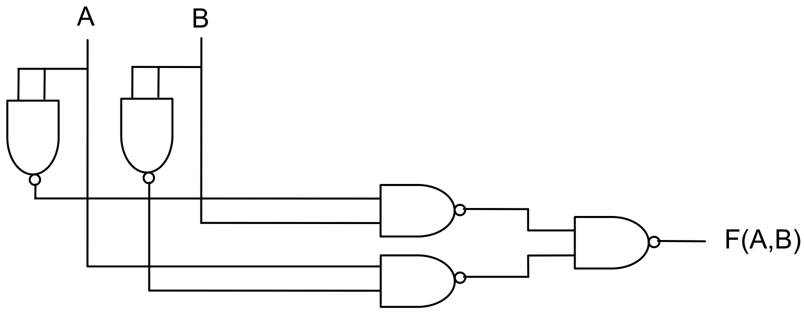


Transforming from one form to another requires only two steps:

1) Complement every input and output.
2) Swap OR and AND gates.

- Example: Design an XOR using only NAND gates.
  F(A,B) = A ⊕ B

  Solution:
  F(A,B) = A’.B + B’.A
  Apply conversion rules “Complement every input and output; Swap ORs and ANDs”
  F(A,B) = ((A’.B)’ . (B’.A)’)’


A
B
F(A,B)

2.7. Canonical or Standard Form of Functions

- Typically, a function has to be written in one of the two standard forms before the minimization step. The two standard forms are:

  - Standard Sum of Products (SOP)
  - Standard Product of Sums (POS)

- Obtaining Standard Sum of Products (SOP) Forms of Functions
  In standard or canonical SOP form, all the variables are present in each product term.

  - Example - for f(A,B) = A+B

    1) System Diagram

    2) Write the Truth table to see all the possible value of F(A,B)

Input		Output
A	B	F(A,B)=A+B
0	0	0
0	1	1
1	0	1
1	1	1

    3) Write the full product term for all the possible combinations

F(A,B) = F(0,0). $\overline{A}.\overline{B}$ + F(0,1). $\overline{A}B$ + F(1,0). $A.\overline{B}$ + F(1,1). $A.B$
F(A,B) = 0. $\overline{A}.\overline{B}$ + 1. $\overline{A}B$ + 1. $A.\overline{B}$ + 1. $A.B$

F(A,B) = 1. $\overline{A}B$ + 1. $A.\overline{B}$ + 1. $A.B$ → Canonical or Standard Form

A standard product or “min-term” is a product of all independent input variables for a function that corresponds to a row of the truth table with output of 1. For example, $A.\overline{B}$ is a min term in the above example.

- Let's take another example from problem statement to truth table to min-terms and the resulting sum of products.

  Step 1) Understand the problem
  Write out an expression for the function that is true, when 2 out of 3 inputs are true. Output is false for all other input combinations.

  Step 2) Develop a truth table for the function

Input X Y Z	Standard Product Terms (min-terms)	Min-term Designators	Output F
0 0 0	$\overline{X}.\overline{Y}.\overline{Z}$	m_0	$F(0,0,0) = F_0 = 0$
0 0 1	$\overline{X}.\overline{Y}.Z$	m_1	$F(0,0,1) = F_1 = 0$
0 1 0	$\overline{X}.Y.\overline{Z}$	m_2	$F(0,1,0) = F_2 = 0$
0 1 1	$\overline{X}.Y.Z$	m_3	$F(0,1,1) = F_3 = 1$
1 0 0	$X.\overline{Y}.\overline{Z}$	m_4	$F(1,0,0) = F_4 = 0$
1 0 1	$X.\overline{Y}.Z$	m_5	$F(1,0,1) = F_5 = 1$
1 1 0	$X.Y.\overline{Z}$	m_6	$F(1,1,0) = F_6 = 1$
1 1 1	XYZ	m_7	$F(1,1,1) = F_7 = 0$

  Note:
  1) The min-term subscript corresponds to the binary value of the input.
  2) All three independent input variables are present in each min-term.
  3) When input is 1, the corresponding variable appears in the Min-term, otherwise the variable is complemented in the min-term.

  Step 3) Write the algebraic function equivalent to the truth table by rule:

  If the output function (F) is 1 for the "min-term", then the value appears in the algebraic form of the expression.

  $F(X, Y, Z) = F_0.m_0 + F_1.m_1 + F_2.m_2 + F_3.m_3 + F_4.m_4 + F_5.m_5 + F_6.m_6 + F_7.m_7$

  $= \sum_{i=0}^{7}(F_i.m_i)$ Generalized compact Min-term form of the function

  $= 0.m_0 + 0.m_1 + 0.m_2 + 1.m_3 + 0.m_4 + 1.m_5 + 1.m_6 + 0.m_7$

  $F(X, Y, Z) = m_3 + m_5 + m_6$ Compact min-term form of the function

$F(X, Y, Z) = \sum m(3,5,6)$ Explicit Compact Min-term form for 1s of the function

$F(X, Y, Z) = \sum(3,5,6)$ Implicit Compact Min-term form for 1s of the function

By the way, the Not (Complement) of F can be written as (write the missing min-terms):

$\overline{F}(X, Y, Z) = \sum m(0,1,2,4,7)$ Explicit Compact Min-term form for 0s of the original function

$\overline{F}(X, Y, Z) = \sum(0,1,2,4,7)$ Implicit Compact Min-term form for 0s of the original function

- Obtaining the Standard Products of Sum (POS) Form of Functions
  Although POS is not used as much, there are times where the POS form is more efficient than SOP.
  - As the name applies, all three independent variables are present in either complemented or un-complemented form.

- For each pattern, if the independent variable value is 0, it is un-complemented, and if 1, it is complemented in the max-term which is the OR of all independent variables.

  For example: X=1, Y=1, Z=0 → M_6 = $\overline{X} + \overline{Y} + Z$

- Each max-term will result in the output for that term being zero.

➢ Here is an example for a 3-input system:

Step 1) Understand the problem

Write the expression for a function that is true when more than 1 input is true, otherwise the function is 0.

Step 2) Develop a truth table for the function and write max-terms:

All independent variables must be present in each Max-term

* It is complemented if the variable value is 1.

* It is un-complemented if the variable value is 0.

Input X Y Z	Standard Sum Terms (Max-terms)	Max-term Designators	Output F
0 0 0	X+Y+Z	M_0	F(0,0,0) = F_0 = 0
0 0 1	$X + Y + \overline{Z}$	M_1	F(0,0,1) = F_1 = 0
0 1 0	$X + \overline{Y} + Z$	M_2	F(0,1,0) = F_2 = 0
0 1 1	$X + \overline{Y} + \overline{Z}$	M_3	F(0,1,1) = F_3 = 1
1 0 0	$\overline{X} + Y + Z$	M_4	F(1,0,0) = F_4 = 0
1 0 1	$\overline{X} + Y + \overline{Z}$	M_5	F(1,0,1) = F_5 = 1
1 1 0	$\overline{X} + \overline{Y} + Z$	M_6	F(1,1,0) = F_6 = 1
1 1 1	$\overline{X} + \overline{Y} + \overline{Z}$	M_7	F(1,1,1) = F_7 = 1

Step 3) Write the algebraic function equivalent to the truth table by rule:

For Compact Max-term Form:

If the Output function (F) is 0 for the max-term, then the value appears in the algebraic form of the expression.

F(X,Y,Z) = $(F_0 + M_0). (F_1 + M_1). (F_2 + M_2). (F_3 + M_3). (F_4 + M_4). (F_5 + M_5). (F_6 + M_6). (F_7 + M_7)$

= $\prod_{i=0}^{7}(F_i + M_i)$ Generalized compact max-term form of the function

Note the when F_i=1, the max-term is not needed --- for our example:

F(X,Y,Z) = $(0 + M_0). (0 + M_1). (0 + M_2). (1 + M_3). (0 + M_4). (1 + M_5). (1 + M_6). (1 + M_7)$

F(X,Y,Z) = $M_0 . M_1 . M_2 . M_4$ Compact Max-term form of the function

Other forms:

F(X,Y,Z) = $\prod M(0,1,2,4)$ Explicit Compact max-term form for 1s of the function

F(X,Y,Z) = $\prod(0,1,2,4)$ Implicit Compact max-term form for 1s of the function

➢ The Not (Complement) of F can be written by writing the missing max-terms for Uncompleneted F:

$\overline{F}$ (X,Y,Z) = $\Pi M(3,5,6,7)$ Explicit Compact max-term form for 0s of the function

$\overline{F}$ (X,Y,Z) = $\Pi(3,5,6,7)$ Implicit Compact max-term form for 0s of the function

- Relationship between Min-terms and Max-terms

  Min-terms and Max-terms are complements of each other : $\overline{M_i} = m_i \quad and \quad M_i = \overline{m_i}$

  - DeMorgan's Theorem is key to proving the min-term/max-term relationship:

    a) $\overline{X_1 + X_2 + \ldots + X_n} = \overline{X_1} \bullet \overline{X_2} \bullet \ldots \bullet \overline{X_n}$

    b) $\overline{X_1 \bullet X_2 \bullet \ldots \bullet X_n} = \overline{X_1} + .\overline{X_2}. + \ldots + \overline{X_n}$

  - Examples:

    Given max-term M_6 = $\overline{X} + .\overline{Y} + Z$, find min-term m_6.

    1) Since it is a max-term, when X=1, Y=1 and Z=0 Then F(X,Y,Z) = 0

    2) To convert to Min-term we can apply DeMorgan's Theorem which in practice is **dividing up the overbar.** This means that the cross bar can be divided across its subpart while accepting the rules:

    $\overline{+} = . \quad and \quad \overline{.} = +$

    Let's see how it applies to our example.

    We know that $\overline{M_i} = m_i \quad and \quad M_i = \overline{m_i}$ so

    $Min - term = m_i = \overline{\overline{X} + \overline{Y} + Z} = \overline{\overline{X}}\overline{+}\overline{\overline{Y}}\overline{+}\overline{Z} = X.Y.\overline{Z}$

  - Example: Apply the overbar to finding Complement of F if F(X,Y,Z) = $(\overline{X} + Y).(\overline{Y} + \overline{Z})$

    Solution:

    Apply the DeMorgan's Theorem in the form of "Dividing up the Overbar".

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

  - Example

    Write Standard SOP and POS form for f(x_3, x_2, x_1, x_0) = $\sum(0,7,12,15)$

    Solution:

- Converting between compact forms of functions

- We can extend the relationship between max-terms and min-terms to include SOP (Sum of Products) and Products of Sum (POS):

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

$$\overline{\prod} = \sum \qquad and \qquad \overline{\sum} = \prod$$

- Example:
  Write the F(A,B,C)= $\prod(0,3,5,6)$ in the compact min-term form.

  We know that $\sum = \overline{\prod}$ Therefore

$$\overline{F(A,B,C)} = \sum(0,3,5,6) = \overline{\prod(0,3,5,6)} = \overline{(A+B+C).(A+\overline{B}+\overline{C}).(\overline{A}+B+\overline{C}).(\overline{A}+\overline{B}+C)}$$

  Since these terms are the 0's of the function, if we write the Min-terms that are not present then we will have the 1's of the function:

$$F(A,B,C) = \sum(1,2,4,7) = \overline{A}.\overline{B}.C + \overline{A}.B.\overline{C} + A.\overline{B}.\overline{C} + A.B.C$$

- Example
  Use only NAND gates to implement $f(a_2, a_1, a_0) = a_2.a_1.a_0 + a_2.a_1'.a_0' + a_2'.a_1'.a_0$

  Solution:

- Example
  Use only NOR gates to implement $f(a_2, a_1, a_0) = (a_2'+a_1+a_0') + (a_2'+a_1'+a_0) + (a_2'+a_1+a_0)$

  Solution:

2.8. Methods of Function Minimization (reducing the number of literals in an expression)

It is important to minimize the function prior to implementation. Minimization of literals and operators reduces the number of gates needed to implement the function therefore reducing the cost of implementation. In this section Systematic Algebraic Reduction (SAR) minimization techniques will be discussed. The SAR technique is effective for automation but tedious for human use. On the other hand, Karnaugh Map (K-Map) that will be discussed later is a visual tool that is more effect for human use to minimize functions.

.

.

❖ Systematic Algebraic Reduction (SAR) technique uses algebraic theorems and postulates. Although you could start applying various algorithms until you find one that reduces the function, our goal is to introduce systematic techniques that can be described in a step-by-step process (algorithm) and consistently applied.

- Usage
  Most Computer Aided Design (CAD) packages use the SAR technique for function minimization. Although SAR is not guaranteed to reduce the function to a minimum, it is the most effective algorithm available.

- Process
  Here is the step-by-step algorithm for a Systematic Algebraic Reduction(SAR):
  (1) Expand the function into its Standard sum of products (SOP) form
  (Include all variables; writing variables in order in all terms makes it easier to recognize patterns.)

  (2) Compare all pairs of products for:
  (a) Adjacency Theorem: "$\exp 1.\exp 2 + \exp 1.\overline{\exp 2} = \exp 1$"
  and
  (b) Idempotency Theorem: "$\exp + \exp = \exp$"

  **Note: The reduction process may have to be repeated a number of times.

  (3) Once you have done all reductions possible in step 2, See if the Consensus Theorem applies
  $$\exp 1.\exp 2 + \overline{\exp 1}.\exp 3 + \exp 2.\exp 3 = \exp 1.\exp 2 + \overline{\exp 1}.\exp 3$$

- Example: Using SAR, minimize the function F= $(A+B+C).(\overline{A}+C).(\overline{B}+C)$
  Solution:
  1) Use the truth table to derive the min-terms

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

2) Write the function in compact Min-term form

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

3) Apply Adjacency Theorem to all pairs as possible.
(another way is drawing double head arrows showing relationship between two terms)

$$(Term1\,\&\,2) \rightarrow \overline{A}.\overline{B}.C + \overline{A}.B.C = \overline{A}.C$$
$$(Term1\,\&\,3) \rightarrow \overline{A}.\overline{B}.C + A.\overline{B}.C = \overline{B}.C$$
$$(Term1\,\&\,4) \rightarrow Not - Applicable$$
$$(Term2\,\&\,3) \rightarrow Not - Appicable$$
$$(Term2\,\&\,4) \rightarrow \overline{A}.B.C + A.B.C = B.C$$
$$(Term3\,\&\,4) \rightarrow A.\overline{B}.C + A.B.C = A.C$$

Therefore:

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

4) Perform a second pass of Adjacency theorem.

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

$$\xleftrightarrow{\quad c \quad}$$
$$\xleftrightarrow{\qquad C \qquad}$$

Therefore:
F= C + C = C

In this case we did not need to apply the Consensus Theorem since the answer can not be simplified further.

- In general, Systematic Algebraic Reduction (SAR) methods are best suited for computer programming. K-maps, which will be described in the next section are best suited for human use up to 4 variables since it is graphic.

2.9. Karnaugh-map or K-map

K-map is the best tool for minimization of five or fewer variables functions for humans. K-maps are graphic and require pattern-matching which is one of human's strongest abilities. Many believe that humans solve problems by creative pattern-matching.

- K-map is a number of squares which are labeled using reflective gray code (each code is only 1 change from an adjacent code). For a given square, the user enters 0 or 1 corresponding to the function value at the inputs represented by the labels.

- Here are K-map examples for 2, 3, and 4 Variables:

A \ B	0	1
0	0	0
1	0	1

F(A,B)
2-Variables

AB \ C	0	1
00	1	0
01	0	0
11	0	1
10	0	0

F(A,B,C)
3-Variables

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

F(A,B,C,D)
4-Variables

- Each of the squares will contain a 1 if the function is 1 (min-term locations) and 0 otherwise. You may also use "-", which reflects the "don't care" (can be 0 or 1, whichever gives us the lowest Literal Count, LC).

  The Literal Count (LC) is proportional to the number of gates needed during the implementation, so the less the better.

- Here is the location of each min-term on a Karnaugh-Map:

A B C D	Min-term,m
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	10
1 0 1 1	11
1 1 0 0	12
1 1 0 1	13
1 1 1 0	14
1 1 1 1	15

A \ B	0	1
0	0	1
1	2	3

F(A,B)=AB
2-Variables

AB \ C	0	1
00	0	1
01	2	3
11	6	7
10	4	5

F(A,B,C)
3-Variables

AB \ CD	00	01	11	10
00	0	1	3	2
01	4	5	7	6
11	12	13	15	14
10	8	9	11	10

F(A,B,C,D)
4-Variables

- Example: Use K-map to minimize F(A,B,C) = $\overline{A}.\overline{B}.C + \overline{A}.B.C + A.\overline{B}.C + A.B.C$

  Solution:

  1. Use a truth table to identify all the Min-terms (Over time you can do this mentally, so it would not be necessary to draw it).

A B C	F	Min-term, m_i
0 0 0	0	0
0 0 1	1	1
0 1 0	0	2
0 1 1	1	3
1 0 0	0	4
1 0 1	1	5
1 1 0	0	6
1 1 1	1	7

  2. Fill in the K-map:
     a. Select the K-Map that matches the number of variables in your function, (3 for the Example)
     b. Draw the K-map (remember the labels are reflective Gray Code)
     c. Enter the value of the function for the corresponding min-term. If the value of the function is unspecified then enter – which means don't care.

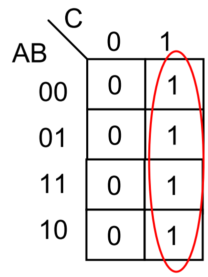
AB \ C	0	1
00	0	1
01	0	1
11	0	1
10	0	1

F(A,B,C)
3-Variables

  3. The next step is to group as many neighboring ones as possible. Cells with one variable is complemented are referred to as neighboring cells:
     a. Grouping adjacent min-terms (boxes) is applying the Adjacency theorem graphically, i.e.

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

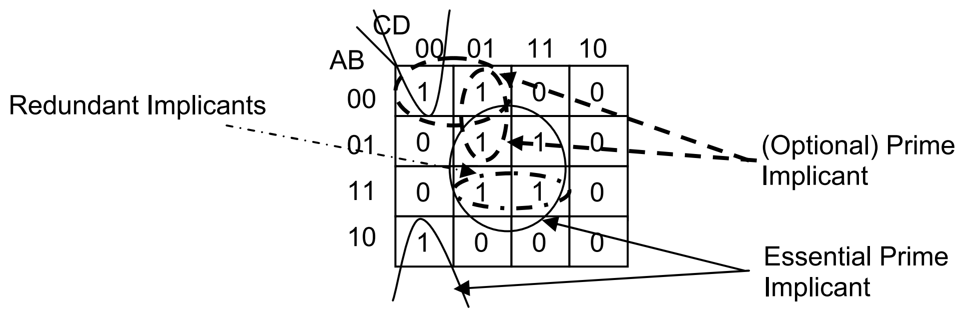
     b. The goal is to get as large a grouping of 1s as possible
        (Must form a full rectangle – cannot group diagonally)



F(A,B,C)
3-Variables

4. For each identified group, look to see which variable has a unique value. In this case, F(A,B,C) = C since F's value is not dependent on the value of A and B.

❖ More K-map related definitions:
  - Example: A function with the following K-Map



F(A,B,C,D)
4-Variables

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

- An ***Implicant*** is the product term where the function is evaluated to 1 or complemented to 0. An Implicant implies the term of the function is 1 or complemented to 0. Each square with a 1 for the function is called an implicant (p). If the complement of the function is being discussed, then 0's are called implicants (r).
  Note: To find the complement of F, apply the same rules to 0 entries in the K-map instead of 1.

- *A **Prime Implicant*** of a function is a rectangular (each side is powers of 2) group of product terms that is not completely contained in a single larger implicant.

- *An **Essential Prime Implicant*** of a function is a product term that provides the only coverage for a given min-term and must be used in the set of product terms to express a given function in minimum form.

- *An **Optional Prime Implicant*** of a function is a product term that provides an alternate covering for a given Min-term and may be used in the set of product terms to express a function in a minimum form. Some functions can be represented in a minimum form in more than one way because of optional prime implicants.

- A ***Redundant Prime Implicant*** or ***Nonessential Prime Implicant*** of a function is a product term that represents a square that is completely covered by other essential or optional prime

Implicants.

- Example: Write the minimized SOP function represented by the following K-Map

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

Solution:

- Example:
  use K-map to write the minimized SOP and POS forms of the following function:
  $F(A,B,C,D,E)=\sum(0,1,2,3,4,5,6,7,8,24,25,26,27)$

  Solution:

Example:
use K-map to write the minimized SOP and POS forms of the following function:
$F(A,B,C,D,E)=\prod(0,2,8,10,13,15)$

Solution:

2.10. Special Case: "Don't Care" Terms

❖ In K-map, we can use the unspecified values of a function "don't care" as 1 or 0, allowing us to create larger cubes to write products with smaller Literal Count (LCs)

- Example: F(W,X,Y,Z) with unspecified values (don't cares, "-")

WX \ YZ	00	01	11	10
00	0	1	-	-
01	1	1	-	0
11	1	0	1	1
10	0	-	1	0

We have an option of assuming "-" as 0 or 1 whichever ends up with a lower Literal Count (LC) and therefore lower hardware (gates) cost during the implementation phase. Here is one minimized function representing the K-Map function:

WX \ YZ	00	01	11	10
00	0	1	-	-
01	1	1	-	0
11	1	0	1	1
10	0	-	1	0

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

For this function the Literal Count (LC) is 10.

Sometimes it makes sense to use the 0s and write the complement to get a lower LC.

WX \ YZ	00	01	11	10
00	0	1	-	-
01	1	1	-	0
11	1	0	1	1
10	0	-	1	0

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

For this function, the Literal Count (LC) is 7.

Function in POS form

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

- Representing "don't care" min-terms in compact form.
  - $\sum md(1,4,5)$ → "md" refers to "don't care" min-terms.

2.11. XOR Properties and Applications

❖ K-map patterns
Checkerboard pattern: alternating cells and diagonal cells of 1s and 0s on a K-map is a sign of XOR or XNOR.

AB \ C	0	1
00	0	1
01	1	0
11	0	1
10	1	0

Checkerboard pattern
F = $A \oplus B \oplus C$

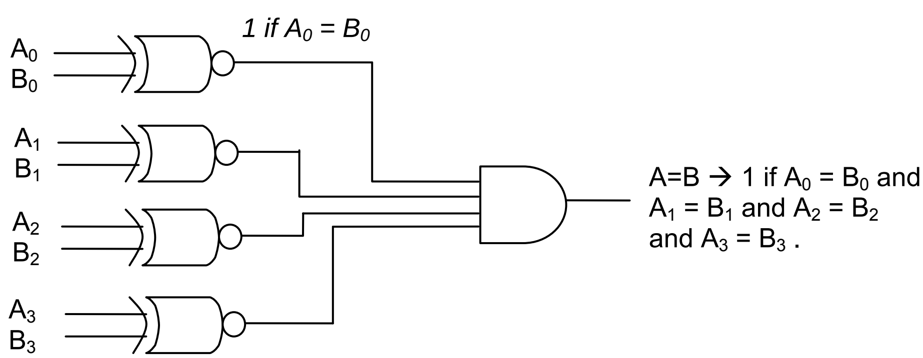
AB \ C	0	1
00	0	1
01	1	0
11	0	0
10	0	0

Diagonal Cells
F = $\overline{A}.(B \oplus C)$

AB \ C	0	1
00	0	0
01	1	1
11	0	0
10	1	1

Alternating Cells
F = $A \oplus B$

- XOR properties:
  - Commutative
    $A \oplus B \oplus C = C \oplus B \oplus A$
  - Associative
    $A \oplus (B \oplus C) = (C \oplus B) \oplus A$
  - Rubber band effect (The bar can be put anywhere and the result remains unchanged)
    $A \oplus B = \overline{A} \oplus \overline{B} = \overline{\overline{A} \oplus B} = \overline{A \oplus \overline{B}}$
  - XOR is 1 when there is an odd number of 1's in the XOR operands
    Note: This feature is used to do single bit error checking, which is adding an extra bit to the data to ensure that the number of 1's is odd. (This is known as odd parity).
  - XNOR is 1 when there is an even number of 1's in the XNOR operands
    Note that this feature is used to do single bit error checking, which is adding an extra bit to the data to ensure that the number of 1's is even. (This is known as even parity).
  - XNOR 4-bit Comparator Design



2.12. Additional Resources

- Wakerly, I. Digital Design. (2006) Prentice Hall
  Chapter 4 “Combinational Logic Design Principles”

2.13. Problems

1. Write the truth table for each of the following functions.

a) $F(X,Y) = X + X.Y$
b) $F(X,Y,Z) = X.Y + X.Z$
c) $F(X,Y,Z) = X.Y.\overline{Z} + \overline{X}.Y + Z$

2. Expand the function, $F(M,R,S) = \overline{S} + \overline{M}.R$, to its canonical or standard Sum-Of-Product(SOP) form:

3. Expand the function, $F(M,R,S) = \overline{S}.R + \overline{M}.R$, to its canonical or standard Product-Of-Sum (POS) form:

4. Expand the function, $f(A_3, A_2, A_1, A_0) = A_3A_2 + (A_1' + A_0).A_3'$ to its canonical or standard Product-Of-Sum (POS) and Sum-Of-Product (SOP) forms:

5. Write the corresponding compact minterm for each of the functions (f1, f2 and f3) shown in the following truth table:

X	Y	Z	F1	F2	F3	F4	Minterm
0	0	0	0	0	1	0	0
0	0	1	0	1	1	0	1
0	1	0	0	0	1	1	2
0	1	1	1	1	0	1	3
1	0	0	1	1	0	1	4
1	0	1	1	0	1	1	5
1	1	0	0	1	0	1	6
1	1	1	1	0	0	0	7

6. Each of the following functions is written in a form that is equivalent to the complement of the standard POS form of F. Write each function in a standard POS form of F and also in a standard SOP form of F. Leave each result in a compact or list form.

a) $\overline{F}$ (P,Q,R,S) = Σ (0, 5, 9, 13)
b) $\overline{F}$ (P,Q,R,S) = Σ (3,6,8,14)
c) $\overline{F}$ (P,Q,R,S) = Σ (4,5,6,7)
d) $\overline{F}$ (P,Q,R,S) = Σ (12,13,14,15)

7. Write the expression for each of the following functions in terms of the independent variables.

a) $F(V,W,X,Y,Z) = m_0 + m_6 + m_{23}$
b) $F(U,V,W,X,Y,Z) = m_{19} + m_{22} + m_{30}$
c) $F(U,V,W,X,Y,Z) = m_{25} + m_{35} + m_{47}$
d) $F(T,U,V,W,X,Y,Z) = m_{55} + m_{67} + m_{93}$

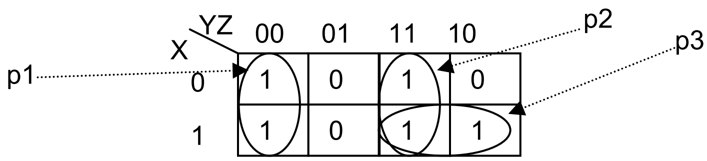
8. Write the following function in implicit SOP and POS compact forms.
f(p,q,r,s,t) = p.q'.r.s + (p + t').s'

9. Enter each of the following functions on a K-map and find a minimum SOP form for each function. Identify each product term by circling adjacent groups of 1s.

a) F(X,Y)= Σ(0,1)
b) F(X,Y)= Σ(1,3)
c) F(X,Y)= Σ(2,3)
d) F(X,Y)= π (1,3)

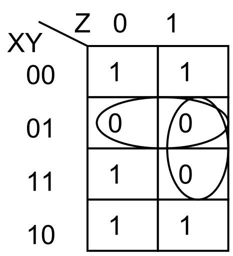
---

10. Obtain the product terms of p1, p2 and p3 circled in the following figure and verify that each product term evaluates to 1 for its respective input combinations. Use the p-terms to write the functions in sum-of-products form.



---

11. Write all the essential prime implicants for the 0s of the function in the map shown in figure below. Use these implicants to obtain a minimum SOP expression for the complement of the function.



---

12. Using a K-map, obtain the minimum POS expression for the following Boolean function.

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

---

13. Obtain the minimum SOP expression for the following Boolean function using a K-map and SAR.

F(W,X,Y,Z)= Σ(0,3,4,5,6,7,11,12,13,14,15) + Σ md(2,8,9)

---

14. For the function

F(s,r,p,q)= Σ(0,3,4,5,6,7,10) + Σ md(1,9)

a) Show the K-map, Essential Prime implicants (SOP) and the minimum SOP expression.
b) Show the K-map, Essential Prime implicants (POS) and the minimum POS expression.
c) Use SAR to find the minimum SOP expression.

Chapter 3. Analyzing and Synthesizing Combinational Logic Circuits

3.1. Key concepts and Overview

- ❖ Standard Logic and Schematic Layout
- ❖ Designing Logic Circuits
- ❖ Compressing Truth Tables & K-Maps
- ❖ Glitches and Hazards
- ❖ Types of Functions and Delays
- ❖ Beyond Standard Logic (Encoders, Decoders, PLD, GAL, ROM, PROM, …)
- ❖ Additional Resources
- ❖ Problems

3.2. Standard Logic and Schematic Layout (Review)

This section describes Small Scale Integration circuits which are commonly used for smaller projects. It also provides an understanding of the basics of computer design.

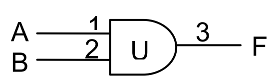
Basically, computers regardless of complexity, can be designed using these simple gates as building blocks. We will start by discussing the most fundamental gates which are AND, OR and NOT.

- ❖ AND Function (F=A.B)

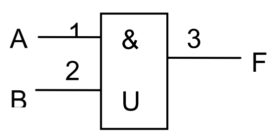
  - ➢ Truth Table

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

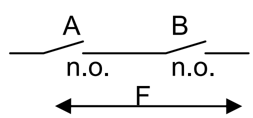
  - ➢ Symbols & Operation
    - Recommendation is to use the shapes for simple gates; and box (non-shape) for more complex logic
    - When using in a schematic mark the IC ID (D#) and Pin # on the schematics
    - A couple of other representations using switches (Relays, Transistors)



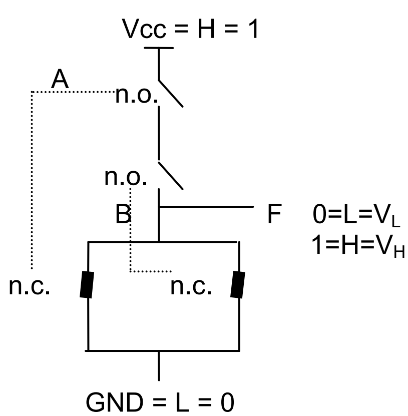
IEEE: Shape Distinctive Graphics



IEC: Non-Shape Distinctive Graphics Symbol



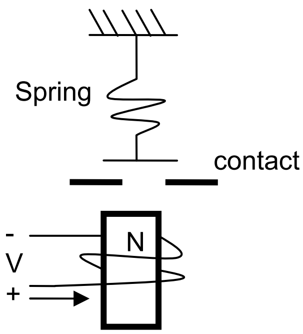
Pass Logic Switching Circuit
0: Button is Not pressed
1: Button is pressed



Regenerative Logic Switching Circuit

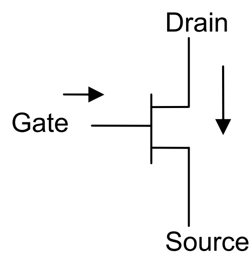
- Different Switches
  Instead of mechanical switches, we can also use electronic switches:

  - A normally open (n.o.) switch is closed when pressed.



Relay as n.o.

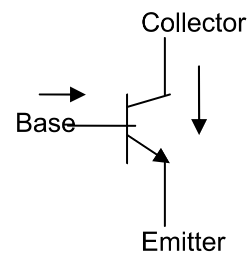
When Current Flows the Relay closes



PMOS FET as n.o.

Positive Channel Metal Oxide Semiconductor

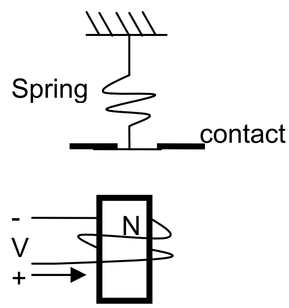
Field Effect Transistor



NPN BJT as n.o.

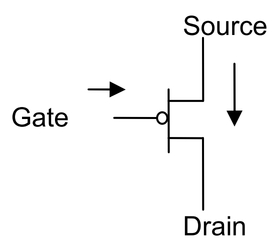
Bipolar Junction Transistors

  - A normally close (n.c.) switch is open when pressed



Relay as n.c.

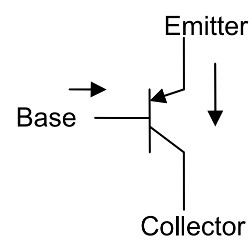
When Current Flows the Relay opens



NMOS FET as n.c.

Négative Channel Métal Oxide Semiconductor

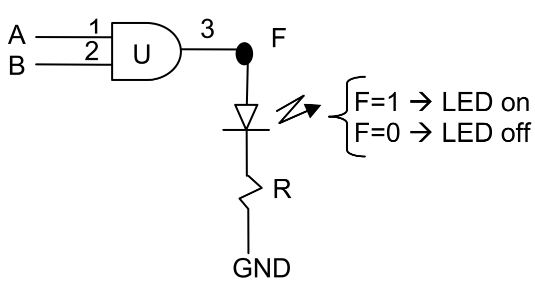
Field Effect Transistor

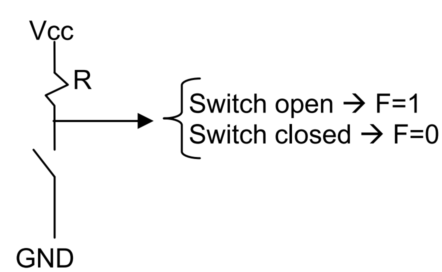


PNP BJT as n.c.

Bipolar Junction Transistors

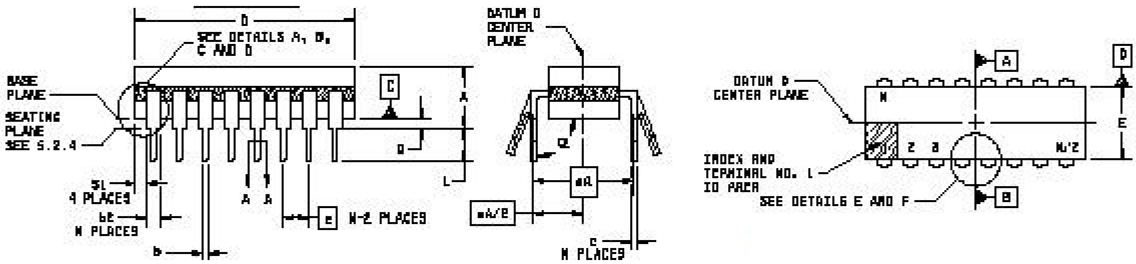
- Example of setting up a LED and a switch



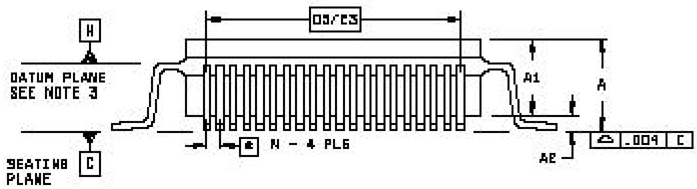


Note: Typically R=1 kΩ is used.

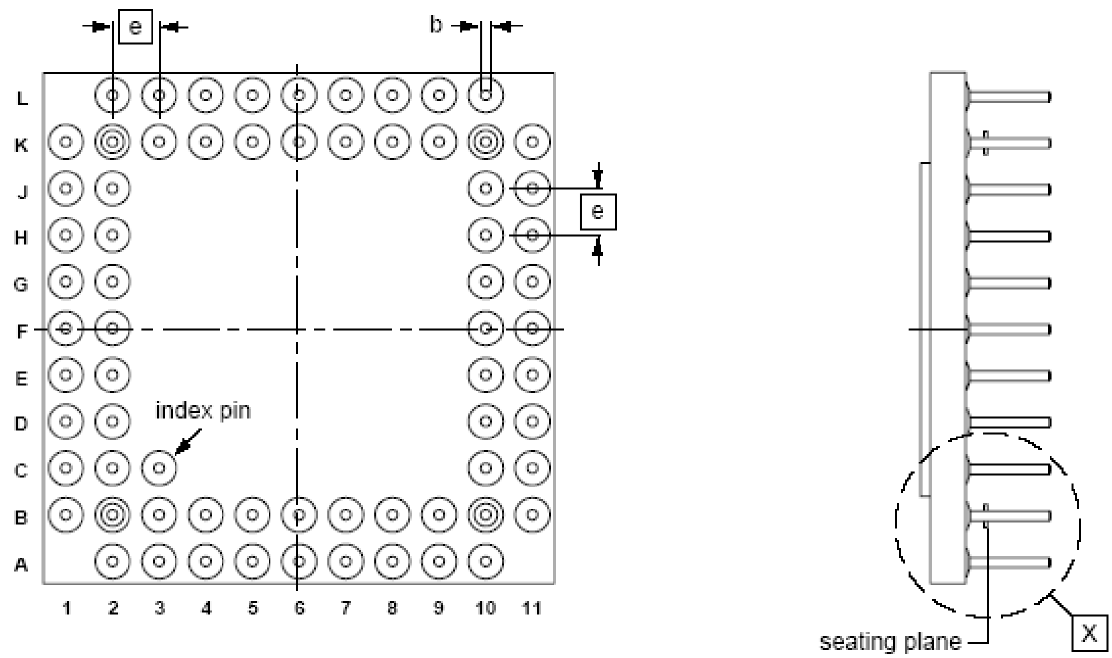
❖ Physical packaging



Through Hole Device (2 to 100 pins)



Surface Mount Device Package (2 to 100 pins)



PIN Grid Array (>100 pins)

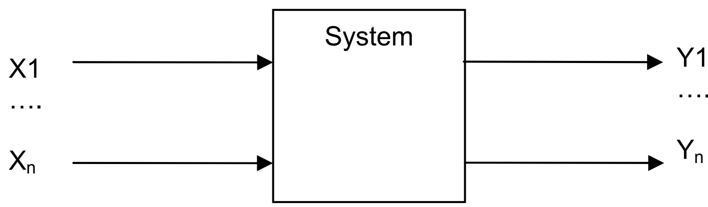
❖ Number of possible functions for an n-variable input equals $2^{(2^n)}$

For example, a device with 2 input may have one of the possible $2^{(2^2)} = 2^4 = 16$ functions.

2-Input X Y	**F0**	**F1**	**F2**	**F3**	**F4**	**F5**	**F6**	**F7**	**F8**	**F9**	**F10**	**F11**	**F12**	**F13**	**F14**	**F15**
0 0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0 1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1 0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1 1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
	GND	And						OR								Vcc

❖ Analyzing Logic Circuits

- Draw a system diagram and identify input/output signals

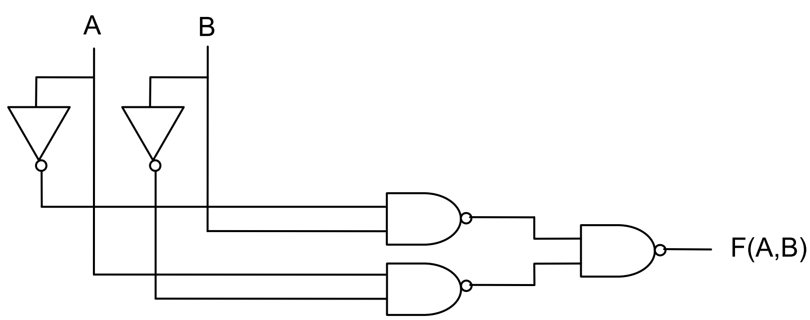


- Based on the schematic, write out the Boolean algebraic equation
  f(x, y, z) = ?

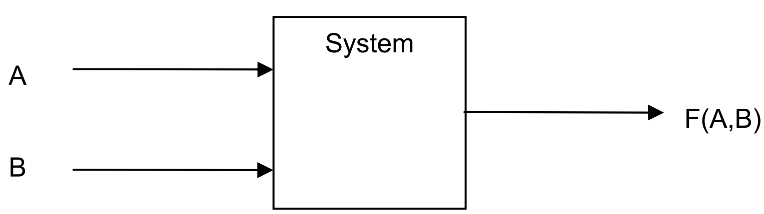
- Based on the equation, do the K-map or truth table

- Review the truth table to understand the function of the circuit

➢ Example
Analyze the following circuit:



- System Diagram



- Algebraic equation
  F(A,B) = {(A'.B)'.(A.B')'}'

- Truth Table

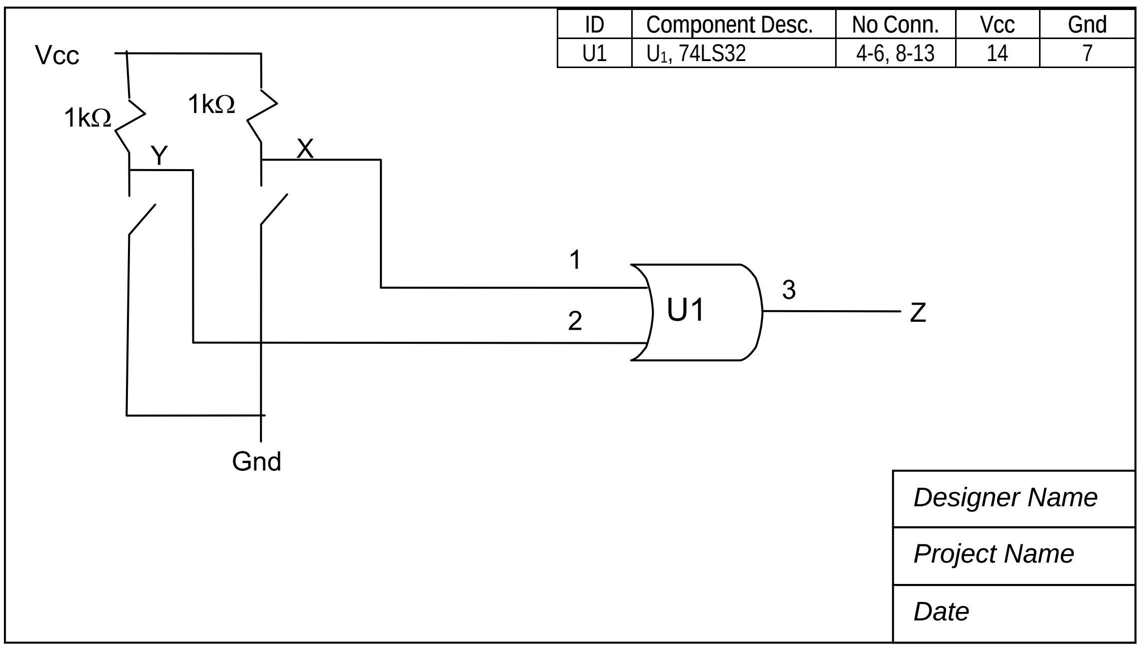
A	B	F(A,B)
0	0	0
0	1	1
1	0	1
1	1	0

- This circuit is implementing an exclusive OR.

3.3. Designing Logic Circuits

The process of combinational logic design is best described in six steps:

1) Draw the system diagram and identify input and output variables.
2) Write out the truth table for the function.
3) Use K-maps or CAD to minimize the function and write the algebraic function.
4) Identify the logic gates required and draw the schematic to implement the terms of the algebraic function. The schematic should include:
   - Designer and project name.
   - Component identification and pin numbers.
   - List of all the pins that are connected to Vcc and Ground, and are not use.
     All Connecting lines must be either vertical or horizontal (No diagonal or curved lines). Additionally, a dot is used to mark an actual connection when two lines cross over each other.



ID	Component Desc.	No Conn.	Vcc	Gnd
U1	U_1, 74LS32	4-6, 8-13	14	7

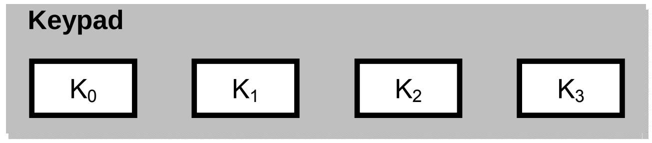
Note: Hardware Description languages such as VHDL or Verilog HDL or State CAD by Visual Software Solutions, can also be used to document the design.

5) Implement (pay attention to layout and ease of support/use).
6) Test (each design must have a test plan).

Remember that the design process is an iterative process; it is important to use the learning from later steps of the design process to improve earlier work. The best designs typically have many design iterations before the final design has been completed.

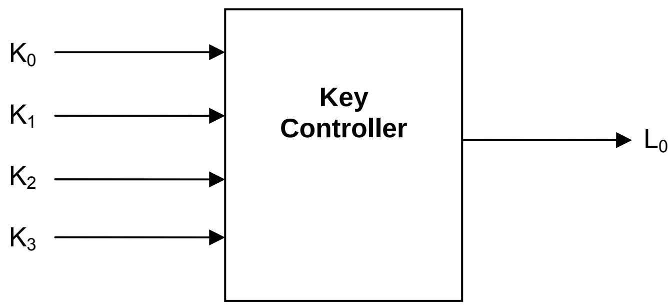
➢ Example

Design a 4-key digital lock that can be opened only when every other keys are pressed.



Solution:

1) Draw the system diagram and Identify input and output variables..



- Input and output value definition:
  K_i =1 when key “i” is pressed and K_i =0 when key “i” is not pressed.
  L_0=1 causes the lock to open

2) Write out the truth table for the function:

Input				Output
K_0	**K_1**	**K_2**	**K_3**	**L_0**
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	1
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	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

3) Use K-map or CAD to minimize the function and write the algebraic function.

4) Identify the logic gates required and draw the schematic to implement the terms of the algebraic function.

ID	Component Desc.	No Conn.	Vcc	Gnd

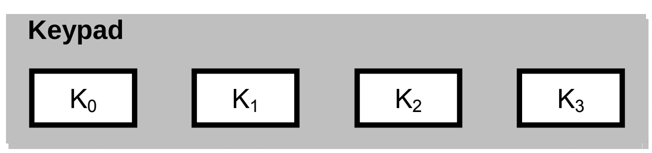
Designer:
Project:
Date:

5) Implement (pay attention to layout and ease of support/use).

6) Test (outline test plan).

- Example

Design a 4-key digital lock that can be opened only when 2 or more adjacent keys are pressed.



Solution: Student Exercise.

- ❖ Fan-out and Fan-in
  1) Fan-out: The number of gate inputs that can be driven by a single output (for LS chips. the fan-out limit is 20)
  2) Fan-in: The number of gate inputs that may be connected together. Typically, fan-out is the limitation factor for LS.

- ❖ Using NAND or NOR gates

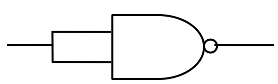
  NAND and NOR gates can be substituted for each other. Basically, NOR implements POS and NAND implements SOP. A function can equally be written as POS or SOP.

  We will explore NAND gates here and similar concepts also apply to NOR gates.

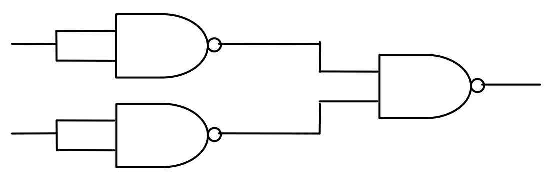
  - ➢ NAND is one the most desirable basic gates for three reasons:

    1) NAND gates are faster than other gates such as AND.
    2) NAND gates can be used to make all other gates such as inverters, ANDs and ORs (Demorgan's Theorem is the basis of this statement.)

       Therefore NAND is said to be "functionally complete" since it can be used to build any other function. Below are examples of Inverter and OR function implementation.



INVERTER

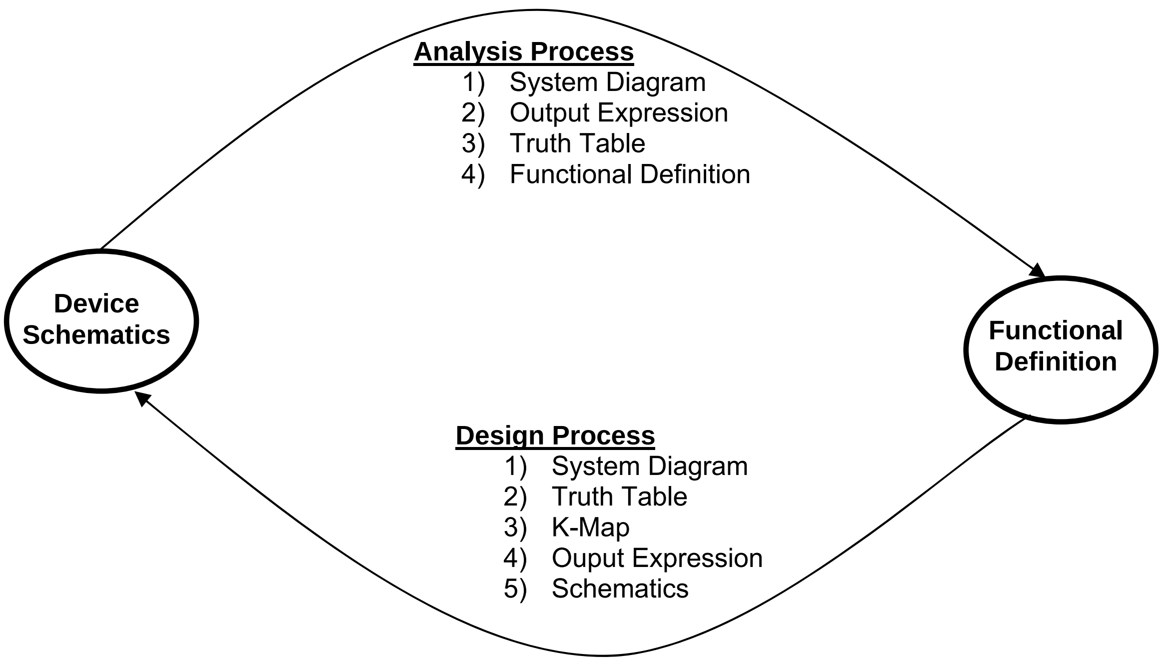


OR

    3) NANDs are available in more variety (#of input) than AND, OR gates

3.4. Combinational Logic Analyis and Design

Below is a summary of design and analysis process steps for a combinational logic device:



3.5. Compressing Truth Tables and K-maps

- The process of compressing truth tables.
  - Truth table compression is done by:
  1) Row compression(use OR function).

A	B	F2	$F2_c$
0	0	0	F2(0,B)=0
0	1	0	
1	0	1	F2(1,B)=
1	1	0	$\overline{B}$

In the above table, we are applying Shannon's Expansion Theorem

$F2' = \overline{B}.F(B=0) + B..F(B=1) = \overline{B}.1 + B.0 = \overline{B}$ which can be compressed to

A	$F2_c$
0	0
1	$\overline{B}$

2) In general, any truth table can be compressed by applying the Shannon's Expansion Theorem with respect to the least significant bit, as shown below:

A	B	C	F3	$F3_c$
0	0	0	0	0
0	0	1	0	
0	1	0	0	C
0	1	1	1	
1	0	0	1	1
1	0	1	1	
1	1	0	1	$\overline{C}$
1	1	1	0	

So we can compress it to

A	B	$F3_c$
0	0	0
0	1	C
1	0	1
1	1	$\overline{C}$

- Example

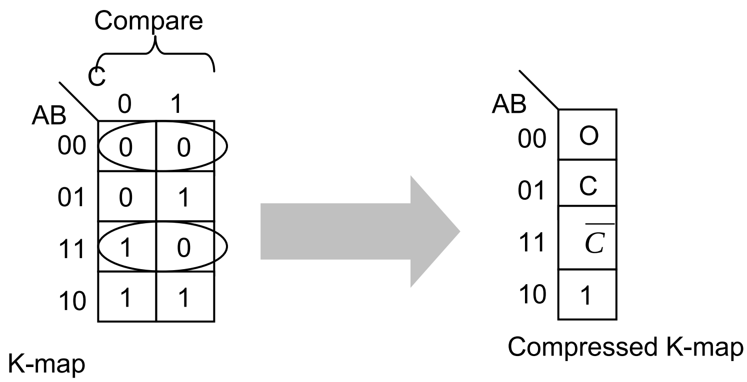
  Compress function F(x,y,z) = Σ(1,3,6,7):

  a) about z

  b) about x

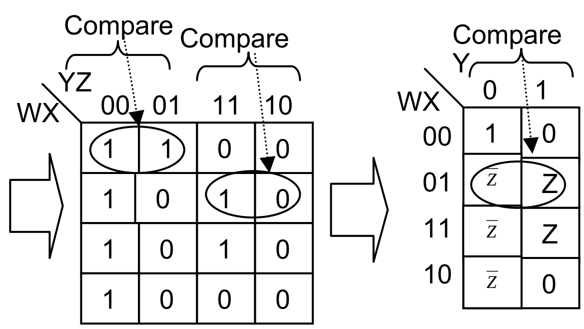
  a) about x, y & z

- The concept of compressing around a variable using Shannon's Expansion Theorem also applies to K-maps. The example below uses C as the reference variable:



- 4-variable (multivariable) expression of K-map Compression.

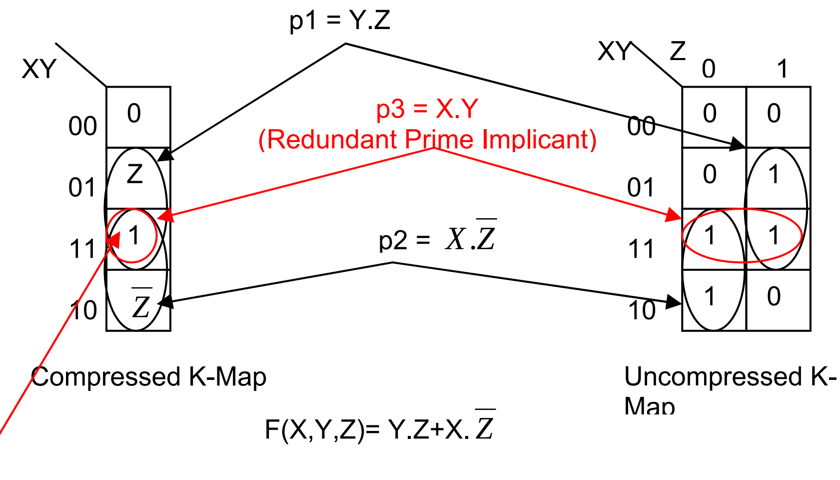
W X Y Z	F
0 0 0 0	1
0 0 0 1	1
0 0 1 0	0
0 0 1 1	0
0 1 0 0	1
0 1 0 1	0
0 1 1 0	0
0 1 1 1	1
1 0 0 0	1
1 0 0 1	0
1 0 1 0	0
1 0 1 1	0
1 1 0 0	1
1 1 0 1	0
1 1 1 0	0
1 1 1 1	1



WX	**F**
0 0	$\overline{Y}$
0 1	$\overline{Y}.\overline{Z}+Y.Z$
1 1	$\overline{Y}.\overline{Z}+Y.Z$
1 0	$\overline{Y}.\overline{Z}$

- Plotting, Filling, and Reducing Compressed K-maps
  - The steps to compress a K-map are:
    (1) Plot the truth table and then compressed K-map.
    (2) Choose cube sizes that result in minimum expressions for the function by covering each map-entered variable separately, treating other map-entered variables as 0s and all 1s as "don't cares".
    (3) Choose cube sizes that result in minimum expressions for the function by covering the 1s in the map that are not complementary-covered.

- Example of plotting, filling and reducing a compressed K-map directly from the compressed K-map and expanding to an uncompressed form.



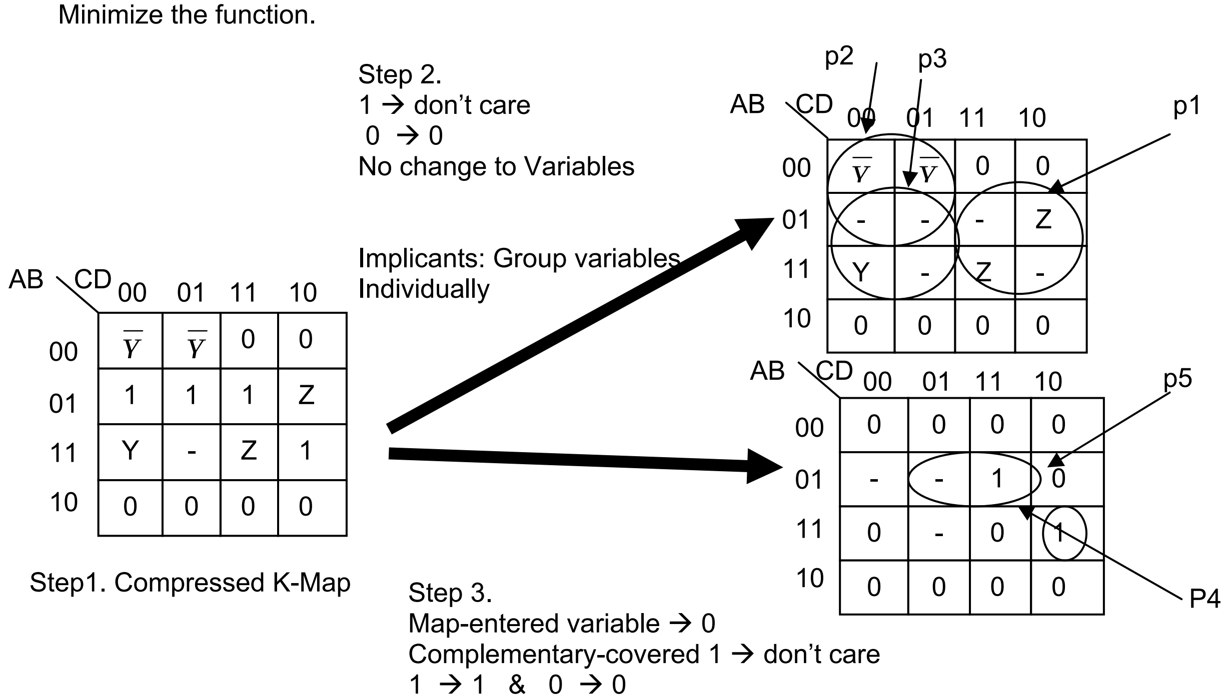
Definition: A complementary covered 1 in a compressed K-map is a 1 that is covered with a map-entered variable and covered again with the complement same map-entered variable.

- A 6-variable example of compressed K-map is described by:

$$F(A,B,C,D,Y,Z)=\sum(0.\overline{Y},1.\overline{Y},4,5,6.Z,7,12.Y,14,15.Z)+\sum md(13)$$

Where m = m(A,B,C,D)

Minimize the function.



F(A,B,C,D,Y,Z) = p1 + p2 + p3 + p4 + p5 = $B.C.Z + \overline{A}.\overline{C}.\overline{Y} + B.\overline{C}.Y + A.B.C.\overline{D} + \overline{A}.B.D$

Although you can use the compressed K-map and apply both steps, it is recommend that you draw both graphs until you are comfortable with the process.

3.6. Glitches and Their Causes

❖ A glitch is a momentary error condition on the output caused by unequal signal paths delays in the circuit. This may appear as an additional pulse high or low that will go away once the circuit reaches a steady state condition (after the signal has propagated through the circuit completely)

- Glitches can occur when a hazard condition exists (Function and Logic Hazards)

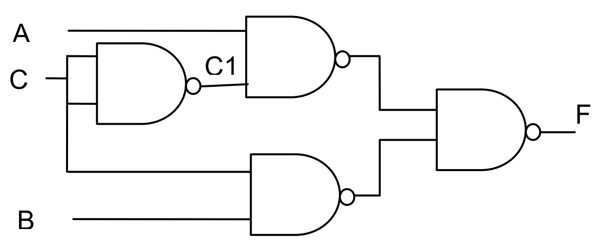
  1) Functional Hazard
     - Exists when there is a problem due to two or more inputs are changing at the same time.
     - May be static when output is not changing or dynamic when the output changes.
     - Cannot be removed by additional circuitry

  2) Logic Hazards
     (1) Exists when there is a problem due to a single input change.
     (2) May be static when output is not changing or dynamic when the output changes.
     (3) Can be removed by additional circuitry

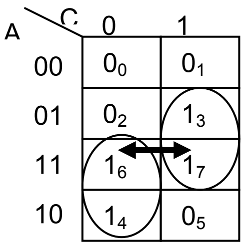
- Example: Use Function F(A,B,C)= $A.\overline{C} + B.C$ to show both function and logic static hazards
  ”Static means before and after the Glitch the output is the same”
  (User DeMorgan’s Theorem to use only NAND Gates)

  Step 1. Generate a K-map and draw a schematic. Both of these drawings will assist in identifying unequal propagation path and, therefore, delays through the circuit.

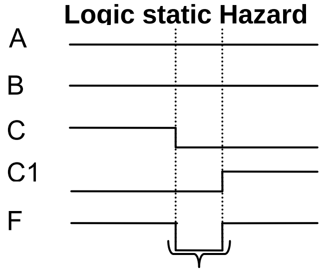
AB \ C	0	1
00	0_0	0_1
01	0_2	1_3
11	1_6	1_7
10	1_4	0_5



  Step 2. Take a look at any logic static hazards that exist and if they may cause a glitch.
  Note: when a single input is changing.

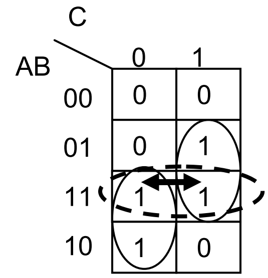


m_7 (111) → m_6 (110)
Logic static Hazard



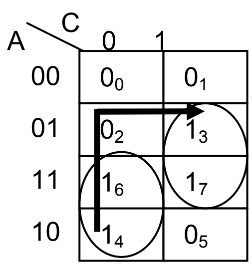
During NAND propagation delay C=C1=0; therefore F=0 during the delay. This is a Logic 0 Glitch.

To remove this logic static hazard, add a term that covers both m6 & m7



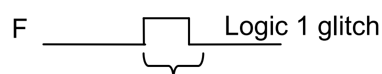
F(A,B,C)= $A.\overline{C} + B.C + A.B$
This is a logic hazard-free function
The process is called the Chain Link Rule.

Step 3. Take a look at any Function Static hazards that exist, and if they may cause a Glitch.
Hint: Look for two or more input changing simultaneously.

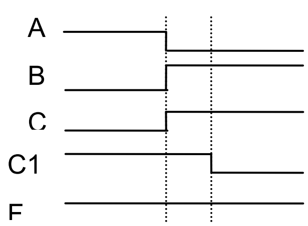


M4(100) → m3 (011)
Function static Hazard

Side Bar
If F=0 → 0 but has a high glitch, then it is called static 0 hazard. For example, m_0 to m_5.



F= 0 → 0 so static 0 hazard



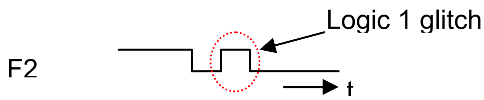
F= 1→ 1
so is a static 1 hazard logic 0 glitch

During NAND Propagation delay C=C1=0 therefore F=0 during the delay which is a Logic 0 Glitch.

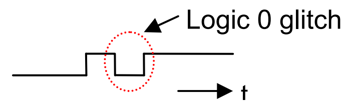
**Note: A system hazard does not have any impact on the functionality if:

- The inputs are not changed to trigger the hazard condition.
  or
- The output will not be used until the input is stabilized.

➢ Dynamic Hazards
A dynamic hazard occurs when an output changes from 0 to 1 or from 1 to 0.
(in static hazard case: output before and after the glitch was the same)



Dynamic 1 to 0 hazard



Dynamic 0 to 1 hazard

- Typically dynamic hazards are produced by multi-level or cascading logic circuits

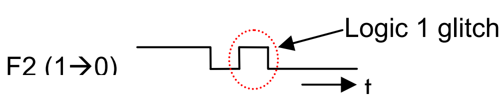
- Example F= $A \oplus B \oplus C$ which means F=1 when odd number of inputs are 1. Identify dynamic hazards in this circuit.

  Step 1. Do the K-map

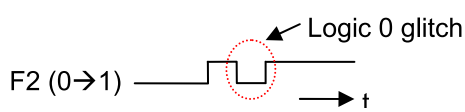
AB \ C	0	1
00	0_0	1_1
01	1_2	0_3
11	0_6	1_7
10	1_4	0_5

Depending on the amount delay through each gate, you may or may not have a dynamic hazard. Typically, simulation software such as B2 Logic , PSpice, and Electronic Work Bench is used to compare maximum and minimum propagation delay for each component to find any dynamic hazards.

If we assume the right amount of delay through the gate, it can be shown that we can cause dynamic hazards in the following paths:



Cell 5 to 2 or Cell 6 to 1
causes Dynamic 1 to 0 hazard



Cell 2 to 5 or Cell 1 to 6
causes Dynamic 0 to 1 hazard

Dynamic hazards are by far the hardest problem to identify. Once the hazards are identified, strategic delays can be implemented to correct the problems.

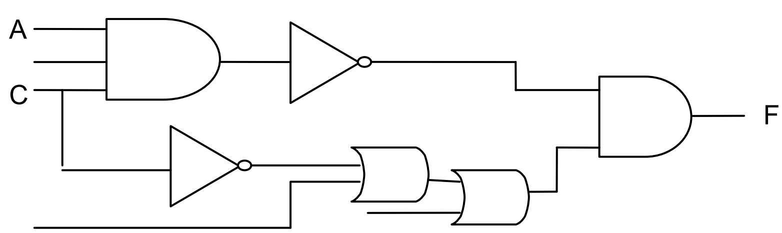
3.7. Types of Functions and Delays

- Trivial Functions, one or zero input
  - GND, Vcc, Buffer, and Inverter
    We use t_p to refer to propagation delay through a gate.

- Simple Functions contain one sum or one product but may be complemented
  - For example:

  - Simple functions may be one or two-level depending on the complements.
  - To find the delay through a gate you need to refer to the part's specification from the manufacturer.
  - For example a 74LS02 (NR gate) for R_L = 2Kohms and C_L = 15 pF:
    (a) Maximum propagation delay time Low to High Level output t_{PLH} = 13 ns
    (b) Maximum propagation delay time High to Low Level output t_{PHL} = 10 ns
    (c) Some times we use the average worst case propagation delay time
    t_{su}=(t_{PLH} + t_{PHL})/2
    (d) To find the maximum delay through a cascading circuit, t_p of each component in the path must be added to find the total worst case delay through the circuit.

- Complex Functions contain multiple levels of sums and/or products
  - The delay for each path needs to be calculated by adding the propagation delay, t_p, for each component in the path. For example



If the t_p is the time delay for each component then:
* delay from input A to output F is 3*t_p.
* delay from input C to output F is 4*t_p.

  - For a multi-output complex function, at times there is an opportunity to share product terms among the output's to lower Literal Count (LC) and, therefore, reducing the number of gates.

- Example: Given the following K-maps for F1 and F2 outputs of a three input system, find the optimum design:

AB \ C	0	1
00	1_0	1_1
01	1_2	0_3
11	1_6	0_7
10	1_4	0_5

F$_1$(A,B,C) = $\overline{A}.\overline{B}.C + \overline{C}$

AB \ C	0	1
00	0_0	1_1
01	1_2	0_3
11	1_6	0_7
10	0_4	0_5

F$_2$(A,B,C) = $\overline{A}.\overline{B}.C + B.\overline{C}$

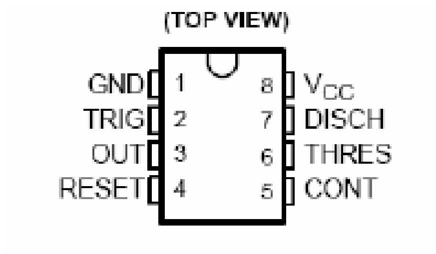
A smaller F_1 could have been written, but the fact that m_1 can be shared between F_1 and F_2, results in a more minimized total solution (F_1 and F_2).

- Commonly-used complex functions have been implemented as ICs, and are crucial to the ability to develop complex functionality. Using individual gates consumes too much PC board space and wiring to be realistic.

  The next section introduces some of the most common complex function available on the market.

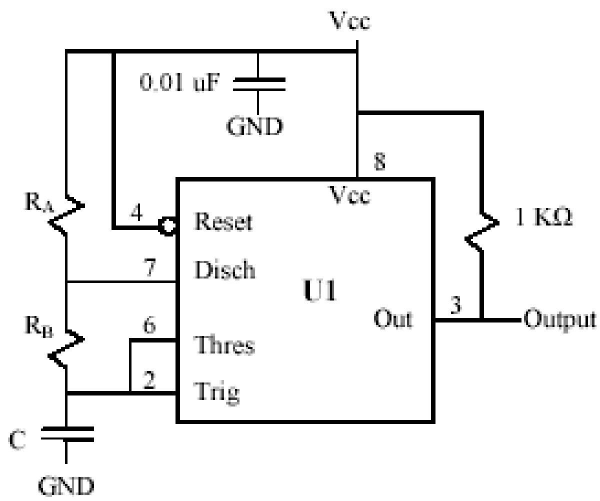
3.8. Beyond Standard Logic: Applications

- Percision Timers "555"
  This device is a precision timer that may be configured for a variety of applications. The most common use of the NE 555 is an application to generate square waves that may be used as a clock signal in digital design. In order for NE 555 to generate the clock signal, it may be configured as shown below:



Device #	Pin#	Value
U1	1	GND
U1	3	Out
U1	8	Vcc
U1	5	Open

$t_H = 0.693\ (R_A + R_B)\ C$
$t_L = 0.693\ (R_B)\ C$
where
$R_A = 1\ M\Omega$
$R_B = 1\ M\Omega$
C = 1 UF



- Example
  Given the component values in the introduction of NE 555 (above), draw the clock (square wave) signal generated and determine the period, frequency and duty cycle for the signal.

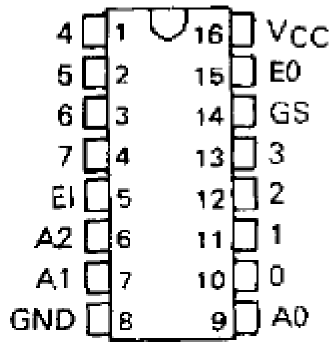
  Solution:

- Example
  Using 555 timer, design a circuit that generates a clock signal with frequency of 2.5 Khz and 75% duty cycle. Show your work including component value calculations, timing diagram and resulting schematics.

  Solution:

- Encoders
  Function: 2^n input → n output
  Example: 74LS148 "8 to 3 encoder"
  Note: You can design these circuits using the 6-step design, K-map, and SSI gates.

'148, 'LS148



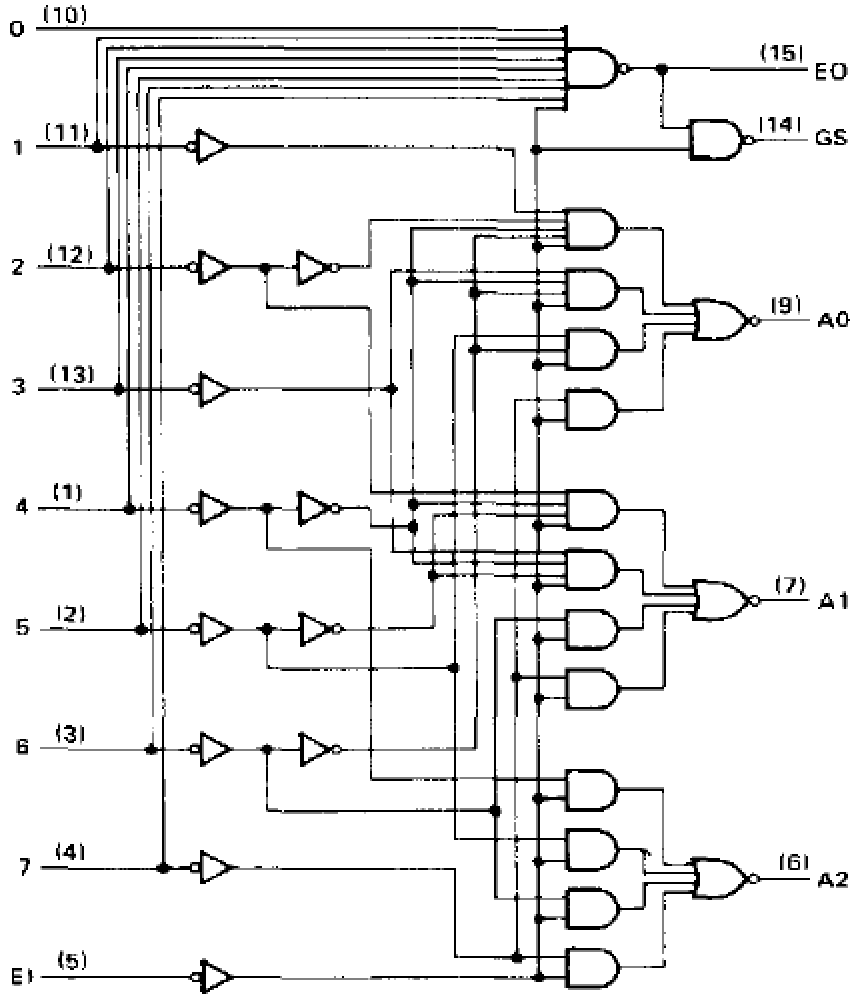
'148, 'LS148
FUNCTION TABLE

INPUTS									OUTPUTS				
EI	0	1	2	3	4	5	6	7	A2	A1	A0	GS	EO
H	X	X	X	X	X	X	X	X	H	H	H	H	H
L	H	H	H	H	H	H	H	H	H	H	H	H	L
L	X	X	X	X	X	X	X	L	L	L	L	L	H
L	X	X	X	X	X	X	L	H	L	L	H	L	H
L	X	X	X	X	X	L	H	H	L	H	L	L	H
L	X	X	X	X	L	H	H	H	L	H	H	L	H
L	X	X	X	L	H	H	H	H	H	L	L	L	H
L	X	X	L	H	H	H	H	H	H	L	H	L	H
L	X	L	H	H	H	H	H	H	H	H	L	L	H
L	L	H	H	H	H	H	H	H	H	H	H	L	H

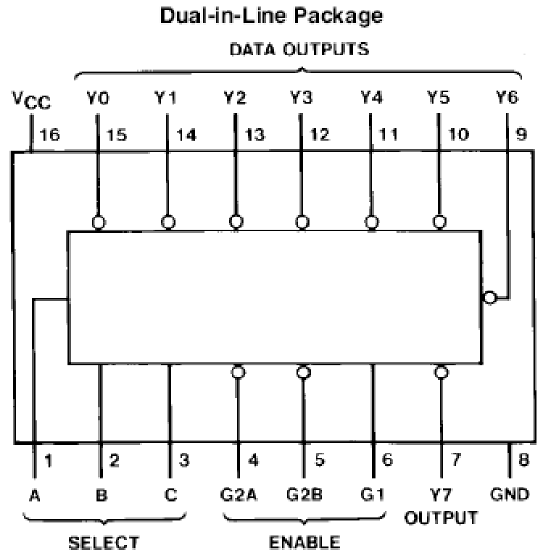
Notes:

- EI is the output-enable input and should be set to "L" for normal operation.
- Remember that:
  "L" is the same as "0", Gnd or 0 volts.
  "H" is the same as "1", Vcc or +5 volts.
  "X" means don't care, it can be high or low.

'148, 'LS148



- Decoder, also called minterm generator
  Function: n input → 2^n output
  Example: 74LS138 "3 to 8 Decoder/Demux"



Function Tables

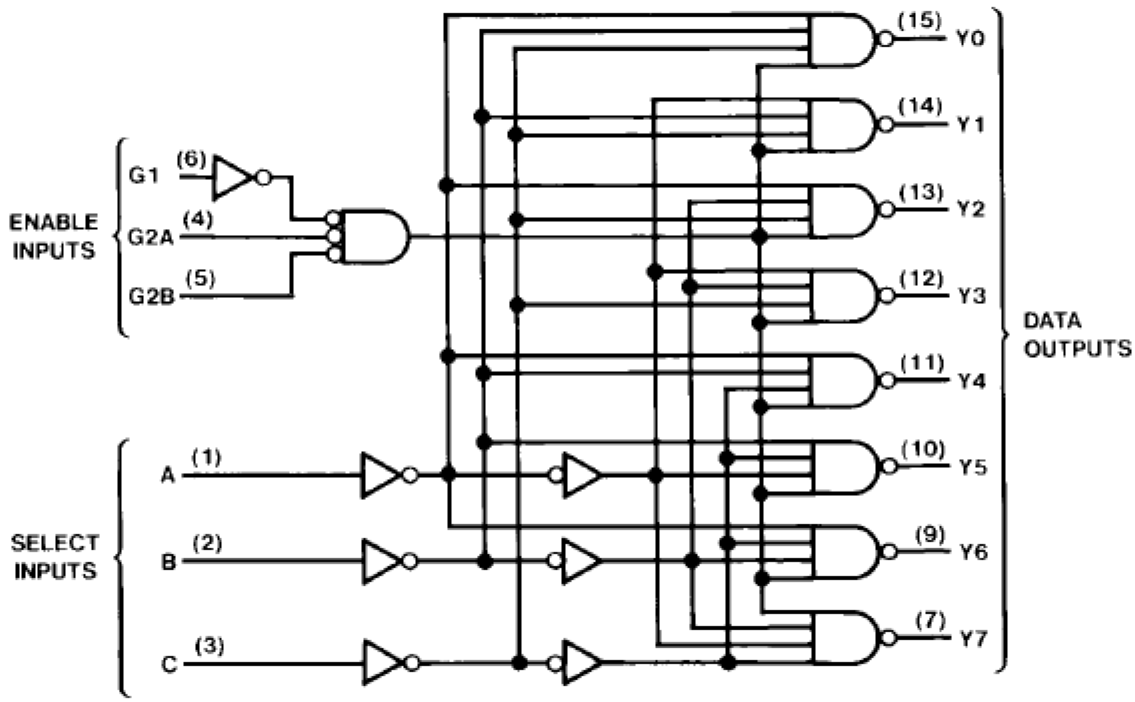
LS138

Inputs					Outputs							
Enable		Select										
G1	G2*	C	B	A	Y0	Y1	Y2	Y3	Y4	Y5	Y6	Y7
X	H	X	X	X	H	H	H	H	H	H	H	H
L	X	X	X	X	H	H	H	H	H	H	H	H
H	L	L	L	L	L	H	H	H	H	H	H	H
H	L	L	L	H	H	L	H	H	H	H	H	H
H	L	L	H	L	H	H	L	H	H	H	H	H
H	L	L	H	H	H	H	H	L	H	H	H	H
H	L	H	L	L	H	H	H	H	L	H	H	H
H	L	H	L	H	H	H	H	H	H	L	H	H
H	L	H	H	L	H	H	H	H	H	H	L	H
H	L	H	H	H	H	H	H	H	H	H	H	L

* G2 = G2A + G2B

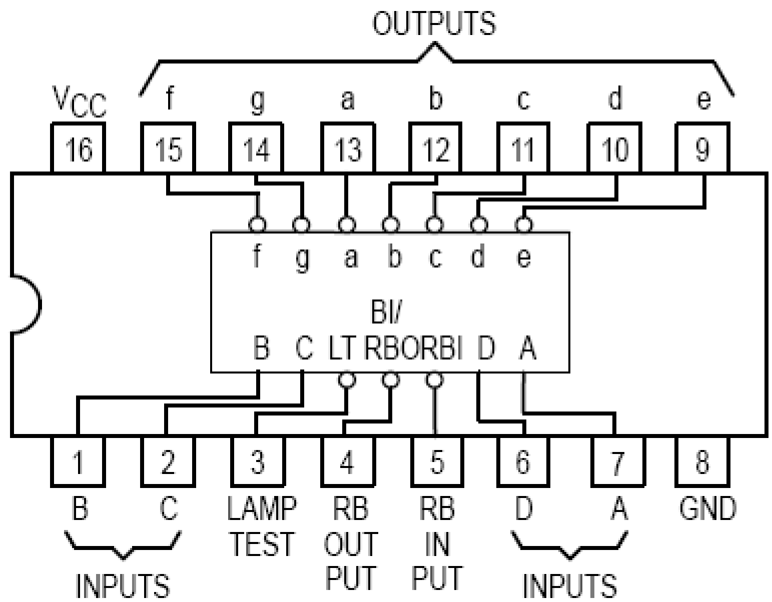
H = High Level, L = Low Level, X = Don't Care

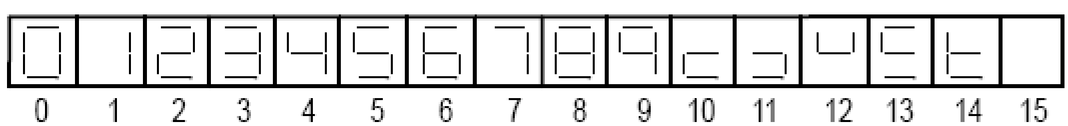
LS138


ENABLE INPUTS
G1
(6)
G2A
(4)
G2B
(5)
SELECT INPUTS
A
(1)
B
(2)
C
(3)
(15)
Y0
(14)
Y1
(13)
Y2
(12)
Y3
(11)
Y4
(10)
Y5
(9)
Y6
(7)
Y7
DATA OUTPUTS

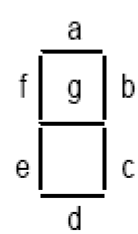
❖ BCD to 7-segment display driver
Function: Binary Coded Decimal Digits (4 inputs "0-9") → 7 output "one per segment"
Example:74LS47

74LS47
(TOP VIEW)





NUMERICAL DESIGNATIONS AND RESULTANT DISPLAYS



SEGMENT IDENTIFICATION

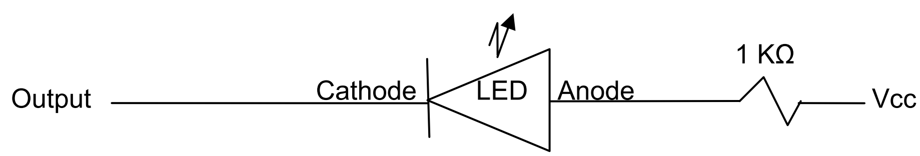
74LS47

FUNCTION TABLE

DECIMAL OR FUNCTION	INPUTS						BI/RBO†	OUTPUTS						
	LT	RBI	D	C	B	A		a	b	c	d	e	f	g
0	H	H	L	L	L	L	H	ON	ON	ON	ON	ON	ON	OFF
1	H	X	L	L	L	H	H	OFF	ON	ON	OFF	OFF	OFF	OFF
2	H	X	L	L	H	L	H	ON	ON	OFF	ON	ON	OFF	ON
3	H	X	L	L	H	H	H	ON	ON	ON	ON	OFF	OFF	ON
4	H	X	L	H	L	L	H	OFF	ON	ON	OFF	OFF	ON	ON
5	H	X	L	H	L	H	H	ON	OFF	ON	ON	OFF	ON	ON
6	H	X	L	H	H	L	H	ON	OFF	ON	ON	ON	ON	ON
7	H	X	L	H	H	H	H	ON	ON	ON	OFF	OFF	OFF	OFF
8	H	X	H	L	L	L	H	ON	ON	ON	ON	ON	ON	ON
9	H	X	H	L	L	H	H	ON	ON	ON	ON	OFF	ON	ON
10	H	X	H	L	H	L	H	OFF	OFF	OFF	ON	ON	OFF	ON
11	H	X	H	L	H	H	H	OFF	OFF	ON	ON	OFF	OFF	ON
12	H	X	H	H	L	L	H	OFF	ON	OFF	OFF	OFF	ON	ON
13	H	X	H	H	L	H	H	ON	OFF	OFF	ON	OFF	ON	ON
14	H	X	H	H	H	L	H	OFF	OFF	OFF	ON	ON	ON	ON
15	H	X	H	H	H	H	H	OFF	OFF	OFF	OFF	OFF	OFF	OFF
BI	X	X	X	X	X	X	L	OFF	OFF	OFF	OFF	OFF	OFF	OFF
RBI	H	L	L	L	L	L	L	OFF	OFF	OFF	OFF	OFF	OFF	OFF
LT	L	X	X	X	X	X	H	ON	ON	ON	ON	ON	ON	ON

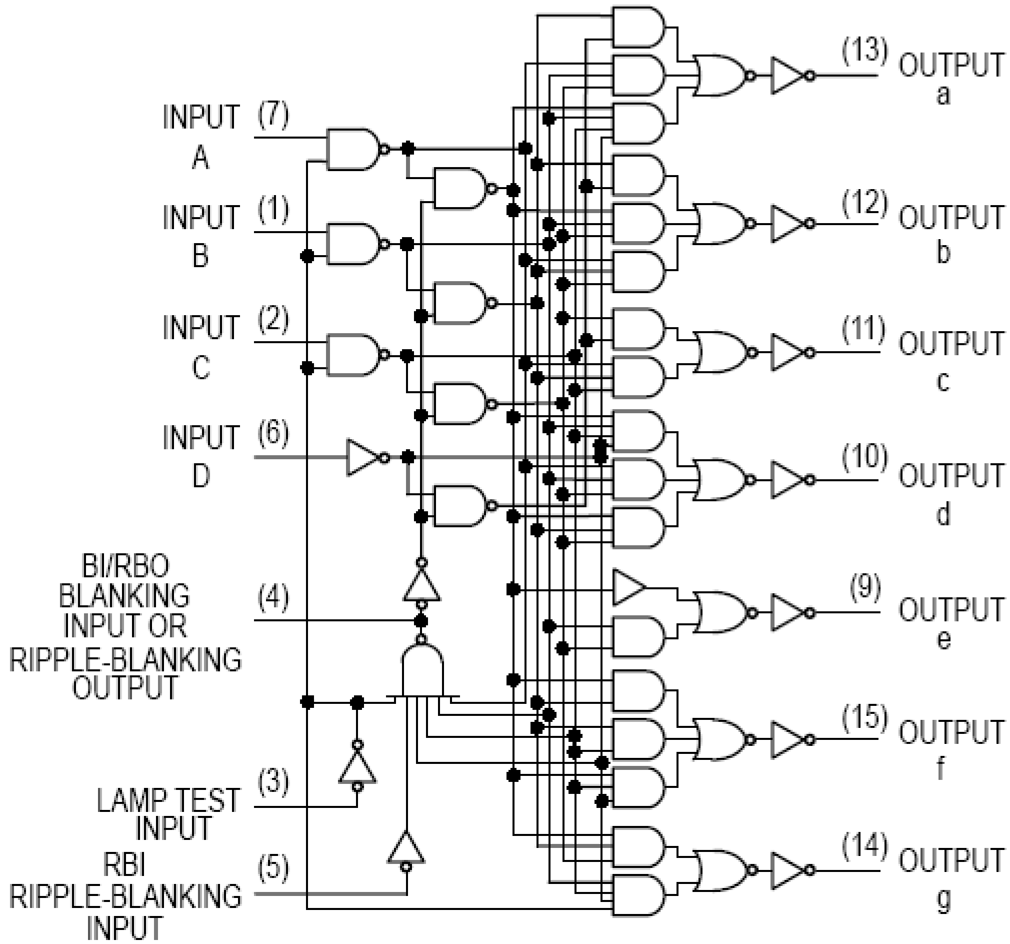
Note: The output is active low and open collector:

- H, Inactive and OFF are same
- Active, L and ON are the same.
- In order to light up LEDs, each LED in the 7-segment display should be wired as shown below.



The 1 KΩ resistors limits the current through the LED to 5 mA.

74LS47


INPUT (7) A
INPUT (1) B
INPUT (2) C
INPUT (6) D
BI/RBO BLANKING INPUT OR RIPPLE-BLANKING OUTPUT (4)
LAMP TEST INPUT (3)
RBI RIPPLE-BLANKING INPUT (5)
(13) OUTPUT a
(12) OUTPUT b
(11) OUTPUT c
(10) OUTPUT d
(9) OUTPUT e
(15) OUTPUT f
(14) OUTPUT g

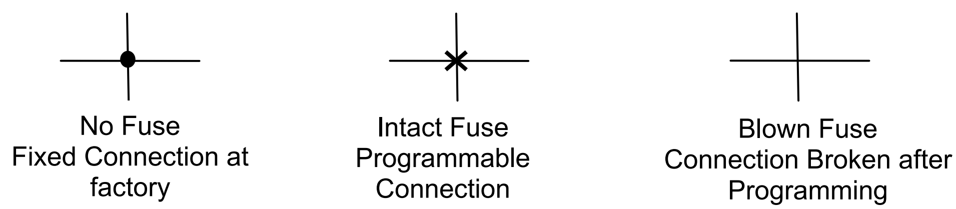
3.9. Programmable Logic Devices (PLDs)

PLDs allow a designer to implement his/her design on a single chip. The main advantages of PLDs are speed of implementation, ease of implementation and low overall cost at low quantities. Most projects use PLDs during the design phase because of the stated reasons. During production when the quantities are higher, the design is typically implemented with one-time factory-programmed devices that are able to implement the design.

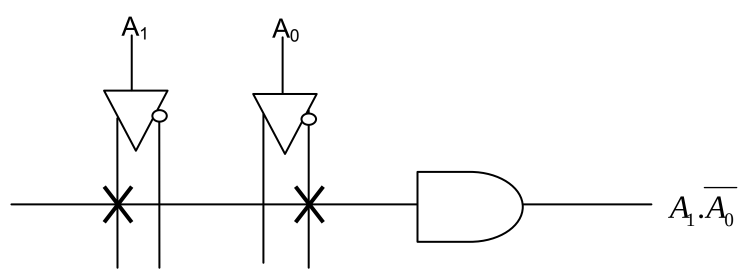
A summary of PLD's is shown below – the size in-terms of input/output and function, is growing on a daily basis. Most designs will prototype using one of these devices until they have enough quantity to justify a custom chip.

Device Type	AND Array Connection	OR Array Connections
PROM -Programmable Read Only Memory	Fixed at the Factory	Customer programmable with Fuses
PLA - Programmable Logic Array	Customer programmable with Fuses	Customer programmable with Fuses
PAL – Programmable Array Logic, also called GAL - Generic Array Logic	Customer programmable with Fuses	Fixed at the Factory

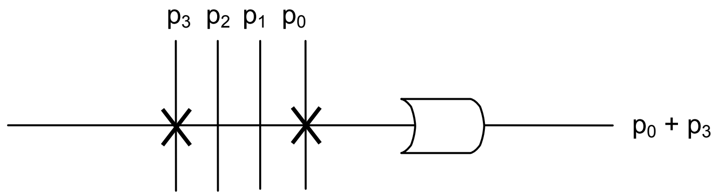
- Introducing Key Symbols used in PLD Design
  - Fuse Types Symbols



(1) Product Terms (Example)
  (a) The output has a pull-up resistor that is not shown, and if all fuses are blown then the output will be H.
  (b) If all fuses are intact, then they may place an X in the And symbol.



(2) Sum Terms (Example)
  (a) The output has a pull-down resistor (not shown), and if all fuses are blown, then the output will be L.
  (b) If all fuses are intact, then they may place an X in the OR Symbol.



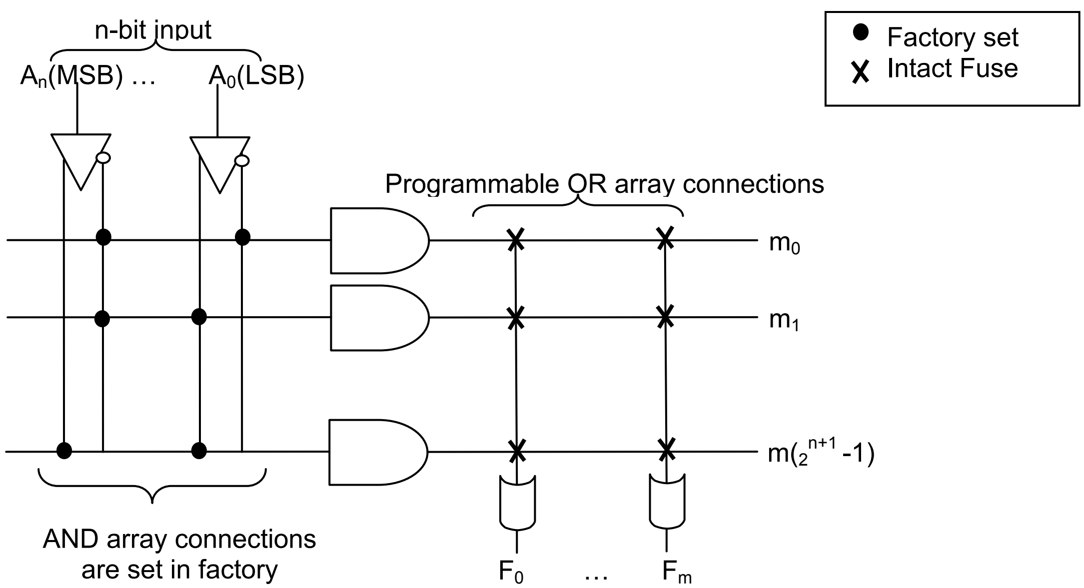
(3) Erase ability
Some devices allow the blown fuses to be re-fused by:
(a) Ultraviolet Light (through a small window on the top)
(b) Electrically (typically at much higher voltage and current than normal operation) "This type is referred to as EE-type."

There are also universal programming units available that allow for fuse map input/output which adheres to the Joint Electronic Device Engineering Council (JEDEC) Standard.

- Programmable Read Only Memory (PROM)
  A typical PROM may have 16 inputs with 2^{16} outputs. This device would be called a 64K PROM. PROM is typically used during development and once the product is in production, Read Only Memory(ROM) will be used.

  ROM is a one-time factory-only programmable device which is the reason why it is called Read Only Memory. The initial ROM set up is costly but the cost per part is significantly lower than that of PROM.

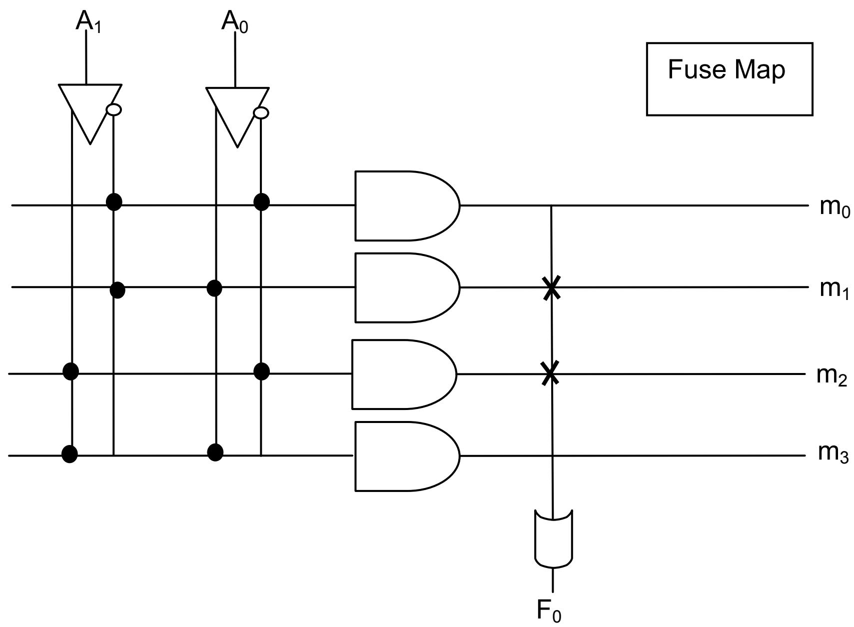
  The following diagram shows the PROM internal diagram:



- PROM Example: Implement F(A_0,A_1)= $\overline{A_0}.A_1 + A_0.\overline{A_1}$

  Since all the minterms are available, all we have to do to get the desired output is to OR the appropriate minterms and then blow fuses for all the minterms not needed for the function.

$F(A_0,A_1)=\sum(1,2)$, which means fuses 0 and 3 need to be blown.



- Example

  Draw the fuse map for the smallest PROM that implements the function:

  f(A,B,C) = A'.B.C + A.B'.C' + AB

  Note: Only draw the minterms that are used

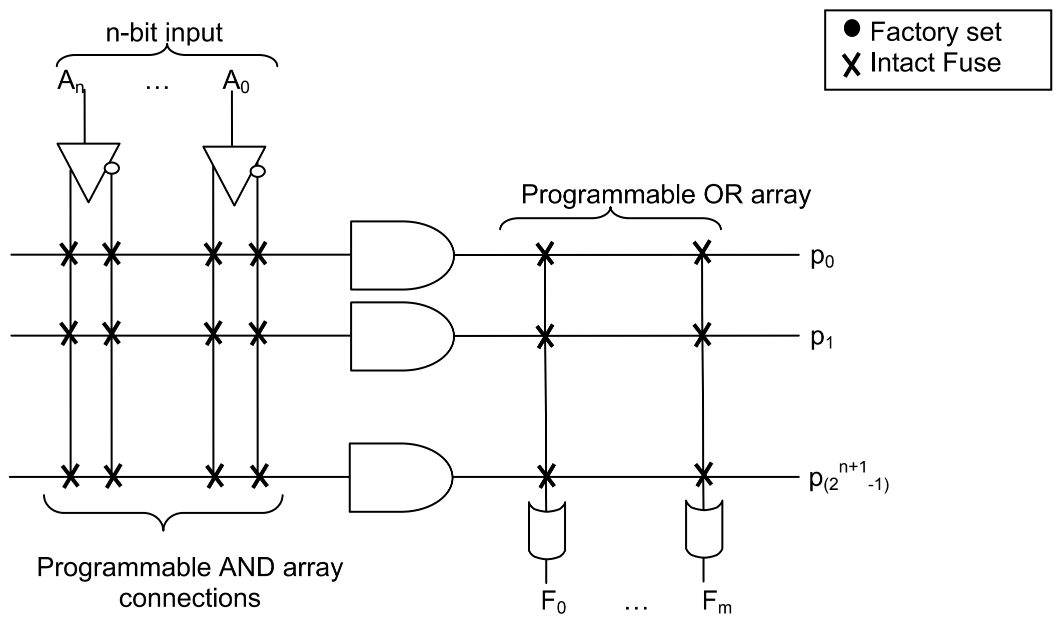
  Solution:

- Example

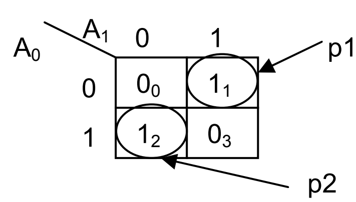
  Use a PROM to design a 4-byte memory that contain 10,50,90 and 20 in location 0 to 3.
  Show the system diagram and PROM fuse map.
  Note: Only draw the minterms that are used

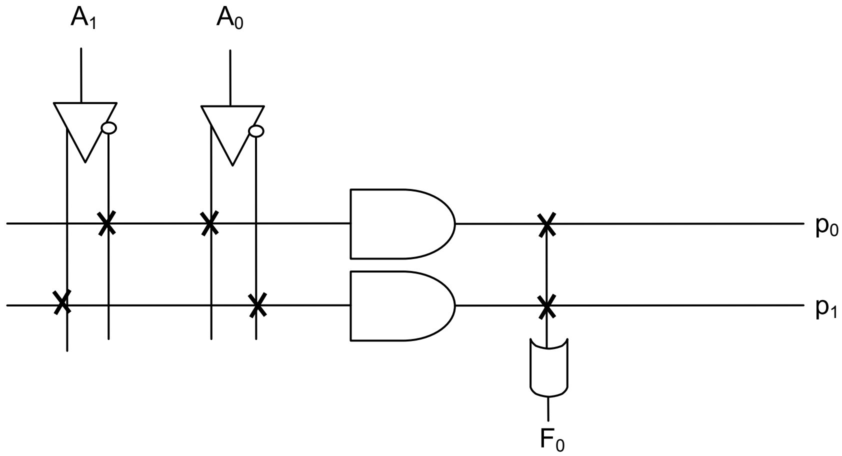
  Solution:

- Programmable Logic Arrays (PLA)
  PLAs are similar to PROMs with the added flexibility of programmable AND array connections. PLAs are the most flexible of the Program Logic Devices since both the AND and OR array connections are field programmable. The drawbacks of the PLA technology are that they have the highest cost per unit and the longest propagation delays.



- PLA Example: Implement $F(A_0,A_1)=\overline{A_0}.A_1 + A_0.\overline{A_1}$
  PLAs allow both the AND and OR array connections to be programmed. The complete fuse map shows both the AND and the OR fuses. There may be opportunities to share product terms between the outputs to improve efficiency.



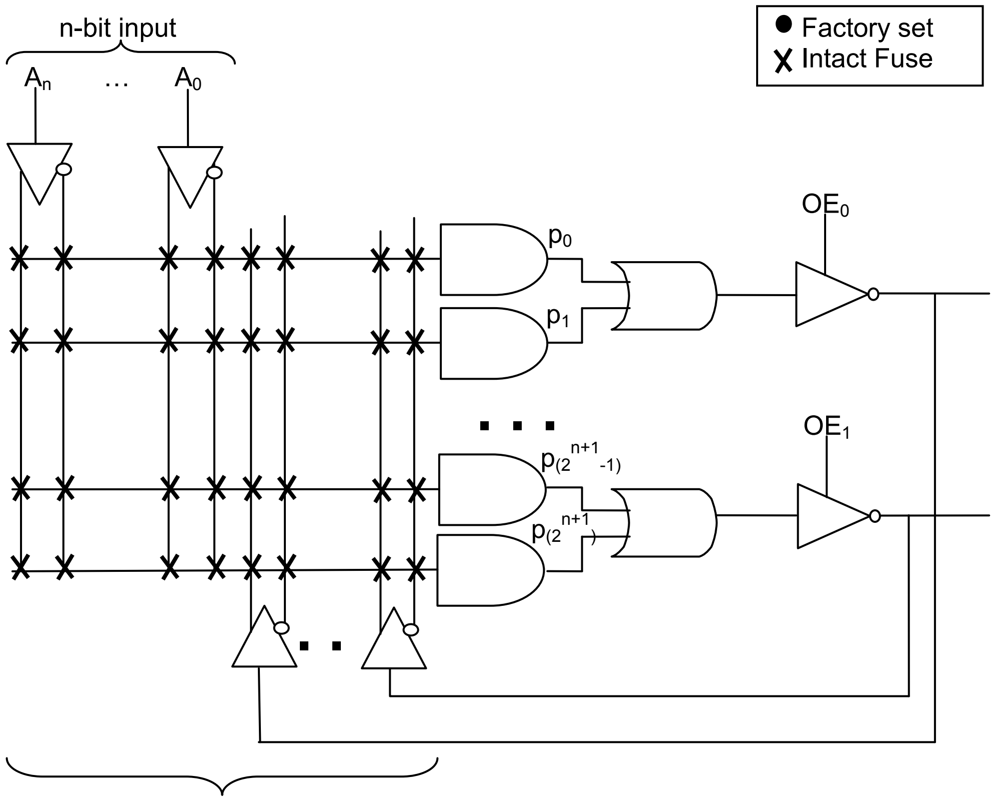


➢ Programmable Array Logic (PAL) or Generic Array Logic (GAL)
PAL and GAL are two different way to refer to this technology. PALs are:

- Easiest to use, since only the AND array connection is programmable.
- Best suited for non-standard complex combination logic implementation.
- Available in variety of sizes. The larger versions are called Field Programmable Gate Array (FPGA). The version that can be configured at the factory for larger volume design is called Gate Arrays.
- Able to implement feeding back the outputs to and array for improved functionality.
- Available with 3-state outputs (1, 0, high impendence) which are controlled by input, OE.

(1) OE=0 → output = open (which can be used to drive the pin as an input which is why sometime 3-state pins are referred to as I/O)

(2) OE=1 → output = $\overline{input}$

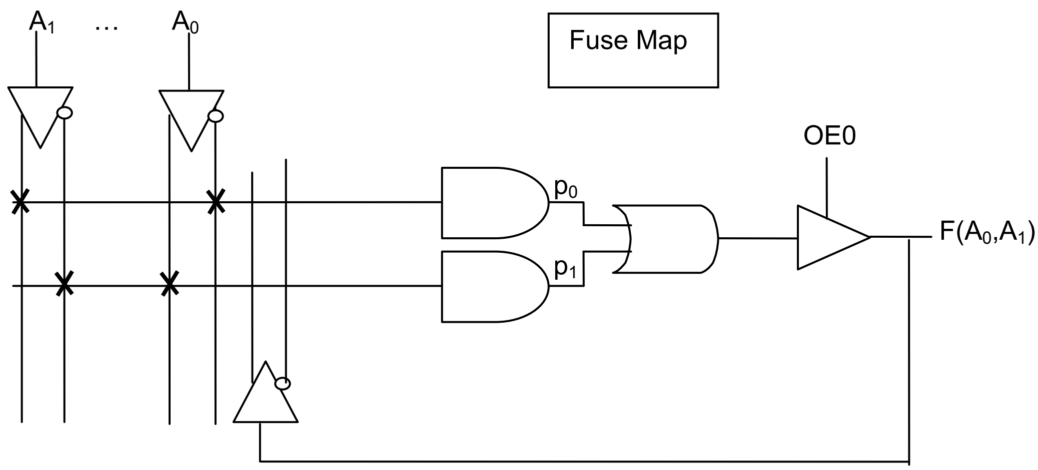


- PALs or GALs are named based on the number of inputs and outputs (PALxxyzz) where:
  - "xx" is the number of maximum AND array inputs
  - "y" represent the type of output
    - Combination output: H is active High, L is active low, P is programmable
    - Registered outputs: R is registered (Contains memory devices), RP is registered with programmable polarity.
    - Versatile: V indicates programmable output macro-cells which can be configured to be either combinational or registered.
  - "zz" represents the maximum number of dedicated outputs.

  For example: PAL16L8 is active-Low output with 16 inputs and 8 outputs.

- PALs and GALs are programmed using Universal programmers which use JEDEC fuse map file format. Some of the other names commonly used to refer to these devices based on their complexity or size are:
  - Simple Programmable Logic Devices (SPLD)
  - Complex Programmable Logic Devices (CPLD)

- PAL usage example: Implement $F(A_0,A_1)=\overline{A_0}.A_1+A_0.\overline{A_1}$
  Here again only one Fuse map is needed (AND array connection). Also need to consider if there are any terms that can be shared. In this case we do not have any shared terms.



- Benefits of PLDs over individual gates:
  - Shorter design time (rapid prototype).
  - Allow for rapid design changes.
  - Decreased PC board real estate.
  - Improved reliability since they require fewer packages and interconnections.

❖ Signal Polarity Convention
There are two types of convention: Positive Logic Convention (PLC) and Direct Polarity Indication (DPI). It is recommended that polarity be consistent unless there is a practical reason to change polarity.

- Positive Logic Convention (PLC)
  - Other name for PLC: True Form.
  - PLC has been used so far and we will continue to use it in the rest of the text. It is best suited for working with logic (ones and zeros).

- Direct Polarity Indication (DPI)
  - Other name for DPI: Complemented Form.
  - DPI is also referred to as the mixed signal since each signal can have polarity attached to it For example W(H) is W in positive logic which is the same as $\overline{W}(L)$
  - DPI is preferred by engineers who need to be aware of voltage levels correlating with the active levels
  - Here is a comparison of DPI and PLC signal naming

PLC Signal Name	DPI signal Name	Type of Signal
A	A(H) or $\overline{A}(L)$	Active High
$\overline{A}$	$\overline{A}(H)$ or A(L)	Active Low
Use bubble "o" to indicate negation	Use wedge ⊿ to indicate polarity (Low)	
	Write signal name with no over bar and change to Active low with polarity indicator.	

(1) Double complementation can be used to write equivalent signal names in a number of formats

(a) $A = A(H) = \overline{A(L)} = \overline{A}(\overline{H}) = \overline{A}(L)$

(b) $\overline{A} = \overline{\overline{A}(H)} = \overline{\overline{A}}(\overline{H}) = A(\overline{H}) = A(L)$

(2) To fully specify a circuit, you need to provide a Signal List (SL) in addition to function. If the SL is not provided, then it will be assumed to be positive logic.

(a) PLC → F=A.B + C.D; ***SL***: $F, A, \overline{B}, \overline{C}, D$

(b) DPI → F=A.B + C.D; ***SL***: F(H), A(H), B(L), C(L), D(H)

3.10. Additional Resources

- Wakerly, I. Digital Design. (2006) Prentice Hall
  Chapter 6 “Combinational Logic Design Practices”

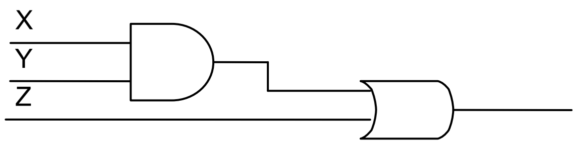
3.11. Problems

1. For each value of N (when N=2, 3, 4 or 5):

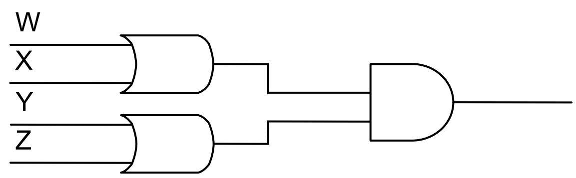
   a) Determine number of unique functions possible with N independent variables?
   b) If the function and its complement are considered as only one function, determine number of unique functions possible with N independent variables?

2. Analyze the logic circuits shown below in order to obtain the logic function for each circuit.

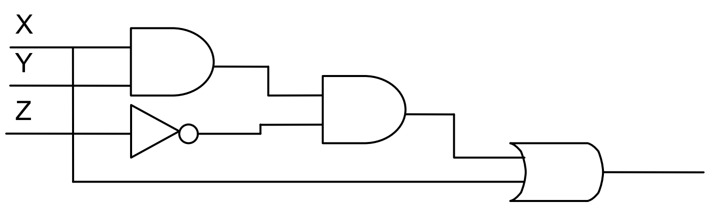
   a)



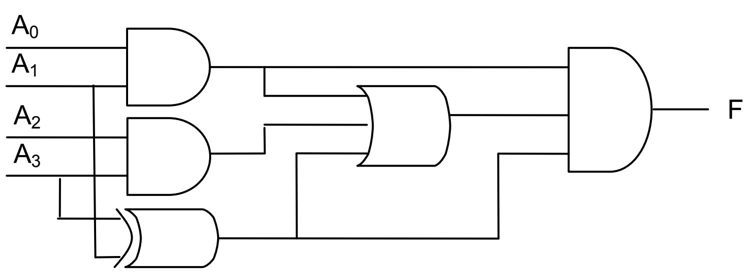
   b)



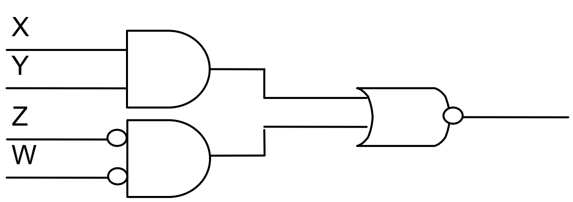
   c)



3. Write the minimized logic function implemented by the following circuit:



4. Draw the following circuit using only NAND gates.



5. Obtain minimum SOP expression for the following function using a compressed K-map.

$$F(X,Y,Z)=\overline{X}+X.Y.Z \quad where \quad m=m(X,Y)$$

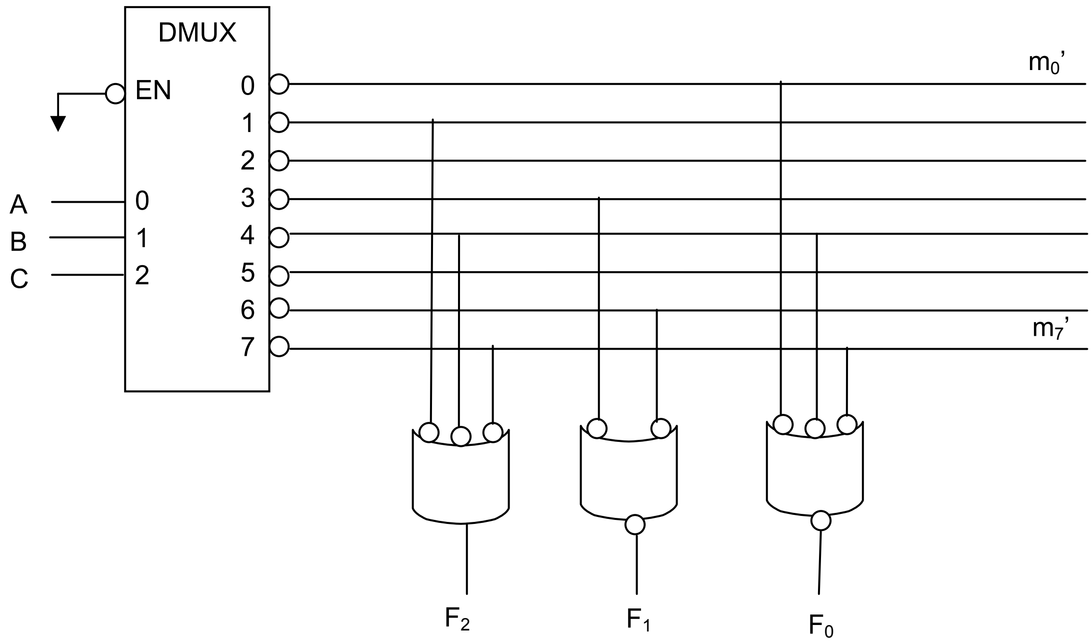
6. The function F1 specified by the following truth table was modified three times. Find a minimum SOP expression for each of the functions F2 through F4. (A dash in the table represents a “don’t care” output.)

A	B	C	F1	F2	F3	F4
0	0	0	0	0	0	0
0	0	1	1	D’	1	1
0	1	0	0	0	0	0
0	1	1	1	1	D’.E’	1
1	0	0	-	-	-	-
1	0	1	-	-	-	-
1	1	0	0	0	0	0
1	1	1	1	1	1	D’+E’

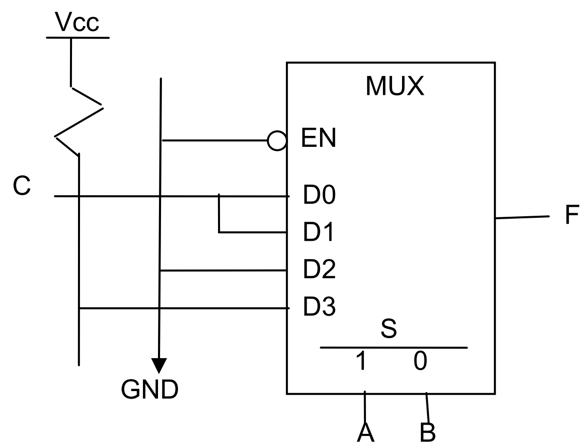
7. For the following K-map, show a reduced K-map one size smaller. Write the minimum SOP expression for the normal and reduced map.

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

8. Analyze the following design to obtain the output Boolean functions in compact min term form.



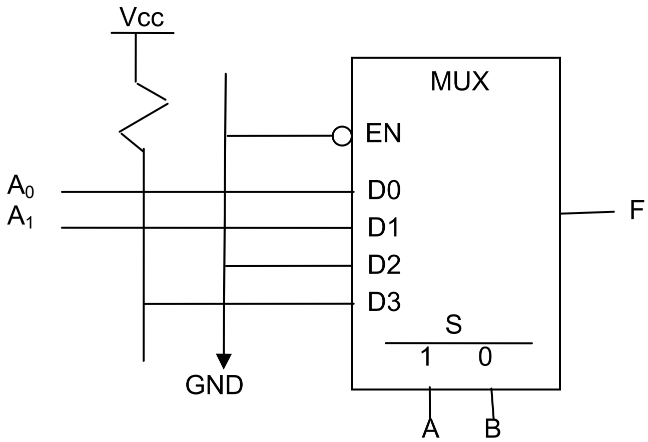
9. Analyze the multiplexer circuit shown below to obtain its output function, f(A,B,C), in compact minterm form.



MUX Functional Truth Table:

A	B	F
0	0	D0
0	1	D1
1	0	D2
1	1	D3

10. Analyze the multiplexer circuit shown below to obtain its output function, F, in compact minterm form.



MUX Functional Truth Table:

A	B	F
0	0	D0
0	1	D1
1	0	D2
1	1	D3

11. Obtain the simplest collective set (lowest number of literals) of SOP expressions for the 0s of the following functions:

$F_1(A,B,C) = \Sigma(1,2,4,6)$
$F_2(A,B,C) = \Sigma(0,1,2,6,7)$
$F_3(A,B,C) = \Sigma(1,2,6)$

Note: In addition to minimizing each function, also look for shared terms between the functions.

12. Obtain the simplest collective set (lowest number of literals) of POS expressions for the following functions:

$F_1(A,B,C,D) = \Sigma(1,2,3,4,6,7)$
$F_2(A,B,C,D) = \Sigma(0,1,2,3,4,6,7,14,15)$
$F_3(A,B,C,D) = \Sigma(1,2,6, 14, 15)$

Note: In addition to minimizing each function, also look for shared terms between the functions.

13. Show fuse map of a PLA that outputs number of 1's in a 4-bit input.

14. Show the fuse map for an XS3-to-BCD code (see table for definition) converter with inputs I_3, I_2, I_1, I_0 and outputs F_3, F_2, F_1, F_0 using a simple PROM with 4 inputs and 8 outputs. (The min-terms not listed are 0 for all functions.)

XS3 Code				**XS3**	**BCD Code**			
I_3	I_2	I_1	I_0	**minterm #**	F_3	F_2	F_1	F_0
0	0	1	1	3	0	0	0	0
0	1	0	0	4	0	0	0	1
0	1	0	1	5	0	0	1	0
0	1	1	0	6	0	0	1	1
0	1	1	1	7	0	1	0	0
1	0	0	0	8	0	1	0	1
1	0	0	1	9	0	1	1	0
1	0	1	0	10	0	1	1	1
1	0	1	1	11	1	0	0	0
1	1	0	0	12	1	0	0	1

15 . Show the fuse map for an 8 input lock that unlocks (output is 1) only when input are set to "10100100", "00100101" or "11100111". Implement the lock control using:

a) PROM
b) PLA

Chapter 4. Introduction to Feedback Circuits and Sequential Logic Analysis

This is the first section in Sequential Logic Design and Analysis.

4.1. Key concepts and Overview

- SR flip-flops
- Asynchronous Sequential Logic Issues
- Finite State Machines (Sequential Logic Circuits)
- Additional Flip-Flop Circuits
- Sequential Circuit Analysis
- Debouncing Switches
- Additional Resources
- Problems

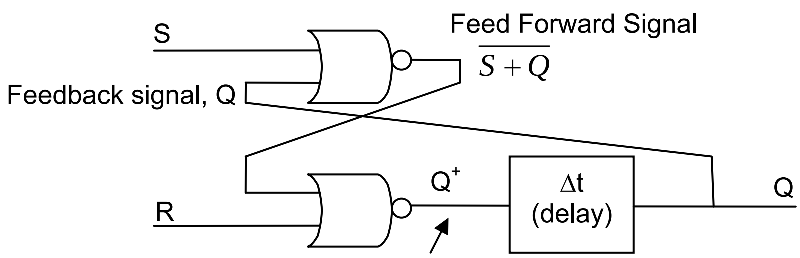
4.2. SR Flip-Flops

SR flip-flops are the simplest form of single-bit memory (latch). A Latch is also known as a bi-stable memory device. Flip-flop is used to store value of input.

- SR flip-flops have set and reset inputs. The circuit for a SR flip-flop is shown below:
  - Q^+ is used to refer to the next output state when the current output state is Q
  - Unlike combinational logic, sequential logic Q^+ is dependent on the value of previous output Q.

    $Q^+(S, R, Q) = \overline{\overline{S+Q}+R}$

    Note: Q^+ is dependent on inputs S and R as well as current state, q.



$$Q^+ = \overline{\overline{S+Q}+R}$$

"Characteristic equation"

- Present-State/Next-State (PS/NS) Table
  The PS/NS table is the sequential circuit equivalent to combinational circuit's truth table. As it can be seen from the following example, instead of output, PS/NS table has current and next state:

Present State(PS) Q	External Input Signal S	R	Next State (NS) Q^+	Comment	Additional Comment
0	0	0	0	hold 0	Q is stable , Q_{new} = Q
0	0	1	0	hold 0	Q is stable , Q_{new} = Q
0	1	0	1	set	Q is unstable, Q_{new} = Q+ after Δt = $\overline{Q}$
0	1	1	0	normally not allowed	Q is stable , Q_{new} = Q, reset dominant
1	0	0	1	hold 1	Q is stable , Q_{new} = Q
1	0	1	0	reset	Q is unstable, Q_{new} = Q+ after Δt = $\overline{Q}$
1	1	0	1	hold 1	Q is stable , Q_{new} = Q
1	1	1	0	normally not allowed	Q is unstable , Q_{new} = Q+ after Δt = $\overline{Q}$, reset dominant

- Compressed Characteristic Table
  Compressed Characteristic Table is another way to describe sequential circuit that is simpler to generate than PS/NS table while provide most of the information required. Compressed Characteristic table is commonly used in sequential circuit design and analysis. Below is an example of Compressed Characteristic table for SR flip-flop.

S	R	Q+
0	0	Q
0	1	0
1	0	1
1	1	0 (reset dominant)

❖ K-map for flip-flops
K-maps can be generated based on flip-flops PS/NS table. The following K-map is for a SR flip-flop.

Q^{+} (S,R,Q) = $Q.\overline{R} + S.\overline{R}$

SR \ Q	0	1
00	<u>0</u>	<u>1</u>
01	<u>0</u>	0
11	<u>0</u>	0
10	1	<u>1</u>

➢ Note:
Minterms 3, 4 and 7 are unstable and shown on K-map without an underlined. As stated earlier, unstable state will change after the Δt delay.

4.3. Asynchronous Sequential Logic Issues

A circuit that uses latches (flip-flops) but does not use a clock to synchronize all signals is called Asynchronous Sequential Logic. Here we will explore the issues that may be present in asynchronous design.

- Race Condition
  For example (SR flip-flop)
  - S R Q = 000
    S changes to 1 → S R Q = 100 (unstable or transitory state) → Q^+ = 1 (refer to table) →
    S R Q = 101 (after delay) A stable state. This state will be maintained until the External input changes.

SR \ Q	0	1
00	$\underline{0}$	$\underline{1}$
01	$\underline{0}$	0
11	$\underline{0}$	0
10	1	$\underline{1}$

If the S and R inputs change quickly (one after another) before the output settles into a new stable state, the input provides a race condition (each trying to change the output first). If the output becomes a predictable stable state, then the race is non-critical.

A critical race occurs if the circuit output ends in an unpredictable stable state.

  - Example of a Critical Race
    S R Q = 110
    SR are changed simultaneously to 00
    - S may change first S R Q = 010 → Q+ = 0 Stable state
      next R changes, SRQ = 0 0 0 → Q+ = 0 Stable state

SR \ Q	0	1
00	$\underline{0}$	$\underline{1}$
01	$\underline{0}$	0
11	$\underline{0}$	0
10	1	$\underline{1}$

    - R may change first S R Q = 100 → Q+ = 1 Unstable state
      next S changes, SRQ = 0 0 1 → Q+ = 1 Stable state

SR \ Q	0	1
00	0	1
01	0	0
11	0	0
10	1	1

Note that depending on if S changed first (case 1) or R changed the first (Case 2), final state will be different, which means we have a critical race.

- ➢ To insure proper operation of well-designed asynchronous sequential logic circuits (no critical race), allow only one external input signal to change at a time. This mode of operation is referred to as “fundamental operating mode”.

❖ Transient, Meta-Stable State or Unstable Equilibrium State Output
This is another failure mode of latch circuits which causes the output to oscillate between 1 and 0 and the final state may be unpredictable.

- ➢ The cause may be:
  - Runt pulses
    If two inputs feeding a gate are changed nearly simultaneously, a runt pulse may be produced at the output of the gate.
  - Positive runt pulse
    A positive-going pulse that begins with a value of 0 but doesn’t achieve the value of 1
  - Negative runt pulse
    A negative-going pulse that begins with a value of 1 but doesn’t achieve the value of 0

4.4. Finite State machine

❖ State Diagram
A state diagram is a graphical method of showing each state and the movement to other steps based on the new values of external inputs. Here are the steps (example for SR flip-flop):

Step 1. Write the Compressed Characteristic Table

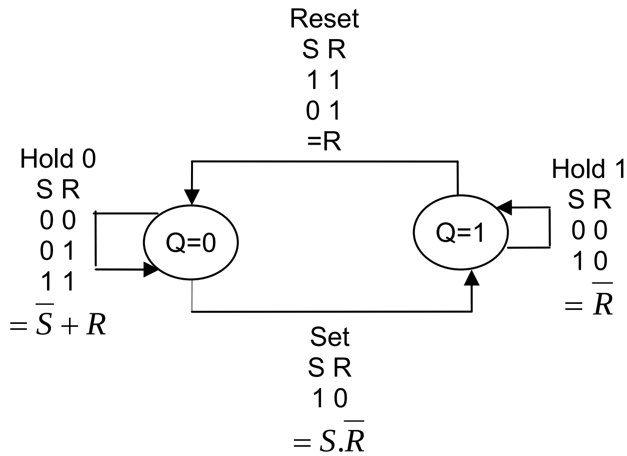
S	R	Q+
0	0	Q
0	1	0
1	0	1
1	1	0

(reset dominant)

Step 2. Present State /Next State (PS/NS) Table

S	R	Q	Q+
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

Step 3. Draw a circle for each state (one output, Q, so 2 –state)
Step 4. Draw arrow showing change for each possible input in each state.



❖ Finite State Machine or Simple State Machine is used to describe a sequential logic circuit

➢ A completely specified state machine is one for which all input conditions are used to specify each next state condition. For a completely specified machine, the “sum = 1 rule” applies which says that all outgoing conditions from a state must sum up to 1.

For example for state 0 of the SR flip-flop, a completely specified state machine, the outgoing conditions sums up to 0

Sum of Outgoing Conditions $= (\overline{S} + R) + S.\overline{R} = 1$

➢ An incompletely specified state machine is one which is missing some input condition. You may interpret them as “don’t care” conditions.

Note: For the incompletely specified state machine, the sum of outgoing conditions of all states are not equal to 1.

❖ Algorithmic State Machine (ASM) Chart
An ASM chart is similar to the programming flow chart and is an equivalent of the state diagram used to describe a sequential logic circuit (or Finite State Machine).

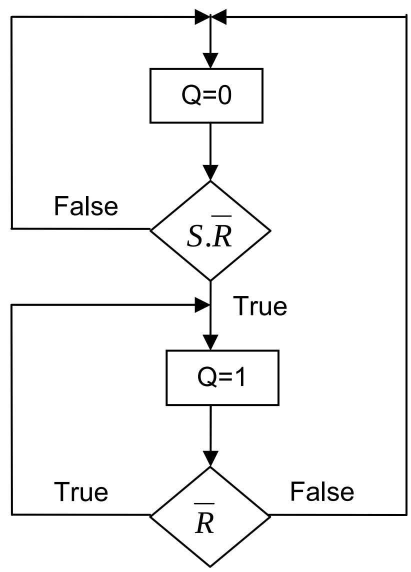
The chart uses three symbols:

- Rectangle is the state box (equivalent to circles in state diagram)
- Diamond is the decision box (equivalent to inputs next to the lines in state diagram) one of two paths provide the exit from decision box:

- If condition is true, T or “1” path is taken
- If condition is false, F or ”0” path is taken

▪ Oval shape is used for start, end and output box

➢ Example - Below is an ASM Chart example for a SR flip-flop derived from the state diagram:
- State exit conditions are the decision conditions in ASM.
- Emphasis is on state changes. All conditions that do not change the stay are not shown on the ASM chart.

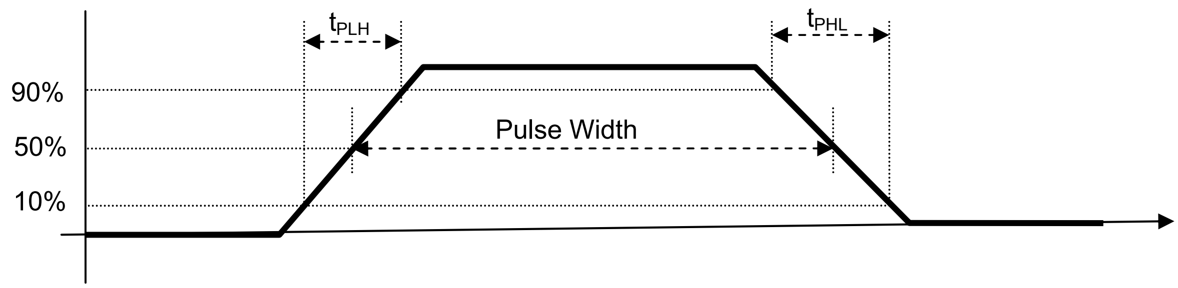


The advantage of ASM is that it has two outputs from each decision so it is clear if both conditions are addressed and therefore the state machine is completely specified.

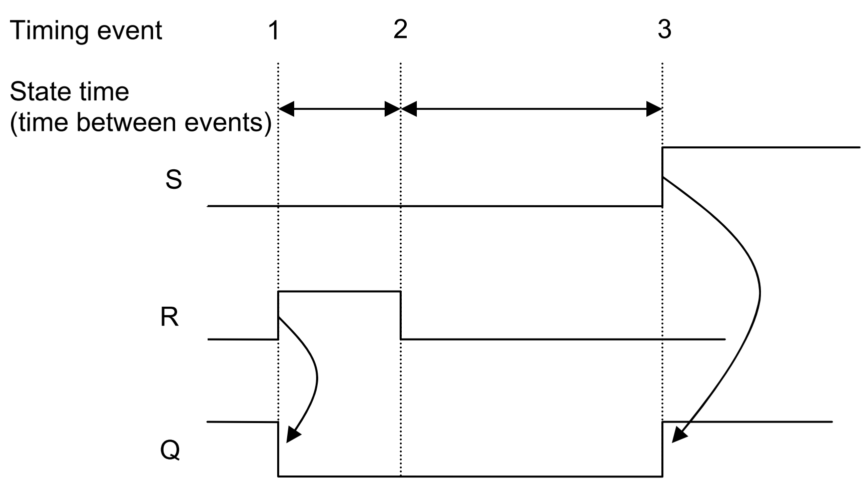
❖ Timing Diagram

Another tool for describing the functionality of a sequential logic circuit is the timing diagram. Although the timing diagram is not as scalable as State diagrams or ASM chart, it provides the timing relationship between input and out signals.

➢ Basic definitions used in timing diagrams
- Timing Events
  External input changes that cause changes in the output of a sequential logic circuit are called timing events.
- Rise Time (t_{PLH})
  The time it takes for a signal to go from a 10% to 90% value (an ideal timing diagram assumes 0 seconds)
- Fall Time (t_{PHL})
  The time it takes for a signal to go from a 90% to 10% value (an ideal timing diagram assumes 0 seconds)
- Pulse Width
  The time it takes for a signal to go from a 50% value on the rising edge to 50% value on the falling edge.
- Average Propagation delay
  In order to simplify a timing diagram, gate delays may be represented by an average propagation delay t_p where $t_p =(t_{PHL} + t_{PLH})/2$
  (an ideal timing diagram assumes 0 seconds)



- An ideal timing diagram for an SR flip-flop
  SR flip-flop discussed so far is an asynchronous sequential logic circuit since this latch circuit does not rely on a system clock for synchronization.



Note: This is an ideal timing diagram where propagation delays are not shown.

- Design of Asynchronous Sequential Logic Circuit
  Asynchronous Circuit does not rely on a clock which means that hazards may be a design problem.

  - A couple of rules to avoid logic hazards and critical races:
    - Rule 1 – One external input signal change at a time (fundamental mode)
    - Rule 2 – Before the next external signal is allowed to change, the circuit must be given time to reach a new stable state. (The circuit path with the longest delay dictates the speed of the circuit.)

➢ Applying the Design Steps to an SR flip-flop

Step 1. Write the compressed characteristic table ⇒ Step 2. Draw the compressed K-map ⇒ Step 3. Draw the schematic (since Q is used for Q+, we need a delay element)

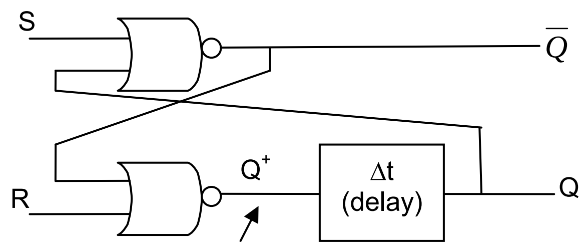
S	R	Q+
0	0	Q
0	1	0
1	0	1
1	1	0

SR	00	01	11	10
	Q	0	0	1

$$Q^{+} = Q.\overline{R} + S.\overline{R}$$

$$= \overline{\overline{(Q+S)} + R}$$

Apply DeMorgan's Theorem twice.

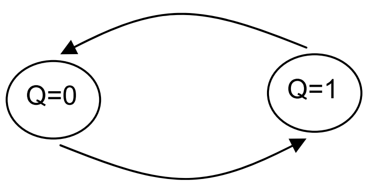


Since Q and $\overline{Q}$ are provided, it is call a "double-rail output". If Q was the only output, then it would be called a single-rail output.

Note: This circuit encounters a critical race condition when SR transitions from 11 to 00.

❖ Designing a Clock Circuit (another simple asynchronous sequential logic circuit)

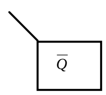
➢ Start with state diagram

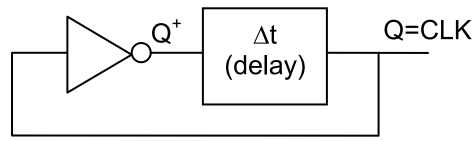


➢ Flow the design process

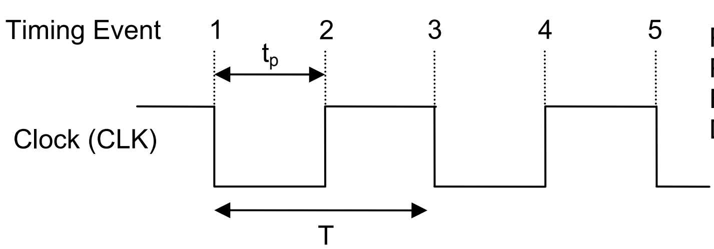
Step 1. Write the Compressed Characteristic Table ⇒ Step 2. Draw Compress K-map ⇒ Step 3. Draw the schematic (since Q is used for Q+, we need a delay element)

Q+ = $\overline{Q}$





➢ Timing Diagram



Propagation Delay, t_p = Not Gate delay + Δt
Period $T = 2t_p$
Frequency $f = 1/T = 1/(2t_p)$
Duty Cycle $DC = t_p/T = t_p/(2t_p) = ½ = 50\%$

You can increase the period T by one of the following methods:

- Adding to Δt (more buffer)
- Adding more NOT gates (must be an odd number of gates)
- Adding an RC circuit and adjusting the time constant

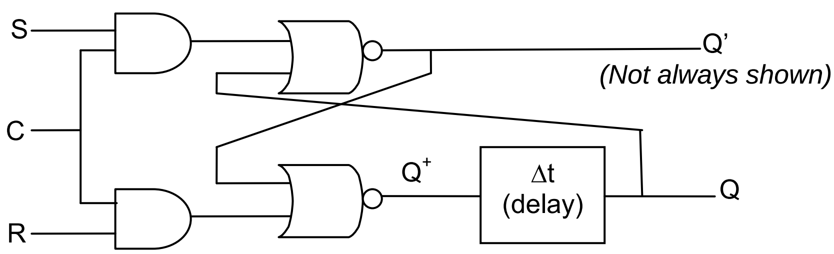
- In practice, most designs use crystal oscillators which oscillate at a precise frequency, providing more reliable system clock.

- Design of Gated Sequential Logic Circuit
  Providing another input which controls when the inputs can affect the outputs (latching the inputs) increases the functionality of the latch circuit. The resulting circuit is called a latched or gated circuit.

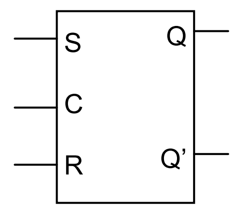
  - Gated Circuit Types
    Latches may be classified based on their input control types:

    - Level activated: Latches input when the control is at a given logic level (High or Low depending on design)
    - Edge Trigger: Latches input when the control changes level (rising- or falling-edge, depending on design)
    - Pulse-triggered: Latches input when the control is pulsed (a rising-edge followed by a falling-edge)

  - Gated S-R flip-flop (latch) Circuit



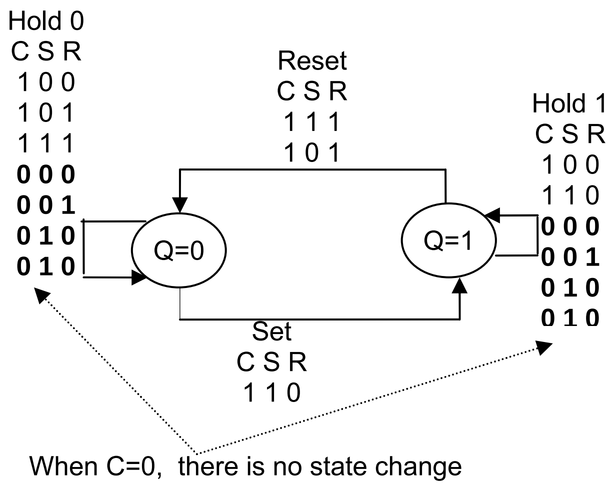
Gated SR Latch Using NOR gates



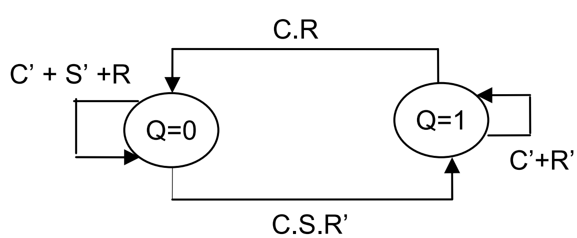
Block Diagram

    - This circuit allows inputs to affect the outputs only when C=1. When C=0, the latch holds the last state value at its output. This is an example of high-level activated SR flip-flop.

- The state diagram showing the effect of Latch Control (C) on the state change.



* Another form of State Diagram for the same circuit is shown below:

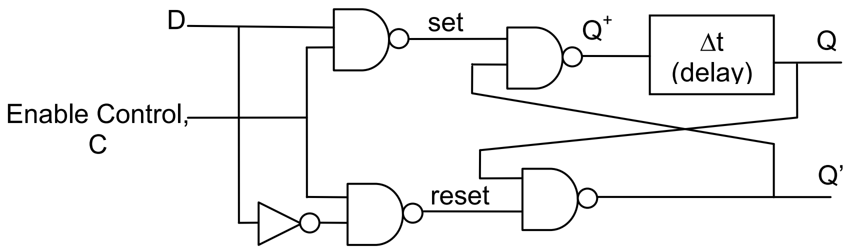


Note: This circuit, like the SR Latch Circuit, has a critical race when CSR transitions from 111 to CSR=100. To prevent this issue, it is possible not to allow CSR=111.

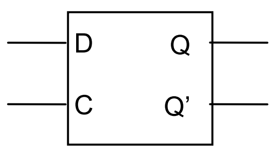
4.5. Additional Flip Flops

❖ D flip-flop (or D-latch)
Although we have talked about SR flip-flop first, there are many other types of flip flops. Each have their own set of advantages and disadvantages. D flip-flop is the most commonly used flip-flops due to its simplicity. Additionally, D flip-flop does not have an inherent critical race.

SR flip-flops can be modified using NAND gates to create a D flip flop as shown in the following diagram:



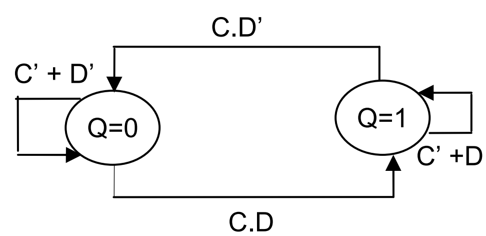
The D flip-flop may be referred to as "gated D Latch", a transparent D latch, a level sensitive flip-flop or data flip-flop. Symbol and Compressed Characteristic table is shown below:



D flip-flop Symbol

C	D	Q+
0	0	Q
0	1	Q
1	0	0
1	1	1

Compressed Characteristic Table



State Diagram

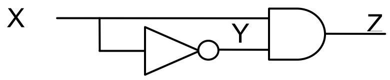
Note: When C=0 (inactive), the last value of D is driving the output. Also it can be shown that it does not contain critical race and is logic hazard free.

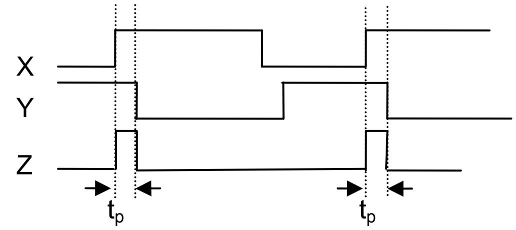
- Explore the specifications for 74LS373 "level activated" and 74LS374 "Positive-edge triggered". Refer to Course Website for the complete specifications including:

  - Set up time (t_{su})
  - Hold time (t_h)
  - Sampling interval, t_{si} = (t_{su}+ t_h)
    The Minimum t_{si} is required for proper operation of the circuit.
  - 3-state output.

➢ Although there are pulse, level activated flip flop and edge-triggered D latches, it is recommended that new design use edge-triggered flip-flops.

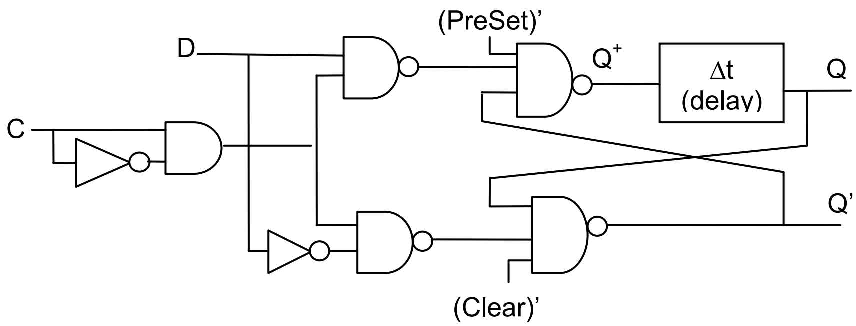
- Basic edge-triggered flip-flop
  One ways to create a narrow pulse is by using the following circuit:





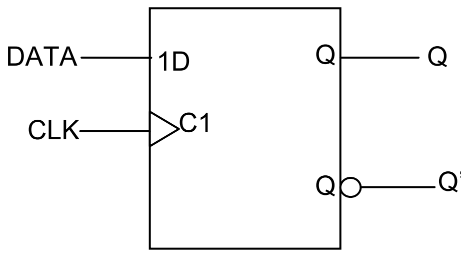
A pulse is generated only during the rising edge

So you can use this design to implement a positive-edge-triggered D flip-flop circuit with preset and clear inputs.

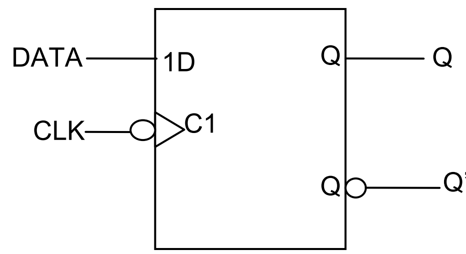


Circuit Diagram

➢ Symbols

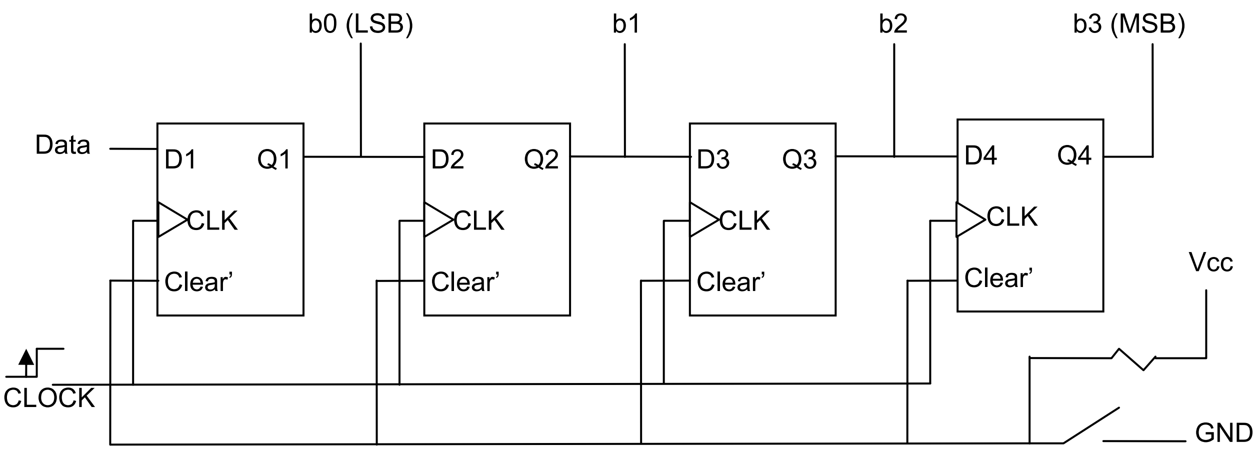


Positive-Edge-Triggered D Flip-Flop
(triangle called dynamic indicator)

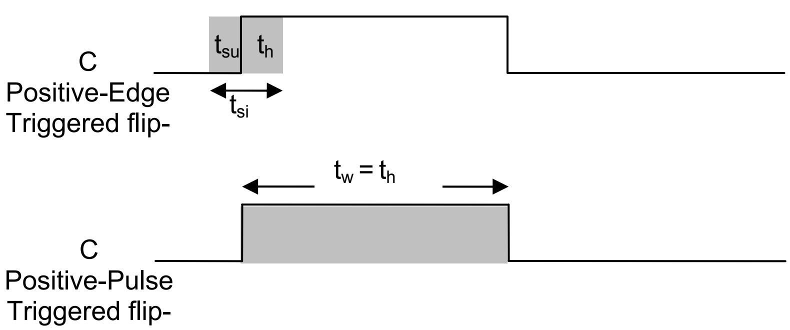


Negative-Edge-Triggered D Flip-Flop
(Bubbled triangle indicator)

➢ Application: using D flip-flops with clear to build a shift register



- Edge-Triggered and Pulse-Triggered Flip Flop Comparison
  A pulse enable the input to change the output. The pulse can be negative or positive depending on the flip-flop design. Here is a comparison of hold time requirement for positive-edge and positive-pulse triggered flip-flop.

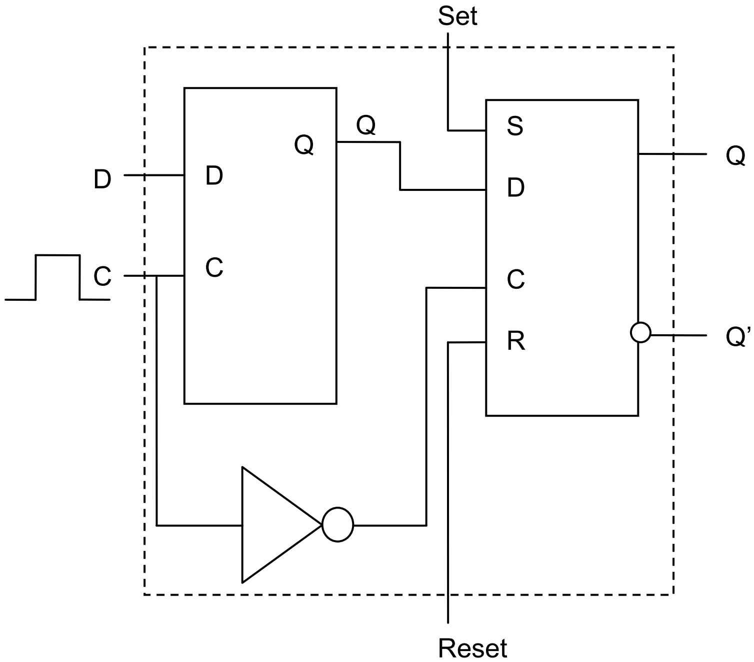


t_s = Set up time
t_h = Hold time
t_w = Pulse width
t_{si} = Sampling interval = t_{su} + t_h

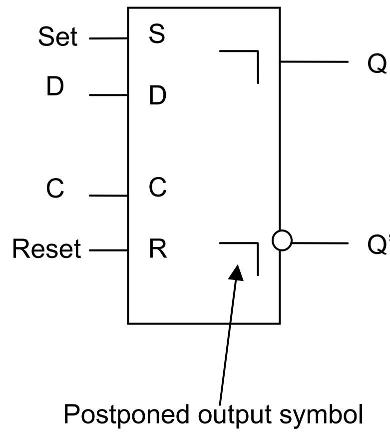
Shaded area is the area where data must be stable.

- A pulse-triggered flip-flop has much longer sampling interval than an edge-triggered flip-flops; therefore, all new designs use Edge-triggered flip-flops to improve speed.

- A pulse-triggered flip-flop is also called "master-salve" due to its implementation which require two D-type flip-flop in a master-salve set up as shown below:

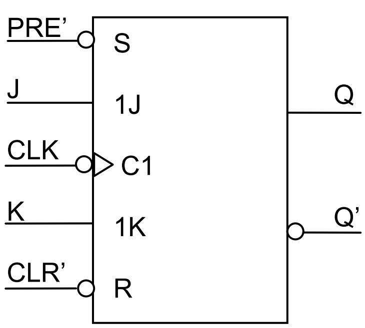


or



The D flip-flop is the most commonly used bi-stable memory device. The other two kinds used are J-K and T flip-flops.

❖ J-K Flip Flops
Below is a negative-edge triggered JK flip flop. As in other flip-flops, the inputs J and K are called "excitation inputs".



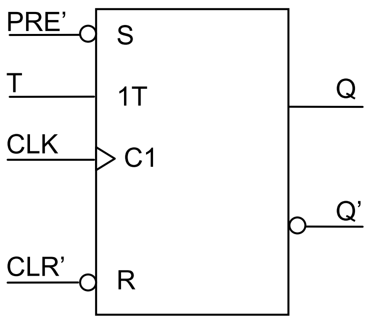
Symbol

Characteristic Table

J	K	Q^+	Comment
0	0	Q	no change
0	1	0	reset condition
1	0	1	set condition
1	1	Q'	toggle

Characteristic equation: $Q^+ = J.Q' + K'.Q$

❖ Toggle or T Flip Flops



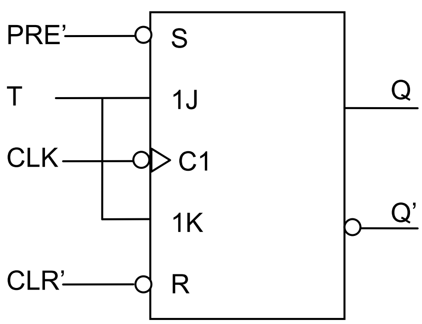
Symbol

Characteristic Table

T	Q^+	Comment
0	Q	no Change
1	Q’	toggle

Characteristic equation: $Q^+ = T'.Q + T.Q'$

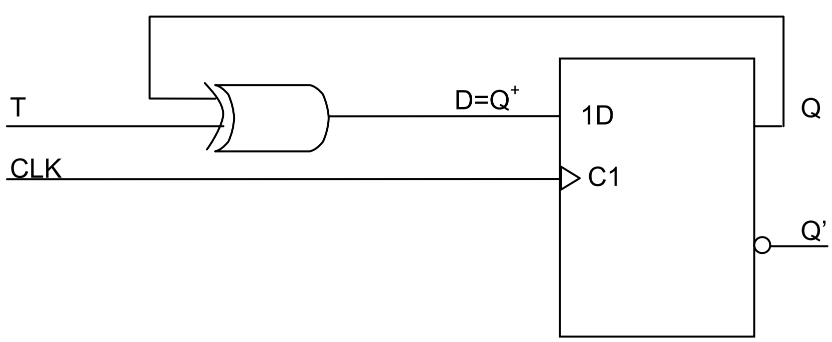
➢ Using a JK flip-flop to implement a negative-edge-triggered T flip-flop:



Symbol

J=K=T	Q^+	Comment
0 0 0	Q	no Change
0 1	Invalid	
1 0	Invalid	
1 1 1	Q’	toggle

➢ Using D flip-flop to implement a positive-edge-triggered T flip-flop:



4.6. Sequential Circuit Analysis

Flip-flops may be used to design circuits with feedback, such as counters, shift registers, sequence detectors and controllers.

Feedback systems are classified as synchronous when all changes are synchronized with the system clock. Feedback systems that do not use the system clock and change as the input change are called asynchronous.

Synchronous systems are preferred over asynchronous systems since they will not have hazards and other synchronization issues.

Synchronous systems are also referred to as synchronous finite state machines (FSM).

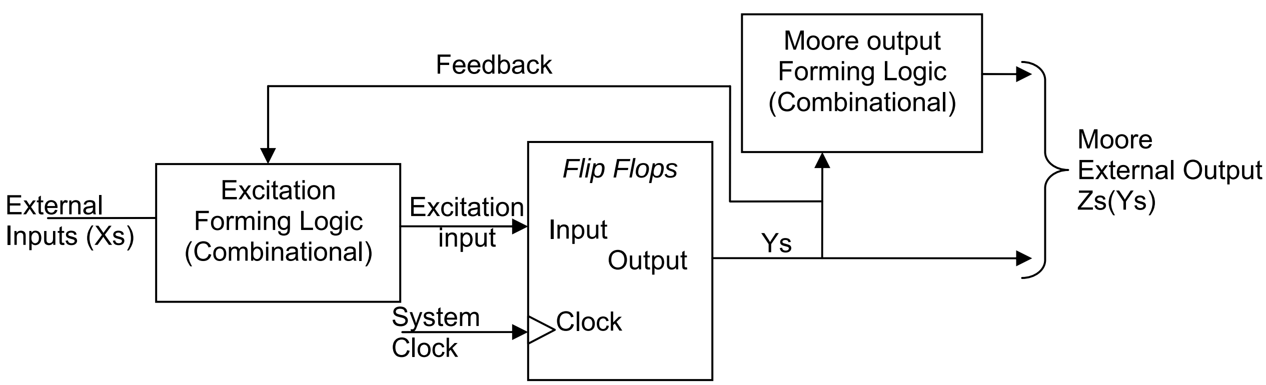
A FSM utilizes a system clock that adheres to the following definitions:

* Each period of system clock represents a State-Time
* A state represents the state of the flip-flop outputs (value of outputs)
* Synchronous external inputs → Xs
* Current machine state, flip-flop outputs → Ys
* Synchronous state machine external outputs → Zs

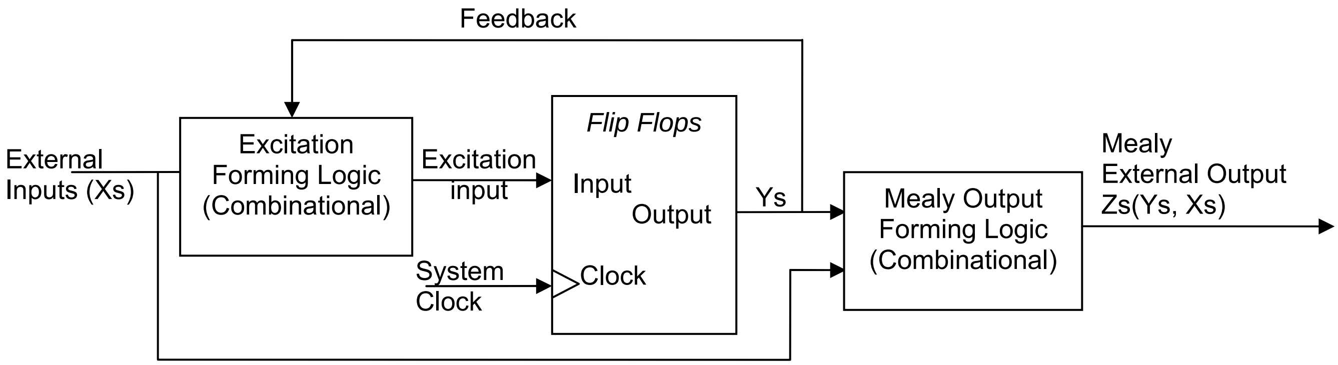
Synchronous state machines may be implemented in one of three models based on the characteristic of its output (Moore, Mealy or mixed-type synchronous state machine):

- Moore-type Synchronous Finite State Machine
  outputs are a function of the state of the machine Z1(Y1, Y2, Y3)

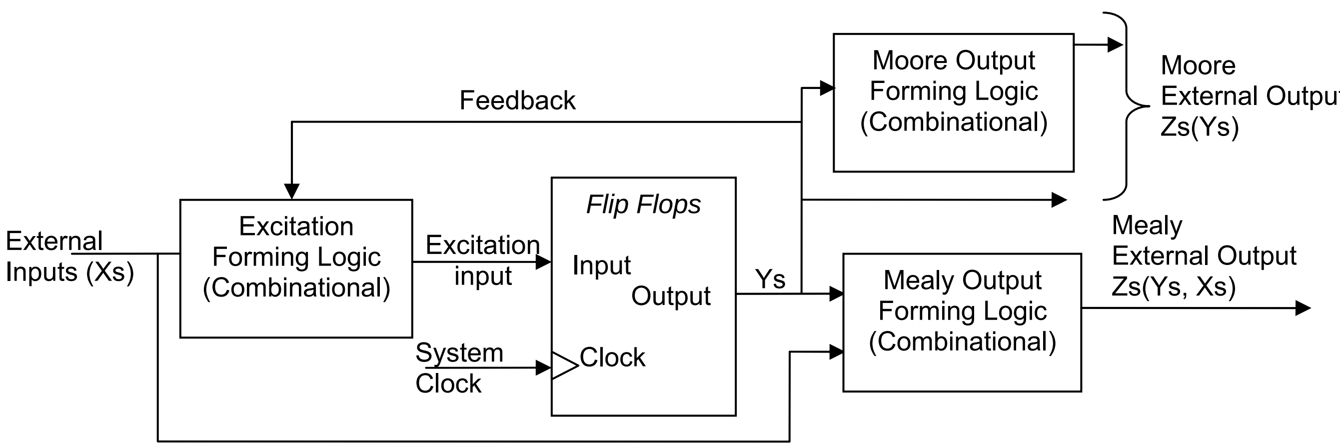
  A Binary counter is a good example of Moore machine since the output of the flip-flops can be used to represent the count.



- Mealy-type Synchronous Finite State Machine
  Outputs are a function of the state of the machine and external inputs. Z1 (Y1, Y2, …X1, X2, …)



➢ Mixed-type Synchronous Finite State Machine
Some outputs are Mealy-type and others are Moore-type.



- Analyzing Synchronous Systems (General)
  There are five steps in analysis of this type of circuit:

  1) Assign a present state variable to each flip flop in the synchronous system.
     Yi represents flip-flop outputs for i = 1, 2, 3, …

  2) Write the excitation-input equation for each of the flip-flops and the external-output (Moore and/or mealy equations). After completing this step, Di, Ji Ki, Ti should be defined where i=1, 2, 3 … {# of flip-flops used}.

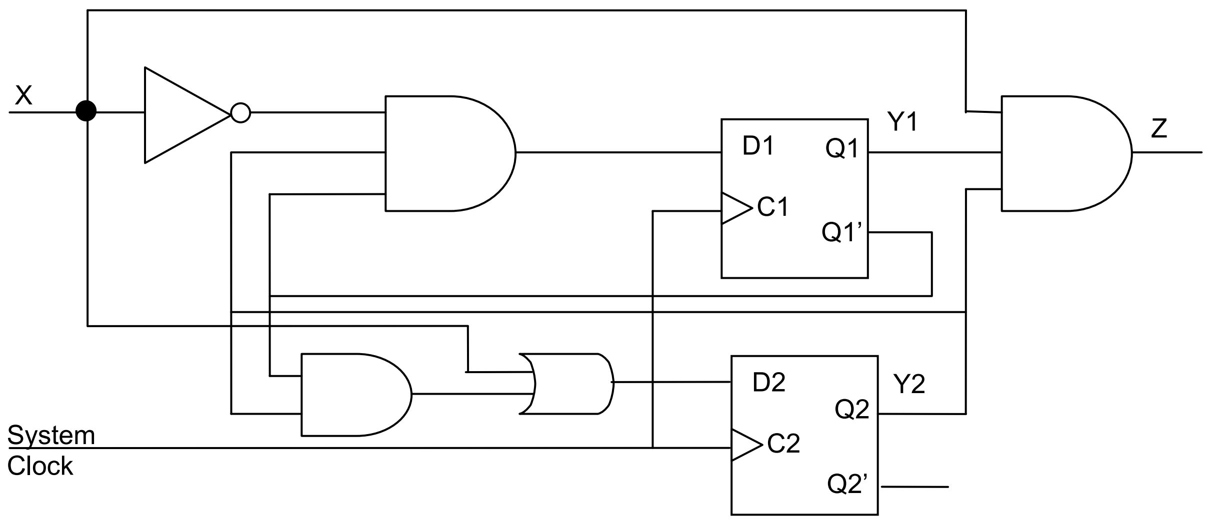
  3) Substitute the excitation input equation into the characteristic equations of the flip-flops to obtain the "next state" equations.
     For D flip-flops → Yi^+ = Di for i=1, 2, 3, …
     For J-K flip-flops → Yi^+ = Ji.Yi' + Ki'.Yi for i=1, 2, 3, …
     For T flip-flops → Yi^+ = Ti <XOR> Yi for i=1, 2, 3, …

  4) Obtain a PS/NS table or a composite K-map using the next state and external-out (Mealy and/or Moore) equations. Separate K-maps can be used for the external

outputs if desired.

5) Use the PS/NS table or the composite K-map to obtain a state diagram, ASM chart or timing diagram to show the behavior of the circuit.

- Apply the five step analysis technique to the following circuit:



Note: This is a Mealy-type machine since the output depends on external input and flip-flop outputs.

1) Assign a present state variable to each flip flop in the synchronous system.
   Yi representing flip-flop outputs for i = 1, 2, 3, …
   Solution: Refer to the schematics

2) Write the excitation-input equation for the flip-flops and the equation for the external-output (Moore and/or mealy equations). After this step is completed, the values of Di, Z should be defined for all flip-flops.

   Solutions:
   D1 = X'.Y1'.Y2
   D2 = Y1'.Y2 + X
   Z = Y1.Y2.X

3) Substitute the excitation-input equation into the characteristic equations for the flip-flops to obtain the "next state" equations.

   D flip-flops → Yi^{+} = Di for i=1, 2, 3, …

   Solutions:
   $Y1^{+}$ = D1 = X'.Y1'.Y2
   $Y2^{+}$ = D2 = Y1'.Y2 + X

4) Obtain a PS/NS table or a composite K-map using the next state and external-output (Mealy and/or Moore) equations. Separate K-maps can be used for the external outputs if desired.
   Solutions:

PS/NS Table

$Y1$	$Y2$	X	$Y1^+$	$Y2^+$	Z
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	1	1	0
0	1	1	0	1	0
1	0	0	0	0	0
1	0	1	0	1	0
1	1	0	0	0	0
1	1	1	0	1	1

OR

Composite K-map where:

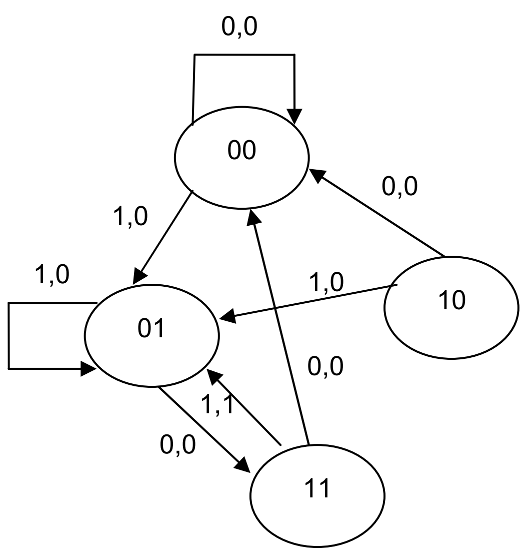
→ Ys and Xs are independent variables
→ Ys^+ and Zs are Dependent

Y_1Y_2 \ X	0	1
00	00,0	01,0
01	11,0	01,0
11	00,0	01,1
10	00,0	01,0

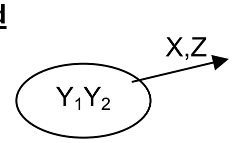
$Y_1^+Y_2^+$, Z

5) Use the PS/NS table or the composite K-map to obtain a state diagram, ASM chart or timing diagram to show the behavior of the circuit.

Solutions:
Since there are two flip-flop, the state machine has 4 states.



Legend



Notes

1) State 00 is reset

2) Output Z=1 only when the input sequence is 101, so this could be "101" pattern detector.

3) State "10" is referred to as "illegal state", "unused state" or an "unreachable state".

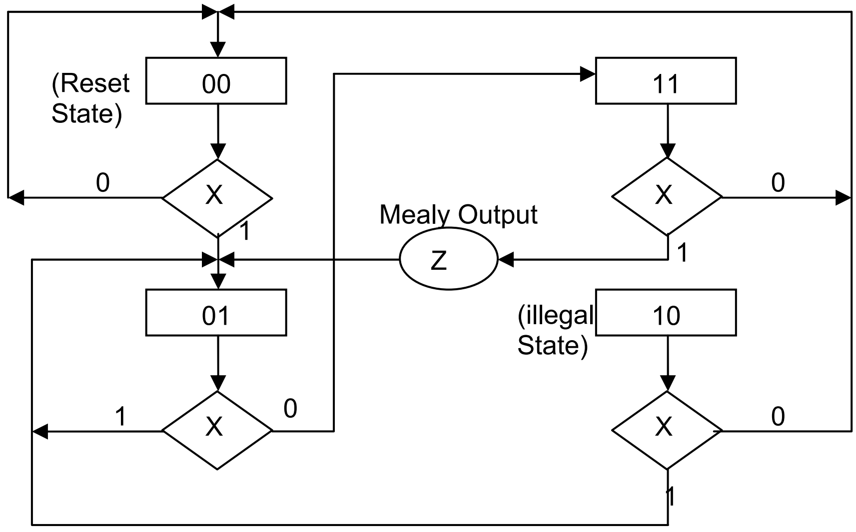
4) One way to ensure you don't end up in illegal state is to have a power on reset.

Classic State machine
* Links show input, output in 1s and 0s
* State is inside the circles

In the case of Moore machines, outputs must be inside the circle because they only depends on the current state.

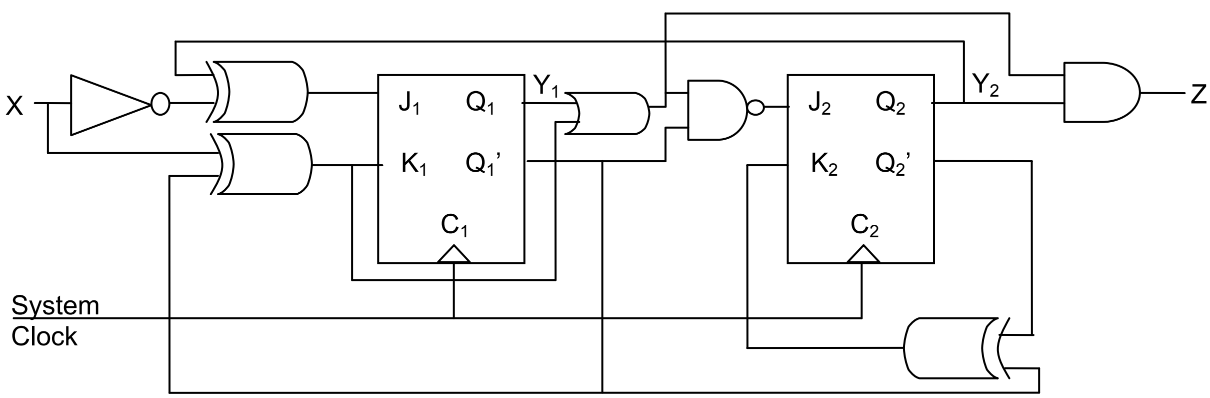
Note: A simplified State machine shows the links between states in Boolean expressions.

An alternative method is the use of Algorithmic State Machine (ASM) chart to describe the functionality.



Note: When Z is not shown, it is assumed the output is 0.

❖ Analysis of JK Flip-Flop Circuits
Apply the 5-step FSM analysis to the following circuits:



1) Assign a present state variable to each flip flop in the synchronous system.
   Yi representing flip-flop outputs for i = 1, 2, 3, …
   Solution: Refer to the schematics

2) Write the Excitation-input equation for JK flip-flop and the equation for the external-output.

3) Substitute the excitation-input equation into the characteristic equations for the flip-flops to obtain the "next state" equations.
   For J-K flip-flops → $Y_i^+ = J_i.Y_i' + K_i'.Y_i$ for i=1, 2, 3, …

$Y_1^+ =$

$Y_2^+ =$

4) Obtain a PS/NS table.

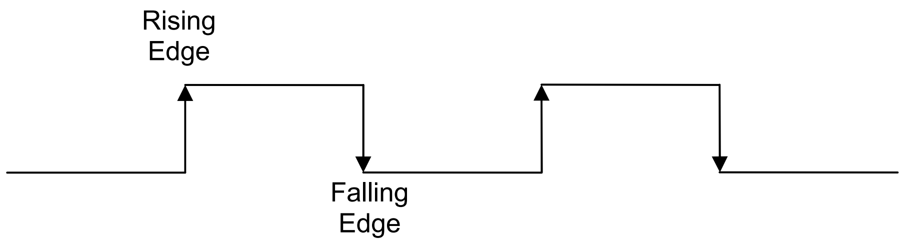
PS/NS Table

Y_1	Y_2	X	Y_1^+	Y_2^+	Z
0	0	0			
0	0	1			
0	1	0			
0	1	1			
1	0	0			
1	0	1			
1	1	0			
1	1	1			

5) Use the PS/NS table or the composite K-map to obtain a state diagram to show the behavior of the circuit.

❖ Input Date Synchronization
Synchronized systems have to accept external-inputs that may not be synchronized with the system clock. Typically, an input is synchronized with the rising or falling edge of the system clock prior to using it in the system:

- In-Phase Synchronization is when the input is synchronized with the rising edge of the system clock.
- Anti-Phase Synchronization is when the input is synchronized with the falling edge of the system clock.

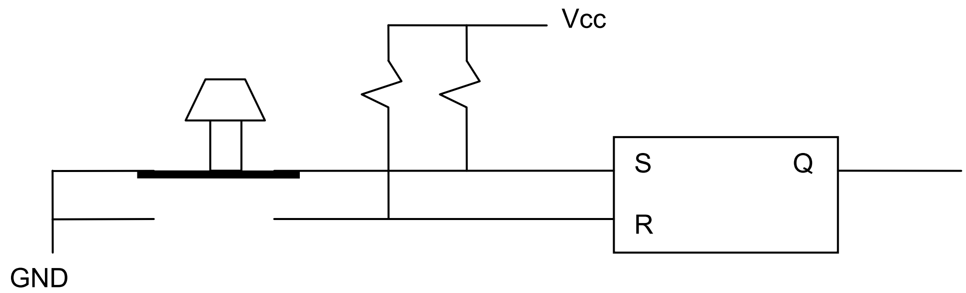


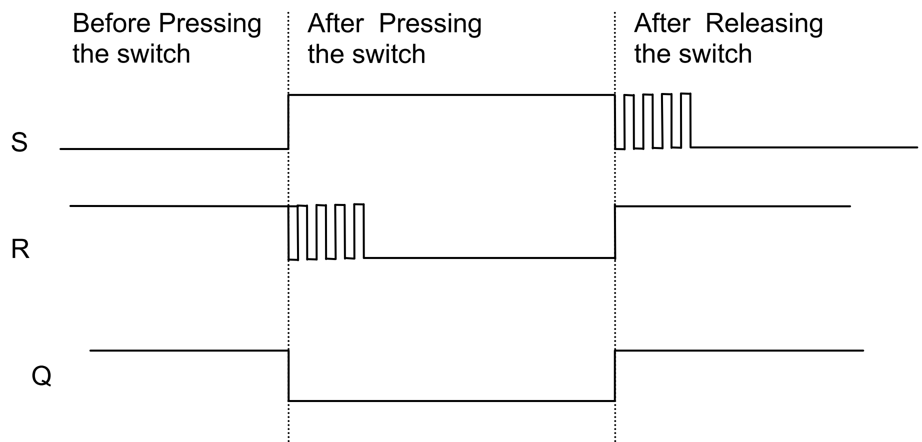
4.7. Debouncing Mechanical Switches

Mechanical switches bounced for many milliseconds before stabilizing on their new position. Meaning, the switch will open and close repeatedly (bounce) when switch position is changed..

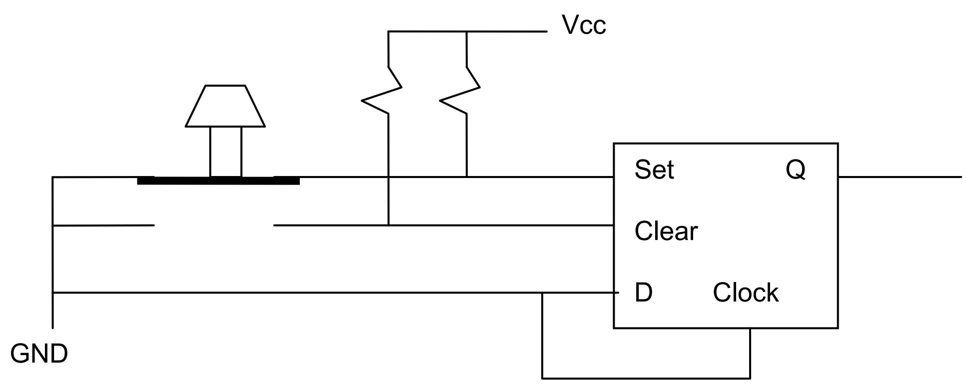
If the switch is used as a clock or an input where each pulse will be considered an input then it is required that the switch output be debounced before using it in the system.

Below is a debounce circuit using SR Flip Flop:





Note: D Flip Flop with Set and Clear may be used to debounce a switch as shown below

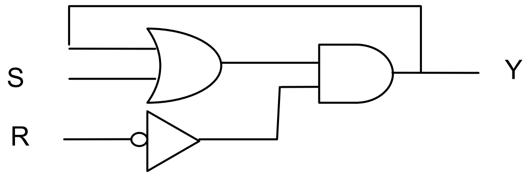


4.8. Additional Resources

- ❖ Wakerly, I. Digital Design. (2006) Prentice Hall
  Chapter 7 “Sequential Logic Design Principles”

4.9. Problems

1. Analyze the latch circuit shown below by obtaining timing diagram for the circuit; include propagation delays.

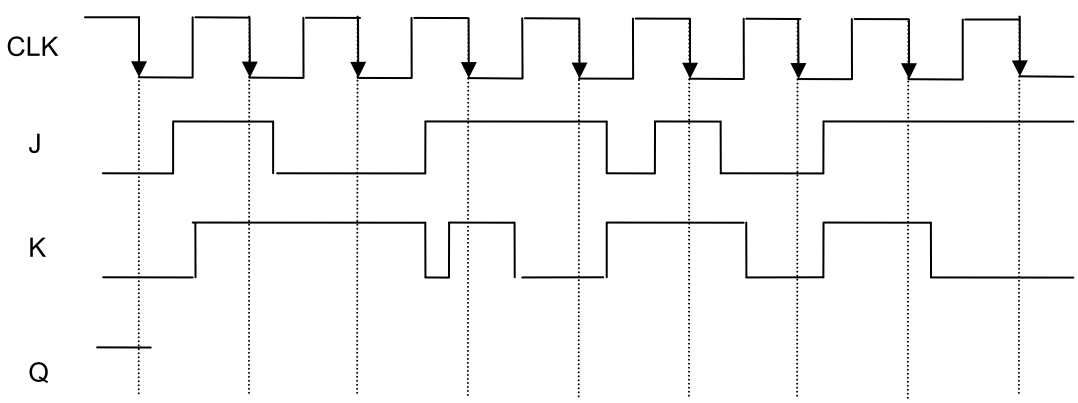


2. Analyze the gated S-R latch circuit to show by K-map that the input conditions SRC=111 are not normally allowed since changing S R to 00 simultaneously with C = 1 can result in a critical race. (Assume Control = 1)

3. Draw a clock signal CLK with four 1-to-0 timing events, a nonzero setup time, t_{su}, and a nonzero hold time, t_h. Also draw one excitation input signal EI that follows the sequence 0101 and meets the setup and hold time requirements. In the excitation input, the first bit in the sequence (0) occurs in time for the first clock timing event, the second bit in the sequence (1) occurs in time for the second clock timing event, and so forth.

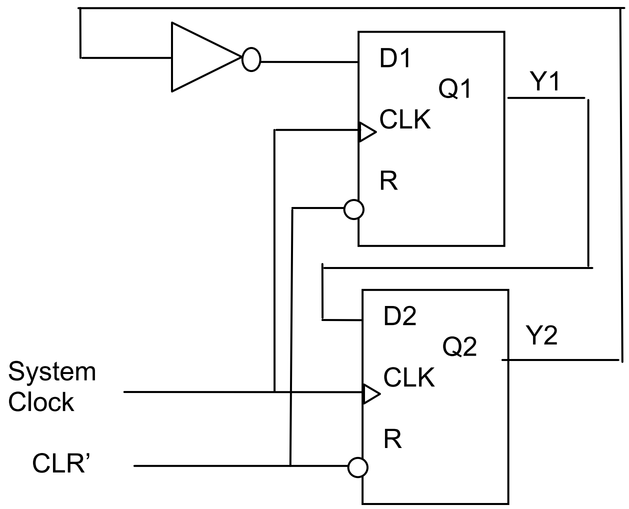
4. Show how a positive-edge-triggered D flip-flop and an inverter can be used as a negative-edge D flip-flop.

5. Complete the timing diagram shown below for a negative-edge-triggered J-K flip-flop. Assume that the J and K inputs always meet the setup and hold time requirements for the J-K flip-flop.
Hint: Use the JK flip-flop characteristic table.



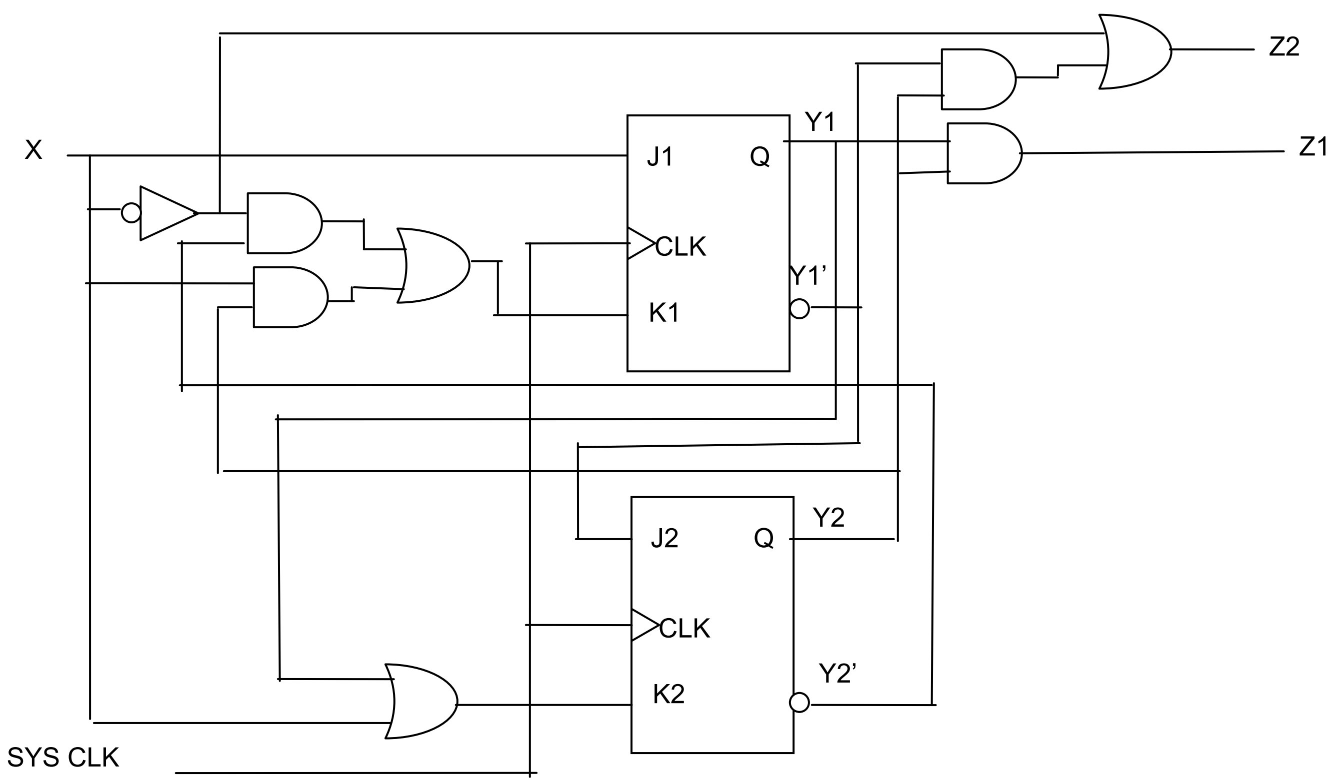
J	K	Q^+
0	0	Q
0	1	0
1	0	1
1	1	Q'

6. Obtain the state diagram for the circuit shown below.

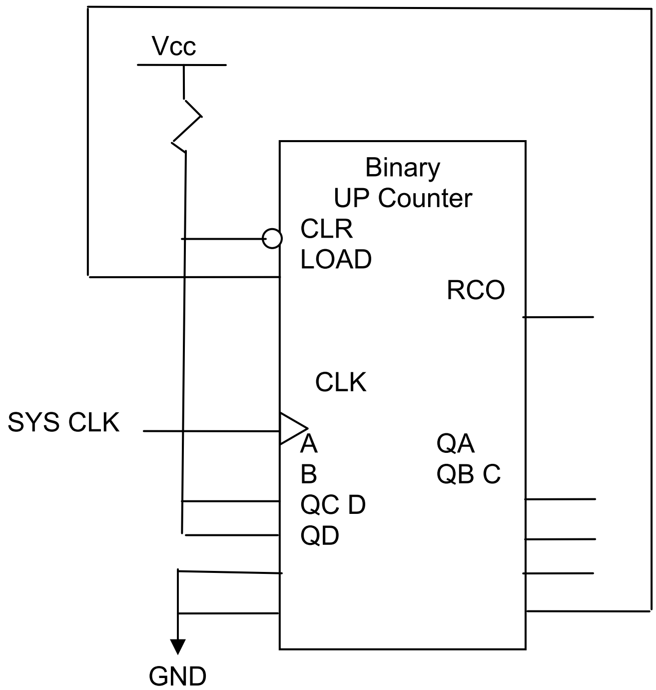


7. Analyze the state machine shown below to obtain the following items:

   a) Excitation input and external output equations
   b) PS/NS
   c) State Diagram



8. Obtain the state diagram for the circuit shown below. What is the mod of the counter? Is this counter useful? Change only the value being loaded at the inputs to obtain a mod 9 counter. What value must be loaded to do this?

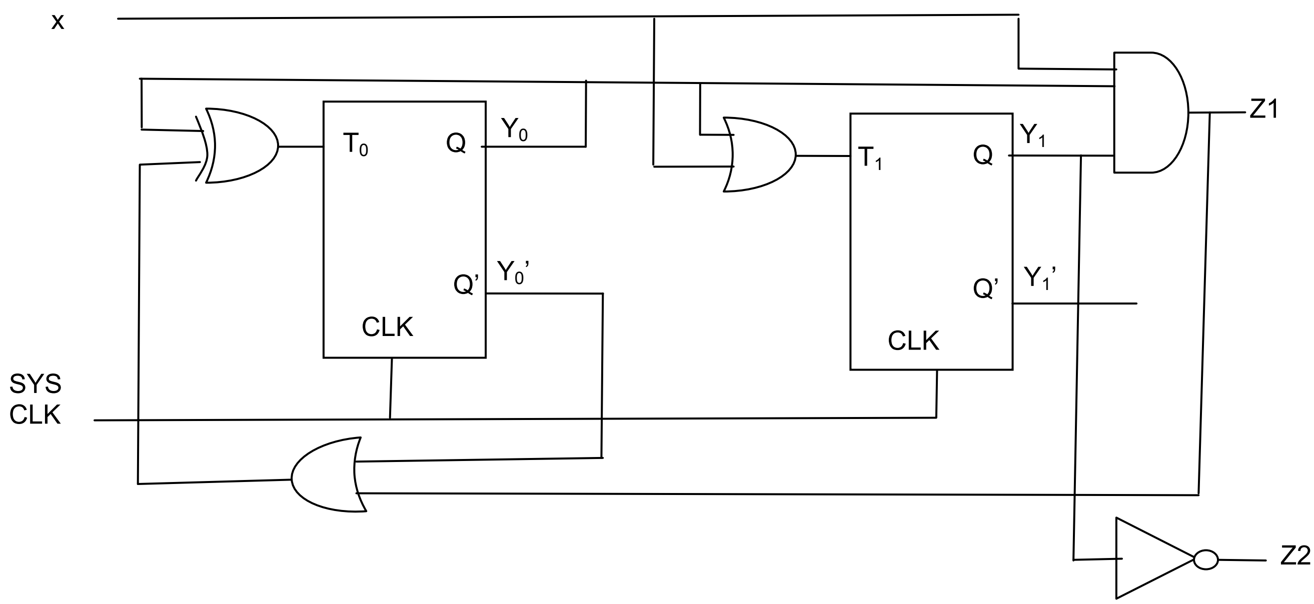


- CLR' and LOAD' are synchronous inputs
- QD is MSB & QA is LSB

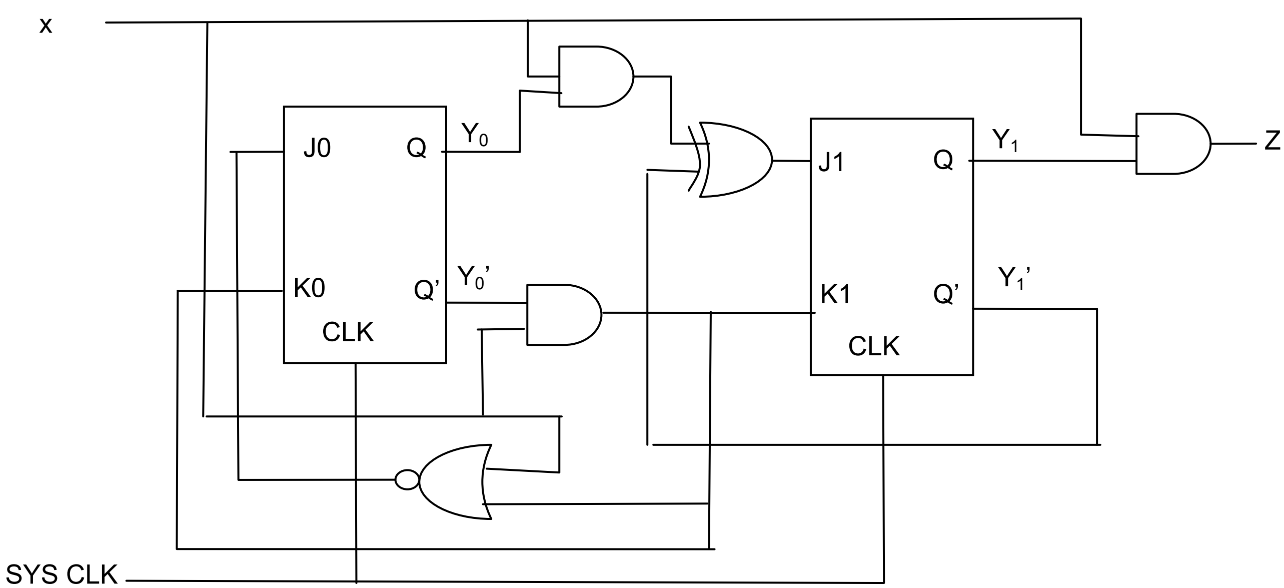
9. What is the frequency of the fastest clock for a circuit using D flip flops with t_{hold} =5 nsec. and t_{setup} = 10 nsec. Assuming there are no other limitations.

10. What is the frequency of the fastest clock for a circuit using D flip flops with t_{hold} =0.5 nsec. and t_{setup} = 1.5 nsec. Assuming there are no other limitations.

11. Analyze the following logic circuit:



12. Analyze the following logic circuit:



Chapter 5. Sequential Circuit Design & Techniques

5.1. Key concepts and Overview

- Synchronous Finite State Machine Design (Classical Technique and Examples)
- State Assignment Encoding, Shift Register Counters, and Enable Inputs
- Inspection Design Methods for Finite State Machines (Inspection Techniques)
- Additional Resources
- Problems

5.2. Synchronous Finite State Machine Design (Classical Design)

- Common Examples of Synchronous FSM
  - Up and Down Binary Counters
  - Shift Registers
  - Sequence Detectors
  - Controllers

- The Seven-Step Design Process for Synchronous Sequential Design (Classical Design)

1) Organizing the Design Specifications –Use one or more of the following tools:
   System Diagram, Timing Diagram, State Diagram or ASM Chart.

2) Determine the number of flip-flops based on the number of states
   At this point, designer may choose to design a full encoding or one-hot encoding. Full encoding utilize all possible combinations of the flip-flops and the following inequality is used to decide the number of flip-flops:

   $2^{\#\text{flip-flop}} \geq \#\text{ States}$

   The other encoding option is one-hot encoding where state is defined by which flip-flop's output is 1. So the number of flip-flop is equal to the number of States.

   Once the number of flip-flops is determined, assign one variable for each of the flip-flop output.

3) Assign a unique code to each state (a specific value for present state variables)

4) Select the flip-flop type to be used, draw the Present State/Next State (PS/NS) table, determine the excitation input equations and the Moore and/or Mealy output equations.

   Remember the excitation for common flip-flop's:
   JK $\rightarrow$ $Y^+ = J.Y' + K'.Y$
   T $\rightarrow$ $Y^+ = T_{XOR}\ Y$
   ….D $\rightarrow$ $Y^+ = D$
   Note: D flip-flops are generally preferred for most synchronous sequential designs.

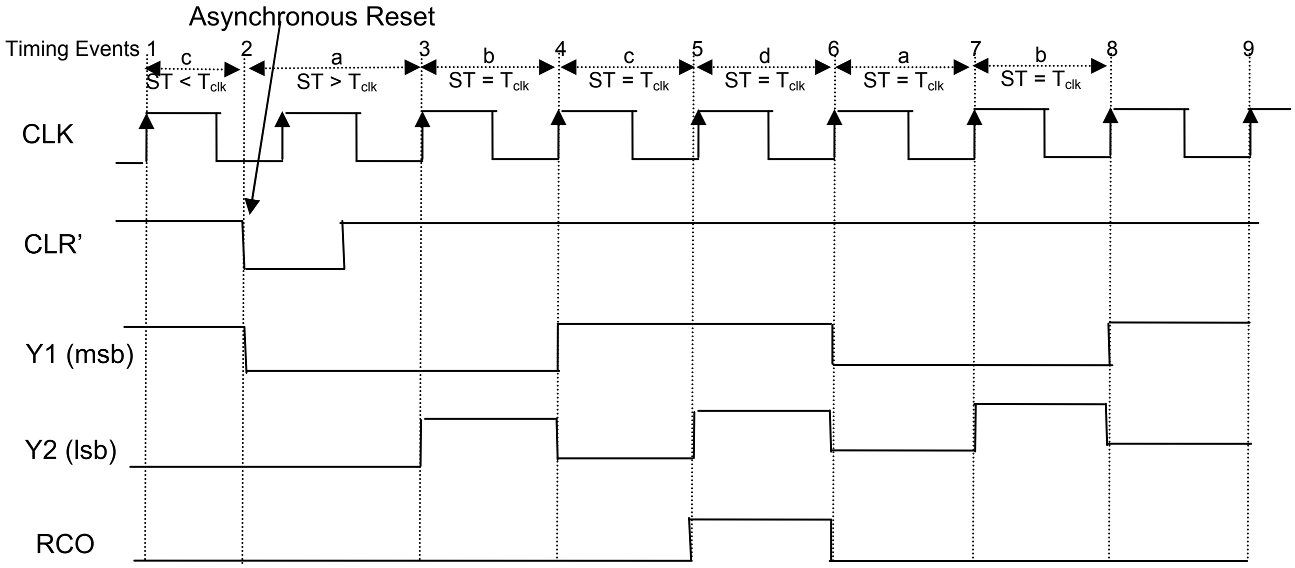
5) Draw the circuit schematic (paper or CAD tool).

6) Perform a simulation to test the functionally of the design.

7) Implement the design with hardware.

❖ EXAMPLE - Design a 2-bit binary up-counter with a ripple carry output (RCO) using D flip-flops. The input CLR' is an asynchronous input that overrides the clock.

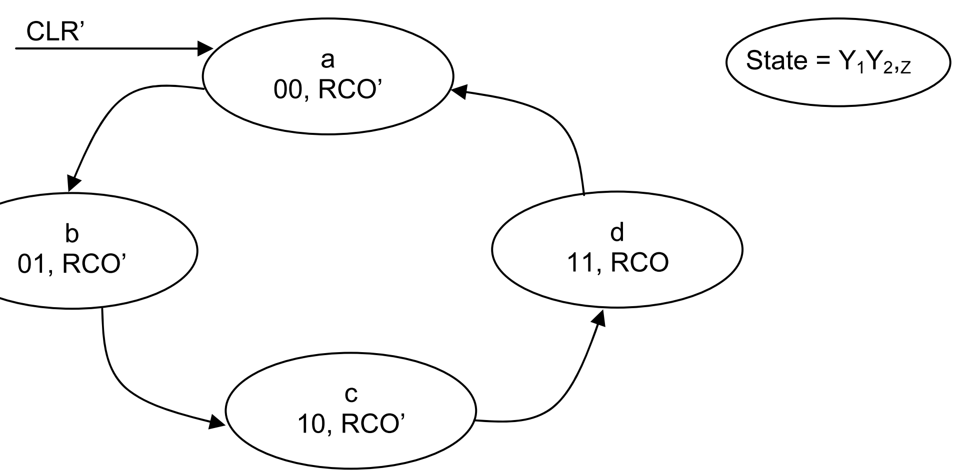
Step1) Design specifications using a timing diagram



**Notes:

1) T_{clk} is the clock period and ST is the state time.
2) The first two events are less and then more than T_{clk}.
3) Y1Y2 are the states (counts).
4) RCO is a Moore output indicating when the maximum count has been reached.

For completeness, we can also show the state diagrams. Although a timing diagram is more complete, the state diagram is simpler to understand, since it does not contain the clock timing information.



Another way to show the functionality of this circuit is to use a PS/NS table.
Note: The "Next state, NS" is the estate of the machine during the next clock cycle.

Asynchronous Clear Input CLR'	Present State Y1	Y2	Next State $Y1^+$	$Y2^+$	Present Output RCO
1	0	0	0	1	0
1	0	1	1	0	0
1	1	0	1	1	0
1	1	1	0	0	1
0	X	X	0	0	0

Present State / Next State (PS/NS) Table

Present State Y1	Y2	Next State $Y1^+$	$Y2^+$	Present Output RCO
0	0	0	1	0
0	1	1	0	0
1	0	1	1	0
1	1	0	0	1

Simplified PS/NS Table
(Note: CLR'=0 → Y1Y2=00)

Step 2) Determine the number of flip-flop based on the number of states.
For full encoding (#states = 4) ≤ $2^{(\#\text{flip-flop} = 2)}$.

Step 3) Assign Unique code to each state.
Already done in the state diagram.

Step 4) Write the excitation-input equations
The D flip-flop excitation equation is $D = Y^+$.
All we need is the K-map for each of the desired outputs Y_1^+ ,Y_2^+, RCO:

Y1Y2	
00	0
01	1
11	0
10	1

D1 = $Y1^+$
D1 = Y1'.Y2 + Y1.Y2'
D1 = Y1 $_{XOR}$ Y2

Y1Y2	
00	1
01	0
11	0
10	1

D2 = $Y2^+$
D2 = Y2'

Y1Y2	
00	0
01	0
11	1
10	0

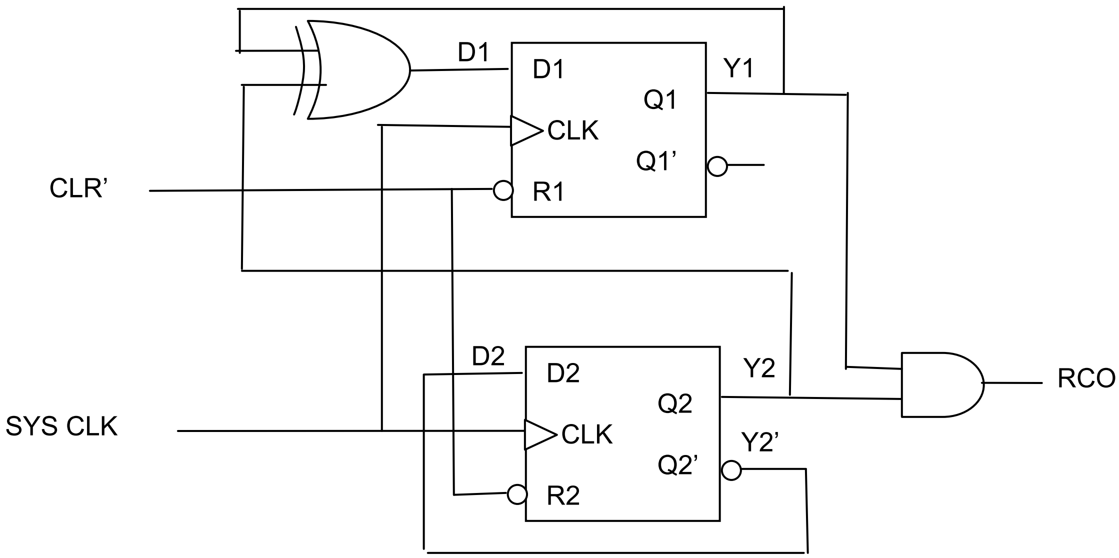
RCO
D2 = Y1.Y2

Excitation-inputs and output RCO equations derived from separate K maps
(These equations are also called design equations)

Y1Y2	$Y1^+$	$Y2^+$	RCO
00	0	1	0
01	1	0	0
11	0	0	1
10	1	1	0

A composite K-map is a short hand for multiple K-maps.

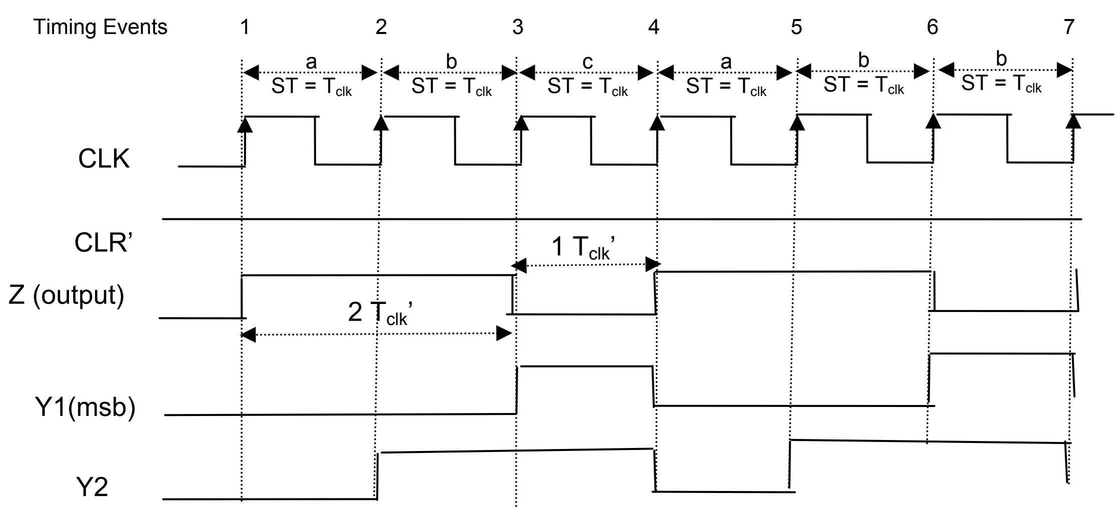
Step 5) Draw the Circuit Schematic.



Steps 6 & 7) Testing and hardware implementation will be skipped for this example.

❖ A Second Application of the Classical Design Process:
Design a synchronous sequential circuit called "Div-by-3", having an output Z that divides the system clock frequency f_{CLK} by 3. The output duty cycle of two-thirds (2 CLK cycle high, 1 cycle low). Design the circuit using positive-edge-triggered flip-flops.

Step1) Design Specifications Using a Timing Diagram



Step 2) Determine the number of flip-flop based on the number of State
(# state = 3) ≤ $2^{(\text{\#flip-flop = 2})}$ Assuming Full Coding

Step 3) Assign a unique code to each state
a: 00, b:01; C:11

Step 4) Write the excitation-input equations
The D flip flop excitation equation is $D = Y^{+}$

All we need is the composite K-map for each of the desired outputs $Y1^+$, $Y2^+$, Z:

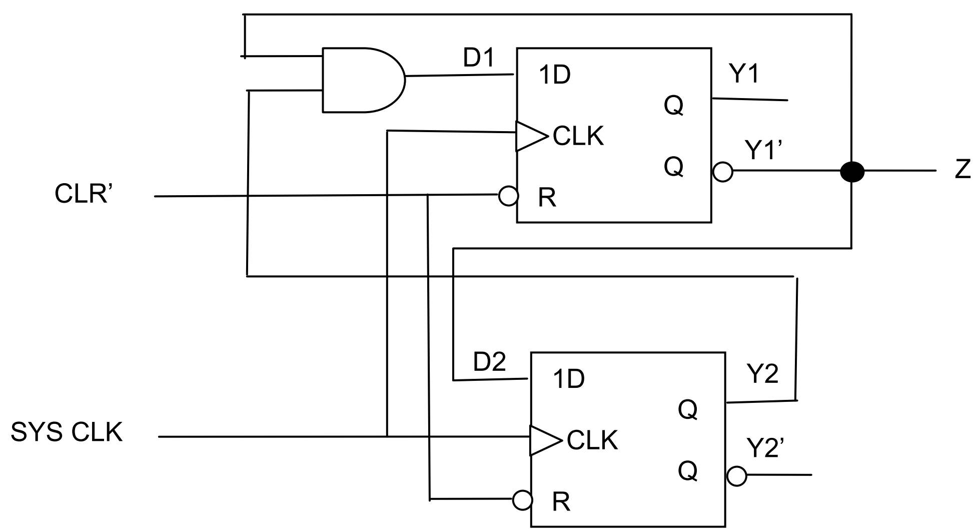
Y1Y2 \	$Y1^+$	$Y2^+$	Z
00	0	1	1
01	1	1	1
11	0	0	0
10	-	-	-

D1 = Y1+ = Y1'.Y2
D2 =Y2+ = Y1'
Z = Y1'

10 State is never reached.

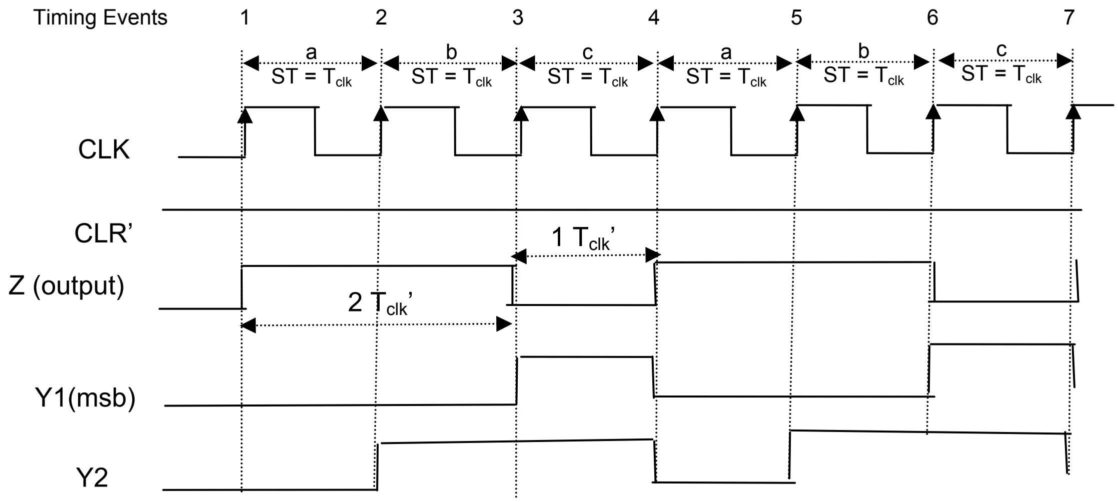
"-" means don't care

Step 5) Draw the Circuit Schematic



❖ A third application of the classical design process (using T flip-flop):
Design a synchronous sequential circuit identical to the previous example, except implement the design using T flip-flops instead of D flip-flops.

Step1) Design Specifications using a Timing Diagram



Step 2) Determine the number of flip-flop based on the number of State
(#state = 3) ≤ $2^{(\text{\#flip-flop} = 2)}$ Assuming Full Coding

Step 3) Assign a Unique code to each state
a: 00, b:01; C:11

Step 4) Write the excitation-input equations:

- The T flip-flop excitation and characteristic equations are $Y^+ = T_{XOR}\ Y$ and $T = Y^+{}_{XOR}\ Y$
  Note: You may derive general excitation equation from re-arranging the characteristic table for the Tflip flop to obtain the excitation table for the T flip-flop as shown below:

T	Y^+
0	Y
1	Y'

Characteristic table

T	Y	Y^+
0	0	0
0	1	1
1	0	1
1	1	0

Output Excitation table

Y	Y+	T
0	0	0
0	1	1
1	0	1
1	1	0

Input Excitation table

$T = Y^+{}_{XOR}\ Y$
Input Excitation Eq.

- Write the PS/NS table (for T & JK, this intermediate step is helpful)

Y_1	Y_2	Y_1^+	Y_2^+	T_1	T_2	Z
0	0	0	1	0	1	0
0	1	1	1	1	0	1
1	1	0	0	1	1	1
1	0	-	-	-	-	-

Unused States

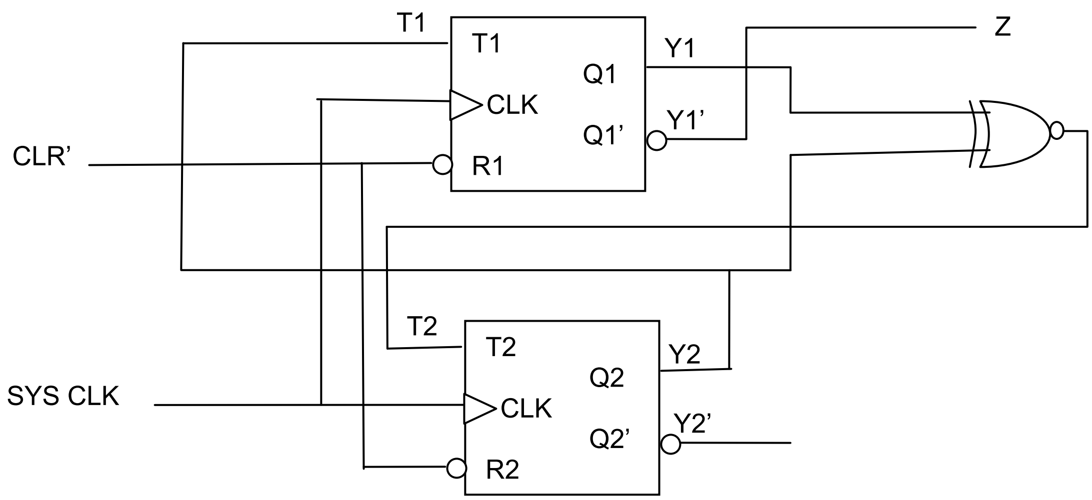
- Draw the composite K-map for each of the desired outputs $Y1^+$, $Y2^+$, Z:

Y1Y2	T1	T2	Z
00	0	1	1
01	1	0	1
11	1	1	0
10	-	-	-

T1 = Y2
T2 = Y1' Y2' + Y1Y2 = Y1 $_{XNOR}$ Y2
Z= Y1'

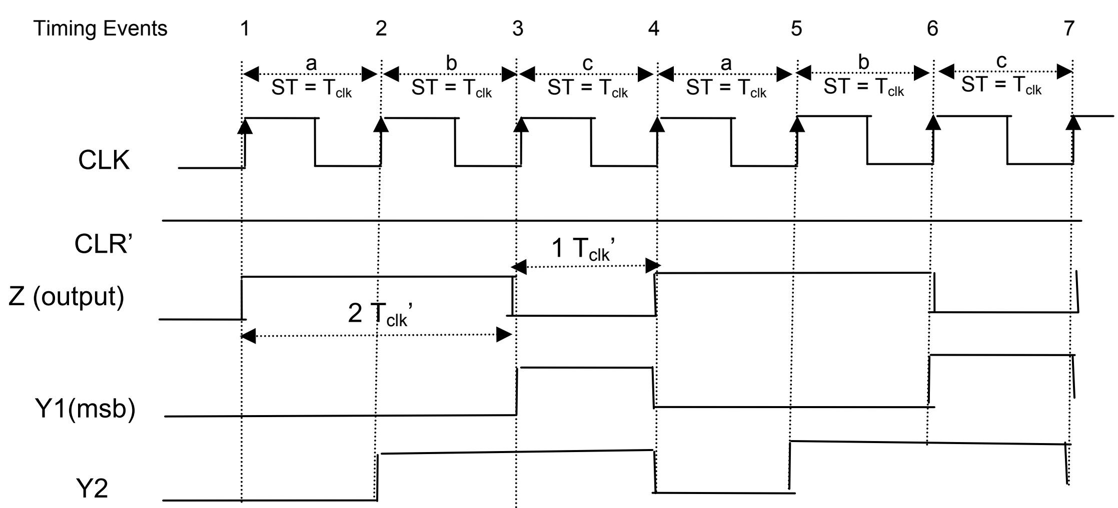
Note: "-" means don't care

Step 5) Draw the Circuit Schematic



- Another Application of the Classical Design Process (using JK flip-flops)
  Design a synchronous sequential circuit identical to the previous example, except implement the design using JK flip-flops.

  Step1) Design Specifications Using a Timing Diagram



Step 2) Determine the number of flip-flop based on the number of states (#states = 3) ≤ $2^{(\#\text{flip-flop} = 2)}$ assuming full encoding.

Step 3) Assign Unique code to each state
a: 00, b:01; C:11

Step 4) Write the excitation-input equations
The JK flip-flop excitation equation is JK → $Y^+ = J.Y' + K'.Y$
You may derive the general excitation equation from the characteristic table for the JK flip-flop to obtain the excitation table for the JK flip-flop, as shown below:

J	K	Y^+
0	0	Y
0	1	0
1	0	1
1	1	Y'

Characteristic table

J	K	Y	Y^+
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

Output Excitation table

Y	Y+	J	K
0	0	0	-
0	1	1	-
1	0	-	1
1	1	-	0

Input Excitation table
Note: "-" = don't care

$J = Y^+$
$K = Y^{+\prime}$
Input-Excitation Eq.

Write the PS/NS table for JK, flip-flops (this intermediate step is helpful)

Y_1	Y_2	Y_1^+	Y_2^+	J_1	K_1	J_2	K_2	Z
0	0	0	1	0	1	1	0	0
0	1	1	1	1	0	1	0	1
1	1	0	0	0	1	0	1	1
1	0	-	-	-	-	-	-	-

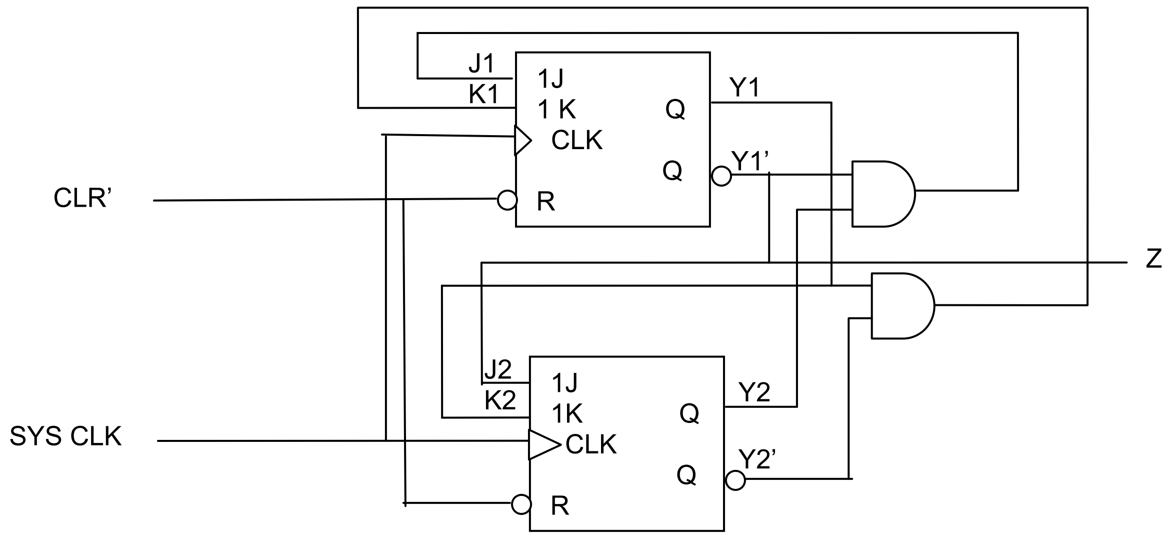
Unused State

Draw the Composite K-map for each of the desired outputs $Y1^+$,$Y2^+$, Z:

Y1Y2	J1	K1	J2	K2	Z
00	0	1	1	0	1
01	1	0	1	0	1
11	0	1	0	1	0
10	-	-	-	-	-

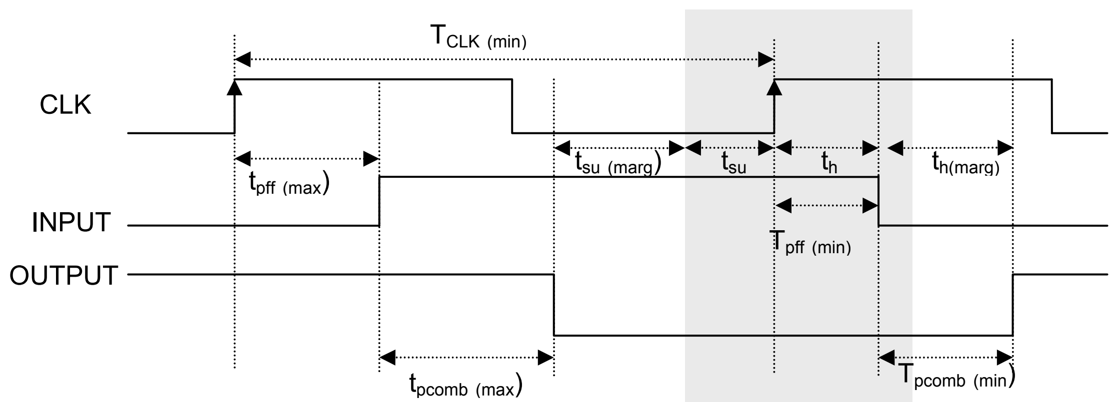
$J_1 = Y1'.Y2$
$K_1 = Y1 + Y2'$
$J_2 = Y1'$
$K_2 = Y1$
$Z = Y1'$
Note: "-" means don't care

Step 5) Draw the Circuit Schematic



Step 6) Test (with a test plan)
Step 7) Implement

❖ Determining the Maximum Clock Frequency of a Synchronous State Machine
The maximum clock frequency that a system can handle is driven by the set-up, hold and margin times required by the flip flops in the synchronous system.



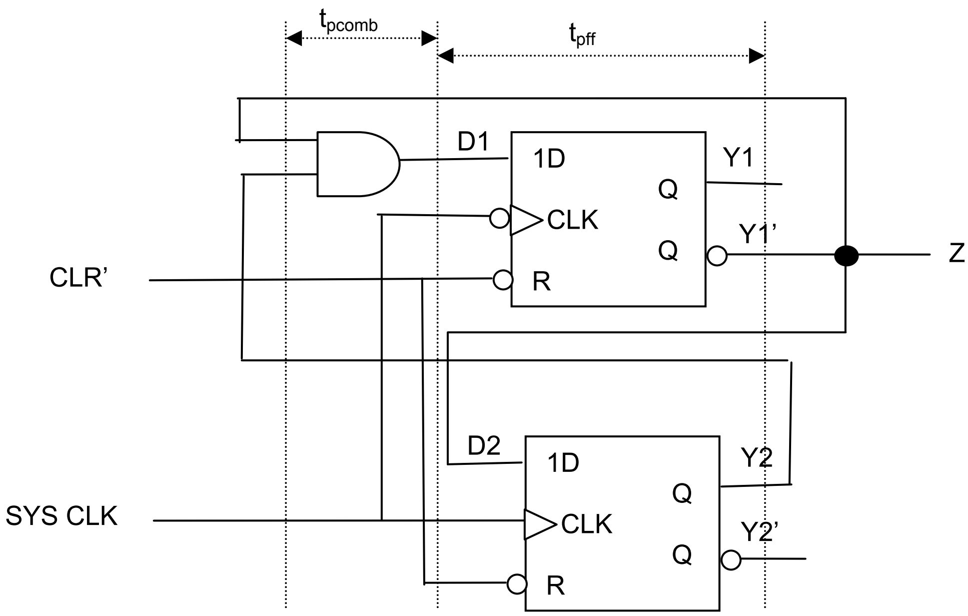
We can see that the clock frequency is limited by $f_{max} = 1/T_{CLK(min)}$ as shown below:

$T_{CLK(min)} = t_{pff(max)} + t_{pcomb(max)} + t_{marg)} + t_{su} + t_h$ where

$t_{pff(max)}$ = Maximum propagation delay time through flip-flop from the clock tick to Q output
$t_{comb(max)}$ = Maximum propagation delay time through combinational logic
$t_{marg)}$ = Margin time, it is always a good design practice to allow for tolerances.
t_{su} = Set-up time requirement
t_h = Hold time requirement

Note: We assume that $t_h + t_{h(marg)} < t_{pff(min)} + t_{pcomb(min)}$

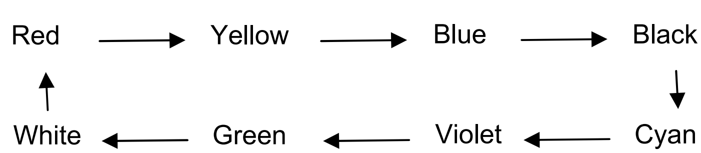
➢ EXAMPLE - Timing
Determine the absolute maximum clock frequency for the divide-by-3 synchronous machine



- 74LS08 AND gate
  t_{pcomb}: Min at 3 ns and Max. at 18 ns
- 75LS175 D-flip-flop
  t_{pff}: Min at 0 ns and Max. at 42 ns
  t_{su}: Min at 20 ns
  t_h: Min at 0 ns
- The fastest clock speed
  $T_{CLK(min)} = t_{pff(max)} + t_{pcomb(max)} + t_{marg)} + t_{su} = 42 + 18 + 0 + 20 = 80$ ns
  $F_{CLK(max)} = 1/T_{CLK(min)} = 1/80*10^{-9} = 12.5$ MHz

  12.5 MHz is significantly slower than today's technology, where the average personal computer clock frequency is many GHz.

➢ Example – Design
Design a system (Finite State Machine, FSM) that cycles through the following colors as shown below:



5.3. State Assignment Encoding, Shift Register Counters, and Adding an Enable Input

- Full-encoding compared to one-hot encoding

    - Full encoding uses all possible combinations of flip flop outputs to represent states, so the equation $2^{\#\text{flip-flop}} \geq$ #states is used to determine the number of flip-flops required. Full encoding:

        - leads to minimum number of Flip Flops.
        - best used with Simple Programmable Logic Devices (SPLDs) and Complex Programmable Logic Devices (CPLDs).

    - One-hot encoding, on the other hand, allows only one flip-flop outputs to be active (or "hot") at any one time. So the equation #flip-flop = #states is used to determine the number of flip-flops required. One-hot encoding:

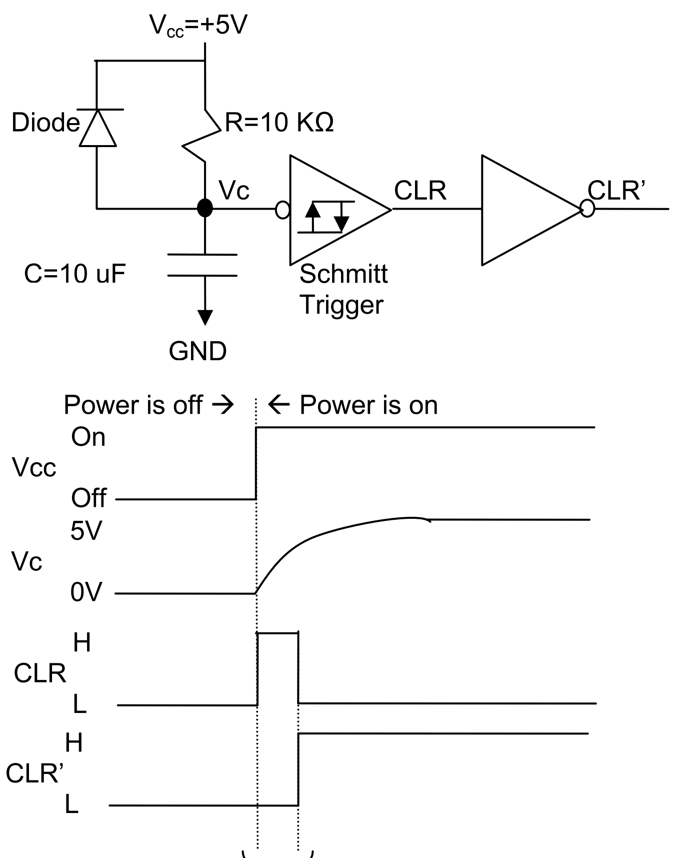
        - leads to larger number of flip-flops.
        - best used with Field Programmable Gate Arrays (FPGAs). FPGAs, which are sometime referred to as "a sea of flip-flops", has made the use of one-hot encoding a viable approach due its overabundance of flip-flops.

- Power-on Reset Circuit
  With either type of encoding there may be illegal and/or unreachable states. Additionally, when your system is turned on initially or regains power after an interruption, it is important for it to recover in a predefined state.

  A power-on reset circuit ensures that a reset is generated immediately after a power up condition. This could be used to preset or reset flip flops into the desired state.

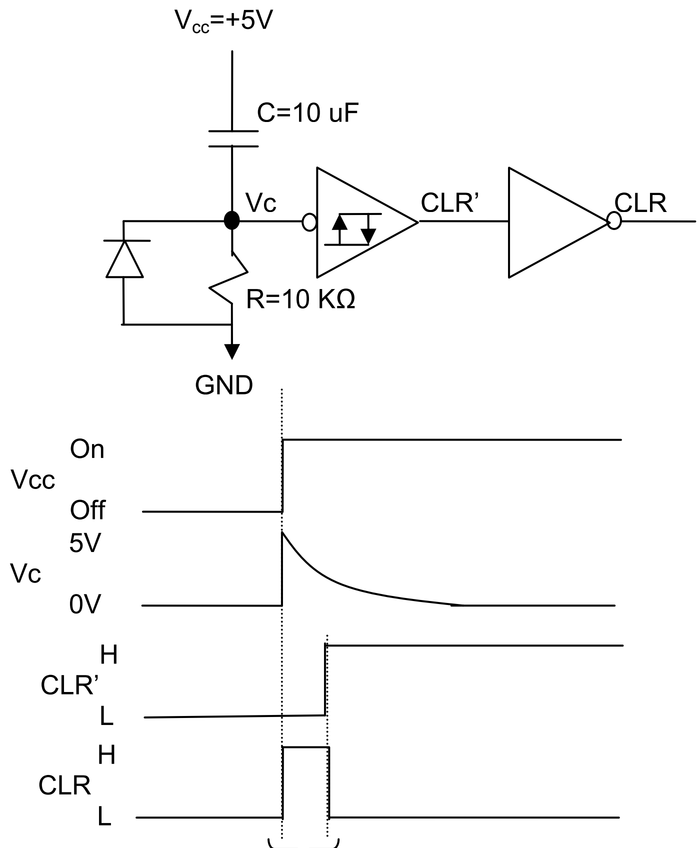
  Using RC circuits, we can design circuits that generate active high or low signals, depending on our needs, as shown below:

RC with C grounded



Time it takes the Vc = $5(1- e^{-t/RC})$ to trigger the Schmitt trigger

RC with R grounded



Time it takes the Vc = $5(e^{-t/RC})$ to trigger the Schmitt trigger

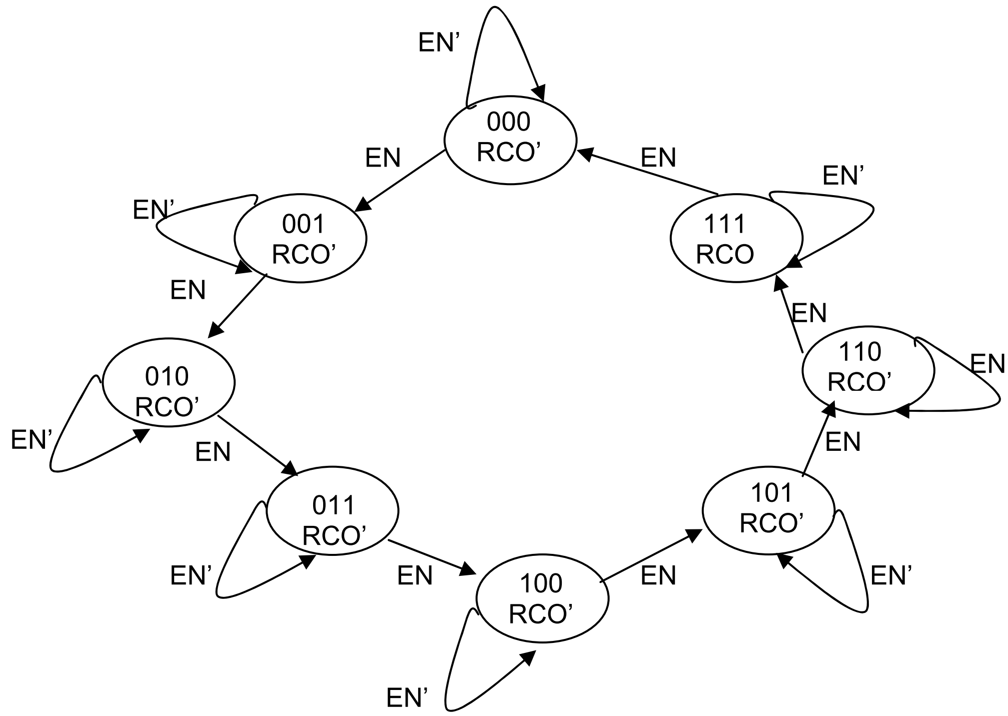
- Additional Types of Shift Registers:
  - Parallel in/Parallel out
  - Parallel in/Serial out
  - Serial in/Parallel out
  - Serial in/Serial out

- Additional Types of Counters:
  - Ring Counters; a 1 is shifted through each flip-flop while all the other flip-flop outputs are 0. (one hot encoding is a recommended design)
  - Twisted Ring Counters (or switch tail ring counter, Johnson counter, mobius counter)
  - Linear Feedback Shift Register Counter (or maximum length shift counters) Depending on design it will count all possible states, but skips all 0s and 1s states.

  Recommendation: Reader is encouraged to explore full definition of these counters and others through independent research.

- Adding an Enable Input
  It may be necessary to stop the count at times and then continue counting. In this section we will design a "Full-Encoded Stoppable Counter". This counter will count up as long as EN is asserted; otherwise it will stop the counting.

- State Diagram for a three-bit ($Y_1Y_2Y_3$) Full-Encoded Stoppable Counter



  - Most standard counters such as 74XX160, 74XX161, 74XX162, 74XX163 have similar designs.
  - RCO can be used to enable the next counter in the cascade (if one exists) to start counting.

- Below is a composite K-map for a 3-bit binary up stoppable counter with enable input EN, asynchronous clear input CLR, and ripple-carry out RCO.

EN \ $Y_1Y_2Y_3$	000	001	011	010	100	101	111	110	
0	000	001	011	010	100	101	111	110	$Y_1^+\ Y_2^+\ Y_3^+$
1	001	010	100	011	101	110	000	111	
	0	0	0	0	0	0	1	0	RCO

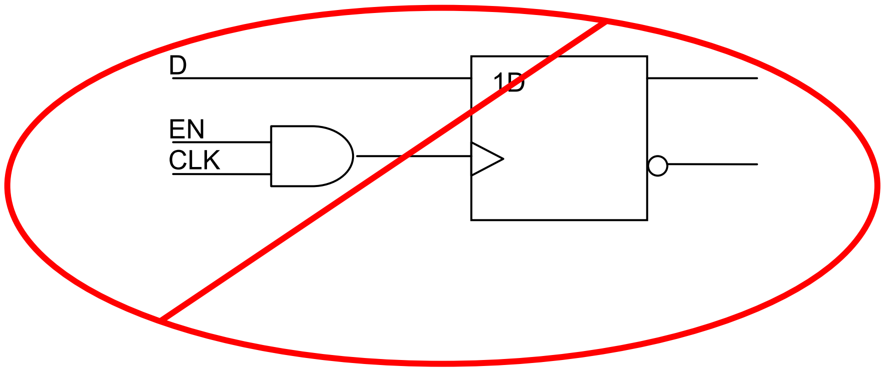
Note: CLR=1 → $Y_1Y_2Y_3$ = 000

The flip-flop input excitation equation and RCO output equation can be derived from the composite K-map or (need 3 flip-flops):

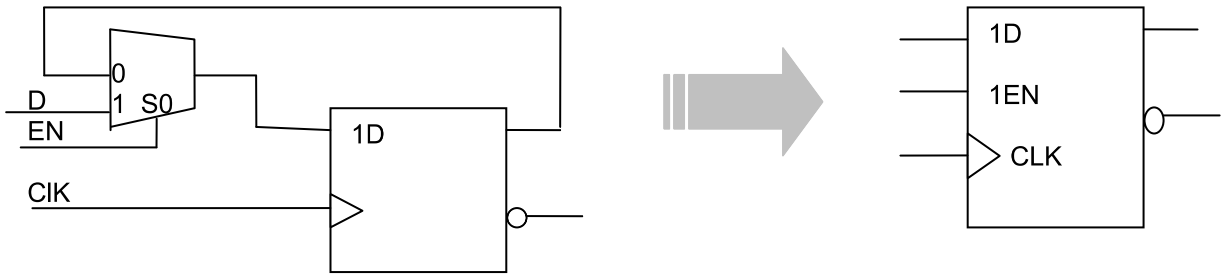
$D1=Y1^+ = EN.Y1'.Y2.Y3 + Y1.Y2'+Y1.Y3'+EN'.Y1$
$D2=Y2^+ = EN.Y2'.Y3 + Y2.Y3'+ EN'.Y2$
$D3=Y3^+ = EN.Y3 + EN'.Y3$
$RCO = Y1.Y2.Y3$

- This counter can be designed with one-hot encoding using 8 flip flops.

❖ Using Enable in Synchronous circuits
In order to maintain the benefits of a synchronous system (avoiding clock glitches), it is important that the clock to all of the components be the same (uninterrupted). Here is what NOT TO DO:



Instead, if you need to enable a flip-flop, use one with enable capability designed in or use the MUX as shown:



Flip-flops with enable allows the designers to focus on the input/output synchronization. Enabled flip-flops simply require a connection to the enable pin, similar to the Clear and Preset signals.

5.4. Inspection Design Methods for Finite State Machines

The classical design methods are limited to a small number of inputs, states and outputs since the K-maps required become too difficult to draw and work with.

The Inspection Design Method provides ways to write the excitation equation for flip-flops by inspection from a timing diagram, a state diagram, or ASM chart of a synchronous Finite State Machine. By observing or inspecting the present state (PS) and next state (NS) for each state variable, the D, T and J-K excitation equations can be written.

The equations derived using inspections are not typically minimum equations. There are two inspection methods:

- Set-Hold 1 Method
  or
- Clear-Hold 0 Method

❖ Set - Hold 1 Method for obtaining **D** excitation-input equations
We use the following table to write D excitation equations directly from a state diagram, ASM chart or timing diagram.

Present State (PS/NS) Yi	**Yi^+**	**Di**	**Comment**	**User for 1s (Set-Hold 1)**	**Use for 0s (Clear-Hold 0)**
0	0	0	Hold 0 transition		Di’
0	1	1	Set transition	Di	
1	0	0	Clear transition		Di’
1	1	1	Hold 1 transition	Di	

- The “Set-Hold 1 Method” can be used to obtain the D excitation equations for the 1s of each state variable (flip-flop outputs)
  Di = ∑ (PS**.**external input conditions for set) + ∑ (PS**.**external input conditions for hold 1)
  for i=1, 2, 3…

  Note: This method solves for the 1’s of the function.

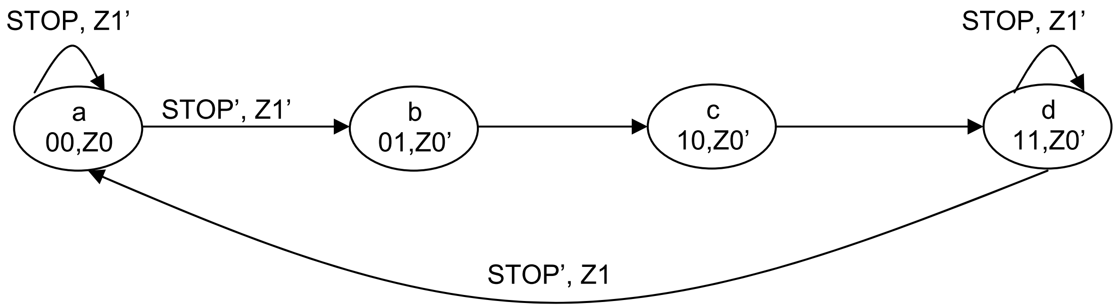
- We could also apply the “Clear-Hold 0 Method” to obtain the D excitation equations for the 0s of each state variable (flip-flop outputs)
  Di’ = ∑ (PS**.**external input conditions for clear) + ∑ (PS**.**external input conditions for hold 0)
  for i=1,2,3,…
  Note: This method solves for the 0’s of the function and it is equivalent to the first method.

For both of the methods, if we have not completely specified FSM meaning and some state values are don’t care, enter them as such so that we can use them in later reduction processes.

➢ Example - Obtaining the D excitation-input equations from a state diagram
Obtain the excitation equations for the following state diagram of a mixed (Mealy-Moore) machine.

State → Y1Y2
Input → STOP

Output → Z0 Z1



- By observing (or inspecting) all set transitions (Y1 =0 → $Y1^{+}$=1) and all Hold 1 transitions (Y1 =1 → $Y1^{+}$=1) we can write the D1 excitation equation from the state diagram:
  D1 = Y1’.Y2 + Y1.Y2’ + Y1.Y2.STOP

- Repeat the previous step for D2 using Y2 transitions
  By observing (or inspecting) all transitions (Y2 =0 → $Y2^{+}$=1) and all Hold 1 transitions (Y2 =1 → $Y2^{+}$=1) we can write the D1 excitation equation, from the state diagram:
  D2 = Y1’.Y2’.STOP’ + Y1.Y2’ + Y1.Y2.STOP

  Note: We could also look for the 0’s function using Clear-hold 0 method to find D1’ and D2’

- Based on the state diagram Z0 is a Moore-type output since it only depends on the state variables (flip-flop outputs).

  We will use a K-map with state variables to find minimized the Z0 equation.

Y1 \ Y2	0	1
0	1	0
1	0	0

Z0 = Y1’.Y2’

Z1 is a Mealy-type output since it depends on both the state variables and external input
We will use a K-map with state variables plus external input to find minimize Z1 equation

STOP \ Y1Y2	00	01	11	10
0	0	0	1	0
1	0	0	0	0

Z0 = Y1.Y2.STOP’

➢ Example - Design a 2-bit up-and-down counter using the inspection design Method.
  - Draw system diagram

- Draw the Present/Next State Table

- Write the Excitation-Input Equations
  Di = Σ(PS.external input for set) + Σ(PS.external input for hold 1)

- Draw the schematics

➢ Set-Clear Method for obtaining **T** Excitation-Input Equations
The following table will be used to write T excitation equations directly from a state diagram, ASM chart, or a timing diagram.

Present State (PS/NS) **Yi Yi$^+$**	**Ti**	**Comment**	**User for 1s (Set-Hold 1)**	**Use for 0s (Clear-Hold 0)**
0 0	0	Hold 0 transition		Ti'
0 1	1	Set transition	Ti	
1 0	1	Clear transition	Ti	
1 1	0	Hold 1 transition		Ti'

- The "Set – Clear Method" can be used to obtain the T excitation equations for the 1s of each state variable (flip flop outputs)
  Ti = ∑ (PS**.**external input conditions for set) + ∑ (PS**.**external input conditions for clear)
  for i = 1,2,3,…

  Note: This method solves for the 1's of the function.

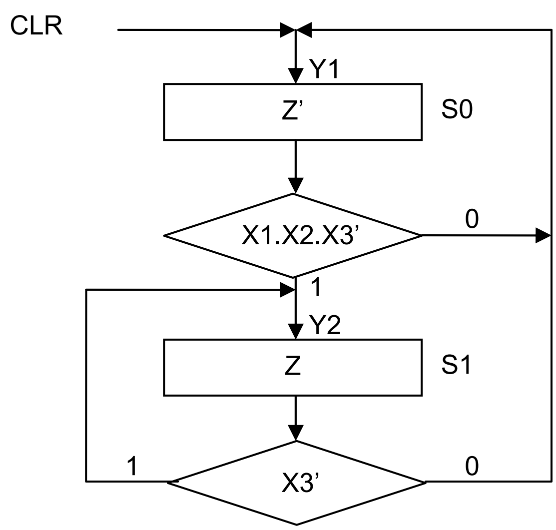
- We could also apply the "Hold 0 - Hold 1 Method" to obtain the T excitation equations for the 0s of each state variable (flip flop outputs)

Ti' = ∑ (PS**.**external input conditions for Hold 0) + ∑ (PS**.**external input conditions for hold 1)
for i = 1,2,3,…

Note: This method solves for the 0's of the function and it is equivalent to the first method.

- Example of obtaining T Excitation-Input Equations from an ASM Chart
  Obtain the excitation equations for the one-hot encoded synchronous Moore-type state machine from the following ASM Chart.

  State → Y1Y2 (S0=10 and S1=01 are used and all others are unreachable)
  Input → X1 X2 X3
  Output → Z



  - By observing all the sets (Y1 =0 → $Y1^+$=1) and all clears (Y1 =1 → $Y1^+$=0), we can write the T1 excitation equation, from the state diagram:

    T1 = Y2.X3 + Y1.(X1.X2.X3')

  - Repeat the previous step for T2 using Y2 transitions
    By observing all the sets (Y2 =0 → $Y2^+$=1) and all clears (Y2 =1 → $Y2^+$=0), we can write the T2 excitation equation, from the state diagram:

    T2 = Y2.X3 + Y1.(X1.X2.X3')

    Note that T1 and T2 were the same. This is not the norm, and just occurred for this machine.

  - Based on the ASM Chart , this is a Moore machine because the output depends only on the state variables (flip-flop output)
    Z = Y2

- Set – Clear method for obtaining **J-K** Excitation-Input Equations
  The following table will be used to write the JK excitation equations directly from state diagram, ASM chart, or a timing diagram.

Present State (PS/NS) Y_i Y_i^+	Ji Ki	Comment	User for 1s (Set-Hold 1)	Use for 0s (Clear-Hold 0)
0 0	0 -	Hold 0 transition		Ji’
0 1	1 -	Set transition	Ji	
1 0	- 1	Clear transition	Ki	
1 1	- 0	Hold 1 transition		Ki’

**Note: “-“ indicates don’t care

- The “Set – Clear Method” can be used to obtain the J-K excitation equations for the 1s of each state variable (flip-lop outputs)
  Ji = ∑ (PS**.**external input conditions for set) when Yi = 0 for i=1,2,3,…
  Ki = ∑ (PS**.**external input conditions for clear) when Yi = 1 for i=1,2,3,…

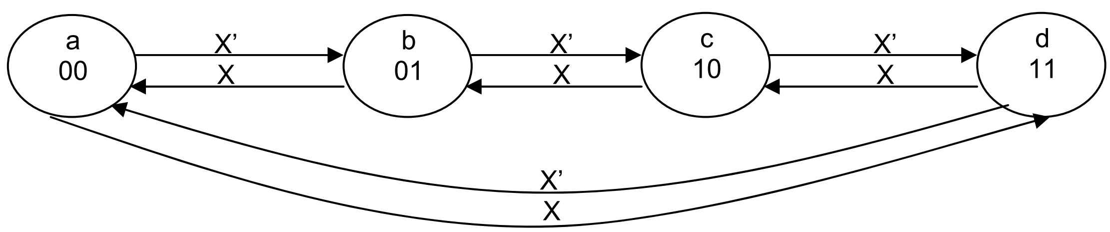
  Note: This method solves for 1’s of the function.

- We could also apply the “Hold 0 - Hold 1 Method” to obtain the T excitation equations for the 0s of each state variable (flip-flop outputs)
  Ji’ = ∑ (PS**.**external input conditions for hold 0) when Yi = 0 for i=1,2,3,…
  Ki’ = ∑ (PS**.**external input conditions for hold 1) when Yi = 1 for i=1,2,3,…

  Note: This method solves for 0’s of the function and it is equivalent to first method.

➢ Example of obtaining J-K excitation Equation from state diagram
Design a synchronous 2-bit Binary up down counter that counts up when input signal X=0 and counts down when input signal x=1

State → Y1Y2
Input → X1



- Use the “Set – Clear Method” to obtain the J-K excitation equations for the 1s of each state variable (flip-flop outputs)
  - By observing all the sets (Y1 =0 → $Y1^+$=1), we can write the J1 excitation equation, from the state diagram:
    J1 = Y1’.Y2.X’ + Y1’.Y2’.X = Y2’.X + Y2.X’
  - By observing all the clears (Y1 =1 → $Y1^+$=0), we can write the K1 excitation equation, from the state diagram:
    K1 = Y1.Y2’.X + Y1.Y2.X’ = Y2’.X + Y2.X’
- Repeat Step 1 for the second Flip Flop
  Use the “Set – Clear Method” to obtain the J-K excitation equations for the 1s of each state variable (flip flop outputs)

- By observing all the sets ($Y2 = 0 \rightarrow Y2^{+} = 1$), we can write the J2 excitation equation, from the state diagram:
  J2 = Y1'.Y2'.X' + Y1'.Y2'.X + Y1.Y2'.X' + Y1.Y2'.X = Y1'.Y2' + Y1.Y2' = Y2'

- By observing all the clears ($Y2 = 1 \rightarrow Y2^{+} = 0$), we can write the K2 excitation equation, from the state diagram:
  K2 = Y1'Y2.X' + Y1'Y2.X + Y1.Y2.X' + Y1.Y2.X = Y1'.Y2 + Y1.Y2 = Y2

5.5. Additional Resources

- Wakerly, I. Digital Design. (2006) Prentice Hall
  Chapter 8 “Sequential Logic Design Practices”

5.6. Problems

1. Design a synchronous counter to provide a BCD output sequence. Provide an asynchronous clear signal CLR to force the counter to state 0000 and force all unused states to state 0000.

   a) Carry out the design using positive-edge-triggered D flip-flops. Draw the circuit diagram.
   b) Carry out the design using positive-edge-triggered T flip-flops. Draw the circuit diagram.
   c) Carry out the design using positive-edge-triggered JK flip-flops. Draw the circuit diagram.

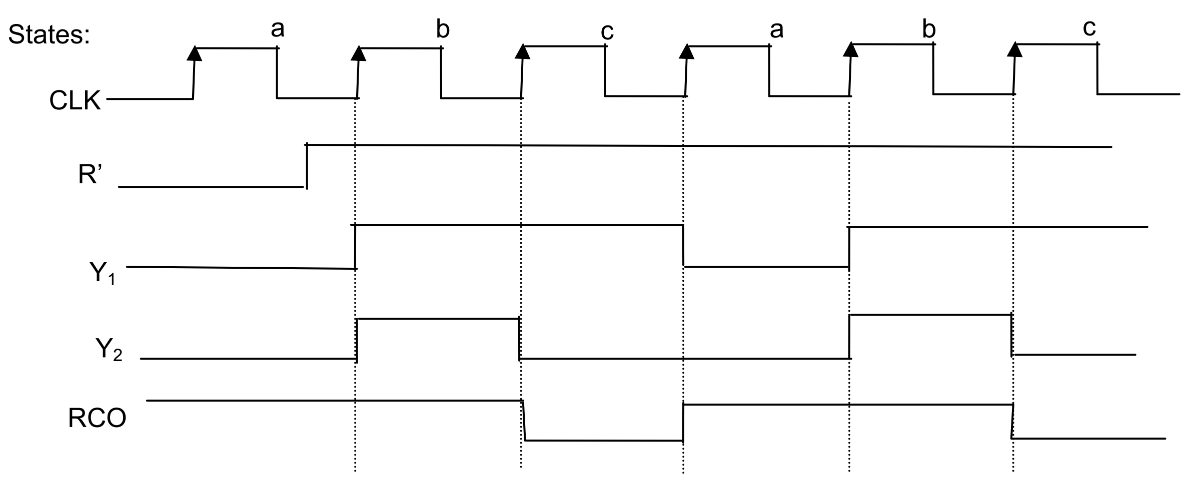
2. Show the state diagram and PS/NS table for a 5-bit binary up-counter design using T flip-flops. Assume that bit position 0 is the Y_0 flip-flop, bit position 1 is the Y_1 flip-flop, etc. Determine T0, T1, etc., show the design using 2-input AND gates.

3. Design a synchronous FSM that performs according the following PS/NS table.

Present State (PS)	Next State (NS) XY 00	XY 01	XY 10	XY 11	Z
a	b	a	a	a	0
b	a	a	b	b	1

   a) Derive the minimum excitation and output equations for a full-encoded design using D flip-flops, and show the implementation.
   b) Derive the minimum excitation and output equations for a one-hot-encoded design using D flip-flops, and show the implementation.

4. Design a synchronous FSM according to the following diagram using rising-edge D flip-flops. Direct any illegal states to state “a”. Asynchronous reset signal R’ returns the counter to state “a”.

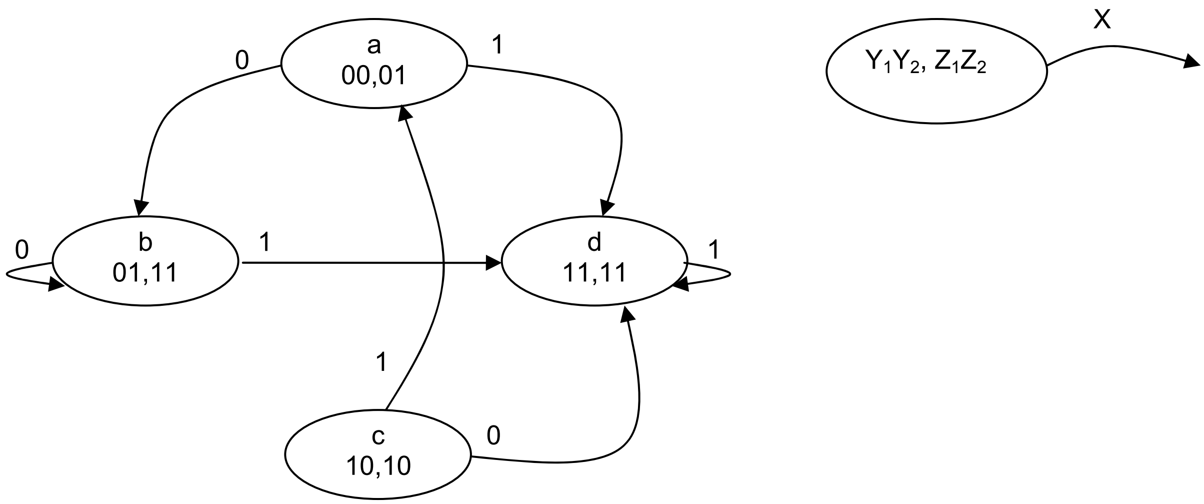


5. Design a synchronous circuit with negative-edge-triggered D flip-flops to provide an output signal Z that is 1/5 the frequency of the system clock CLK.

   a) Draw the timing diagram showing the system clock (CLK) and output (Z).
   b) Obtain the PS/NS table.

c) Obtain the K-map for the design. Use a straight binary counting sequence for the state assignments beginning with a=0, b=1, c=2, etc., and force all unused states to go to 0.

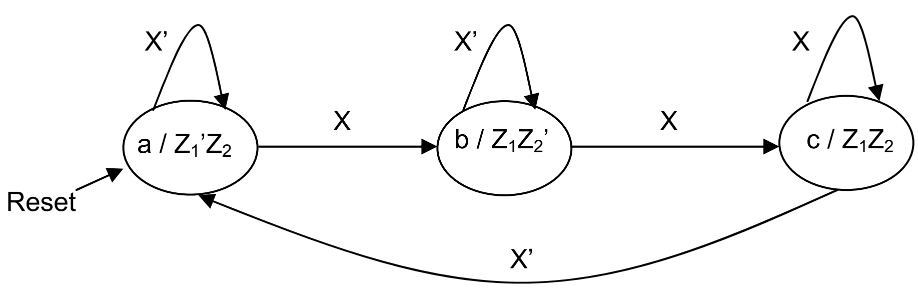
6. Design a synchronous circuit using positive-edge-triggered T flip-flops for the following state diagram.



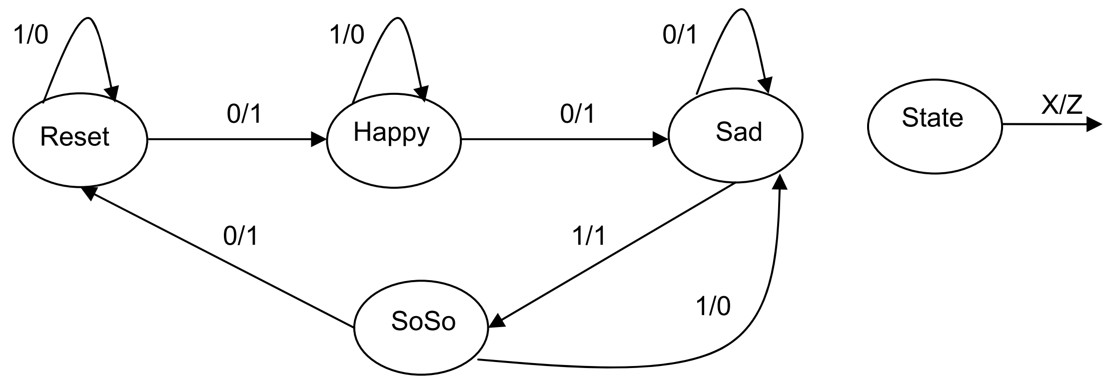
a) Derive the PS/NS table for the machine.
b) Derive the minimum excitation and external output equations.
c) Draw the circuit schematics.

7. Use D flip-Flops to design an automatic room light control that turns the light on only someone is in the room. Light should be off when no one is in the room. Assume the room capacity is 2. (Door has two sensors, one detecting a person entering "in=1" and one detecting a person leaving "out=1"). Start with no one in the room.

8. Design a ILCW sequence detector using the classical design technique, full encoding and rising edge D flip flops. ILCW operation is described by the following state diagram when X is the input and Z_1 Z_2 is the 2-bit output. *Show your work for each of the classical design steps.*



9. Design a circuit (FSM) that performs the function described by the following state:



a) Using D-ff
b) Using JK –ff
c) Using T-ff
d) Using SR-ff

10. Design a 3-bit binary up counter that counts from 2 to 6 with roll over from 6 to 2.

a) Using D-ff
b) Using JK–ff
c) Using T-ff
d) Using SR-ff

11. Design a 2-bit gray code counter (00 → 01 → 11 → 10 → 00 → …).

a) Using D-ff
b) Using JK–ff
c) Using T-ff
d) Using SR-ff

Chapter 6. Finite State Machine Optimization & Testing

6.1. Key concepts and Overview

- State Minimization and FSM Design Process
- State Minimization/Reduction Using Implication Chart (or Table)
- Design for Testability (Scan test, Linear Feedback Shift Register and primitive Polynomials)
- Additional Resources
- Problems

6.2. State Minimization and FSM Design Process

The state minimization is done after the fourth step of the seven steps of Finite State Machine (FSM) classical design:

1) Organize the Design Specifications –Using one or more of the following:
   Timing Diagram, State Diagram, ASM Chart or Present State/Next State (PS/NS) table

2) Determine the number of flip-flops based on the number of states.
   Full encoding → $2^{\#\text{flip-flop}} \geq \#$ states
   or
   one-hot encoding → #flip-flop = # States

   Next assign one present state variable to each flip-flop output.

3) Assign a unique code to each state (a specific value for present-state variables).

4) Select the flip-flop type to be used, then determine the excitation input equations and the Moore and/or Mealy output equations.

   The excitation-input equations for common flip-flops are shown below:
   JK → $Y^+ = J.Y' + K'.Y$
   T → $Y^+ = T_{XOR}\ Y$
   ….D → $Y^+ = D$

"State Minimization using implication Chart is used at this point"

5) Draw the circuit schematic (pencil/paper or CAD tools).

6) Perform a simulation to test the functionally of the design.

7) Implement the design in hardware.

6.3. State Minimization Using an Implication Chart (or Table)

The Implication Chart Method is a systematic approach to find the states that can be combined into a single reduced state. This method is cumbersome to do by pencil and paper, but it is well-suited for automation because it is a systematic approach.

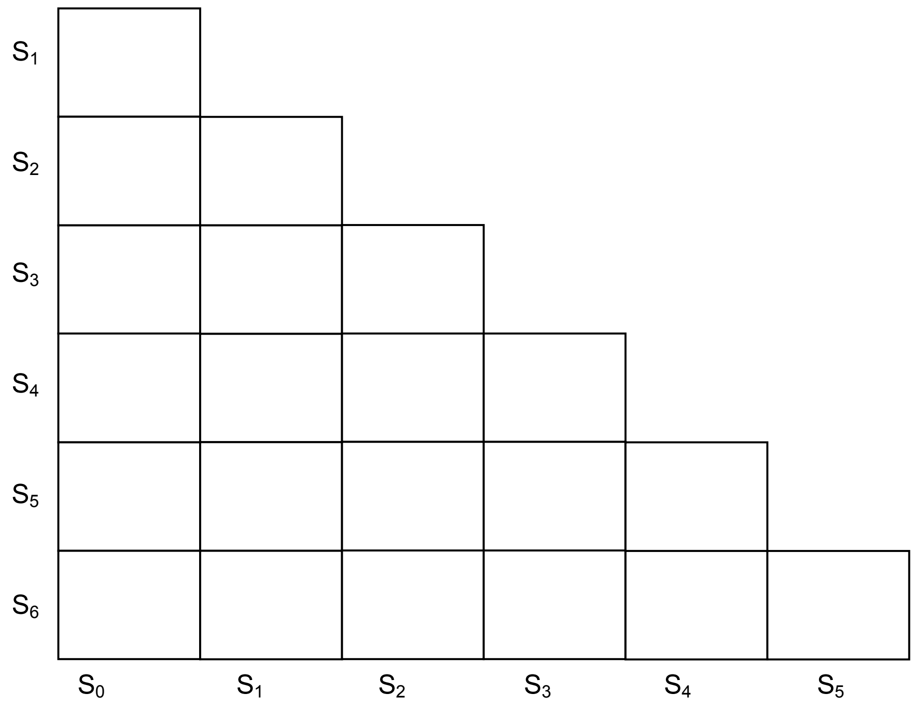
- Minimization procedure with an Implication Chart
  A 3-bit sequence detector example is used here to demonstrate the Implication Chart use in state minimization.

  - Problem Statement
    Design a binary sequence detector with the minimum number of states that outputs a 1 whenever the machine has observed the serial input sequence 010 or 110.

  - Step 1) Use the problem statement to write the Present/Next State Table
    (It may help to first do a state diagram.)

		Next State		Output	
Input Sequence	**Present State**	**X=0**	**X=1**	**X=0**	**X=1**
Reset	S_0	S_1	S_2	0	0
0	S_1	S_3	S_4	0	0
1	S_2	S_5	S_6	0	0
00	S_3	S_3	S_4	0	0
01	S_4	S_5	S_6	1	0
10	S_5	S_3	S_4	0	0
11	S_6	S_5	S_6	1	0

  - Step 2) Draw an implication Chart which allows entries relating every state with every other state as shown below:
    - Label vertically from last state (S6) to second state (S1)
    - Label horizontally from first state (S0) to next to the last state (S6)



In general for an n-state machine, we will have $(n^2 - n)/2$ cells. Each of the cells in the implication chart relates State Sj with State Si.

Note: The order is not important.

- Step 3) Fill-in each cell (Xij) in the implication table with one of the following two options:
  - **X** if the Si and Sj have different outputs.
    (state output for Moore machine and transition output for the Mealy machine)
  - Transition states for Sj and Si for each of the inputs
    This means that the next states for all possible inputs must be equivalent for these states to be equivalent.

After the application of previous two rules we will end up with the following table.

S_1	$S_1 - S_3$ $S_2 - S_4$					
S_2	$S_1 - S_5$ $S_2 - S_6$	$S_3 - S_5$ $S_4 - S_6$				
S_3	$S_1 - S_3$ $S_2 - S_4$	$S_3 - S_3$ $S_4 - S_4$	$S_5 - S_3$ $S_6 - S_4$			
S_4	X	X	X	X		
S_5	$S_1 - S_3$ $S_2 - S_4$	$S_3 - S_3$ $S_4 - S_4$	$S_5 - S_3$ $S_6 - S_4$	$S_3 - S_3$ $S_4 - S_4$	X	
S_6	X	X	X	X	$S_5 - S_5$ $S_6 - S_6$	X
	S_0	S_1	S_2	S_3	S_4	S_5

Mean S0 → S1 and S1 → S3 when X=0.

Mean S0 → S2 and S1 → S4 when X=1.

Note: At this stage, many of the states have been eliminated.

- Step 4)
  We go through the chart repeatedly until a complete pass can be done through the chart without making any additional X markings.

  - First Marking Pass – we are looking for cases where the dependencies are not valid.

    For example, for States S0 and S1 to be equivalent, we must have S1 – S3 equivalent and S2 –S4 equivalent.

    Since the cells relating S2 and S4 are crossed out, then the S0 and S1 cell must be crossed out. Continue this process (top-down and left to right) through the chart.

S_1	~~$S_1 - S_3$ $S_2 - S_4$~~					
S_2	~~$S_1 - S_5$ $S_2 - S_6$~~	$S_3 - S_5$ $S_4 - S_6$				
S_3	~~$S_1 - S_3$ $S_2 - S_4$~~	$S_3 - S_3$ $S_4 - S_4$	$S_5 - S_3$ $S_6 - S_4$			
S_4	X	X	X	X		
S_5	~~$S_1 - S_3$ $S_2 - S_4$~~	$S_3 - S_3$ $S_4 - S_4$	$S_5 - S_3$ $S_6 - S_4$	$S_3 - S_3$ $S_4 - S_4$	X	
S_6	X	X	X	X	$S_5 - S_5$ $S_6 - S_6$	X
	S_0	S_1	S_2	S_3	S_4	S_5

- Second marking pass.
  Repeat the process with the resulting chart from the previous pass.

S_1	X					
S_2	X	$S_3 - S_5$ $S_4 - S_6$				
S_3	X	$S_3 - S_3$ $S_4 - S_4$	$S_5 - S_3$ $S_6 - S_4$			
S_4	X	X	X	X		
S_5	X	$S_3 - S_3$ $S_4 - S_4$	$S_5 - S_3$ $S_6 - S_4$	$S_3 - S_3$ $S_4 - S_4$	X	
S_6	X	X	X	X	$S_5 - S_5$ $S_6 - S_6$	X
	S_0	S_1	S_2	S_3	S_4	S_5

In this pass, no change was made to the table so this is the last pass. The table indicates that the following states are equivalent:

- S_0 does not have equivalent, so it will need a new designator Y_0 state
- S_1 – S_2 – S_3 – S_5 are equivalent so they can be called Y_1 state
- S_4 – S_6 are equivalent so they both can be called Y_2 state

- Step 5) Present/Next State Table for the minimized state machine:

Input Sequence	Present State	Next State		Output	
		X=0	X=1	X=0	X=1
Reset	Y_0	Y_1	Y_1	0	0
00,01 or 10	Y_1	Y_1	Y_2	0	0
01 or 11	Y_2	Y_1	Y_2	1	0

6.4. Design for Testability (DFT)

During the design phase, you need to consider the testing needs. Here are a few key types of testing to consider:

- Go/No Go Testing
  The goal of this test is to ensure that the product is functional before delivering it to the customer. This type of test indicates whether the product is functional and can be shipped or not.

- Diagnostic Test
  As the name implies, this test is typically used to find which subsystem is failing, so it can be replaced or repaired. This type of test benefits from testability consideration during the design phase.

  With the proper attention to Design For Testability (DFT), the diagnostic test will:

  1) Be easier to develop.
  2) Be more effective in finding problems earlier.
  3) Reduce downtime, and may even test while the system is operating, which leads to failure prediction.
  4) Reduce cost of a failed product in production phase as well as within warranty.

- Testing
  Digital designs are tested by applying test vectors, which are a set of input values and expected output values.

  - Simplification Assumptions
    In the worst case scenario, we require 2^n vectors to test an n-input combinational circuit. So, engineers make assumptions about the type of errors in order to simplify the process:

    - Single bit fault
      Here the assumption is that only one bit (or line or pin) may be stuck at 1 or 0 incorrectly.

      Using an 8-input AND gate to demonstrate the benefit of this simplification, instead of needing 2^8 or 256 vectors, we can fully test this circuit with the following nine vectors (walking the 0):

      [11111111] [01111111] [10111111] . . . [11111011] [11111101] [11111110]

    - Test-generation programs
      When the system is more complex, it is hard to impossible to create test vectors by hand. There are programs designed to create test vectors based on circuit design to ensure that the product functions so that all design requirements (customer needs) are met.

      DFT methods attempt to simplify test-pattern generation by enhancing the "controllability" and "observeability" of logic elements in the circuit.

      - In a circuit with good controllability, it is easy to produce any desired values on the internal signals of the circuit by applying an appropriate test-vector input combination to the primary input. You may even add additional inputs just for testing.

      - In a circuit with good observeability, any internal signal value can be easily propagated to a primary output for comparison with the expected output. You may even add additional outputs just for testing.

➢ "Bed-of-Nails" and "In-Circuit" Testing
In a digital circuit that is on a PC board (PCB), most manufacturers use a cushion of probes (nails) that makes contact with every signal in the PCB. Then it can be used to drive through the control points and observe the results at observation points.

Although these devices are expensive, they allow the manufacturer to test a circuit in seconds and have the confidence that all critical circuits operate to the specification.

Agilent and Tektronix are two of the largest in-circuit test solution vendors.

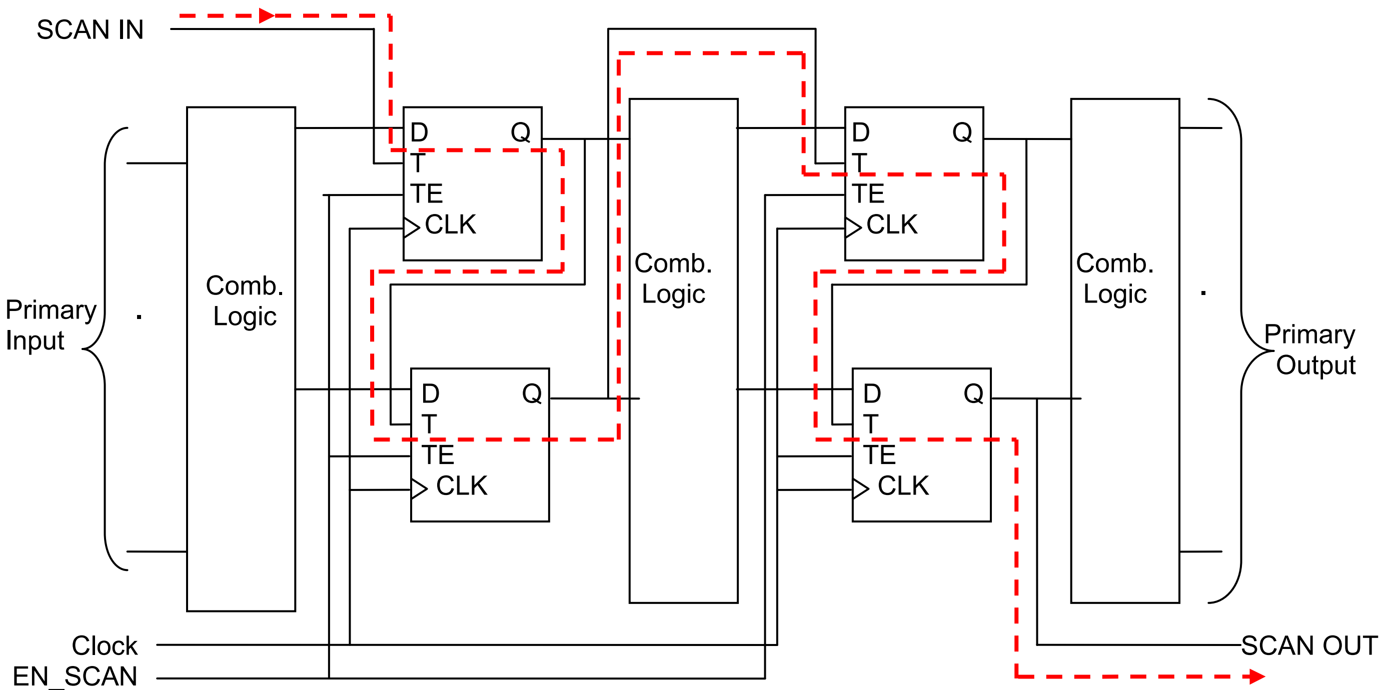
➢ Scan Method
An in-circuit test can not test custom ICs and FPGAs, since internal signals are not accessible. Even with many PCBs, the high-density and surface mounting have limited their effectiveness.

A scan method attempts to control and observe the internal signals of a circuit using only a small number of test points.

A scan-path method considers any digital circuit to be a collection of flip-flops or other storage elements interconnected by combinational logic.

The basic idea of a scan test is to control and observe the state of storage elements. It does this by providing a normal operation mode and a separate scan operation mode where the storage elements are reorganized into a giant shift register (Linear Feedback Shift Register) to test the storage elements

Here is a sample:



Note: Heavier dashed lines indicate the Scan Path

Note: For a more robust scan test, input pattern are designed using Primitive polynomials. Each polynomial offers a different level of coverage and error detection. This area represents an opportunity for further research by the reader.

6.5. Additional Resources

- TBC

6.6. Problems

1. Identify the equivalent states using the shown implication table for a single input sequential circuit.
Note: "X" in a cell indicates that the corresponding states are not equivalent

S_1	$S_0–S_1$ $S_4–S_2$			
S_2	$S_2–S_1$ $S_3–S_2$	$S_2–S_2$ $S_0–S_4$		
S_3	$S_2–S_2$ $S_1–S_2$	$S_1–S_2$ $S_0–S_2$	X	
S_4	$S_2–S_3$ $S_1–S_1$	X	$S_3–S_3$ $S_2–S_2$	$S_3–S_1$ $S_1–S_2$
	S_0	S_1	S_2	S_3

2. Design a binary sequence detector with the minimum number of states that outputs a "1" whenever the machine has observed the serial input sequences "0110" or "1010".Use an implication table to minimize the number of states.

3. Design a mod 6 up counter with the minimum number of states. Apply the implication chart method to find the most reduced state diagram.

4. Identify a 4-bit system with its Present/next State Diagram. The selected system should show a reduction of more than 20% in the number of states compared with the initial Present/Next State Table after the application of Implication Chart Method.

Chapter 7. Hardware Description Language(HDL)

7.1. Key concepts and Overview

- History
- Steps in HDL design
- Architecture and Program Structure
- Declarations
- Operators
- Structural Design Elements
- Behavioral Design Elements
- Dataflow Design Elements
- Additional Resources
- Problems

7.2. History

Hardware Description Language (HDL) is used by designers to describe circuit functionality in high level language. This VHDL design is then processed to an implementable hardware circuit design.

The two main HDL solutions on the market are Verilog Hardware Design Language (Verilog) and Very high-speed integrated circuit Hardware Design Language (VHDL). Although Verilog came on the market much earlier than VHDL, they both have equal market share currently.

- Verilog
  - Introduced by Gateway Automation in 1984 as a proprietary language.
  - Purchases by Synopsis in 1988 which was eventually purchased by Cadence Design Systems.
  - Cadence Design System has successfully has successfully market Verilog to a market power house.
  - The syntax is similar to C language.

- VHDL

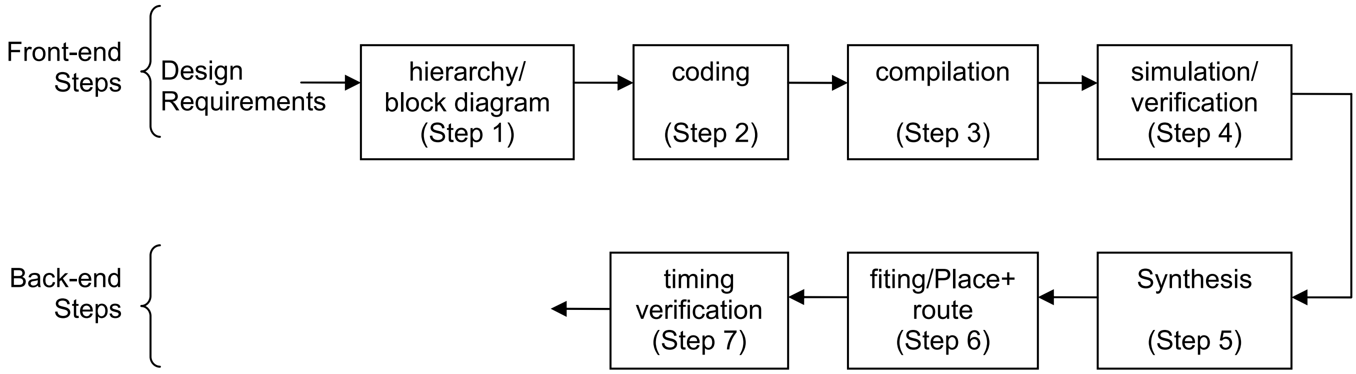
  The US department of Defense (DOD) and the IEEE sponsored the development of VHDL – Standardized by IEEE in 1993.
    - Design may be decomposed hierarchically.
    - Each design element has a well-defined interface and a precise behavioral specification.
    - Behavioral specification can use either an algorithm or by a hardware structure.
    - Concurrency, timing, and clocking can all be modeled (asynchronous & synchronous sequential circuit).

This chapter focuses on VHDL exclusively. There are sufficient similarity in structure and concepts between VHDL and Verilog that learning Verilog is expected to be a simple process.

7.3. Steps in VHDL design

The process of design may be divided into front-end and backend. Where:

- The Front-end section includes all the decision are made and the design is documented.
- The Back-end section includes the implementation and testing and the product.



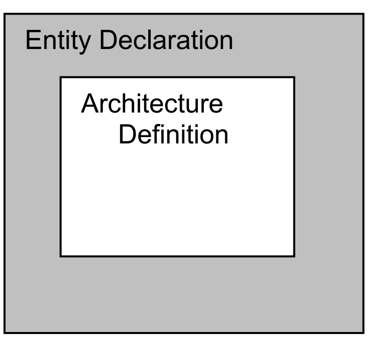
Although this process is iterative by its nature, as the distance between the step that an error is discovered and the step that the correction is made increases, the cost (time & resource) increases exponentially.

- ❖ Program Structure

  - ➢ VHDL was designed with principles of structured programming in mind and borrowed ideas from Pascal and Ada Software Languages.

  - ➢ VHDL Code has two parts (entity & architecture)
    - Entity
      A declaration of a module's inputs and outputs. Entity is viewed as a wrapper for the architecture, hiding what's inside, while providing access for another module to use the functionality.

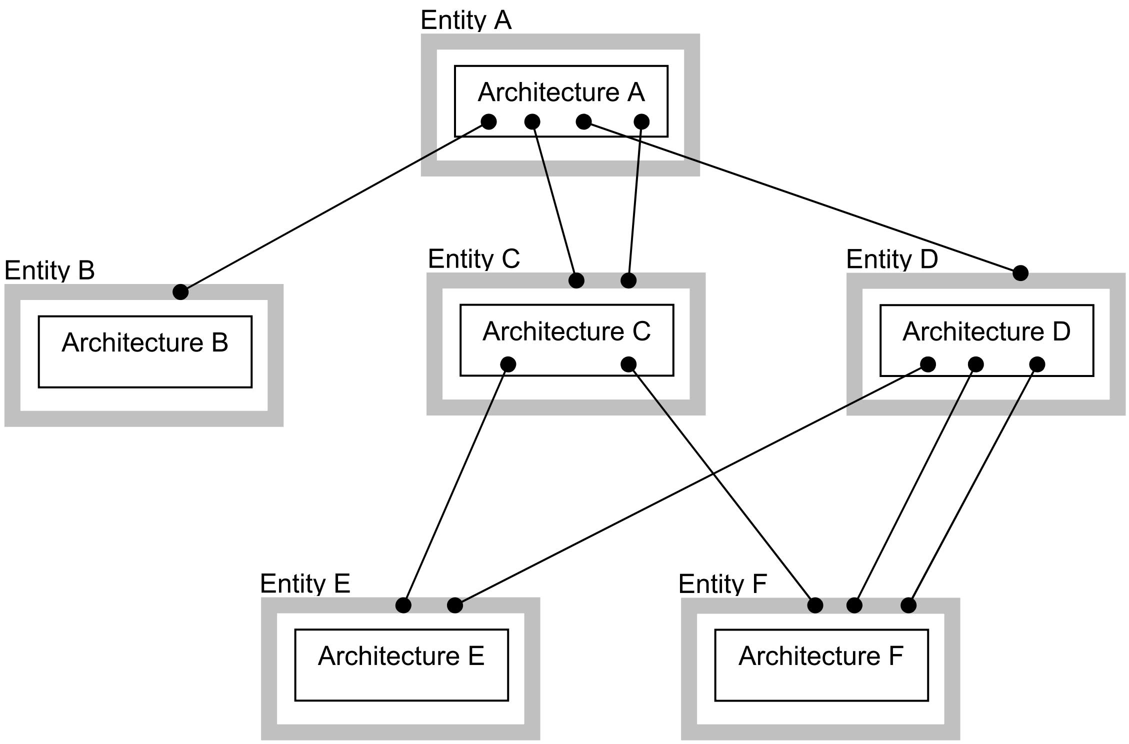
    - Architecture
      A detailed description of the module's internal structure or behavior.




```
-- Eight-bit comparator
entity ent_compare is
        port( A, B: in bit_vector(0 to 7);
         EQ: out bit);
end ent_compare;

architecture arc_compare of ent_compare is
begin
        EQ <= '1' when (A = B) else '0';
end arc_compare1;
```

- ➢ VHDL is hierarchical, meaning that a higher-level entity may use other entities while hiding lower level entities from the higher level ones as shown by the following diagram:



- General VHDL Semantics
  VHDL similar to other languages has many constructs and rules. The following list contain some of the most common Semantics:

  - Code can span multiple lines and files for larger designs.
  - Comment field starts with “--" and ends at the end of line.
  - Each statement must be terminated with a “**;**”.
  - VHDL ignores space and line breaks which allows for readability formatting.
  - VHDL has many reserved words (or keywords) that can not be redefined such as:
    Entity, Port, Is, In, Out, End, Architecture, Begin, When, Else, Not, ...
  - Reserve words and identifiers are not case-sensitive
  - User Defined Identifiers
    These are names used to refer to variables, signals, types, processes, function, types, architecture and entities. User defined identifiers names must adhere to the following requirements:

    - Must begin with a letter and contain letters, digits and underscores.
    - Underscore can not follow each other and can not be the first or last character.
    - Reserve words are not allowed.

7.4. Entity and Architecture

The remainder of this document discusses the VHDL infrastructure, common structures and syntax. The reader is encouraged to use the VHDL development tools such as Active-HDL from Aldec to implement the ideas discussed in this text. Additionally, the reader is encouraged to use the online documentation and help section of these products to explore related capabilities of VHDL.

This section focuses on the core framework of VHDL (Entity and Architecture).

- Entity Declaration
  Entity code describes the system diagram which includes definition of input and output. It does not provide any information on the internal function of the device.

```
entity entity_name is
port (
        signal_names : mode signal_type;
        signal_names : mode signal_type;
        . . .
        signal_names : mode signal_type);
end entity_name;
```

  - "entity_name"
    A user defined identifier to name the entity.

  - "signal_names"
    A comma-separated list of one or more user-selected identifiers to name external-interface signals.

  - "mode"
    "Signal_type" for mode may be set to one of the following four reserved words in order to specifying the signal direction:

    - "in"
      The signal is an input to the entity.
    - "out"
      The signal is an output of the entity. Note that the value of such a signal cannot be "read" inside the entity's architecture, only by other entities that use it.
    - "buffer"
      The signal is an output of the entity, and its value can be also be read inside the entity architecture.
    - "Inout"
      The signal can be used as an input or an output of the entity. This mode is typically used for three-state input/output pins on PLDs.

  - Signal-type
    A built-in or user-defined signal type. Discussed later. Note there is no ";" after the last signal-type.

- Architecture Definition
  Architecture code defines the function of the device. It is highly recommend that pseudo code or other high level design be completed prior to writing the architecture code.

```
architecture architecture_name of entity_name is
        signal declarations;
        type declarations;
        constant declarations;
        function declarations;
        procedure declarations;
        component declarations;
begin
        concurrent_statement;
        . . .
        concurrent_statement;
end architecture_name;
```

- "architecture_name" is a user-defined identifier, and "entity_name" is also a user defined identifier for the entity. The concurrent-statements can appear in any order since they are executed currently. The Declaration statement may also appear in any orders..

7.5. Declarations

- Signal and Variable Declarations
  Signal declaration gives the same information about a signal as in a port declaration, except that mode specification is not required. Syntax for signal declaration is shown below:

  signal signal_names : signal_type;

  Any number of signals can be defined within architecture, and they roughly correspond to the named wires in a logic diagram.

  It is important to note that symbol "**<=**" is used to assign a value to a signal. For example to assign a value of 4 to a signal stemp, it needs to be written as follows:
  stemp **<=** 4;

  VHDL variables are similar to signals except that they do not have a physical significance in a circuit. Variables are used within functions, procedures and processes (not used in architecture definition). The variable declaration syntax is as follows:

  variable variable_name : variable_type;

  It is important to note that symbol "**:=**" is used to assign a value to a variable. For example to assign a value of 4 to a variable vtemp, it need to be written as follows:
  vtemp **:=** 4;

- "type" Declarations
  All signals, variables and constants in a VHDL program must have an associated "type." Each "type" specifies the range of values that object can take on. "type" may be pre-defined or user defined.

  - Pre-Defined Types:

    - Bit
      Takes on '0' and '1' values
    - Bit-vector
      array of bits
    - Boolean
      True, False { EQ <= True;}
    - Integer
      A whole number ranging from $-2^{31}+1$ through $+2^{31}-1$ {count <= count + 2;}
    - Real
      1.0, -1.0E5 {V1 := V2 / 5.3}
    - Character
      All of the ISO 8-bit character set – the first 128 are the ASCII Characters. {CharData <= 'X';}
      Note: The symbol **'** is used for character definition.
    - String
      An array of characters {msg<="MEM:" & Addr;}
      Note: The symbol **"** is used for string definition.
    - Time
      1 us, 7 ns, 100 ps {Q<='1' after 6 ns;}

- Predefined Operators
  VHDL is a strongly typed language which means that the complier issues error messages if types in an operation or assignment do not perfectly match.

  The integer and Boolean Operations are the most commonly used VHDL operation and operands in each operation group must have the correct type in order for the operation to be compiled correctly.

  Following table list some of the most common operations:

Integer Operators		**Boolean Operators**	
+	addition	and	AND
-	subtraction	or	OR
*	multiplication	nand	NAND
/	division	nor	NOR
mod	module division	xor	Exclusive OR
rem	module remainder	xnor	Exclusive NOR
abs	absolute value	not	Complementation
**	exponentiation		

- User-Defined Types
  Although VHDL provides an extensive list of pre-defined types, user may need to define new types using the user-defined type capabilities of VHDL. The flowing pages, describe the most common user-defined type constructs:

  - Numeration
    Numeration enables the user to define a type that can only accept a predefined set of values. The following syntax, allow definition of numeration type and its use to build two different type of arrays:

    type type_name **is** (value_list); -- Value-list is a comma-separated list of all
    -- possible values of the type

    -- create an array of type-name with an ascending order from start to end
    subtype subtype_name **is** type_name **range** start **to** end;

    -- create an array of type-name with a descending order from start to end
    subtype subtype_name **is** type_name **range** start **downto** end;

    - Example – Write a code segment to define an array that starts from 20 to -4 with each element value restricted to either red, green, or blue.

```
type COLORS is (“red”,     -- User-define types are typically in Capital Letters
                “green”,
                “blue”,
                );
subtype my_colors is COLORS range 20 downto -4;
```

    - Example- Define a complete logic type that includes hi-z, weak and forcing.

```
type STD_ULOGIC is (
        ‘U’,    -- Uninitialized
        ‘X’,    -- Forcing Unknown
        ‘0’,    -- Forcing 0
        ‘1’,    -- Forcing 1
        ‘Z’,    -- High Impedance
```

```
'W',    -- Weak        Unknown
'L',    -- Weak        0
'H',    -- Weak        1
'-',    -- Don't care
);
subtype STD_LOGIC is resolved STD_ULOGIC
```

- Array
  The following list represent the most common use of array constructs:

  type type_name **is array** (start **to** end) **of** element_type;
  type type_name **is array** (start **downto** end) **of** element_type;
  type type_name **is array** (range_type) **of** element_type;
  type type_name **is array** (range_type range start **to** end) **of** element_type;
  type type_name **is array** (range_type range start **downto** end) **of** element_type;

- Inside the VHDL program statement array element can be accessed using array name of indices. Note that the leftmost element is the first.

  Examples:
  type monthly_count is array (1 to 12) of integer; -- 12 element array m(5)
  type byte is array (7 downto 0) of STD_Logic; -- 8 element array b(3)
  type statcount is array (traffic_light_state) of integers; -- 4 element array s(reset)

- Array literals can be specified by listing values in parentheses or using one of the pattern shortcuts.

  Examples (N is a 4-bit array):
  N := ('1', '1', '1','1'); -- set all elements to character 1
  N := ("1111"); -- set all elements to character 1

  Examples (B is a 8-bit array):
  B:= (0=>'0', 4=>'0', others =>'1'); -- set B="01110111"
  B:= ('0','1','1','1','0','1','1','1'); -- set B="01110111"

- Array Slice
  A subset of an array can be accessed using the array slice functionality. For example, to only look at sixth to ninth elements of an array M, use one of the following expressions:

  M(6 to 9) or M(9 downto 6) -- Element in these arrays are stored in opposite
  -- order.

- Concatenation Operator, "**&**"
  A Concatenation Operator is used to combine (Concatenate) arrays or array elements as shown by the following examples:

  '0' **&** '1' **&** "1Z" results in the string "011Z"
  B(6 **downto** 0) & B(7) results in a 1-bit rotate left of the 8-bit array B.

- Unconstrained array
  In some application, the designer required an array but at the declaration, its number of elements or range is unknown. For these applications, array may be declared using the unconstrained range definition "**<>**". The following example demonstrates the syntax for declaring a unconstrained range array:

type type_name **is array** (type **range <>**) **of** element_type;

The most important array type in VHDL is the IEEE 1164 standard user-defined logic type std_logic_vector which is defined as an ordered set of std_logic bits. If we want to create unconstrained array of std_logic_vector with an integer index, use the following declaration:

type STD_LOGIC_VECTOR **is array** (integer **range <>**) **of** STD_LOGIC;

- Constant declarations
  Constants are used to improve readability, portability and maintainability of the code. Constant name is typically in capital letters and is descriptive of its use. The constant declaration syntax is shown below:

  constant constant_name **:** type_name **:=** value;

  Below are some examples constant declarations and note the assignment operation is the same as one used for variable "**:=**":

```
constant BUS_SIZE: integer := 32;           -- Width of component
constant MSB: integer := BUS_SIZE-1;        -- Bit number of MSB
constant DEF_OUT : character := 'Z';        -- Default Output constant as character Z
```

- Function definitions
  A function is a subprogram that accepts a number of parameters (Parameters must be defined with mode "**in"**) and returns a result. Each of the parameters and the result must have a pre-determined type.

  Functions may contain local types, constants, variables, nested functions and procedures. All statements in the function body will be executed sequentially. Below is the simplified syntax of function definition:

```
function function-name (
        signal_names : signal_type;          -- arguments (mode is in)
        signal_names : signal_type;
        . . .
        signal_names : signal_type;
) return return_type is                      -- one return value which replaces the function
        type declaration
        constant declaration
        variable declaration
        function definitions
        procedure definitions
begin                                        -- Start of the main body of the function
        sequential_statement
        …
        sequential_statement
end function_name;
```

  - Example—implementing "A but not B" function

```
entity AbutNotB is
        port (X, Y, in BIT;    -- X, Y are input of BIT type
              Z:  out Bit);    -- Z is output of BIT type
end AbutNotB
```

```
architecture AbutNotB_arch of AbutNotB

function ButNot (A, B: bit) return bit is        -- function definition
begin
        if B = '0' then return A;
        else return '0';
end if;
end ButNot;

Begin
        Z<= ButNot (X,Y);                        -- function call
end AbutNotB_arch;
```

- Procedure Definitions

A procedure is similar to the function in that it is a subprogram that accepts input parameters but:

- A procedure does not have a return values.
- A procedure's parameters may be constants, signals, or variables, each of whose modes may be **in**, **out**, or **inout**. This means that by setting the value of the arguments (**out**, **inout**), the value may be returned to the calling program.

Here is the simplified syntax for procedures:

```
procedure procedure_name ( formal_parameter_list )
procedure procedure_name ( formal_parameter_list ) is
        procedure_declarations
begin
        sequential statements
end procedure procedure_name;
```

- Example – A procedure to implement the functionality of a rising edge-triggered D flip-flop.

```
procedure dff (signal Clk,Rst,D; in std_ulogic;
        signal Q: out std_ulogic) is
begin
        if Rst <= '1' then Q <= '0';
        elsif rising_edge(Clk) then Q <= D;
        end if;
end procedure
```

- Libraries

Similar to other high Level languages, VHDL uses libraries to aggregate already completed functionality and make it available to designer for reuse. VHDL supplies a number of general libraries such as IEEE standard libraries and the designer can create local libraries for future use.

The following syntax is used to include a library in a design. This statement should be included prior to the entity and architecture definitions.

```
library library_name;
```

Each of the general VHDL library packages contain definitions of objects that can be used in other programs. A library package may include signal, type, constant, function, procedure, and component declarations.

Once a library is included using the library statement, use statement shown below is used to include the desired library package in the design.

use package_name

When using VHDL functions, the description of function includes guidance on which library packages are required for the function.

➢ Example – The following two statements brings in all the definitions from the IEEE standard 1164 package:

    library IEEE;
    use IEEE.Std_Logic_1164.**all**;

Std_Logic_1164.all contains the following:

- type std_ulogic: unresolved logic type of 9 values;
- type std_ulogic_vector: vector of std_ulogic;
- function resolved resolving a std_ulogic_vector into std_ulogic;
- subtype std_logic as a resolved version of std_ulogic;
- type std_logic_vector: vector of std_logic;
- subtypes X01, X01Z, UX01, UX01Z: subtypes of resolved std_ulogic containing the values listed in the names of subtypes (i.e. UX01 contains values 'U', 'X', '0', and '1', etc.);
- logical functions for std_logic, std_ulogic, std_logic_vector and std_ulogic_vector;
- conversion functions between std_ulogic and bit, std_ulogic and bit_vector, std_logic_vector and bit_vector and vice-versa;
- functions rising_edge and falling_edge for edge detection of signals.
- x-value detection functions, is_x, which detect values 'U', 'X', 'Z', 'W', '-' in the actual parameter.

IEE 1164 Standard Logic Package (released in the 1980s) defines many functions that operate on the standard types of std_logic and std_logic_vector. IEEE 1164 replaces these proprietary data types (which include systems having four, seven, or even thirteen unique values) with a standard data type having nine values, as shown below:

Value	Description
'U'	Uninitialized
'X'	Unknown
'0'	Logic 0 (driven)
'1'	Logic 1 (driven)
'Z'	High impedance
'W'	Weak 1
'L'	Logic 0 (read)
'H'	Logic 1 (read)
'-'	Don't-care

These nine values make it possible to accurately model the behavior of a digital circuit during simulation.

- The std_ulogic data type is an unresolved type, meaning that it is illegal for two values (such as '0' and '1', or '1' and 'Z') to be simultaneously driven onto a signal of type std_ulogic.

- If you are describing a circuit that involves multiple values being driven onto a wire, then you will need to use the type std_logic. Std_logic is a resolved type based on std_ulogic.

Resolved types are declared with resolution functions.

- Example: NAND gate coupled to an output enable
  Note: Even though it is not necessary we will use the resolved type "std_logic"

```
library ieee;
use ieee.std_logic_1164.all;

entity nandgate is
    port (A, B, OE: in std_logic; Y: out std_logic);
end nandgate;

architecture arch1 of nandgate is
    signal n: std_logic;
begin
    n <= not (A and B);
    Y <= n when OE = '0' else 'Z';
end arch1;
```

7.6. Operators

This section provides an overview of logical, relational, arithmetic and other operators. Although, this is an extensive listing, reader is encouraged to explore additional operators through the online documentation available on the development environment.

❖ Logical operators
The logical operators and, or, nand, nor, xor and xnor are used to describe Boolean logic operations, or perform bit-wise operations, on bits or arrays of bits.

Operator	Description	Operand Types	Result Types
and	And	Any Bit or Boolean type	Same Type
or	Or	Any Bit or Boolean type	Same Type
nand	Not And	Any Bit or Boolean type	Same Type
nor	Not Or	Any Bit or Boolean type	Same Type
xor	Exclusive OR	Any Bit or Boolean type	Same Type
xnor	Exclusive NOR	Any Bit or Boolean type	Same Type

❖ Relational operators
Relational operators are used to test the relative values of two scalar types. The result of a relational operation is always a Boolean true or false value.

Operator	Description	Operand Types	Result Type
=	Equality	Any type	Boolean
/=	Inequality	Any type	Boolean
<	Less than	Any scalar type or discrete array	Boolean
<=	Less than or equal	Any scalar type or discrete array	Boolean
>	Greater than	Any scalar type or discrete array	Boolean
>=	Greater than or equal	Any scalar type or discrete array	Boolean

❖ Arithmetic Operations
The Arithmetic operators have been grouped into add/subtract, multiply/divide and sign operators.

➢ Add/Subtract Operators
The adding operators can be used to describe arithmetic functions or, in the case of array types, concatenation operations.

Operator	Description	Operand Types	Result Type
+	Addition	Any numeric type	Same type
-	Subtraction	Any numeric type	Same type
&	Concatenation	Any numeric type	Same type
&	Concatenation	Any array or element type	Same array type

➢ Multiply/Divide Operators
These operators can be used to describe mathematical functions on numeric types. It is important to note that synthesis tools vary in their support for multiplying operators.

Operator	Description	Operand Types	Result Type
*	Multiplication	Left: any integer or floating point type. Right: same type	Same as left
*	Multiplication	Left: any physical type.	Same as left

		Right: integer or real type.	
*	Multiplication	Left: integer or real type. Right: any physical type.	Same as right
/	Division	Left: any integer or floating point type. Right: same type	Same as left
/	Division	Left: any integer or floating point type. Right: same type	Same as left
/	Division	Left: integer or real type. Right: any physical type.	Same as right
mod	Modulus	Any integer type	Same type
rem	Remainder	Any integer type	Same type

❖ Sign Operators
A Sign operator can be used to specify the sign (either positive or negative) of a numeric object or literal.

Operator	Description	Operand Types	Result Type
+	Identity	Any numeric type	Same type
-	Negation	Any numeric type	Same type

❖ Other operators
The exponentiation and absolute value operators can be applied to numeric types, in which case they result in the same numeric type. The logical negation operator results in the same type (bit or Boolean), but with the reverse logical polarity. The shift operators provide bit-wise shift and rotate operations for arrays of type bit or Boolean.

Operator	Description	Operand Types	Result Type
**	Exponentiation	Left: any integer type Right: integer type	Same as left type
**	Exponentiation	Left: any floating point type Right: integer type	Same as left type
abs	Absolute value	Any numeric type	Same as left type
not	Logical negation	Any Bit or Boolean type	Same as left type
sll	Shift left logical	Left: Any one-dimensional array of Bit or Boolean Right: integer type	Same as left type
srl	Shift right logical	Left: Any one-dimensional array of Bit or Boolean Right: integer type	Same as left type
sla	Shift left arithmetic	Left: Any one-dimensional array of Bit or Boolean Right: integer type	Same as left type
sra	Shift right arithmetic	Left: Any one-dimensional array of Bit or Boolean Right: integer type	Same as left type
rol	Rotate left	Left: Any one-dimensional array of Bit or Boolean Right: integer type	Same as left type
ror	Rotate right	Left: Any one-dimensional array of Bit or Boolean Right: integer type	Same as left type

7.7. Behavioral Design

VHDL design may be conducted using structural or behavioral approach. In structural design, the basic building blocks are defined using components and the rest of design defines the interconnection between these components. Structural design is the closest approximation to using the physical component with wiring diagram. In other words, it is the simply a textual description of a schematic.

The strength of VHDL is based on its ability to compile description of circuit behavior to a fully defined and implementable design. This is referred to as behavioral design which is much simpler than the structural design and is commonly used for design.

Behavioral design relies of data flow elements to define functionality which is described in the next section. Another useful VHDL statement is process:

- Characteristics
  - A process is a list of sequential statements that executes in parallel with other concurrent statements and processes in the architecture..
  - Using process, a designer can specify a complex interaction of signals and events in a way that executes in essentially zero simulated time during the simulation. This characteristic is useful in synthesizing and modeling combinational or sequential circuits.
  - A process statement can be used anywhere a concurrent statement can be used.
  - A process statement has visibility within the scope of an enclosing architecture. This means that the types, signals, constants, functions and procedures defined in architecture are visible to the process. But the variable, type, constant, function and procedure defined in the process are not visible to the architecture.
  - A process can not declare signals therefore only variable declarations are available in Process.

- Syntax of a VHDL process statement

```
process (signal_name, signal_name, … , signal_name)
        type declarations
        variable declarations
        constant declarations
        function declarations
        procedure declarations
begin
        sequential_statement
        . . .
        sequential_statement
end process;
```

  As a quick reminder, process executes statements sequentially and does not allow signal declaration within its scope. As discussed earlier, variable assignment operation is "**:=**" which is different from signal assignment "**<=**". But the declaration is similar to signal declaration as shown below:

  variable variable_names **:** variable_type**;**

- Process operations

  A process is always either running or suspended. The list of signals passed is called the "sensitivity list" which determines when the process runs. Below is an overview of process life cycle:

  - Process is initially suspended.
  - When any of the signals in the sensitivity list changes value, the process starts execution with the first sequential-statement in the process.

Process runs until no other signal in the sensitivity list changes value as a result of running the process.

- In simulation, all the statements in the process execute instantly (no elapsed time from start to end of the process).

It is possible to write a process that never suspends. For example, a process with X in its sensitivity list and containing the statement "X <= not X". This process will never suspend will continuously change. This is not a useful process and is similar to infinite loop. Most simulators will detect the error and terminate after few thousand iterations.

Finally, the sensitivity list is optional; a process without a sensitivity list starts running at time zero in simulation. One application of such a process is to generate an input waveform for the test bench.

❖ Example – Design a prime number detector using process-based data flow architecture.

```
architecture prime4_arch of prime is
begin
        process(N)
            variables N3L_N0, N3L_N2L_N1, N3L_N1_N0, N2_N1L_N0: STD_LOGIC;
        begin
            N3L_N0      := not N(3) and N(0);
            N3L_N2L_N1       := not N(3) and not N(2) and N(1);
            N3L_N1_N0        := not N(3) and not N(1) and N(0);
            N2_N1L_N0        := N2 and not N(1) and N(0);
            F <= N3L_N0 or N3L_N2L_N1 or N2L_N1_N0 or N2_N1L_N0;
        end process
end prime4_arch;
```

Note: Within the prime4_arch we have only one concurrent statement and that is the process.

❖ Example – Design a Rising Edge D-Flip Flop.

```
entity ent_DFF is
begin
        port(
            D, clk, : in std_logic;
            Q : out  std_logic
            );
end ent_DFF;

architecture arc_DFF of ent_DFF is
begin
pdf:    process(clk)
        begin
            if (clk = '1') then
                q <= D;
            end if;
        end process pdf;
end arc_DFF;
```

7.8. Dataflow Design Elements

A behavioral design relies on VHDL's dataflow elements in describing the desired behavior. The remainder of this section is focused on the most commonly used dataflow elements.

- Concurrent "when signal" assignments
  - Syntax

```
signal_name <= expression;    -- Concurrent signal assignment statement

signal_name <= expression when boolean_expression else    -- conditional concurrent
               expression when boolean_expression else    -- signal assignment statements
               . . .
               expression when boolean_expression else
               expression ;
```

  - Example— Use the Dataflow elements to write the architecture for the prime number detector (behavioral design).

```
architecture prime2_arch of prime is
        signal N3L_N0, N3_N2L_N1, N2L_N1_N0, N2_N1L_N0: STD_LOGIC;
begin
        N3L_N0         <= 1 when (not N(3) and N(0)) else 0 ;
        N3L_N2L_N1     <= 1 when (not N(3) and not N(2) and N(1)) else 0;
        N2L_N1_N0      <= 1 when (not N(2) and N(1) and N(0)) else 0;
        N2_N1L_N0      <= 1 when (N(2) and not N(1) and N(0)) else 0;
        F <= 1  when (N3L_N0 or N3L_N2L_N1 or N2L_N1_N0 or N2_N1L_N0) ;
end prime2_arch;
```

The prime number detector can also be implemented using conditional concurrent assignment statements.

- Concurrent "selected signal" assignment
This statement evaluates the given expression when it matches one of the choices, then it assigns the corresponding signal_value to signal_name.

  - Syntax

```
with expression select
        Signal_name <=  signal_value when choices,
                        signal_value when choices,
                        . . .
                        signal_value when choices,
                        signal_value when others;
```

    - The choices for the entire statement must be mutually exclusive.
    - The statement with keyword **others** will be used when none of the other choices matches the expression results.
    - Choices may be a single value of expression of a list of values, separated by vertical bars "|".

  - Example – Implement a prime number detector using selected signal assignment.

```
architecture prime3_arch of prime is
begin
        with N select
```

```
        F <= '1' when "0001",
             '1' when "0010",
             '1' when "0011" | "0101" | "0111",
             '1' when "1011" | "1101",
             '0' when others;
end prime2_arch;
```

- Sequential "If-then-else" statement
  This sequential statement will give us the ability to make decisions, based on the value of a Boolean-expression to execute a sequential statement or not.

  - Syntax of "If-then-else" statement simple to fully nested

```
if Boolean-expression then sequential-statement              -- do only on true
end if;

if Boolean-expression then sequential-statement              -- handle true and false
else sequential-statement
end if;

if Boolean-expression then sequential-statement              -- nested if statements
elsif Boolean-expression then sequential-statement
…
elsif Boolean-expression then sequential-statement
end if;

if Boolean-expression then sequential-statement
elsif Boolean-expression then sequential-statement
…
elsif Boolean-expression then sequential-statement
else sequential-statement                                    -- catch all else
end if;
```

  - Example – using "If-then-else" statements to implement the prime number detector

```
architecture prime5_arch of prime is
begin
        process(N)
            variable NI: Integer;
        begin
                NI :=CONV_INTEGER(N);
                if NI=1 or NI=2 then F <= '1';
                elsif NI=3 or NI=5 or NI=7 or NI=11 or NI=13 then F <= '1';
                else F <= '0';
                end if;
        end process;
end prime5_arch;
```

- Sequential "Case" statement
  This statement evaluates the given expression, finds a matching value in one of the choices, and executes the corresponding sequential-statements.

  Note: Choice may take multiple values using vertical bar operator "|"

  - Syntax
    case expression is

```
        when choices => sequential-statements
        …
        when choices => sequential_statements
        when others sequential_statements                  -- do if none of choices match
end case;
```

Use case statement instead of if-then-else if possible since it is easier to synthesize.

- Example – Prime number detector using case statement

```
architecture prime6_arch of prime is
begin
        process(N)
        begin
            case CONV_INTEGER(N) is
                when 1                    =>      F <= '1';
                when 3 | 5 | 7| 11 | 13   =>      F <= '1';
                when others               =>      F <= '0';
            end case;
        end process;
end prime6_arch;
```

- Sequential "Loop" Statements
  There are three types of loops that are useful in synthesizing repeated structures.

  - Sequential "Basic Loop" Statement syntax
    This creates an infinite loop which is useful when doing modeling.

```
loop
        sequential-statement
        . . .
        sequential-statement
end loop;
```

  - Sequential "For Loop" Statement syntax
    the identifier is implicitly declared and will have the same type as the range. The identifier may be used inside the loop only.

```
for identifier in range loop
        sequential-statement
        . . .
        sequential-statement
end loop;
```

    The two sequential statements "exit" and "next" may be used in the loop body:
    - "exit" terminates the loop and continues with the next statement after the loop.
    - "next" starts the next iteration through the loop, bypassing the remaining statement in the current iteration.

  - Sequential "While Loop" statement syntax
    The identifier is implicitly declared and will have the same type as the range. The identifier may be used inside the loop only.

```
while Boolean_expression loop
        sequential_statement
        . . .
```

```
        sequential_statement
    end loop;
```

7.9. Additional Resources

- Wakerly, I. Digital Design. (2006) Prentice Hall
  Chapter 5 “Hardware Description Language”

7.10. Problems

(Include system diagram for Entity code and pseudo code for Architecture code)

1. Write a VHDL behavioral program to implement a pulse-triggered D latch.

2. Write entity code for a (2-input) XNOR.

3. Complete a VHDL behavioral design for an edge-triggered D latch .
   *Note: The expression (**signal'event)** is only true when the signal is changing. This is referred to as the event attribute of the signal. This attribute in conjunction with process() should be considered in completing this design. <Why is "**signal'event**" even mentioned; the way that this is written suggests that **signal'event** was mentioned before.>*

4. Write a VHDL behavioral program to implement a rising-edge-triggered JK flip flop.

5. Design a VHDL model for a 16-bit register with Clock Enable, active-low Output Enable, and active-low Clear.

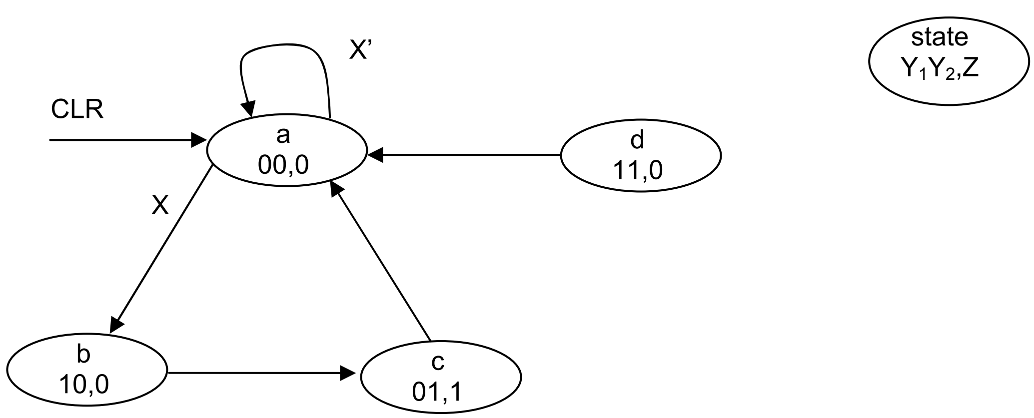
6. Using VHDL, design a 4-to-1 MUX. The two bit input "sel", selects the input d_0-d_3 that is connected to output "f". For example when sel="01", f=d_1.

7. Using VHDL, design a 8-to-1 MUX. The three-bit input "sel", selects the input d_0-d_8 that is connected to output "f". For example when sel="010", f=d_2.

8. Design a 2–Bit Ripple Carry Adder with carry in and a carry out using VHDL.

9. Design the architectures for a rising edge D flip-flop for each of the following cases:
   a) with asynchronous reset using an **if** statement.
   b) with synchronous reset using an **if** statement.
   c) with synchronous reset using a **wait until** statement.

10. Create a complete VHDL design for the system described by the following state diagram:.



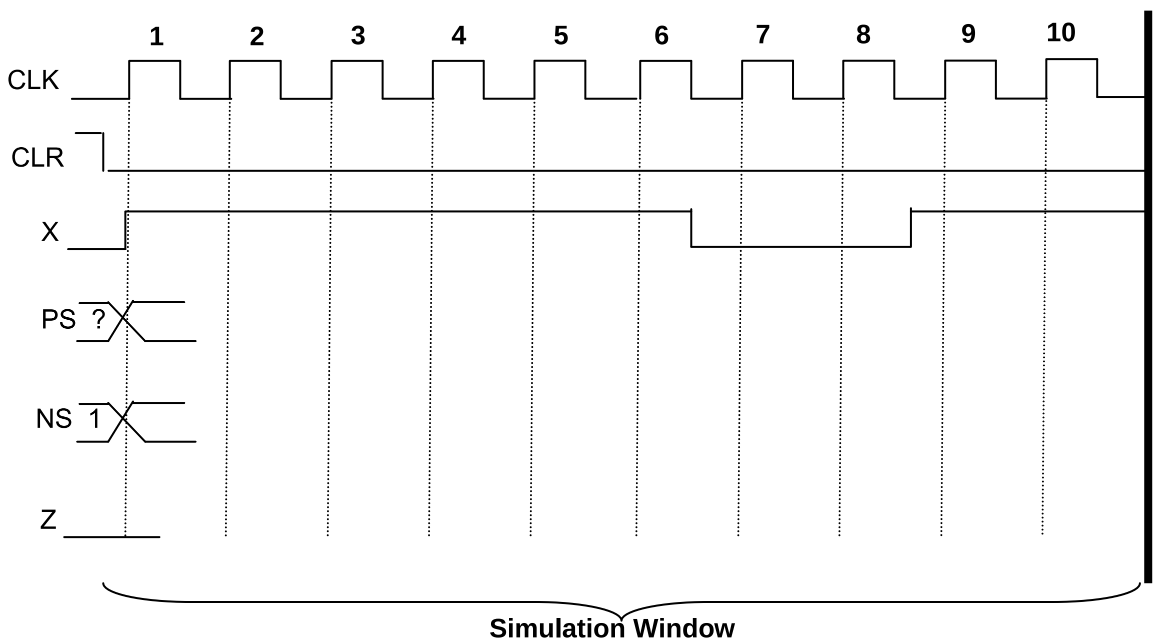
11. Complete the timing diagram in simulation window drawn at the bottom of the page based on the following the VHDL Code segment:

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
```

```
entity quiz_ent is
            port (CLK, CLR, X: in std_logic;
                        Z: out std_logic
                  );
end quiz_ent;
architecture quiz_arch of quiz_ent is
            Signal PS, NS: integer;
begin
            proc_a: process (CLK, CLR)
            begin
                        if (CLR = '1') then          NS <= 1;
                        elsif rising_edge (CLK) then
                                    PS <= NS;
                                    If ( NS < 2**8 and x='1') then NS <= NS*2;
                                    elsif ( NS > 0 and x='0') then NS <= NS/2;
                                    end if;
                        end if;
            end process proc_a;

            proc_b: process (PS, X)
            begin
                        if (PS > 31) then Z <= '1';
                        else Z <= '0';
                        end if;
            end process proc_b;
end quiz_arch;
```

TIMING DIAGRAM



12. Write VHDL code for a vending machine that accepts 5, 10 and 25 cents coins and deliver the product when $1 has been deposited.

Chapter 8. Commercial Digital Integrated Circuits and Interface Design

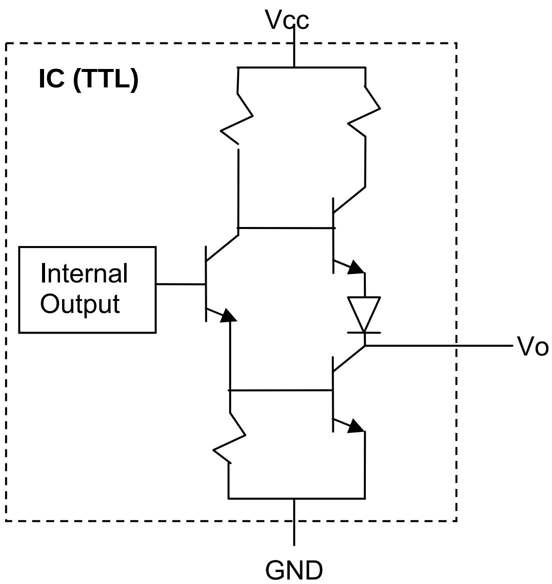
8.1. Key concepts and Overview

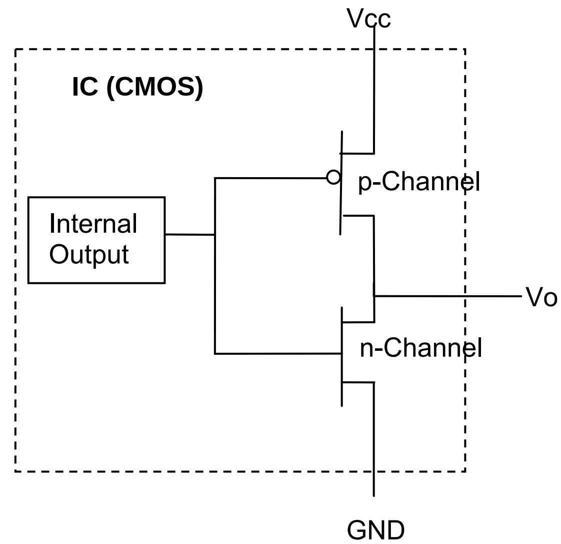
- Output Types
- Logic Families
- XOR Properties and Applications
- Multiplexers and DeMultiplexers (MUXes and DEMUXes)
- Adder & Subtractor Design
- Multiplier Design
- Arithmetic Logic Unit (ALU)
- Additional Resources
- Problems

8.2. Output Types

❖ Totem-Pole or Push-Pull Output
Totem-Pole output uses two complementary transistors to force the output to Vcc or ground. The advantage is that the output is set to one value. Disadvantages are:

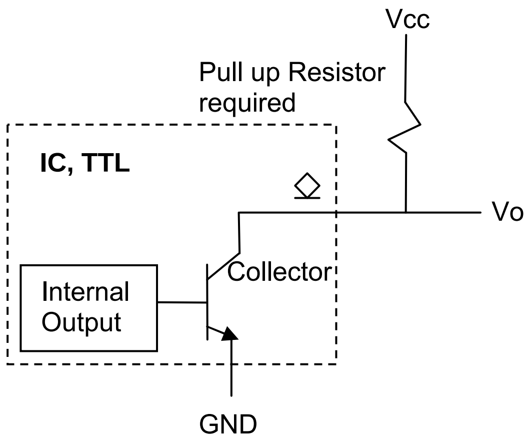
- Multiple outputs can not be connected together.
- Circuit is constantly using power since there is a path between Vcc and ground.

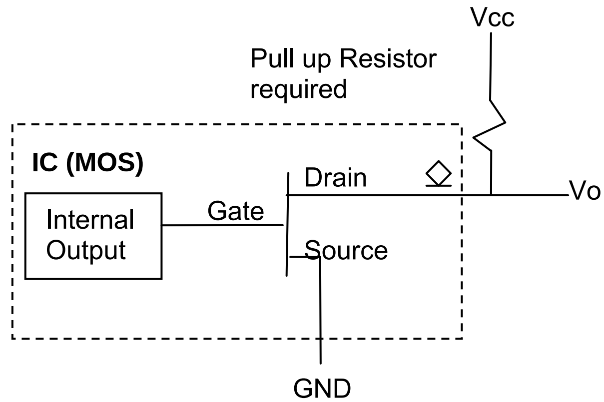




❖ Open Collector or Drain Output
This type of output will connect to low voltage when the output is 0, but it is not connected to anything (High Impedance) when the output is 1. Therefore it needs a pull up resistor to make sure it is connected to high, otherwise, it is floating resulting in an unknown value.

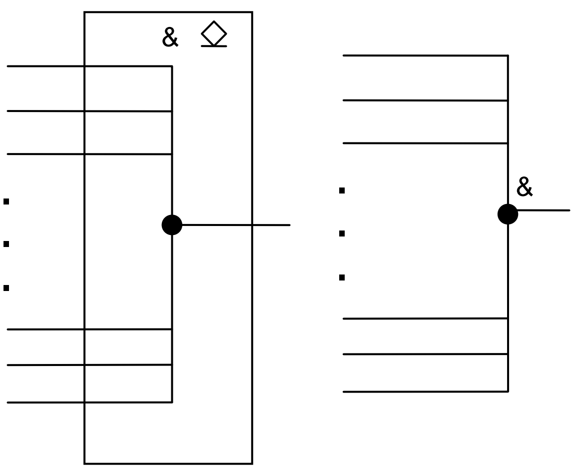
The advantage of this type of output is that the designer can connect multiple outputs together to create a wired AND.



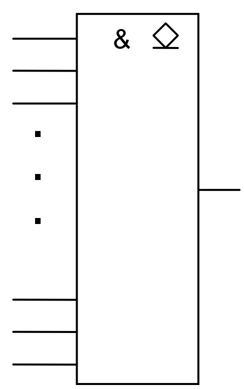


◇ Is used to indicate an open collector or Drain output

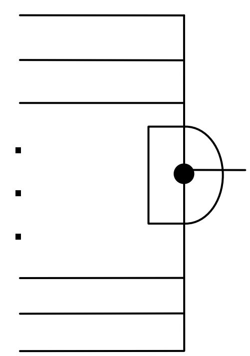
- Open Collector/drain is useful for creating a wired "AND" by connecting the outputs together and have one pull up resistor. This is also known as: Virtual AND, Dot-AND or Distributed AND. Below are three points to consider in relation to this type of configuration:
  - When all are high, then the pull up resistor provides the 1 output since all outputs are open
  - If any one of the outputs go low then the output will short to ground and output is 0 (Logic AND)
  - Symbols used to show the Wired-AND are shown below:



IEC International Symbols for the Function

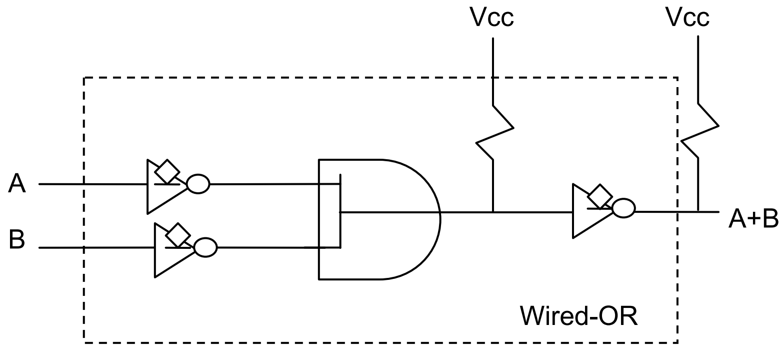


IEC alternate Symbol



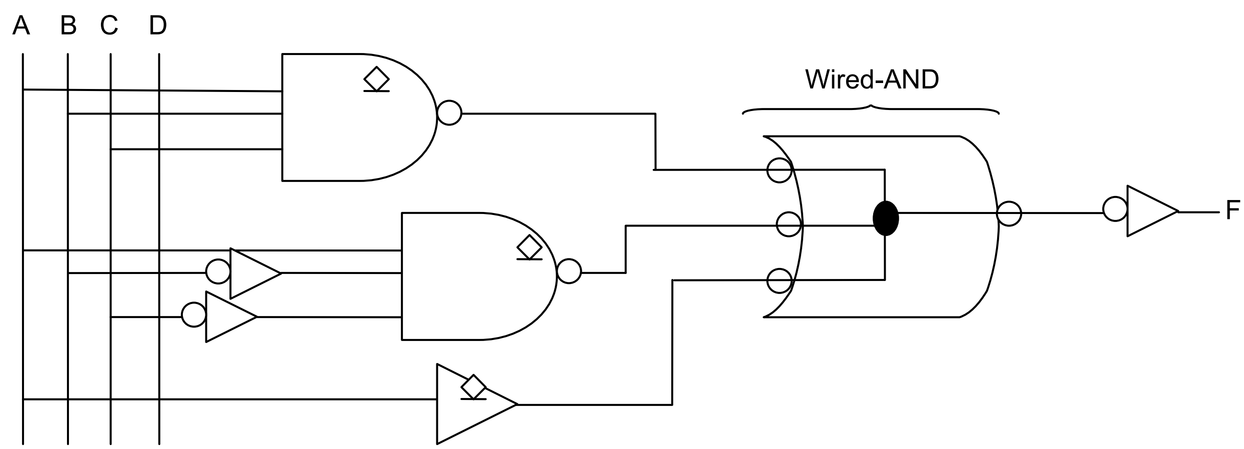
IEEE alternate Symbol

- A wired-OR can be created by using the DeMorgan's Theorem: $A+B=\overline{\overline{A}.\overline{B}}$



- Example - Using Wired-OR to implement F= $A.B.C + A.\overline{B}.\overline{C} + \overline{D}$ with the Signal List SL: F, A, B, C, D.

.



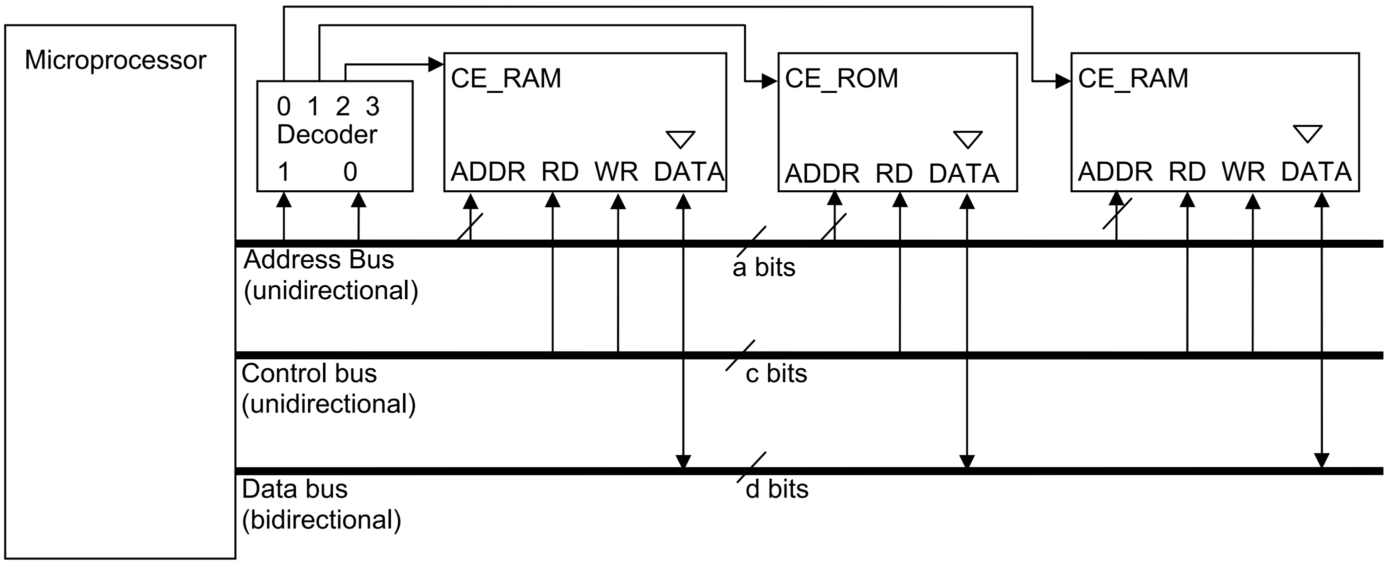
- Tri-State, 3-State or High impendence-State Output
  An input is used to decide if the output is being driven (enabled).
  - If the output is enabled, then it behaves like a normal 2-state device
  - If the output is disabled, then the output has high impedance (referred to as "hi-Z").
  - 74LS125 is a good example:

Input		Output
OE	A	F
1	X	Z
0	1	1
0	0	0

Notes:

- Z indicates high impendence (output is not connected internally)
- X indicates “don’t care”

▪ One of the most common uses of a tri-state output is for a microprocessor memory bus where you may have multiple memory banks connected but you only want one to be interacting with the processor at a time.



8.3. Logic Families

- TTL (Bipolar Junction Transistor Logic)
  First technology to get to market

- CMOS (Complementary Metal Oxide Semiconductor)
  Used for low power

- Integrated-Injection Logic (I^2L)
  Bipolar Transistor and Open-collector output used for the wired-AND function.

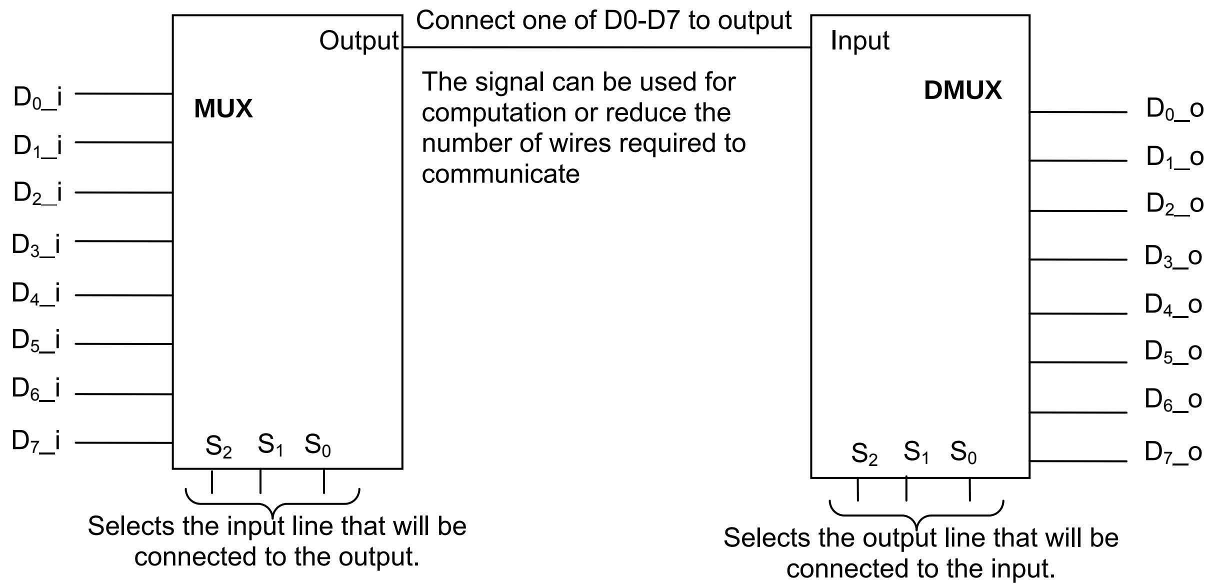
- Emitter-Coupled Logic (ECL)
  High-speed and high-power-requirement solutions.

8.4. Multiplexer (MUX)/DeMultiplexer (DMUX) Design

Multiplexers (MUX) and DeMultiplexers (DMUX) are used to route signals between networks with unequal number of signal lines. There are many applications that need one line for control or monitoring, but also need to analyze the data in a more compressed format. The application can be in communication, power, control, etc.

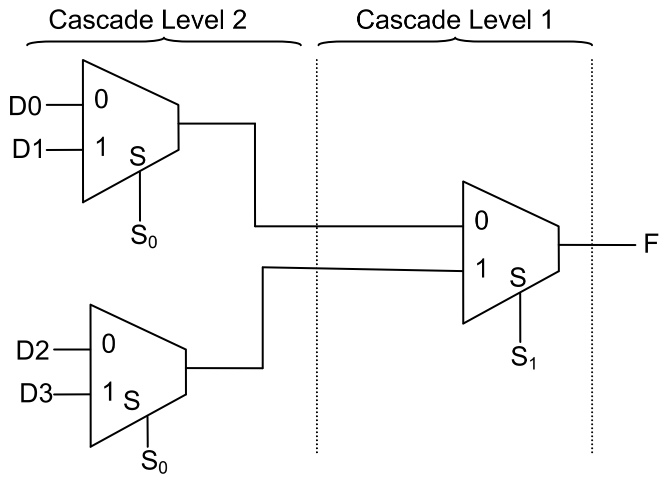
For example: You are building a security system that needs to control 200 entry ways. Each entry way will provide one input (Open/Close). Instead of running the 200 wires to the control, we could DMUX it into 8-bits ($2^n = 256$ when n=8). This means that only 8 lines are needed to go to the control instead of 200.

- ❖ An example of using a 1 to 8 DMUX and a 8 to 1 MUX



- ❖ Building a Large Scale MUX from Smaller MUXes
  - ➢ Typically use a cascading tree of MUXes to build a larger MUX :
    - ▪ Identify the number of MUX-ed outputs needed:
      $n \geq$ {ln(#input lines} / {ln 2} where n is the smallest integer that satisfies the equation and indicates the number of outputs.
    - ▪ If you have J-to-K MUX available then you will need n/K levels

  - ➢ Example of implementing a 4-to-1 MUX using 2-to-1 MUXes



- Example of implementing a 256-to-1 MUX using 8-to-1 MUXes.

Diagram of 256 to 1 MUX.

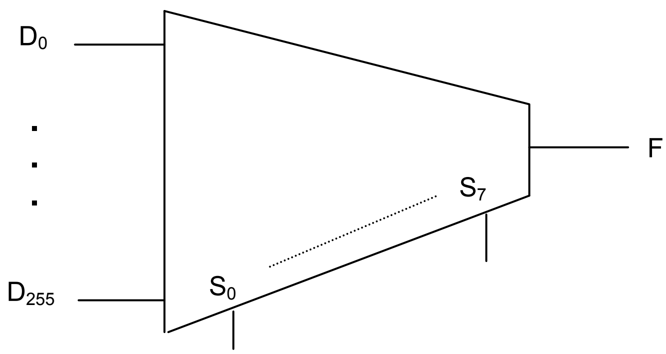
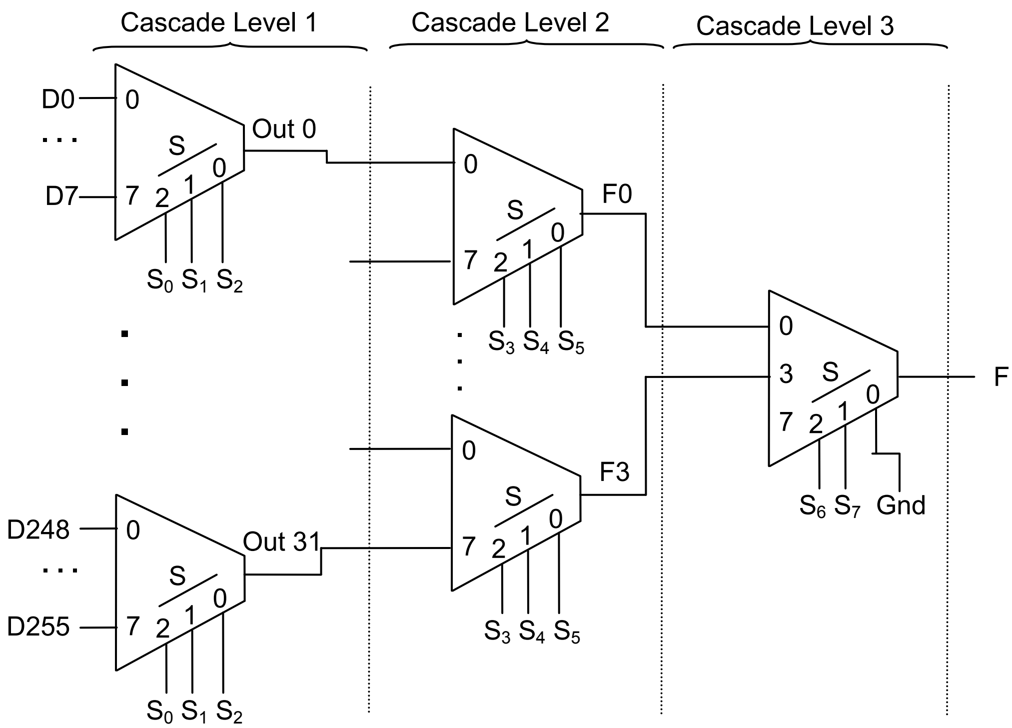
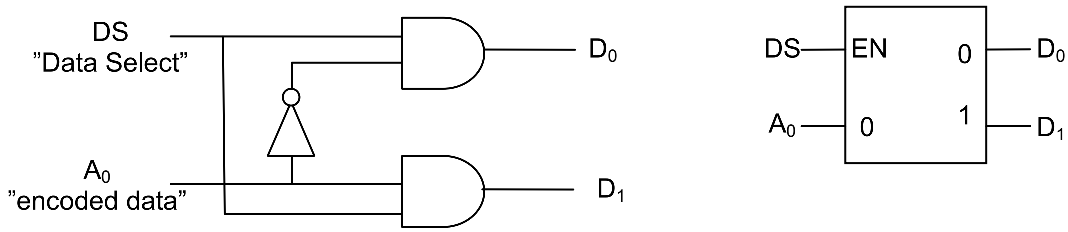


Diagram of a 256-to-1 MUX using 8-to-1 MUXes:

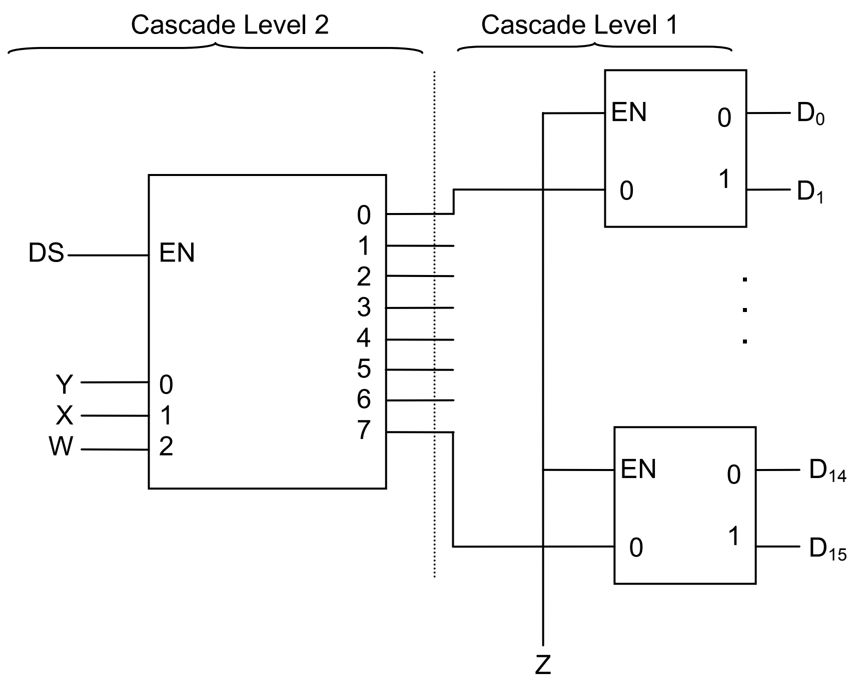

Cascade Level 1
Cascade Level 2
Cascade Level 3
D0
D7
Out 0
F0
D248
D255
Out 31
F3
F
Gnd
S0 S1 S2
S3 S4 S5
S6 S7

- Larger DMUX from smaller DMUX
  - 1-to-2 DMUX Design & Symbol



- The larger DMUX can be built from smaller DeMUXes by cascading the DeMUXes similar to MUXes.

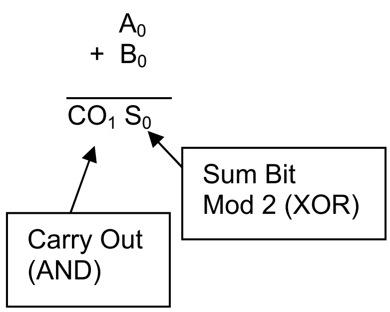
- For example: building a 3-to-8 DMUX using 1-to-2 DeMUXes:



8.5. Adder & Subtractor Design

Small adders can be shown at gate level, but for larger designs we will use an iterative modular design process. This process allows us to define a circuit for the i^{th} module, and then use it to show the overall design without redrawing the circuit each time.

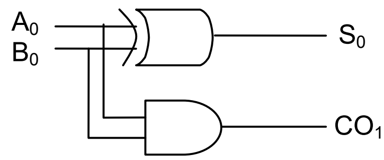
- Half Adder
  A Half Adder is the simplest form of an adder. It simply treats the carry and binary bit-addition separately.
  - Example: 1-bit adder



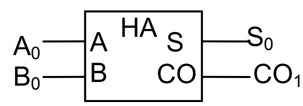
Half Adder Process

A_0	B_0	$C0_1$	S_0
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Truth Table

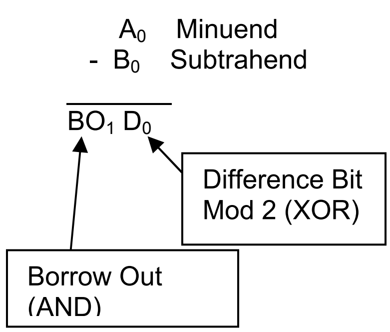


Gate-level Logic Circuit



Half Adder Symbol

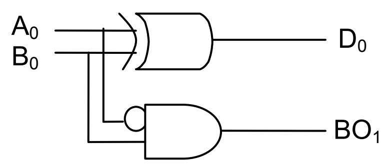
- Half Subtractor
  A Half Subtractor is the simplest form of Subtractor circuit.
  - Example: 1-bit Subtractor



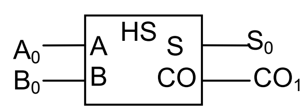
Half Subtractor Process

A_0	B_0	$B0_1$	D_0
0	0	0	0
0	1	1	1
1	0	0	1
1	1	0	0

Truth Table



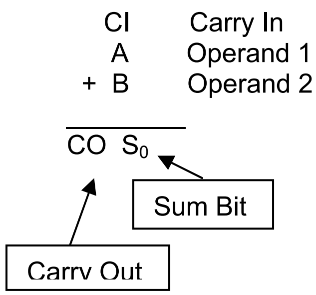
Gate-level Logic Circuit



Half Subtractor Symbol

Note: The Half Subtractor is the same as the Half Adder except for one input inversion. Universally, Subtractor are created by adding additional circuitry to an adders.

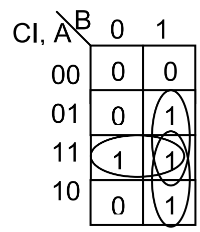
- Full Adder
  A Full Adder is a set of single bit adders combined to make an n–bit adder. It accepts a carry from a lower-significant-digit adder and produces a carry to be used by the next-higher-significant-digit adder.
  - Example:1-bit full adder module design for i^{th} bit.



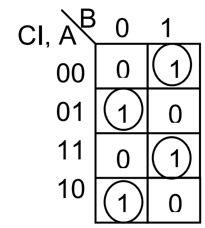
Full Adder Process

CI	A	B	C0	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

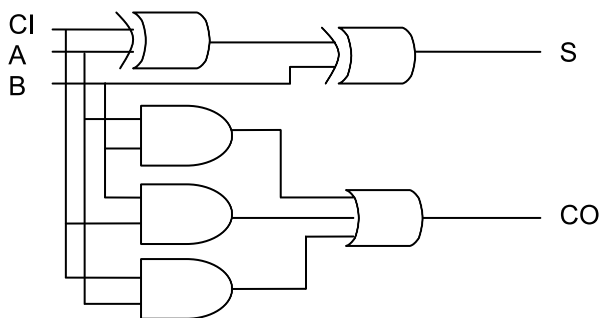
Truth Table



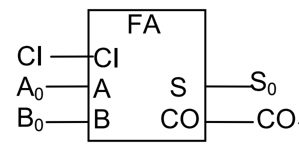
Carry Out CO =
A.B + CI.A + CI.B



Sum Bit S = $A \oplus B \oplus C$

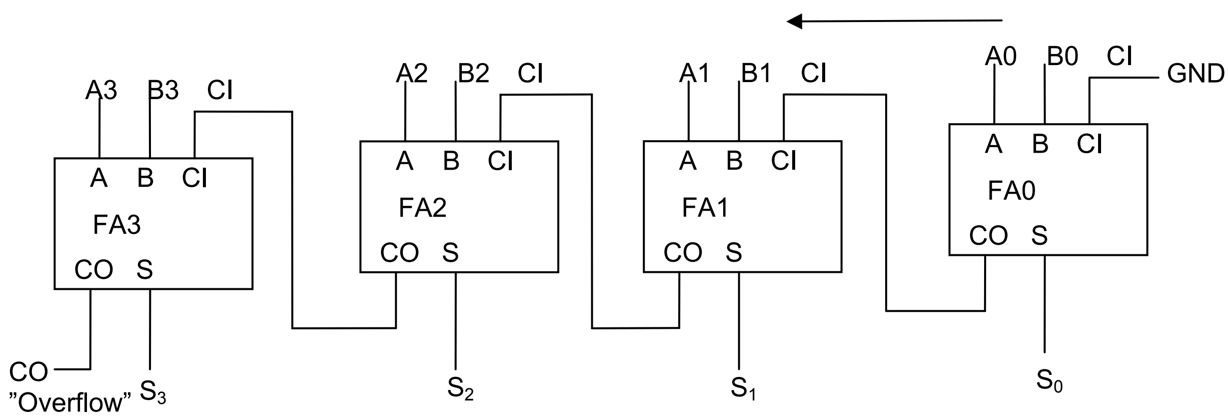


Full Adder Circuit



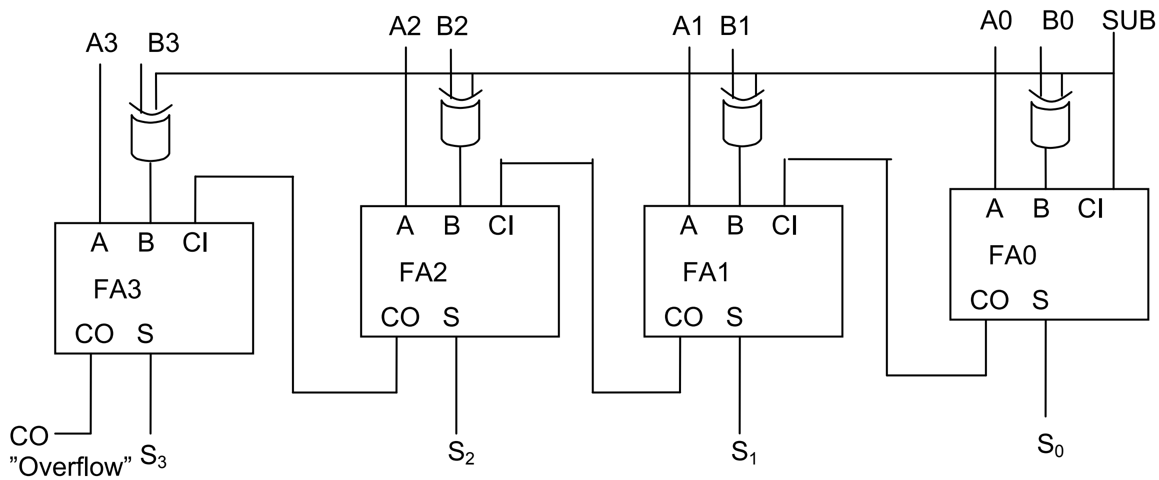
Full Adder Symbol

- Ripple-Carry Adder (RCA)
  A Ripple-Carry Adder uses Full Adders in a cascading form. The carry from one adder is fed to the next most significant bit-adder.



  - The Ripple-Carry Adder will have to wait until all the carries have propagated through the circuit before output stabilizes and results are valid. Carry-Look-ahead or carry-anticipation is often used to speed up the addition.

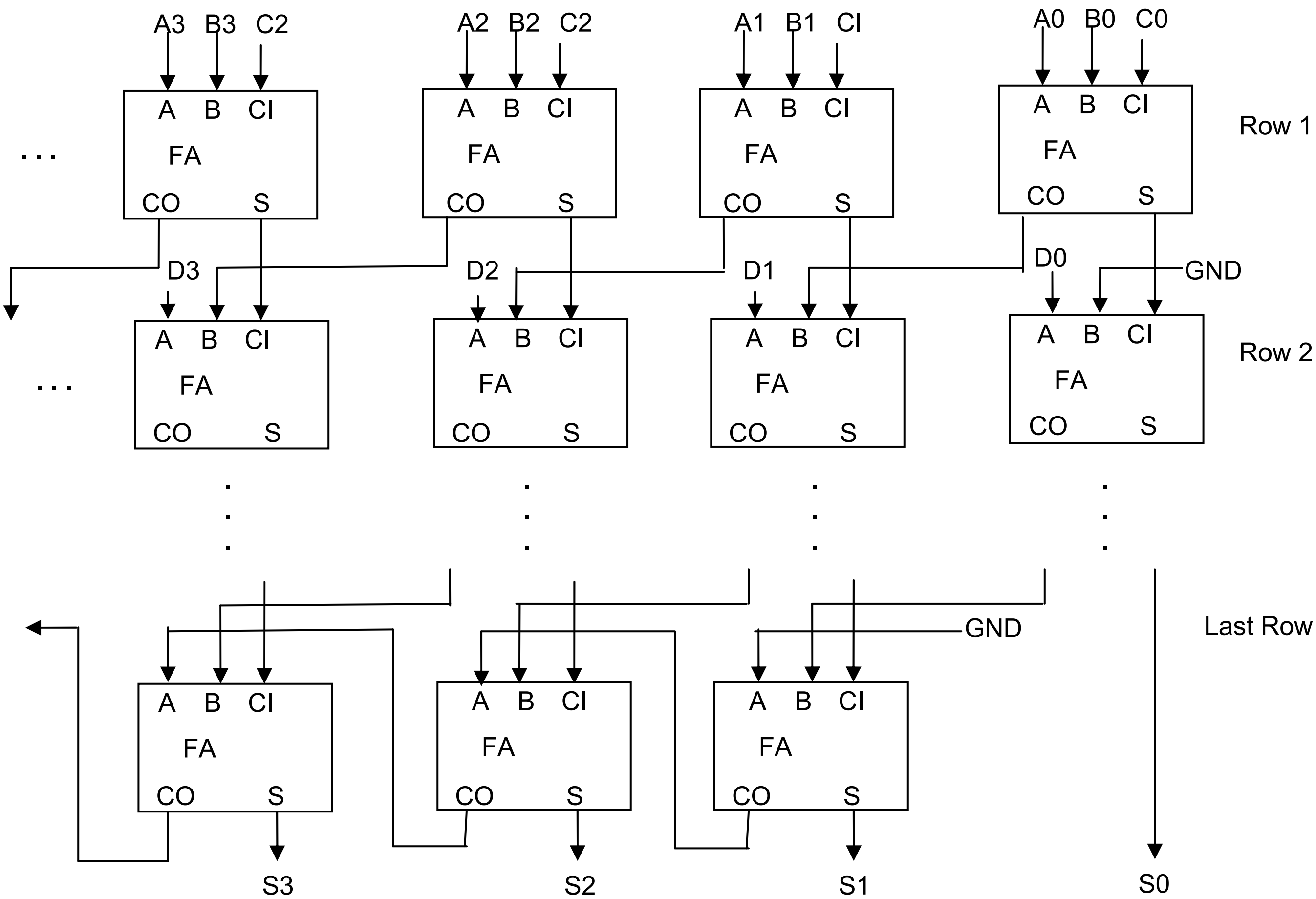
- Indirect Subtraction
  Given the following facts, a Subtractor can be designed from an RCA:
  - Given A – B = A + (-B)
  - From the two's complement, $(-B)_{2RC} = \overline{B} + 1$
  - Use an XOR to invert B when SUB=1 (subtraction) and B when SUB=0 (addition).



- Carry-Anticipation or Carry Look-Ahead Adder
  This solution reduces the settling time of adders.
  - Ripple-Carry for an n-bit adder will have settling time of $3t_p + 2(n-1)t_o$ since each stage will generate an output based on the last stage and would require $2t_p$ (Gate propagation) to complete the result.
  - Carry-Look-Ahead basically adds the circuitry to calculate the carry without having to wait for the propagation from each stage, effectively cutting the settling time to $6t_p$ for an n-bit adder when n>2. For a 1-bit adder, the setting time is $3t_p$, and for a two-bit adder, the settling time is $4t_p$.

- Carry-Save Adders
  Carry-Save Adders (CSAs) are designed to add more than two operands.
  - CSA's are designed using Full Adders (FA)
    - The carry from one level is fed into the next significant bit of the next stage.
    - The last stage shifted by one to the left but no new output is generated.
    - The number of rows of Adders = (The number of operands to be added) - 1
  - Example (five operands):

			A0	Operand 1
			B0	Operand 2
			+ C0	Operand 3
			-----------	.
			S10	Sum, Row 1
		CO11		Carry Row 2 (carry Save)
	+		D0	Operand 4
		---------------		.
		S21	S20	Sum, Row 2
		CO21		Carry Row 3 (carry Save)
	+		E0	Operand 5
		---------------		.
		S31	S30	Sum, Row 1
	CO32	CO31		Carry Row 4 (carry Save)
+ CO43	CO42	CO41		Carry Row 4 (no carry Save)
----------------------------------				.
S43	S42	S41	S40	Sum, Row 4 (Last Row)



A3 B3 C2
A2 B2 C2
A1 B1 CI
A0 B0 C0
A B CI
FA
CO S
Row 1
D3
D2
D1
D0
GND
Row 2
Last Row
S3
S2
S1
S0

8.6. Multiplier Design

First some basics of multiplication:

	A_m … A_2 A_1 A_0	Multiplicand (m bits)
	B_n … B_2 B_1 B_0	Multiplicand (n bits)
	X … X X X X	Partial Product 0
	X … X X X X	Partial Product 1
	.	
	.	
	.	
	X … X X X X	Partial Product n
$R(m+n)$	… R_2 R_1 R_0	Result (m+n bits)

Truth-table for a 2-bit by 2-bit multiply

A1	A0	B1	B0	R3	R2	R1	R0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0
0	0	1	0	0	0	0	0
0	0	1	1	0	0	0	0
0	1	0	0	0	0	0	0
0	1	0	1	0	0	0	1
0	1	1	0	0	0	1	0
0	1	1	1	0	0	1	1
1	0	0	0	0	0	0	0
1	0	0	1	0	0	1	0
1	0	1	0	0	1	0	0
1	0	1	1	0	1	1	0
1	1	0	0	0	0	0	0
1	1	0	1	0	0	1	1
1	1	1	0	0	1	1	0
1	1	1	1	1	0	0	1

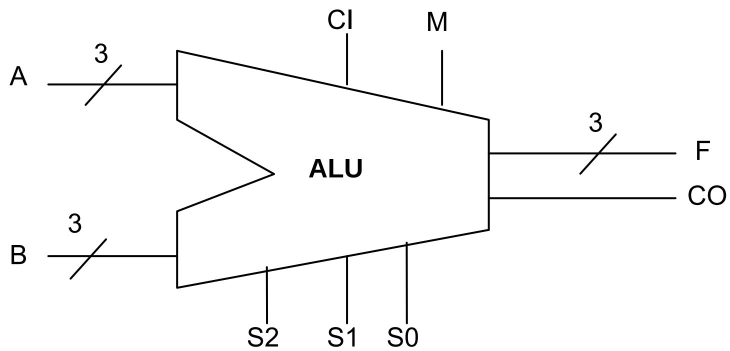
There are two methods to implement the above multiplication operation.

- Multiplier Design using 2-operand Adders
- Multiplier Design using Multiple-Operand Adder

8.7. Arithmetic Logic Unit (ALU) Design

The Arithmetic Logic Unit (ALU) is the heart of the computational capability of a computer.

A typically ALU Block Diagram (74LS382) is shown below:



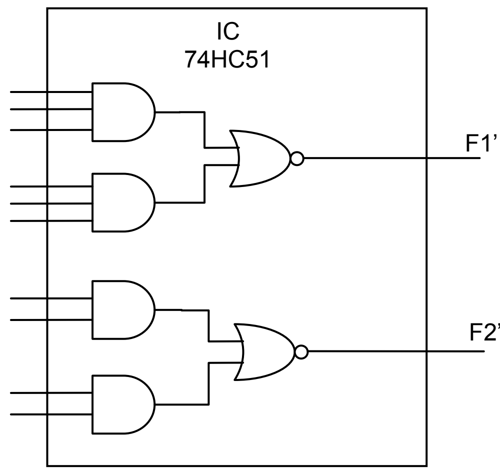
S2	S1	S0	Output: F and CO
0	0	0	Clear
0	0	1	B $_{minus}$ A
0	1	0	A $_{minus}$ B
0	1	1	A $_{plus}$ B
1	0	0	A $_{XOR}$ B
1	0	1	A $_{or}$ B
1	1	0	A $_{and}$ B
1	1	1	PRESET

8.8. Additional Resources

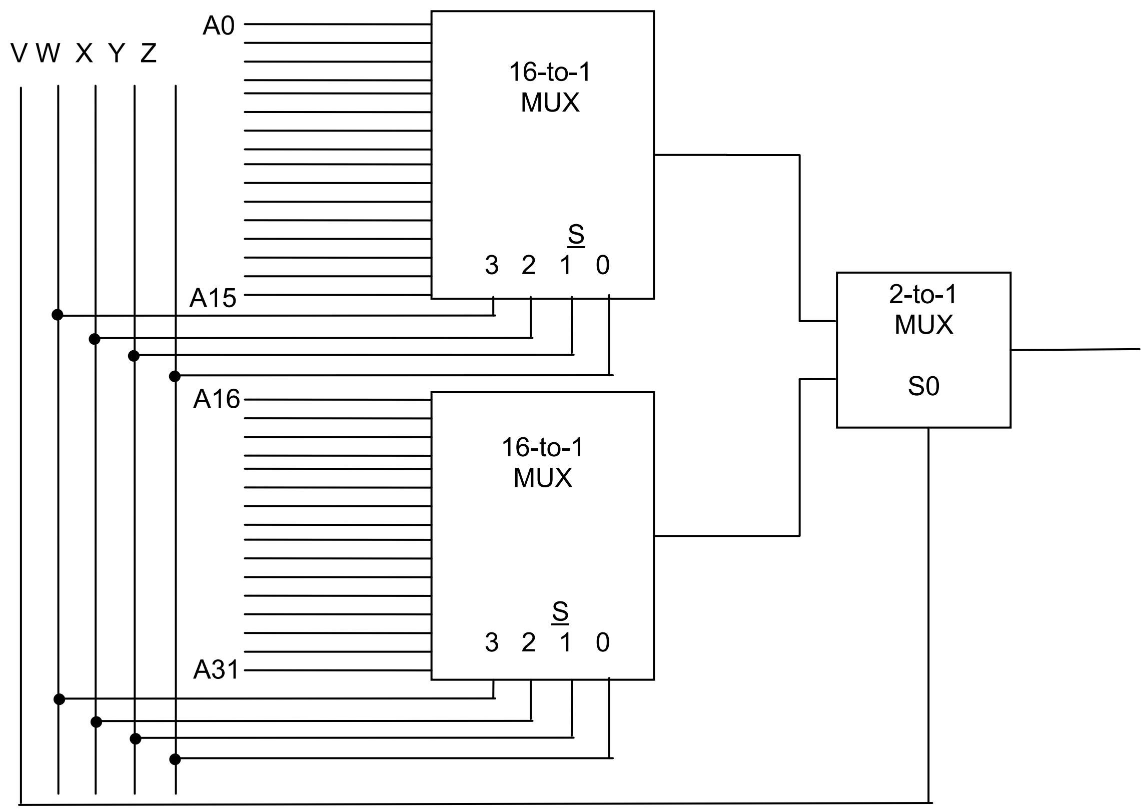
- Wakerly, I. Digital Design. (2006) Prentice Hall
  Chapter 4 & 6, “Combinational Logic Design Principles” & ”Combinational Logic Design Practices”

8.9. Problems

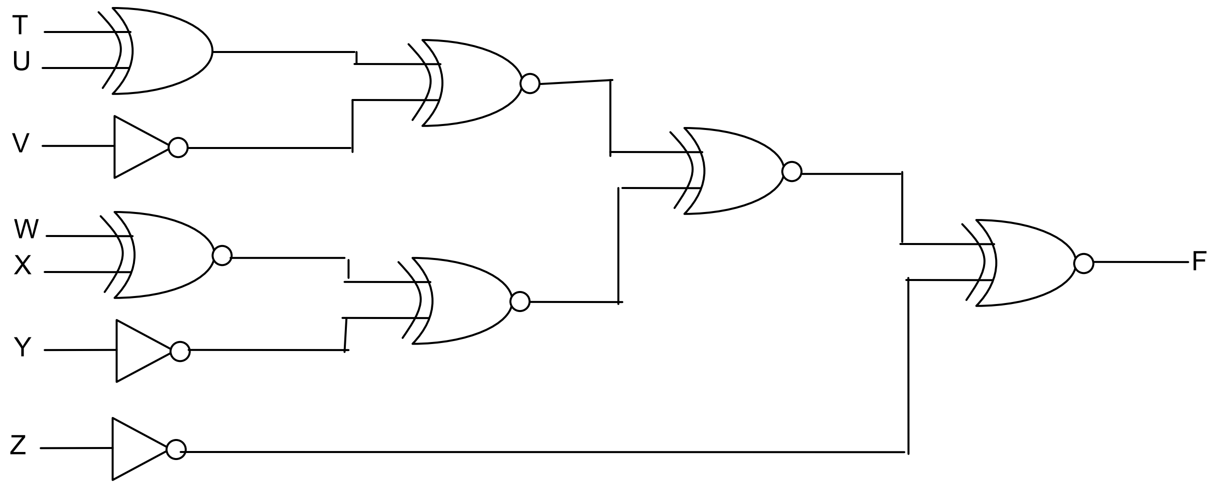
1. Design a circuit to realize the logic functions F1 = X'.Y'.Z + X.Y and F2 = Y' + X.Z' using the IC shown below. Assume all signals are uncomplemented positive-logic signals and use inverters as necessary.



2. Use two 16-bit multiplexers and one 2-bit multiplexer to construct a 32-bit MUX.



3. For the following circuit, determine if the output function is odd or even.

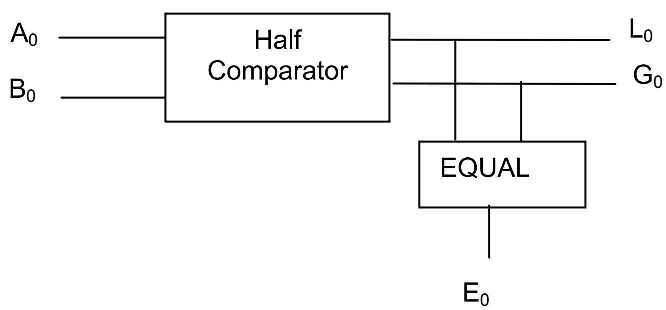


Definitions:

- An odd function will output a 1 when there is an odd number of 1s (1, 3, 5 …) in the input (XOR of all inputs).
- An even function will output a 1 when there is an even number of 1s (2, 4, 6 …) in the input (XNOR of all inputs).

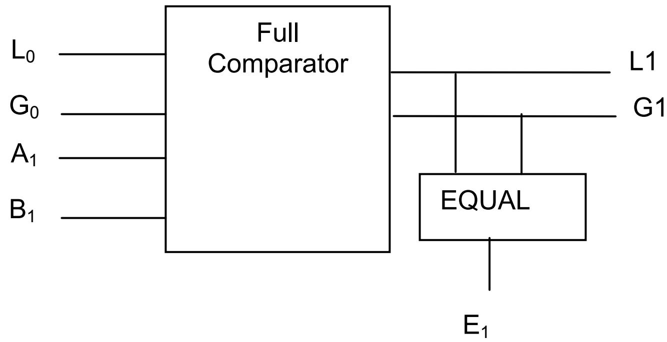
---

4. Design the half-comparator represented by the block diagram shown below. L_0 represents $A_0<B_0$, G_0 represents $A_0>B_0$, and E_0 represents $A_0=B_0$. Determine E_0 from $L_0 = G_0=0$.



---

5. Design a full comparator represented by the block diagram shown in figure below. L_0 represents $A_0<B_0$, G_0 represents $A_0>B_0$, L_1 represents $A_1A_0< B_1B_0$, G_1 represents $A_1A_0> B_1B_0$, and E_1 represents $A_1A_0= B_1B_0$. Determine E_1 from $L_1 = G_1=0$.



6. Design a gate-level 2-bit full adder that adds A1 A0, B1 B0 (2-bit numbers), and C0 (carry out) to obtain C2 S1 S0.

7. Design a modular 2-bit full subtractor using two 1-bit full subtractors.

8. Design a modular 4-bit full adder using four 1-bit full adders with carry look-ahead circuitry.

9. Obtain a modular circuit design for a 4-bit x 4-bit binary multiplier using two-operand adders. Before you begin the design, calculate:

a) the number of result bits required.
b) the number of AND gates required.
c) the number of rows of independent adders required.
d) the number of bits in each independent adder.

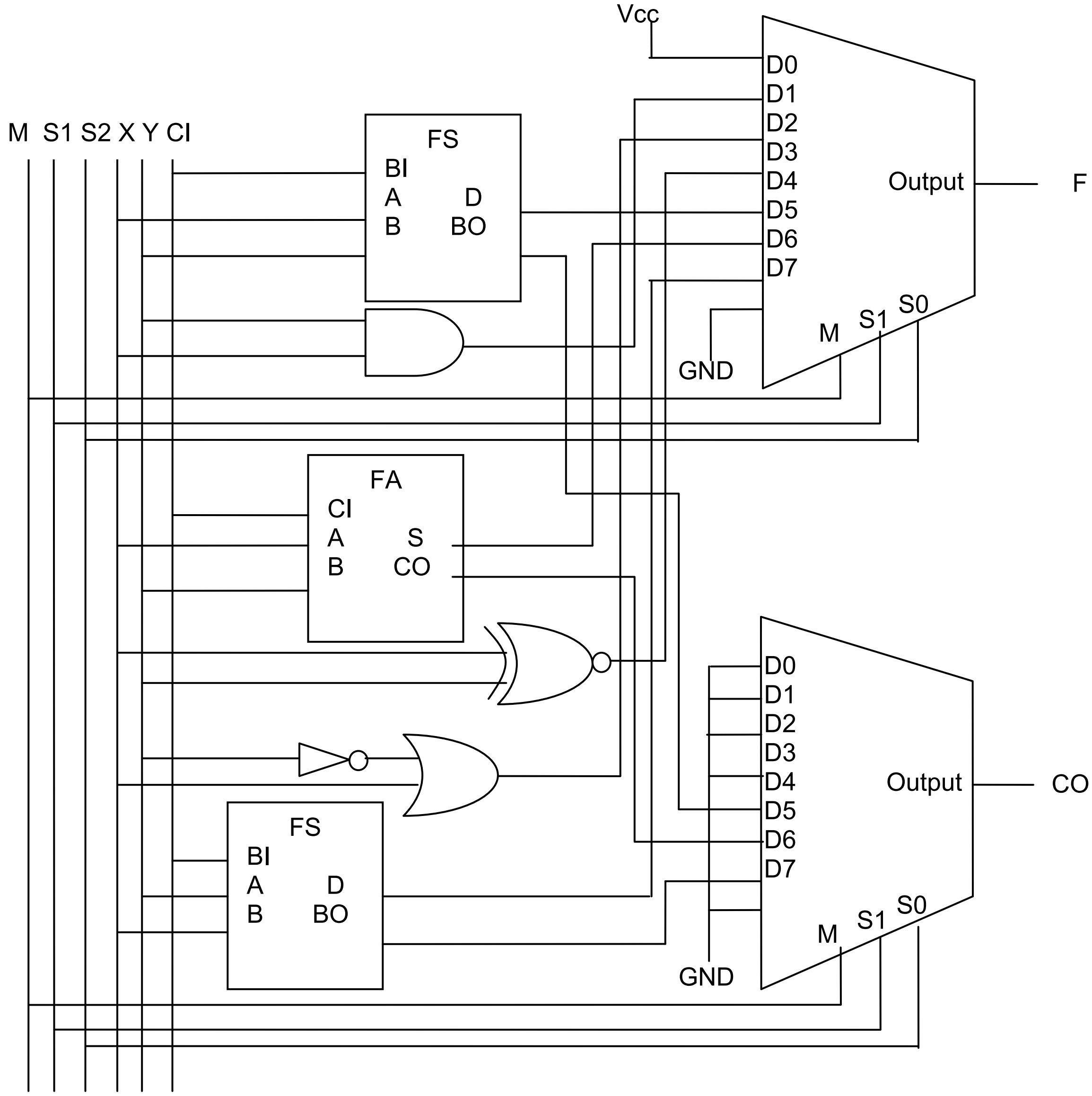
10. Design a single-bit ALU, using the modular design process, that will perform the following specification:

Selection		Arithmetic/Logic Operation
M	P	CO F
0	0	Increment A
0	1	A plus B plus CI
1	0	A = B
1	1	Transfer A

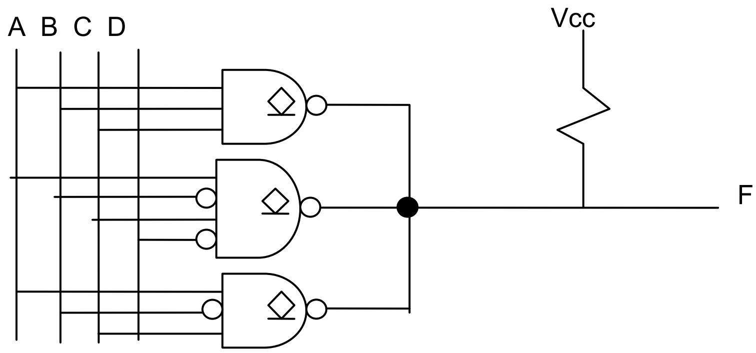
11. Design a single-bit ALU, using the modular design process that will perform the following specification:

Selection			Arithmetic/Logic
M	S1	S0	Operation
0	0	0	PRESET
0	0	1	X.Y
0	1	0	X + Y’
0	1	1	X $_{XNOR}$ Y
1	0	0	X MINUS Y
1	0	1	X PLUS Y

1	1	0	Y MINUS X
1	1	1	CLEAR



12. Is the circuit shown below a valid design? Explain your answer. If the answer is yes, write the Boolean expression for F(A, B, C, D).


A B C D
Vcc
F

Appendix A. Additional Resources

- ❖ Additional resources are available at the author's website www.EngrCS.com