NEHRU COLLEGE OF ENGINEERING AND RESEARCH CENTRE
(NAAC Accredited)
(Approved by AICTE, Affiliated to APJ Abdul Kalam Technological University, Kerala)

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

COURSE MATERIALS

EC 207: LOGIC CIRCUIT DESIGN

VISION OF THE INSTITUTION

To mould true citizens who are millennium leaders and catalysts of change through excellence in education.

MISSION OF THE INSTITUTION

NCERC is committed to transform itself into a center of excellence in Learning and Research in Engineering and Frontier Technology and to impart quality education to mould technically competent citizens with moral integrity, social commitment and ethical values.

We intend to facilitate our students to assimilate the latest technological know-how and to imbibe discipline, culture and spiritually, and to mould them in to technological giants, dedicated research scientists and intellectual leaders of the country who can spread the beams of light and happiness among the poor and the underprivileged.

ABOUT DEPARTMENT

- Established in: 2002
- Course offered : B.Tech in Electronics and Communication Engineering

  M.Tech in VLSI
- Approved by AICTE New Delhi and Accredited by NAAC
- Affiliated to the A P J Abdul Kalam Technological University.

DEPARTMENT VISION

Provide well versed, communicative Electronics Engineers with skills in Communication systems with corporate and social relevance towards sustainable developments through quality education.

DEPARTMENT MISSION

1) Imparting Quality education by providing excellent teaching, learning environment.

2) Transforming and adopting students in this knowledgeable era, where the electronic gadgets (things) are getting obsolete in short span.

3) To initiate multi-disciplinary activities to students at earliest and apply in their respective fields of interest later.

4) Promoting leading edge Research & Development through collaboration with academia & industry.

PROGRAMME EDUCATIONAL OBJECTIVES

PEO1. To prepare students to excel in postgraduate programmes or to succeed in industry / technical profession through global, rigorous education and prepare the students to practice and innovate recent fields in the specified program/ industry environment.

PEO2. To provide students with a solid foundation in mathematical, Scientific and engineering fundamentals required to solve engineering problems and to have strong practical knowledge required to design and test the system.

PEO3. To train students with good scientific and engineering breadth so as to comprehend, analyze, design, and create novel products and solutions for the real life problems.

PEO4. To provide student with an academic environment aware of excellence, effective communication skills, leadership, multidisciplinary approach, written ethical codes and the life-long learning needed for a successful professional career.

PROGRAM OUTCOMES (POS)

Engineering Graduates will be able to:

1. **Engineering knowledge**: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
2. **Problem analysis**: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
3. **Design/development of solutions**: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
4. **Conduct investigations of complex problems**: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
5. **Modern tool usage**: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.

6. **The engineer and society**: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
7. **Environment and sustainability**: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
8. **Ethics**: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
9. **Individual and team work**: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
10. **Communication**: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
11. **Project management and finance**: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
12. **Life-long learning**: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

PROGRAM SPECIFIC OUTCOMES (PSO)

PSO1: Facility to apply the concepts of Electronics, Communications, Signal processing, VLSI, Control systems etc., in the design and implementation of engineering systems.

PSO2: Facility to solve complex Electronics and communication Engineering problems, using latest hardware and software tools, either independently or in team.optimization.

SYLLABUS

EC 207	LOGIC CIRCUIT DESIGN	CATEGORY	L	T	P	CREDIT
		PCC	3	1	0	4

Preamble: This course aims to impart the basic knowledge of logic circuits and enable students to apply it to design a digital system.

Prerequisite: Basics of Electrical and Electronics Engineering

Course Outcomes: After the completion of the course the student will be able to

CO 1	Explain the elements of digital system abstractions such as digital representations of information, digital logic and Boolean algebra
CO 2	Create an implementation of a combinational logic function described by a truth table using and/or/inv gates/ muxes
CO 3	Compare different types of logic families with respect to performance and efficiency
CO 4	Design a sequential logic circuit using the basic building blocks like flip-flops
CO 5	Design and analyze combinational and sequential logic circuits through gate level Verilog models.

Mapping of course outcomes with program outcomes

	PO 1	PO 2	PO 3	PO 4	PO 5	PO 6	PO 7	PO 8	PO 9	PO 10	PO 11	PO 12
CO 1	3	3										
CO 2	3	3	3									
CO 3	3	3										
CO 4	3	3	3									
CO 5	3	3	3		3							

Assessment Pattern

Bloom's Category	Continuous Assessment Tests		End Semester Examination
	1	2	
Remember	10	10	10
Understand	20	20	20
Apply	20	20	70
Analyse			
Evaluate			
Create			

Mark distribution

Total Marks	CIE	ESE	ESE Duration
150	50	100	3 hours

Continuous Internal Evaluation Pattern:

Attendance : 10 marks
Continuous Assessment Test (2 numbers) : 25 marks
Course project : 15 marks

Course Level Assessment Questions

Course Outcome 1 (CO1) : Number Systems and Codes

1. Consider the signed binary numbers A = 01000110 and B = 11010011 where B is in 2's complement form. Find the value of the following mathematical expression (i) A + B (ii) A – B
2. Perform the following operations (i)$D9CE_{16}$-$CFDA_{16}$ (ii) 6575_8-5732_8
3. Convert decimal 6,514 to both BCD and ASCII codes. For ASCII, an even parity bit is to be appended at the left.

Course Outcome 2 (CO2) : Boolean Postulates and combinational circuits

1. Design a magnitude comparator to compare two 2-bit numbers A = A_1A_0 and B = B_1B_0B
2. Simplify using K-map F(a,b,c,d) = Σ m (4,5,7,8,9,11,12,13,15)
3. Explain the operation of a 8x1 multiplexer and implement the following using an 8x1 multiplexer F(A, B, C, D) = Σ m (0, 1, 3, 5, 6, 7, 8, 9, 11, 13, 14)

Course Outcome 3 (CO3) : Logic families and its characteristics

1. Define the terms noise margin, propagation delay and power dissipation of logic families. Compare TTL and CMOS logic families showing the values of above mentioned terms.
2. Draw the circuit and explain the operation of a TTL NAND gate
3. Compare TTL, CMOS logic families in terms of fan-in, fan-out and supply voltage

Course Outcome 4 (CO4) : Sequential Logic Circuits

1. Realize a T flip-flop using NAND gates and explain the operation with truth table, excitation table and characteristic equation
2. Explain a MOD 6 asynchronous counter using JK Flip Flop
3. Draw the logic diagram of 3 bit PIPO shift register with LOAD/SHIFT control and explain its working

Course Outcome 5 (CO5) : Logic Circuit Design using HDL

1. Design a 4-to-1 mux using gate level Verilog model.
2. Design a verilog model for a half adder circuit. Make a one bit full adder by connecting two half adder models.
3. Compare concurrent signal assignment versus sequential signal assignment.

Syllabus

Module 1: Number Systems and Codes:

Binary and hexadecimal number systems; Methods of base conversions; Binary and hexadecimal arithmetic; Representation of signed numbers; Fixed and floating point numbers; Binary coded decimal codes; Gray codes; Excess 3 code. Alphanumeric codes: ASCII. Basics of verilog -- basic language elements: identifiers, data objects, scalar data types, operators.

Module 2: Boolean Postulates and Fundamental Gates

Boolean postulates and laws – Logic Functions and Gates De-Morgan's Theorems, Principle of Duality, Minimization of Boolean expressions, Sum of Products (SOP), Product of Sums (POS), Canonical forms, Karnaugh map Minimization. Modeling in verilog, Implementation of gates with simple verilog codes.

Module 3: Combinatorial and Arithmetic Circuits

Combinatorial Logic Systems - Comparators, Multiplexers, Demultiplexers, Encoder, Decoder. Half and Full Adders, Subtractors, Serial and Parallel Adders, BCD Adder. Modeling and simulation of combinatorial circuits with verilog codes at the gate level.

Module 4: Sequential Logic Circuits:

Building blocks like S-R, JK and Master-Slave JK FF, Edge triggered FF, Conversion of Flipflops, Excitation table and characteristic equation. Implementation with verilog codes. Ripple and Synchronous counters and implementation in verilog, Shift registers-SIPO, SISO, PISO, PIPO. Shift Registers with parallel Load/Shift, Ring counter and Johnsons counter. Asynchronous and Synchronous counter design, Mod N counter. Modeling and simulation of flipflops and counters in verilog.

Module 5: Logic families and its characteristics:

TTL, ECL, CMOS - Electrical characteristics of logic gates – logic levels and noise margins, fan-out, propagation delay, transition time, power consumption and power-delay product. TTL inverter - circuit description and operation; CMOS inverter - circuit description and operation; Structure and operations of TTL and CMOS gates; NAND in TTL and CMOS, NAND and NOR in CMOS.

Text Books

1. Mano M.M., Ciletti M.D., "Digital Design", Pearson India, 4th Edition. 2006
2. D.V. Hall, "Digital Circuits and Systems", Tata McGraw Hill, 1989
3. S. Brown, Z. Vranesic, "Fundamentals of Digital Logic with Verilog Design", McGraw Hill
4. Samir Palnikar"Verilog HDL: A Guide to Digital Design and Syntheis", Sunsoft Press
5. R.P. Jain, "Modern digital Electronics", Tata McGraw Hill, 4th edition, 2009

Reference Books

1. W.H. Gothmann, "Digital Electronics – An introduction to theory and practice", PHI, 2nd edition ,2006
2. Wakerly J.F., "Digital Design: Principles and Practices," Pearson India, 4th 2008

.

3. A. Ananthakumar ,"Fundamentals of Digital Circuits", Prentice Hall, 2nd edition, 2016
4. Fletcher, William I., An Engineering Approach to Digital Design, 1st Edition, Prentice Hall India, 1980

Course Contents and Lecture Schedule

No	Topic	No. of Lectures
1	**Number Systems and Codes**:	
1.1	Binary, octal and hexadecimal number systems; Methods of base conversions;	2
1.2	Binary, octal and hexadecimal arithmetic;	1
1.3	Representation of signed numbers; Fixed and floating point numbers;	3
1.4	Binary coded decimal codes; Gray codes; Excess 3 code :	1
1.5	Error detection and correction codes - parity check codes and Hamming code-Alphanumeric codes:ASCII	3
1.6	Verilog basic language elements: identifiers, data objects, scalar data types, operators	2
2	**Boolean Postulates and Fundamental Gates:**	
2.1	Boolean postulates and laws – Logic Functions and Gates, De-Morgan's Theorems, Principle of Duality	2
2.2	Minimization of Boolean expressions, Sum of Products (SOP), Product of Sums (POS)	2

2.3	Canonical forms, Karnaugh map Minimization	1
2.4	Gate level modelling in Verilog: Basic gates, XOR using NAND and NOR	2
3	**Combinatorial and Arithmetic Circuits**	
3.1	Combinatorial Logic Systems - Comparators, Multiplexers, Demultiplexers	2
3.2	Encoder, Decoder, Half and Full Adders, Subtractors, Serial and Parallel Adders, BCD Adder	3
3.3	Gate level modelling combinational logic circuits in Verilog: half adder, full adder, mux, demux, decoder, encoder	3
4	**Sequential Logic Circuits:**	
4.1	Building blocks like S-R, JK and Master-Slave JK FF, Edge triggered FF	2
4.2	Conversion of Flipflops, Excitation table and characteristic equation.	1
4.3	Ripple and Synchronous counters, Shift registers-SIPO.SISO,PISO,PIPO	2
4.4	Ring counter and Johnsons counter, Asynchronous and Synchronous counter design	3
4.5	Mod N counter, Random Sequence generator	1
4.6	Modelling sequential logic circuits in Verilog: flipflops, counters	2
5	**Logic families and its characteristics**:	
5.1	TTL,ECL,CMOS- Electrical characteristics of logic gates – logic levels and noise margins, fan-out, propagation delay, transition time, power consumption and power-delay product.	3
5.2	TTL inverter - circuit description and operation	1
5.3	CMOS inverter - circuit description and operation	1
5.4	Structure and operations of TTL and CMOS gates; NAND in TTL, NAND and NOR in CMOS.	2

QUESTION BANK

Module-1

1. Perform inter-conversions of the following:
(a) $AC0BFE_{16}$ to Binary, Octal and Decimal
(b) 77504_{8} to Binary, Hexadecimal and Decimal
(c) 11101_{2} to Octal, Hexadecimal and Decimal
(d) 77504_{10} to Binary, Hexadecimal and Octal
(e) 77504_{10} to BCD 8421and 2421
(f) 137_{10} to IEEE 754 Floating point number

2. Perform the subtraction of 65_{10} from 110_{10} using 2's complement arithmetic on 8-bit signed numbers and validate your answer.

3. Verify the following Verilog relational statement:

Given A, B, C, and D are operands, show with steps,
Given that A=110, B=111, C=011000, D=111011
4. Show the floating-point representation of the decimal number 228, in version-1, version-2 and IEEE 754 notation.
5. Represent decimal number 228 using excess-3 code.
6. Explain with the aid of examples the 8421, 2421, Excess-3 and Gray Code. State which are weighted codes and unweighted codes.
7. Add the numbers $FADE.BEE_{16}$ and $BAD.FAB_{16}$ using hexadecimal arithmetic.
8. Explain the Data types in Verilog with examples.
9. Show the floating-point representation of the decimal number 232, in version-1, version-2 and IEEE 754 notation.

10. Add the numbers $DEAD.BEEF_{16}$ and 100011.101001_{2} using hex arithmetic.

11. Add the numbers 677.432_{8} and 333.123_{8} using octal arithmetic with the aid of octal number line.

12. Explain the operation of each of the Verilog statements given below:

```
reg [7:0] b= 8'hA3;
{Carry, Sum} = a+ b;
integer signed a= 16'hBEEF;
```

13. Show the floating-point representation of the decimal number 238, in version-1, version-2 and IEEE 754 notation.

14. State what is Gray code? Draw the 4-bit Gray code representation. Provide the applications.

15. Explain about use of Parity bits for error detection.

16. Perform the following additions using bcd arithmetic and validate your answer: Add 12345 and 7234.

Module-2

1. State De Morgan's Theorem and the rules. Apply the theorem as many times as needed to obtain the complement of the following function in standard canonical form:
   f ' = (x'.y. z' + x'.y'.z)'
2. Explain the Principle of Duality with the aid of examples.
3. State and prove the Involution Theorem and the Absorption Theorem.
4. State and prove the Associative Theorem and the Idempotent Theorem.
5. What is a Truth Table? With the aid of Truth Table prove the De Morgan's Theorem.
6. Examine the different sets of Logic Gates with the aid of symbol, function and truth table.
7. Compare and contrast Buffer gate and Invertor gate.
8. Decompose the Exclusive OR Function using universal gates with the aid of truth table, logic diagram and modified logic diagram. Write Verilog program for the result.
9. Simplify the following Boolean expression using appropriate Karnaugh Map and provide the logic implementation of the minimized expression using gates.

   $$f(A, B, C, D) = \sum m(4, 5, 7, 8, 9, 11, 12, 13, 15)$$

10. Construct the Verilog program module for the implementation in Q.9 and carefully provide comments for each line of the code.
11. Explain the standard canonical forms for representing Boolean functions with the aid of two examples each.
12. Explain the meaning of Literal, Minterm, Maxterm, Don't Care term, SOP and POS. Give examples to support your answers.
13. Simplify the following Boolean expression using appropriate Karnaugh Map and provide the logic implementation of the minimized expression using universal gates with the aid of De Morgan's laws.

$f=\sum m(0,2,3,4,5,6)$

14. Construct the Verilog program module for the above implementation in Q.13 and carefully provide comments for each line of the code.

15. Decompose the Exclusive NOR Function using universal gates with the aid of truth table, logic diagram and modified logic diagram. Write Verilog program for the result.

16. State De Morgan's Theorem and the rules. Apply the theorem as many times as needed to obtain the complement of the following function in standard canonical form:
$f_2' = [(x.(y'. z' + y.z)]'$

17. Simplify the following Boolean expression using appropriate Karnaugh Map and provide the logic implementation of the minimized expression using universal gates with the aid of De Morgan's laws.

$\pi M (0,3,5,6,7)$.

18. Construct the Verilog program module for the above implementation in Q.17 and carefully provide comments for each line of the code.

19. Write a well commented Verilog program for a circuit that has four input signals, x_1, x_2, x_3, and x_4, and three output signals, f, g, and h, and implements the logic functions:
$g = x_1.x_3 + x_2.x_4$
$h = (x_1 + x_3')(x_2' + x_4)$
$f = g + h$

20. Obtain the complement of the following function in standard canonical form by applying the De Morgan's rule as many times as needed, as well as Principle of Duality:
$f_1 = (x + y + z)(x + y' + z)(x' + y + z')(x' + y' + z)$

21. Construct an XOR gate using NAND gates and perform gate level modeling using Verilog.

22. Construct an XOR gate using NOR gates and perform gate level modeling using Verilog.

23. Repeat the above for XNOR Gate.

Module-3 (Part-1)

1. State what is meant by a Combinational circuit? Draw a generic block diagram. Give four examples.
2. Analyze the Comparator circuits with the aid of graphic symbol and logic implementations.
3. Define the Multiplexer function. Analyze the 4:1 MUX with the aid of graphic symbol, truth table and logic implementation.
4. Implement the following Boolean function using an appropriate Multiplexer after judicious manipulation of the original truth table.

$f = w1 \oplus w2 \oplus w3$

5. Give a practical application of Multiplexer circuits in Digital Electronics and Communications or Entertainment. Try to specify a MUX of specific size and provide the blueprint for your design.
6. State what is meant by a Decoder? Analyze the 2:4 Decoder with the aid of graphic symbol, truth table and logic circuit.
7. Analyze how the basic Decoder can be made more efficient by means of the Enable input. Draw the graphic symbol, truth table and logic circuit.
8. Define the Demultiplexer function. Analyze the 1:8 DEMUX with the aid of graphic symbol, truth table and logic implementation. Give an application of this function.
9. Implement the following Boolean function using an appropriate Multiplexer after judicious manipulation of the original truth table.
   $f = x \textcircled{S} y \textcircled{S} z$,
   where Ⓢ→ Ex-NOR function
10. State what is meant by a Magnitude Comparator? Analyze the 4-bit Magnitude Comparator with the aid of graphic symbol, truth table and logic circuit.
11. Analyze the 3:8 Decoder by means of the graphic symbol, truth table and logic circuit. How would an extra enable input improve its robustness?
12. State what is meant by Encoder? Analyze the 4:2 Encoder by means of the graphic symbol, truth table and logic circuit.
13. Investigate the Priority Encoder circuit as an improvement over plain ol' Encoder with the aid of an example.
14. Analyze the Full Adder and Half Adder with the aid of truth table and logic circuit diagram.

Module-3 (Part-2)

1. Describe the BCD Adder with the help of Logic Diagram and Functional table of operation.
2. Analyze the Ripple Carry Adder with the help of Logic Diagram and Functional table of operation.

3. Describe the Binary Subtractor with overflow detection with the help of Logic Diagram and operation. Also explain the detection of Overflow.
4. Describe the Priority Encoder with the help of Logic Diagram and Functional table of operation
5. Analyze the 4-bit Magnitude Comparator with the help of logic expressions, Logic Diagram and operational description.
6. Analyze the 16:1 Multiplexer using the 4:1 Multiplexer with logic diagram and develop the Hierarchical Verilog Code for the 16:1 Multiplexer using functional description of 4:1 Multiplexer code. Describe the steps in arriving at the final code.
7. State what is meant by a Decoder? Analyze the 2:4 Decoder with the aid of graphic symbol, truth table and logic circuit.
8. Analyze how the basic Decoder can be made more efficient by means of the Enable input. Draw the graphic symbol, truth table and logic circuit.
9. Analyze the 2:4 Decoder with the aid of truth table and logic diagram and develop the Verilog Code using **`case statement`** for the 2:4 Decoder system. Describe the steps in arriving at the final code.
10. Use Verilog code to model a Full Adder system using Gate primitives, by starting with the logic diagram of a Full Adder.
11. Use Verilog code to model a 4-bit Binary Adder system using Gate primitives, by starting with the functional description of a Full Adder.
12. Use Verilog code to model a 4:1 MUX using Gate primitives, by starting with the logic diagram of a Full Adder.
13. Use Verilog code to model a 16:1 MUX using Hierarchical coding, by starting with the functional description of a 4:1 MUX.
14. Use Verilog code to model a 2:4 Decoder using Gate primitives, by starting with the logic diagram.
15. Use Verilog code to model a 2:4 Decoder in an alternative way using case statement along with Gate primitives, by starting with the logic diagram.

Module-4

1. What is a Sequential system? With the aid of a block diagram, explain how a sequential circuit can be constructed.
2. Analyze the S R Latch with the aid of Logic diagram and Functional table. How does the addition of an enable input make the latch operate with clock signal. What is the main drawback of this latch?
3. Analyze the Transparent Latch with the aid of Logic diagram and Functional table. Represent the Graphic symbols for this latch.
4. Analyze the ET Flip Flop or MSD Flip Flop with the aid of Logic diagram and Functional table. Represent the Graphic symbols for this flip flop.
5. Compare and Contrast Latch and Flip Flop. With diagrams, dissect the structure of a periodic clock signal that caters for each of these.
6. Describe the J-K Flip Flop with the aid of Logic Diagram and Functional table or Characteristic Table. Why is this called Universal Flip Flop?
7. Describe the realization of T Flip Flop and D Flip Flop from the Universal Flip Flop, with clear reference to additional hardware requirements.
8. Draw the Characteristic Tables, write down the characteristic equations and draw the Excitation Tables for the S-R, D, J-K and T flip flops.
9. Analyze the Binary Ripple Counter using D Flip Flops/ T Flip Flops and describe the operation with the aid of Count table.
10. Configure a 4-bit Synchronous Counter using J-K Flip Flops and provide the functional table, logic diagram, Count table and describe its operation. Compare with Ripple Counter of same capacity.
11. Configure a 4-bit Parallel Access Shift Register using D Flip- Flops and provide the functional table, logic diagram and describe its operation for the various modes such as SISO, SIPO, PISO and PIPO.
12. Describe the Ring Counter and the Johnson Counter with the aid of Logic Diagram, Functional table and details of operation.
13. Construct the hierarchical Verilog code for a 4-bit Shift Register with the help of functional code of D Flip Flop. Comment on the suitability of the code.
14. Construct the Verilog code for an Up Counter. Comment on the suitability of the code.

15. Analyze the Verilog construct of use in Sequential circuits and construct the code for a D Flip Flop operating on the rising edge of the clock, with use of Asynchronous reset input. Comment on the sensitivity list.

Module-5

1. Analyze the CMOS Logic Levels for Input voltage and Output voltages with the aid of diagrams.
2. Analyze the TTL Logic Levels for Input voltage and Output voltages with the aid of diagrams.
3. Justify the need for Noise Immunity in Digital Logic Systems.
4. Define Noise Margin. Explain the quantitative measures for Noise Margin with the aid of diagrams of CMOS 5 V family.
5. Determine the High-level and LOW-level noise margins for CMOS and for TTL using their logic level voltage ranges. Which is preferable for a noise prone environment?
6. Analyze the Power Dissipation in logic circuits and obtain quantitative measures of the same.
7. A certain gate draws 3 μA when its output is HIGH and 4.6 μA when its output is LOW. What is its average power dissipation if V_{cc} is 5 V and the gate is operated on a 50% duty cycle?
8. A certain IC gate has an I_{CCH} = 1.5μA and I_{CCL} = 2.8 μA. Determine the average Power dissipation for 50% duty cycle operation if V_{cc} is 5 V.
9. Analyze the Binary Ripple Counter using D Flip Flops/ T Flip Flops and describe the operation with the aid of Count table.
10. Compare and contrast the Power Dissipation in CMOS and TTL circuits.
11. Analyze the Propagation time Delay in Logic circuits and justify how the Speed- Power product can be used as a benchmark.
12. Explain the Loading and Fan-out of the Gates. How does excessive loading affect the Noise Margin of the gates?
13. Describe the CMOS Loading with the help of diagrams.
14. Describe the TTL Loading with the help of diagrams.
15. Analyze the CMOS Inverter circuit with the aid of Circuit Diagram and Operational Diagrams.

16. Analyze the TTL Inverter circuit with the aid of Circuit Diagram and Operational Diagrams.
17. Analyze the TTL NAND Gate circuit with the aid of Circuit Diagram and Operational Diagrams.
18. Analyze the CMOS NAND gate circuit with the aid of Circuit Diagram and functional table and operation.
19. Analyze the CMOS NOR gate circuit with the aid of Circuit Diagram and functional table and operation.
20. Explain the ECL Family of digital logic circuits.
21. Analyze the ECL NOR/OR gate with the aid of circuit diagram, operation and transfer characteristic.
22. Explain the Noise margin of ECL circuits.
23. Compare and contrast the ECL, CMOS and TTL family of logic gates.

ECT 203 LOGIC CIRCUIT DESIGN

This course aims to impart the basic knowledge of Logic Circuits and enable students to apply it to design a Digital System.

Module – I: Number Systems and Codes:

Introduction

A digital computer stores data in terms of digits (numbers) and proceeds in discrete steps from one state to the next. The states of a digital computer typically involve binary digits which may take the form of the presence or absence of magnetic markers in a storage medium , on-off switches or relays. In digital computers, even letters, words and whole texts are represented digitally.

Digital Logic is the basis of electronic systems, such as computers and cell phones. Digital Logic is rooted in binary code, a series of zeroes and ones each having an opposite value. This system facilitates the design of electronic circuits that convey information, including logic gates. Digital Logic gate functions include and, or and not. The value system translates input signals into specific output. Digital Logic facilitates computing, robotics and other electronic applications.

Digital Logic Design is foundational to the fields of electrical engineering and computer engineering. Digital Logic designers build complex electronic components that use both electrical and computational characteristics. These characteristics may involve power, current, logical function, protocol and user input. Digital Logic Design is used to develop hardware, such as circuit boards and microchip processors. This hardware processes user input, system protocol and other data in computers, navigational systems, cell phones or other high-tech systems.

1.2 NUMBER SYSTEMS

Decimal Numbers

A decimal number such as 7,392 represents a quantity equal to 7 thousands, plus 3 hundreds, plus 9 tens, plus 2 units. The thousands, hundreds, etc., are powers of 10 implied by the position of the coefficients (symbols) in the number. To be more exact, 7,392 is a shorthand notation for what should be written as:

$7\times10^3 + 3\times10^2 + 9\times10^1 + 2\times10^0$

However, the convention is to write only the numeric coefficients and, from their position, deduce the necessary powers of 10, with powers increasing from right to left. In general, a number with a decimal point is represented by a series of coefficients:

a5a4a3a2a1a0**.** a−1a−2a−3

The coefficients aj are any of the 10 digits (0, 1, 2, . . . , 9), and the subscript value *j* gives the place value and, hence, the power of 10 by which the coefficient must be multiplied.

So 7392 can be expanded with a3=7, a2=3, a1=9, and a0=2, and the other coefficients equal to zero.

The radix of a number system determines the number of distinct values that can be used to represent any arbitrary number. The decimal number system is said to be of base, or radix, 10 because it uses 10 digits and the coefficients are multiplied by powers of 10.

The radix point (e.g., the decimal point) distinguishes positive powers of 10 from negative powers of 10.

Binary Numbers

The binary system is a different number system. The coefficients of the binary number system have only two possible values: 0 and 1. So the radix is 2. Each coefficient aj is multiplied by a power of the radix, for example, 2^j, and the results are added to obtain the decimal equivalent of the number. The radix point or the binary point distinguishes positive powers of 2 from negative powers of 2.

For example, consider the binary number 11010.11_2

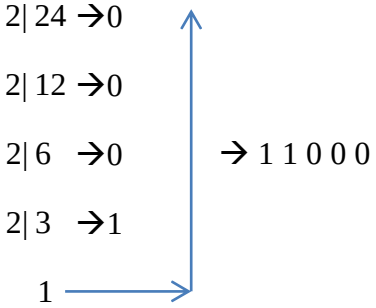
Find the decimal equivalent of the binary number 11010.11_2

11010.11_2 can be expanded as $1\times2^4 + 1\times2^3 + 0\times2^2 + 1\times2^1 + 0\times2^0 + 1\times2^{-1} + 1\times2^{-2} = 26.75_{10}$

So 26.75_{10} is the decimal equivalent of 11010.11_2

Given a decimal number, the binary equivalent can be found by repeatedly dividing by 2 and extracting the remainder of the series of divisions, till the quotient becomes 1 and then arrange the remainders from bottom to top.

For example, find the binary equivalent of 24_{10}



Therefore, $24_{10} \equiv 11000_2$

Octal Numbers

The octal system is a different number system. The coefficients of the octal number system have only eight possible values: 0, 1, 2, 3, 4, 5, 6 and 7. So the radix is 8. Each coefficient aj is multiplied by a power of the radix, for example, 8^j, and the results are added to obtain the

decimal equivalent of the number. The radix point or the octal point distinguishes positive powers of 8 from negative powers of 8.

For example, consider the octal number 1004.02_8

Find the decimal equivalent of the octal number 1004.02_8

1004.02_8 can be expanded as $1\times8^3 + 0\times8^2 + 0\times8^1 + 4\times8^0 + 0\times8^{-1} + 2\times8^{-2} = 516.03125_{10}$

So 516.03125_{10} is the decimal equivalent of 1004.02_8

The conversion between octal numbers and binary numbers is much simpler and faster to perform. Simply represent each octal digit as a combination of 3 binary digits or bits. Similarly to convert binary numbers to octal, group sets of 3 bits from the lsb to the msb. If you run out of bits add zeros from the msb.

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

Example: Convert the octal number 7765431_8 to binary

msb lsb

7765431_8 = 111 111 110 101 100 011 001 ≡ $1111111101011000110 01_2$

Example: Convert the binary number 1100011_2 to octal

= 001 100 011

1 4 3 → 143_8

Hexadecimal Numbers

The hexadecimal system is a different number system. The coefficients of the hex number system have 16 possible values: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E and F. So the radix is 16. Each coefficient aj is multiplied by a power of the radix, for example, 16^j, and the results are added to obtain the decimal equivalent of the number. The radix point or the hex point distinguishes positive powers of 16 from negative powers of 16.

For example, consider the hex number $100A_{16}$

Find the decimal equivalent of the hex number $100A_{16}$

$100A_{16}$ can be expanded as $1\times16^3 + 0\times16^2 + 0\times16^1 + 10\times16^0 = 4106_{10}$

So 4106_{10} is the decimal equivalent of $100A_{16}$

The conversion between hex numbers and binary numbers is much simpler and faster to perform. Simply represent each hex digit as a combination of 4 binary digits or bits. Similarly to convert binary numbers to hex, group sets of 4 bits from the lsb to the msb. If you run out of bits add zeros from the msb.

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

Example: Convert the HEX number $ABCDE_{16}$ to binary

msb lsb

$ABCDE_{16}$= 1010 1011 1100 1101 1110≡ 10101011110011011110_2

Example: Convert the binary number 1100011_2 to hex

= 0000 0110 0011

0 6 3 → 063_{16}

More examples:

$$1000110011110_2 = 100\ 011\ 001\ 110_2 = 4316_8$$
$$1110110111010100 1_2 = 011\ 101\ 101\ 110\ 101\ 001_2 = 355651_8$$

The procedure for binary to hexadecimal conversion is similar, except we use groups of four bits:

$$1000110011110_2 = 1000\ 1100\ 1110_2 = 8CE_{16}$$
$$1110110111010100 1_2 = 0001 1101\ 1011\ 1010\ 1001_2 = 1DBA9_{16}$$

In these examples we have freely added zeroes on the left to make the total number of bits a multiple of 3 or 4 as required.

Table 1 Binary, decimal, octal, and hexadecimal numbers.

Binary	*Decimal*	*Octal*	*3-Bit String*	*Hexadecimal*	*4-Bit String*
0	0	0	000	0	0000
1	1	1	001	1	0001
10	2	2	010	2	0010
11	3	3	011	3	0011
100	4	4	100	4	0100
101	5	5	101	5	0101
110	6	6	110	6	0110
111	7	7	111	7	0111
1000	8	10	—	8	1000
1001	9	11	—	9	1001
1010	10	12	—	A	1010
1011	11	13	—	B	1011
1100	12	14	—	C	1100
1101	13	15	—	D	1101
1110	14	16	—	E	1110
1111	15	17	—	F	1111

If a binary number contains digits to the right of the binary point, we can convert them to octal or hexadecimal by starting at the binary point and working right. Both the left-hand and right-hand sides can be padded with zeroes to get multiples of three or four bits, as shown in the example below:

$$10.1011001011_2 = 010\,.\,101\ 100\ 101\ 100_2 = 2.5454_8$$
$$= 0010\,.\,1011\ 0010\ 1100_2 = 2.B2C_{16}$$

Converting in the reverse direction, from octal or hexadecimal to binary, is very easy. We simply replace each octal or hexadecimal digit with the corresponding 3- or 4-bit string, as shown below:

$$1357_8 = 001\ 011\ 101\ 111_2$$
$$2046.17_8 = 010\ 000\ 100\ 110\,.\,001\ 111_2$$
$$BEAD_{16} = 1011\ 1110\ 1010\ 1101_2$$
$$9F.46C_{16} = 1001\ 111\,.\,0100\ 0110\ 1100_2$$

The octal number system was quite popular 25 years ago because of certain minicomputers that had their front-panel lights and switches arranged in groups of three. However, the octal number system is not used much today, because of the preponderance of machines that process 8-bit *bytes*. It is difficult to extract individual byte values in multibyte quantities in the octal representation; for example, what are the octal values of the four 8-bit bytes in the 32-bit number with octal representation 12345670123_8?

In the hexadecimal system, two digits represent an 8-bit byte, and $2n$ digits represent an n-byte word; each pair of digits constitutes exactly one byte. For example, the 32-bit hexadecimal number $5678ABCD_{16}$ consists of four bytes with values 56_{16}, 78_{16}, AB_{16}, and CD_{16}. In this context, a 4-bit hexadecimal digit is sometimes called a *nibble*; a 32-bit (4-byte) number has eight nibbles. Hexadecimal numbers are often used to describe a computer's memory address space. For example, a computer with 16-bit addresses might be described as having read/write memory installed at addresses $0–EFFF_{16}$, and read-only memory at addresses $F000–FFFF_{16}$. Many computer programming languages use the prefix "0x" to denote a hexadecimal number, for example, `0xBFC0000`.

Table -2 Conversion methods for common radices.

Conversion	Method	Example
Binary to		
Octal	Substitution	$10111011001_2 = 10\ 111\ 011\ 001_2 = 2731_8$
Hexadecimal	Substitution	$10111011001_2 = 101\ 1101\ 1001_2 = 5D9_{16}$
Decimal	Summation	$10111011001_2 = 1\cdot 2^{10} + 0\cdot 2^9 + 1\cdot 2^8 + 1\cdot 2^7 + 1\cdot 2^6 + 0\cdot 2^5 + 1\cdot 2^4 + 1\cdot 2^3 + 0\cdot 2^2 + 1\cdot 2^1 + 1\cdot 2^0 = 1497_{10}$
Octal to		
Binary	Substitution	$1234_8 = 001\ 010\ 011\ 100_2$
Hexadecimal	Substitution	$1234_8 = 001\ 010\ 011\ 100_2 = 0010\ 1001\ 1100_2 = 29C_{16}$
Decimal	Summation	$1234_8 = 1\cdot 512 + 2\cdot 64 + 3\cdot 8 + 4\cdot 1 = 668_{10}$ How??
Hexadecimal to		
Binary	Substitution	$C0DE_{16} = 1100\ 0000\ 1101\ 1110_2$
Octal	Substitution	$C0DE_{16} = 1100\ 0000\ 1101\ 1110_2 = 1\ 100\ 000\ 011\ 011\ 110_2 = 140336_8$
Decimal	Summation	$C0DE_{16} = 12\cdot 4096 + 0\cdot 256 + 13\cdot 16 + 14\cdot 1 = 49374_{10}$
Decimal to		
Binary	Division	$108_{10} \div 2 = 54$ remainder 0 (LSB) $\div 2 = 27$ remainder 0 $\div 2 = 13$ remainder 1 $\div 2 = 6$ remainder 1 $\div 2 = 3$ remainder 0 $\div 2 = 1$ remainder 1 $\div 2 = 0$ remainder 1 (MSB) $108_{10} = 1101100_2$
Octal	Division	$108_{10} \div 8 = 13$ remainder 4 (least significant digit) $\div 8 = 1$ remainder 5 $\div 8 = 0$ remainder 1 (most significant digit) $108_{10} = 154_8$
Hexadecimal	Division	$108_{10} \div 16 = 6$ remainder 12 (least significant digit) $\div 16 = 0$ remainder 6 (most significant digit) $108_{10} = 6C_{16}$

Arithmetic: Decimal Addition/Subtraction

Use decimal number line:

0	1	2	3	4	5	6	7	8	9

0+1 = 1; 1+1 = 2; 2+1 = 3; 3+1 = 4; 4+1 = 5; 5+1 = 6; 6+1 = 7; 7+1 = 8; 8+1=9; 9+1=10
0+2 = 2; 1+4 = 5; 2+6 = 8; 3+5 = 8; 4+5 = 9; 5+2 = 7; 6+1 = 7; 7+2 = 9; 8+0=8; 9+2=11

9-2 = 7; 8-4 = 4; 2-6 = -4; 3-5 = -2; 4-5 = -1; 5-2 = 3; 6-1 = 5; 7-5 = 9; 8-8=0; 9-2=7

Sample additions:

```
 1           1
 7920711.716+
   234020.019
---------------------------
 8154731.735
---------------------------
```

Arithmetic: Binary Addition/Subtraction

Use binary number line:

0	1

0+1 = 1; 1+1 = 10; 10+1 = 11; 11+1 = 100;
1-1 = 0; 1-0 = 1; 11-1=10; 10-1=1; 0-1 = -1;

Sample additions:

```
      11 1111  1
    1110111.111+
     111000 .010
-----------------------------------
   10110000. 0 0 1
-----------------------------------
```

Arithmetic: Octal Addition/Subtraction

Use octal number line:

0	1	2	3	4	5	6	7

0+1 = 1; 1+1 = 2; 2+1 = 3; 3+1 = 4; 4+1 = 5; 5+1 = 6; 6+1 = 7; 7+1 = 10; 7+2=11; 7+3=12
0+2 = 2; 1+4 = 5; 2+6 = 10; 3+5 = 10; 4+5 = 11; 5+2 = 7; 6+1 = 7; 7+2 = 11; 11+7=20; 17+2=21

2-6 = -4; 3-5 = -2; 4-5 = -1; 5-2 = 3; 10-1 = 7; 11-2 = 7; 12-5=5;
Sample addition:

```
 1   1     1   1
 6 7 2 0 7 1 1 . 7 1 6 +
   2 3 4 1 2 0 . 1 1 2
-----------------------------
 7 1 5 5 0 3 2 . 0 3 0
-----------------------------
```

Arithmetic: Hex Addition/Subtraction

Use hexadecimal number line:

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

8 + 1 = 9; 9 + 1 = A; A + 1 = B; B + 1 = C; C + 1 = D; D + 1 = E; E + 1 = F; F + 1 = 10;
7 + 4 = B; 6 + 6 = C; 5 + 5 = A; 4 + 7 = B; F + 2 = 11; 6 + A = 10;
F-2 = D; F-4 = B; E-6 = 8; 3-F = -C; A-5 = 5;
Sample additions:

```
             1
 B 9 2 0 7 1 1 . E 1 A +
   2 3 4 0 2 0 . 0 1 9
-----------------------------
 B B 5 4 7 3 1 . E 3 3
-----------------------------
ABC.DEF+
012.347
----------
ACF.136
```

Representation of Negative Numbers

Signed-Magnitude Representation

In the *signed-magnitude system*, a number consists of a magnitude and a symbol indicating whether the magnitude is positive or negative. Thus, we interpret decimal numbers +98, –57, +123.5, and –13 in the usual way, and we also assume that the sign is "+" if no sign symbol is written. There are two possible representations of zero, "+0" and "–0", but both have the same value.

The signed-magnitude system is applied to binary numbers by using an extra bit position to represent the sign (the *sign bit*). Traditionally, the most significant bit (MSB) of a bit string is used as the sign bit (0 = plus, 1 = minus), and the lower-order bits contain the magnitude. Thus, we can write several 8-bit signed-magnitude integers and their decimal equivalents:

$01010101_2 = +85_{10}$ $\quad$ $11010101_2 = -85_{10}$

$01111111_2 = +127_{10}$ $\quad$ $11111111_2 = -127_{10}$

$00000000_2 = +0_{10}$ $\quad$ $10000000_2 = -0_{10}$

The signed-magnitude system has an equal number of positive and negative integers. An n-bit signed-magnitude integer lies within the range $-(2^{n-1}-1)$ through $+(2^{n-1}-1)$, and there are two possible representations of zero.

Practical design of digital logic circuit to add/sub signed magnitude numbers is complicated and is seldom undertaken.

One's (1's) Complement representation:

The diminished radix-complement system for binary numbers is called the *ones' complement*. the most significant bit is the sign, 0 if positive and 1 if negative. Thus there are two representations of zero, positive zero ($00\cdots00$) and negative zero ($11\cdots11$). Positive number representations are the same for both ones' and two's complements. However, negative number representations differ by 1. A weight of $-(2^{n-1}-1)$, rather than -2^{n-1}, is given to the most significant bit when computing the decimal equivalent of a ones'-complement number. The range of representable numbers is $-(2^{n-1}-1)$ through $+(2^{n-1}-1)$. Some 8-bit numbers and their ones' complements are shown below:

$17_{10} = 00010001_2$ $\Downarrow$ $11101110_2 = -17_{10}$

$-99_{10} = 10011100_2$ $\Downarrow$ $01100011_2 = 99_{10}$

$119_{10} = 01110111_2$ $\Downarrow$ $10001000_2 = -119_{10}$

$-127_{10} = 10000000_2$ $\Downarrow$ $01111111_2 = 127_{10}$

$0_{10} = 00000000_2$ (positive zero) $\Downarrow$ $11111111_2 = 0_{10}$ (negative zero)

The main advantages of 1's complement are its symmetry and ease of implementation. But the adder design for 1's complement numbers is not easy. Also zero detectors in a 1's complement system must check for both representations of zero, or must always convert 11…11 to 00…00.

Two's (2's) Complement representation:

For binary numbers, the radix complement is called the *two's complement*. The MSB of a number in this system serves as the sign bit; a number is negative if and only if its MSB is 1. The decimal equivalent for a two's-complement binary number is computed the same way as for an unsigned number, except that the weight of the MSB is -2^{n-1} instead of $+2^{n-1}$. The range of representable numbers is $-(2^{n-1})$ through $+(2^{n-1}-1)$. Some 8-bit examples are shown below:

$17_{10} = 00010001_2$ $\Downarrow$ complement bits

11101110

$+1$

$11101111_2 = -17_{10}$

$-99_{10} = 10011101_2$ $\Downarrow$ complement bits

01100010

$+1$

$01100011_2 = 99_{10}$

$119_{10} = 01110111$ $\Downarrow$ complement bits

10001000

$+1$

$10001001_2 = -119_{10}$

$-127_{10} = 10000001$ $\Downarrow$ complement bits

01111110

$+1$

$01111111_2 = 127_{10}$

$0_{10} =$ 00000000_2 ⇓ complement bits 11111111 $+1$ $\overline{1\ 00000000_2} = 0_{10}$

$-128_{10} =$ 10000000_2 ⇓ complement bits 01111111 $+1$ $\overline{10000000_2} = -128_{10}$

A carry out of the MSB position occurs in one case, . As in all two's-complement operations, this bit is ignored and only the low-order n bits of the result are used.

In the two's-complement number system, zero is considered positive because its sign bit is 0. Since two's complement has only one representation of zero, we end up with one extra negative number, $-(2^{n-1})$, that doesn't have a positive counterpart.

2's Complement numbers support sign extension. Most modern computers incorporate 2's complement system for subtraction.

Exercise!!!

- **1**. Subtract **5** from **17** using 8-bit signed numbers in 2's complement representation. Validate ur answer.
- Sol:

17: 0 0010001+
+5: 0 0000101
-5: 1 1111010 (1's Complement)
1 1 1 11010+
1
-5: 1 1 1 1 1011 (2's Complement)
---
1 0 0 0 0 1100 (+17 + -5)

0 0 0 0 1 1 0 0 → +12

*** 2**. Subtract **17** from **16** using 8-bit 2s' complement representation and validate ur answer.

Binary Coded Decimal Codes

- Binary numbers or bits are most suited for internal computations of digital systems.
- But humans prefer decimal numbers (Why?)
- So, external interfaces of digital system may read or display decimal numbers.
- Also some digital devices process decimal numbers directly.
- So, to do this, a decimal digit is represented by a string of bits.
- Different combinations of bit values in the string represent different decimal digits.
- For example, using a 4-bit string, we represent decimal 0 · 0000, 1 · 0001, 2 · 0010, and so on.

- **Code** is defined as a set of n-bit strings in which diff string patterns represent diff numbers or other things.
- **Code word** is a particular combination of n-bit string value.
- There may or may not be an arithmetic relationship between the code word and the thing it represents.
- A Code that uses n-bit strings need not contain 2^n valid code words.

(8421) BCD

- Binary Coded Decimal (BCD) encodes the digits 0 thro' 9 by their 4-bit unsigned bin representations 0000 thro' 1001.
- The code words 1010 thro' 1111 are not used.
- BCD is a weighted code as each decimal digit can be obtained from its code word by putting a fixed weight (2^3=8, 2^2=4, 2^1=2,2^0=1) for each code word bit.
- Place 2 BCD Digits in one 8-bit byte in Packed BCD representation.
- So one byte represents values from 0 to 99 in packed BCD repr as opposed to 0 to 255 for unsigned 8-bit binary num.
- BCD numbers with any desired number of digits can be got by using one byte for each two digits.

Decimal digit	BCD (8421)
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
	1010
	1011
	1100
	1101
	1110
	1111

Negative BCD numbers

- Signed BCD numbers have <u>one extra digit position</u> (4-bits) for the sign.
- Both the signed-magnitude and 10's complement representations are used.
- In signed-magnitude repr, the encoding of the sign bit is arbitrary.
- In 10's complement, **+** · 0000, **-** · 1001

Additions of BCD digits (Use correction of 6)

5	0101		4	0100	
+ 9	+ 1001		+ 5	+ 0101	
14	1110		9	1001	
	+ 0110	— correction			
10+4	1 0100				
8	1000		9	1001	
+ 8	+ 1000		+ 9	+ 1001	
16	1 0000		18	1 0010	
	+ 0110	— correction		+ 0110	— correction
10+6	1 0110		10+8	1 1000	

2421 Code

- This is also a weighted code having weights 2,4,2 and 1 for the code word bits from msb.
- The advantage of this code is that it is self-complementing, i.e., the code word for the 9's complement can be got by flipping the individual bits of the code word.

Decimal digit	BCD (8421)	2421
0	0000	0000
1	0001	0001
2	0010	0010
3	0011	0011
4	0100	0100
5	0101	1011
6	0110	1100
7	0111	1101
8	1000	1110
9	1001	1111
	1010	0101
	1011	0110
	1100	0111
	1101	1000
	1110	1001
	1111	1010

Excess-3 Code

- This is an unweighted code.
- The code word for a decimal digit is got by adding 0011_2 to the corresponding BCD code word.
- As the code words follow a standard binary counting sequence, standard binary counters can easily be made to count in excess-3 code.
- This is also a self-complementing code.

Decimal codes.

Decimal digit	*BCD (8421)*	*2421*	*Excess-3*
0	0000	0000	0011
1	0001	0001	0100
2	0010	0010	0101
3	0011	0011	0110
4	0100	0100	0111
5	0101	1011	1000
6	0110	1100	1001
7	0111	1101	1010
8	1000	1110	1011
9	1001	1111	1100
		Unused code words	
	1010	0101	0000
	1011	0110	0001
	1100	0111	0010
	1101	1000	1101
	1110	1001	1110
	1111	1010	1111

Gray Code

- A Digital Code in which there is only one bit changes between a pair of successive code words is called a Gray Code.
- Eg. 3-bit Gray code

Decimal number	Binary code	Gray code
0	000	000
1	001	001
2	010	011
3	011	010
4	100	110
5	101	111
6	110	101
7	111	100

- Is Gray Code a Weighted code?

Represent 4-bit Gray Code…

Gray Code

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

- The Gray code is used in applications where normal sequence of binary number generated by hardware produce an error during transition from one number to the next.
- If binary numbers are used, a change from 0111 to 1000 may produce an intermediate error number 1001, if lsb takes longer to change than do the values of the other three bits.
- This could have serious issues for the machine.
- The Gray code eliminates this problem, since only one bit changes its value during any transition between two numbers.

ASCII Character Code

- An alphanumeric character code is a set of code words that encodes the 10 decimal digits, the 26 letters of the alphabet, and a number of special characters.
- Such a set contains between 64 and 128 elements if both uppercase and lowercase letters are included, we need a binary code word of seven bits. (Why? · using **7 bit** can repr [**0 – 2^7-1**] · [**0 to 127**] = **128 combinations**)
- The standard binary code for the alphanumeric characters is the **American Standard Code for Information Interchange** (**ASCII**), which uses 7 bits to code 128 characters.

American Standard Code for Information Interchange (ASCII)

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

Control Characters

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

Error-Detecting Code

- To detect errors in data communication and processing, an eighth bit is sometimes added to the ASCII character to indicate its parity.
- A *parity bit* is an extra bit included with a message to make the total number of 1's either even or odd.
- Consider the following two characters and their even and odd parity:

	With even parity	With odd parity
ASCII A = 1000001	01000001	11000001
ASCII T = 1010100	11010100	01010100

Parity bit –Error Detection

- Insert an extra bit in leftmost position of code to produce an even no: of 1's in character for even parity or an odd number of 1's in character for odd parity.
- In general, one or the other parity is used, with even parity being more common.
- Parity bit is helpful in detecting errors during transmission of info from one location to another.
- This function is handled by generating an even parity bit at the sending end for each character.
- The eight-bit characters that include parity bits are transmitted to their destination.
- The parity of each character is then checked at the receiving end.
- If the parity of the received character is not even, then at least one bit has changed value during the transmission.
- This method detects one, three, or any odd combination of errors in each character that is transmitted.
- An even combination of errors, however, goes undetected, and additional error detection codes may be needed to take care of that possibility.

Basics of Error Correction

- After an error is detected ,need to correct it!
- Make request for retransmission of the message on the assumption that the error was random and will not occur again.
- If the receiver detects a parity error, it sends back the ASCII NAK (negative acknowledge) control character consisting of an even parity eight bits 10010101.
- If no error is detected, the receiver sends back an ACK (acknowledge) control character, namely, 00000110.

Basics of Error Correction

- The sending end will respond to an NAK by transmitting the message again until the correct parity is received.
- If, after a number of attempts, the transmission is still in error, a message can be sent to the operator to check for malfunctions in the transmission path.

Introduction to Verilog programming…

- **Verilog** HDL(Hardware Description Language) was invented by *Phil Moorby* and *Prabhu Goel* around 1984.
- It served as a proprietary hardware modeling language owned by Gateway Design Automation Inc. ...
- In 2001, extensions to **Verilog**-95 were submitted back to IEEE and became IEEE standard 1364-2001, known as **Verilog**-2001.

- In 1990, **Gateway Design Automation Inc.** was acquired by **Cadence Design System**, which is now one of the biggest suppliers of electronic design technologies and engineering services in the electronic design automation (**EDA**) industry.
- Cadence recognized the value of Verilog, and realized that if **Verilog** remained as a closed language, the pressure of standardization would eventually drive people to shift to **VHDL**.
- So in 1991 the **Open Verilog International** (OVI) (now known as **Accellera**) was organized by Cadence and the documentation of Verilog was transferred to public domain under the name of OVI.
- It was later submitted to **IEEE** and became **IEEE standard 1364-1995**, commonly referred as **Verilog-95**.

Verilog of the 20th Century ….

- In 2001, extensions to **Verilog-95** were submitted back to IEEE and became **IEEE standard 1364-2001**, known as **Verilog-2001**.
- The extensions covered some deficiencies that users had found in Verilog-95.
- One of the most significant upgrades was that signed variables (in 2's complement) became supported.
- **Verilog-2001** is now dominant edition of Verilog supported by most design tools.
- In 2005, **Verilog-2005** (**IEEE Standard 1364-2005**) was published with minor corrections and modifications.
- Also in 2005 **System Verilog**, a superset of **Verilog-2005**, with many new features and capabilities to aid design verification, was published.
- As of 2009, System Verilog and Verilog language standards were merged into **System Verilog 2009** (**IEEE Standard 1800-2009**), which is one of the most popular languages for IC design and verification today.
- **Xilinx® Vivado Design Suite**, released in 2013, can support **System Verilog** for **FPGA** design and verification.
- At our famed worthy NCERC Labs, we are devout followers of the same…

Verilog of the 21st Century ….

Comments

- A single line comment starts with // and tells Verilog compiler to treat everything after this point to the end of the line as a comment.
- A multiple-line comment starts with /* and ends with */ and cannot be nested.

```
1  // This is a single line comment
2
3  integer a;   // Creates an int variable called a, and treats everything to the r
4
5  /*
6  This is a
7  multiple-line or
8  block comment
9  */
10
11 /* This is /*
12 an invalid nested
13 block comment */
14 */
15
16 /* However,
17 // this one is okay
18 */
19
20 // This is also okay
21 ////////////// Still okay
```

Data Types

- Verilog has two main groups of data types: the *variable* data type and the *net* data type.
- These two groups differ in the way that they are assigned and hold values.
- They also represent different hardware structures.
- The *net* data types can represent physical connections between structural entities, such as gates.
- Generally, it does not store values.
- Instead, its value is determined by the values of its drivers, such as a continuous assignment or a gate.
- A very popular *net* data type is the **wire**.
- There are also several other predefined data types that are part of nets.
- Examples are **tri** (for tri-state), **wand** (for wired and), **wor** (for wired or).

The *variable* data type

- The *variable* data type is an abstraction of a data storage element.
- A variable shall store a value from one assignment to the next.
- An assignment statement in a procedure acts as a trigger that changes the value in the data storage element.
- A very popular *variable* data type is the **reg**.
- There are also several other predefined data types that are part of *variables*.
- Examples are **reg**, **time**, **integer, real,** and **real-time.**

Predefined Types

nets	connections between hardware elements (declared with keywords such as **wire**)
variables	data storage elements that can retain values (declared with the keywords such as **reg**)
integer	an integer is a variable data type (declared with the keyword **integer**)
real	real number constants and real variable data types for floating-point number (declared with the keyword **real**)
time	a special variable data type to store time information (declared with the keyword **time**)
vectors	wire or reg data types can be declared as vectors (multiple bits) (vectors can be declared with [range1 : range2])

IEEE Standard

- In previous versions of the Verilog standard, the term **register** was used to encompass the **reg**, **integer**, **time**, **real**, and **realtime** types.
- But starting with the 2005 IEEE 1364 Standard, that term is no longer used as a Verilog data type.
- A net or reg declaration without a range specification shall be considered 1 bit wide and is known as a **scalar**.
- Multiple bit net and reg data types shall be declared by specifying a range, which is known as a **vector**.

Verilog Operators

1. Unary sign and reduction operators:

+ −	Unary sign operators
&	Reduction and (unary operator to and bits in a vector and reduce to one bit)
~&	Reduction NAND
|	Reduction or (unary operator to or bits in a vector and reduce to one bit)
~|	Reduction NOR
^	Reduction XOR
~^ or ^~	Reduction XNOR
!	Logical negation
~	Bit-wise negation

2. Arithmetic: Exponent

**	Arithmetic (POWER)

3. Arithmetic: Multiplying, Modulus operators:

*	Multiply
/	Divide
%	Modulus

4. Arithmetic: Addition:

+	Add
−	Subtract

5. Shift operators:

<<	Logical left shift
>>	Logical right shift
<<<	Arithmetic left shift
>>>	Arithmetic right shift

6. Relational operators:

>	Greater than
<	Less than
>=	Greater than or equal
<=	Less than or equal

7. Logical and bitwise operators: Equality and inequality

==	Logical equality
!=	Logical inequality
===	Case equality
!==	Case inequality

8. Bitwise operators:

&	Bit-wise and (binary operator)

9. Logical and bitwise operators:

^	Bit-wise exclusive or (binary operator)
^~ or ~^	Bit-wise equivalence (binary operator)
|	Bit-wise inclusive or (binary operator)

10. Logical and:

&&	Logical and

11. Logical or:

||	Logical or

12. Conditional

? :	Conditional

13. Concatenation and replication

{} Concatenation

{{}} Replication

Precedence of operators

- When parentheses are not used, operators in class 1 have highest precedence and are applied first, followed by class 2, then class 3, and so forth.
- Class 13 operators have lowest precedence and are applied last.
- Operators in the same class have the same precedence and are applied from left to right in an expression.

- Precedence order can be changed by using parentheses.
- The { } operator can be used to concatenate two vectors (or an element and a vector, or two elements) to form a longer vector.
- For example, **{010, 1}** is **0101** and **{"ABC", "DEF"}** is **"ABCDEF"**
- In expression where *A*, *B*, *C*, and *D* are vectors:
- **({A, ~B} | C >> 2 & D) == 110010**
- Relational expression performing an *equality test*.
- It is not an assignment statement.
- To evaluate the expression inside ( ), operator precedence shows highest precedence for three operators in order: >>, &, |
- In order to evaluate | , one of the operands of | has to be obtained by concatenation, which forces expression inside concatenate to be evaluated and operators ~, { }, are applied before the | can be evaluated.

In expression where *A*, *B*, *C*, and *D* are vectors:

({A, ~B} | C >> 2 & D) == 110010

If $A = 110$, $B = 111$, $C = 011000$, and $D = 111011$, the computation proceeds as follows:

```
C >> 2                        = 000110          (shift right 2
                                                 places)
C >> 2 & D                    = 000010          (bit-wise and)
~ B = 000                                       (bit-wise negate)
{A, ~ B}                      = 110000          (concatenation)
({A, ~ B}) | (C >> 2&D)       = 110010          (bit-wise or)
[({A, ~ B} | C >> 2) & D) == 110010] = TRUE     (the parentheses
                                                 force the equality)
                                                 test to be done
                                                 last and the
                                                 result is TRUE)
```

- The result of applying a relational operator is always a Boolean (FALSE or TRUE).
- Equals (==) and not equals (!=) can be applied to almost any type.
- The other relational operators can be applied to many numeric as well as to some array types.
- For example, if A = 5 B =4 and C =3, the expression **`(A >= B) && (B <= C)`** evaluates to FALSE.
- It is legal to use concatenate operator on the left side of the assignment.

- For example, **`{Carry, Sum} = A + B;`**
- It adds A and B and the result goes into Sum and Carry.
- The most significant bit (msb) of the result is assigned to Carry.
- Shift operators can be applied to signed and unsigned registers.
- One can declare a register to be signed/unsigned in the following manner:

```
reg signed [7:0] A = 8'hA5;//signed register A
// number 0xA5 is unsigned, size (8) but repr in A in signed form msb =1

reg [7:0] B = 8'hA5; //unsigned register B
// number 0xA5 is unsigned, size (8) repr in B in unsigned form
```

- The `'h` indicates that the value is hex.

- If the register is unsigned, arithmetic and logic shifts do same operation.
- The following example illustrates the difference between signed and unsigned shifts on signed and unsigned data…

```
reg signed [7:0] A = 8'hA5; // A is signed 1010 0101

A >> 4 is 00001010   (shift right unsigned by 4, filled with 0).
A >>> 4 is 11111010  (shift right signed by 4, filled with sign
                     bit).
A  << 4 is 01010000  (shift left unsigned, filled with 0).
A <<< 4 is 01010000  (shift left signed, filled with 0
                     irrespective of rightmost bit).

reg [7:0] B = 8'hA5; // B is unsigned 1010 0101

B >> 4 is 00001010     (shift right unsigned by 4, filled with 0)
B >>> 4 is 00001010    (shift right signed by 4, but B is unsigned,
                       filled with 0)
B  << 4 is 01010000    (shift left unsigned, filled with 0)
B <<< 4 is 01010000    (shift left signed, filled with 0
                       irrespective of rightmost bit)
```

- If A is declared as integer as in:

```
integer signed A = 8'hA5;
```

- `A >>> 4` yields `00001010` (shift right signed by 4, but integer type is 32 bits and so bits fill up with 0's)
- But if A is initialized to 8'shA5 as in:
- `integer A = 8'shA5;`
- `A >>> 4` yields `11111010` (shift right signed by 4, A's sign bit is 1).
- However, in **reg** declarations, if a signed register is desired, it should be explicitly mentioned.
- For instance,

```
reg [7:0] A = 8'shA5
```

- does not make the register signed. It should be declared as:

```
reg signed [7:0] A =8'hA5
```

- The + and - operators can be applied to any types, including integer or real numeric operands.
- When types are mixed, the expression self-evaluates to a type according to the types of the operands.
- If a and b are 16 bits each, (a + b) will evaluate to 16 bits.
- However, (a + b + 0) will evaluate to integer.
- If any operand is real, the result is real.
- **If any operand is unsigned, the result is unsigned, regardless of the operator**.

- When expressions are evaluated, if the operands are of unequal bit lengths and if one or both operands are unsigned, the smaller operand shall be zero-extended to the size of the larger operand.
- If both operands are signed, the smaller operand shall be sign-extended to the size of the larger operand.
- If constants need to be extended, signed constants are sign-extended and unsigned constants are zero-extended.
- The * and / operators perform multiplication and division on integer or floating point operands.
- The ** operator raises an integer or floating-point number to an integer power.
- The % (modulus) operator calculates the remainder for integer operands.

Keywords

always	endfunction	input	signed
and	endgenerate	integer	task
assign	endmodule	join	time
automatic	endprimitive	localparam	real
begin	endspecify	module	realtime
case	endtable	nand	reg
casex	endtask	negedge	unsigned
casez	event	nmos	wait
deassign	for	nor	while
default	force	not	wire
defparam	forever	or	
design	fork	output	
disable	function	parameter	
edge	generate	pmos	
else	genvar	posedge	
end	include	primitive	
endcase	initial	specify	
endconfig	inout	specparam	

Fixed-Point Number Systems

- Fixed-point notation has an implied binary point between the integer and fraction bits.
- For ex, a fixed-point number with four integer bits and four fraction bits is shown:

(a) 01101100

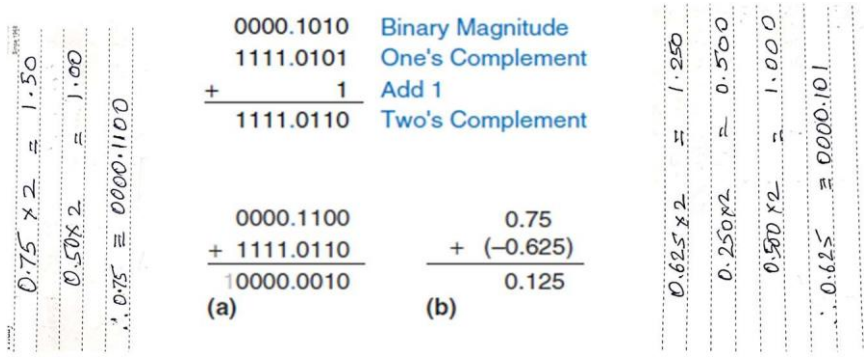
(b) 0110.1100

(c) $2^2 + 2^1 + 2^{-1} + 2^{-2} = 6.75$

- Signed fixed-point numbers can use either two's complement or sign/magnitude notation.

ARITHMETIC WITH FIXED-POINT NUMBERS

- Compute 0.75 + −0.625 using 8-bit fixed-point numbers.



- Fixed-point number systems are commonly used for banking and financial applications that require precision but not a large range.

Floating-Point Number Systems

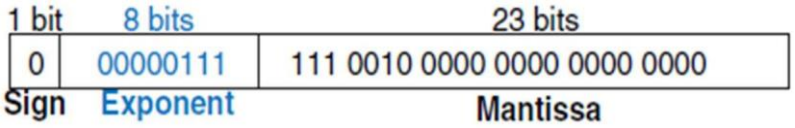
- Floating-point numbers are analogous to scientific notation.
- They remove constraint of constant number of integer and fractional bits, allowing the representation of very large and very small numbers.
- Like scientific notation, floating-point numbers have a sign, mantissa (M), base (B), and exponent (E).

$$\pm M \times B^E$$

- For example, the number 4.1×10^3 is the decimal scientific notation for 4100.
- It has a mantissa of 4.1, a base of 10, and an exponent of 3.
- The decimal point floats to the position right after the most significant digit.
- Floating-point numbers are base 2 with a binary mantissa.
- 32 bits are used to represent 1 sign bit, 8 exponent bits, and 23 mantissa bits.

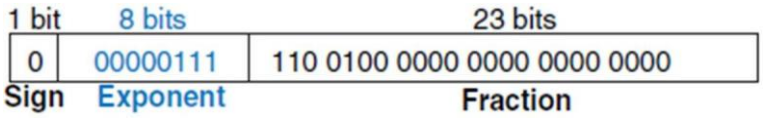
32-BIT FLOATING-POINT NUMBERS

- Show the floating-point representation of the decimal number 228.
- Sol:
- First convert the decimal number into binary: $228_{10}=11100100_2=1.11001_2 \times 2^7$.



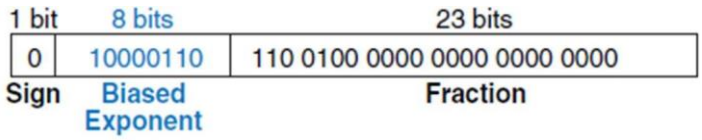
Floating-point version 2

- In binary floating-point, the first bit of the mantissa (to the left of the binary point) is always 1 and therefore need not be stored.
- It is called the implicit leading one.
- The modified floating-point representation
- of $228_{10} = 11100100_2 \times 2^0 = 1.110012 \times 2^7$.
- The implicit leading one is not included in the 23-bit mantissa for efficiency.
- Only the fraction bits are stored.
- This frees up an extra bit for useful data.



IEEE 754 floating point notation

- Need to make one final modification to the exponent field.
- The exponent needs to represent both positive and negative exponents.
- To do so, floating- point uses a biased exponent, which is the original exponent plus a constant bias.
- 32-bit floating-point uses a bias of 127.
- For example, for the exponent 7, the biased exponent is 7 + 127 = 134 = 10000110_2.
- For the exponent −4, the biased exponent is: −4 + 127 = 123 = 01111011_2.



Special Cases: 0, ±∞, and NaN

- The IEEE floating-point standard has special cases to represent numbers such as zero, infinity, and illegal results.
- For example, representing the number zero is problematic in floating-point notation because of the implicit leading one.
- Special codes with exponents of all 0's or all l's are reserved for these special cases.

Number	Sign	Exponent	Fraction
0	X	00000000	00000000000000000000000
∞	0	11111111	00000000000000000000000
−∞	1	11111111	00000000000000000000000
NaN	X	11111111	Non-zero $\sqrt{-1}$ or $\log_2(-5)$.

Boolean Postulates and Fundamental Gates

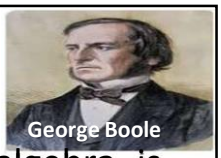
Module - 2

ECT 203

Boolean Algebra

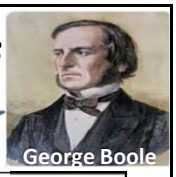
- In 1849 George Boole published a scheme for the algebraic description of processes involved in logical thought and reasoning.
- This scheme and its further refinements became known as **Boolean algebra**.
- It was almost 100 years later that this Algebra found application in the Engineering sense.
- In the late 1930s, Claude Shannon showed that Boolean Algebra provides an effective means of describing circuits built with switches.
- The Algebra can be used to describe logic circuits.
- A Boolean variable can take value of either logic 0 or logic 1.

Principle of Duality



- This important property of Boolean algebra is called the Principle of Duality and states that every algebraic expression deducible from the postulates of Boolean algebra remains valid if the operators and identity elements are interchanged.
- In a two-valued Boolean algebra, the identity elements and elements of set are same: 1 and 0.
- The duality principle has many applications.
- If the dual of an algebraic expression is desired, we simply interchange OR and AND operators and replace 1's by 0's and 0's by 1's.

Postulates and Theorems of Boolean Algebra



Postulate/Theorem	Rule -a	Principle of Duality	Rule-b
Postulate 2	x + 0 = x		x .1 = x
Postulate 5	x + x' = 1		x . x' = 0
Theorem 1	x + x = x		x . x = x
Theorem 2	x + 1 = 1		x . 0 = 0
Theorem 3, Involution	(x')' = x		
Postulate 3, Commutative	x + y = y + x		x.y = y.x
Theorem 4, Associative	x + (y + z) = (x + y) + z		x.(y.z) = (x.y).z
Postulate 4, Distributive	x.(y + z) = x.y + x.z		x + y.z = (x + y).(x + z)
Theorem 5, DeMorgan	(x + y)' = x'. y'		(x.y)' = x' + y'
Theorem 6, Absorption	x + x.y = x		x.(x + y) = x

Proof of the Theorems

THEOREM 1(a): $x + x = x$.

Statement	Justification
$x + x = (x + x) \cdot 1$	postulate 2(b)
$= (x + x)(x + x')$	5(a)
$= x + xx'$	4(b)
$= x + 0$	5(b)
$= x$	2(a)

THEOREM 1(b): $x \cdot x = x$.

Statement	Justification
$x \cdot x = xx + 0$	postulate 2(a)
$= xx + xx'$	5(b)
$= x(x + x')$	4(a)
$= x \cdot 1$	5(a)
$= x$	2(b)

Dual

- Note that theorem 1(b) is the dual of theorem 1(a) and that each step of the proof in part (b) is the dual of its counterpart in part (a).
- Any dual theorem can be similarly derived from the proof of its corresponding theorem.

THEOREM 2(a): $x + 1 = 1$.

Statement	Justification
$x + 1 = 1 \cdot (x + 1)$	postulate 2(b)
$= (x + x')(x + 1)$	5(a)
$= x + x' \cdot 1$	4(b)
$= x + x'$	2(b)
$= 1$	5(a)

THEOREM 2(b): $x \cdot 0 = 0$ by duality.

THEOREM 3: $(x')' = x$. From postulate 5, we have $x + x' = 1$ and $x \cdot x' = 0$, which together define the complement of x. The complement of x' is x and is also $(x')'$.

- Therefore, since the complement is unique, we have (*x'*)' = *x*.
- The theorems involving two or three variables may be proven algebraically from postulates and theorems that have already been proven.

THEOREM 6(a): $x + xy = x$.

Statement	Justification
$x + xy = x \cdot 1 + xy$	postulate 2(b)
$= x(1 + y)$	4(a)
$= x(y + 1)$	3(a)
$= x \cdot 1$	2(a)
$= x$	2(b)

THEOREM 6(b): $x(x + y) = x$ by duality.

Truth Table

- The theorems of Boolean algebra can be proven by means of truth tables.
- In truth tables, both sides of the relation are checked to see whether they yield identical results for all possible combinations of the variables involved.

Proof of Absorption Theorem

- The following truth table verifies the absorption theorem:

x	y
0	0
0	1
1	0
1	1

xy	x + xy
0	0
0	0
0	1
1	1

Theorem 6, Absorption	x + x.y = x	x.(x + y) = x

Proof of DeMorgan's Theorem

Theorem 5, DeMorgan	(x + y)' = x'. y'	Rule – a

x	y	x + y	(x + y)'
0	0	0	1
0	1	1	0
1	0	1	0
1	1	1	0

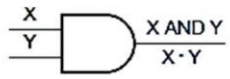
x'	y'	x'y'
1	1	1
1	0	0
0	1	0
0	0	0

Augustus De Morgan

27 June 1806 – 18 March 1871

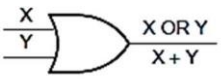
- An *AND gate* produces a 1 output if and only if all of its inputs are 1.
- An *OR gate* produces a 1 if and only if one or more of its inputs are 1.
- A *NOT gate*, usually called an *inverter*, produces an output value that is the opposite of its input value.

(a)



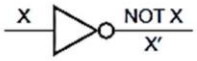
X	Y	X AND Y
0	0	0
0	1	0
1	0	0
1	1	1

(b)



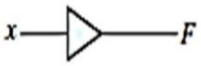
X	Y	X OR Y
0	0	0
0	1	1
1	0	1
1	1	1

(c)



X	NOT X
0	1
1	0

Buffer



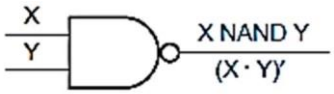
$F = x$

x	F
0	0
1	1

- A buffer produces the *transfer* function, but does not produce a logic operation, since the binary value of the output is equal to the binary value of the input.
- This circuit is used for power amplification of the signal and is equivalent to two inverters connected in cascade.

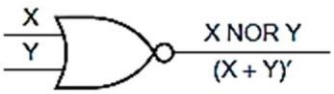
- A *NAND gate* produces the opposite of an AND gate's output, a 0 if and only if all of its inputs are 1.
- A *NOR gate* produces the opposite of an OR gate's output, a 0 if and only if one or more of its inputs are 1.

(a)



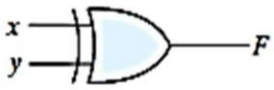
X	Y	X NAND Y
0	0	1
0	1	1
1	0	1
1	1	0

(b)



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

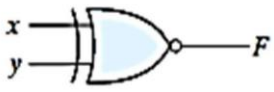
Exclusive-OR (XOR)



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

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

Exclusive-NOR or equivalence



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

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

Boolean Function

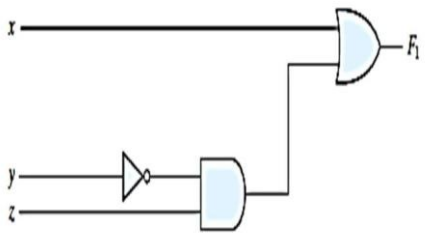
- $F_1 = x + y'.z$
- The function F_1 is equal to 1 if *x* is equal to 1 or if both y' and z are equal to 1.
- F_1 is equal to 0 otherwise.
- The complement operation y = 1, y' = 0.
- F_1 = 1 if x = 1 or if y = 0 and z = 1.

Truth Tables for F_1

x	y	z	F_1
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

$F_1 = x + y'.z$

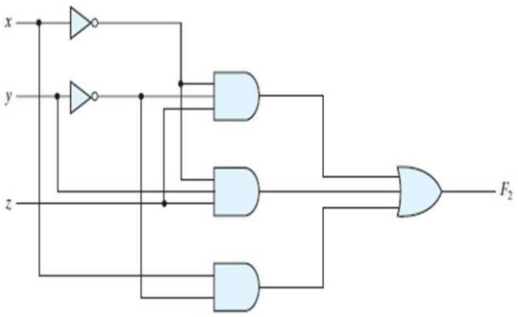
Gate implementation



$F_2 = x'.y'.z + x'.y.z + x.y'$

Truth Tables for F_2

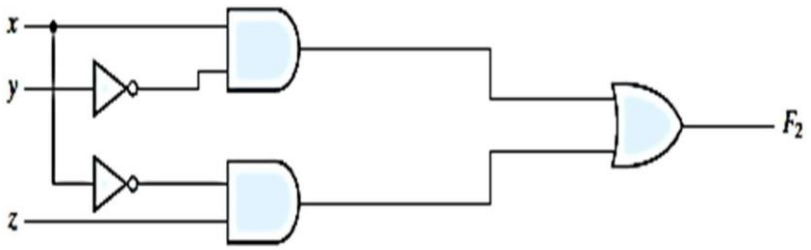
x	y	z	F_2
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0



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

Now consider the possible simplification of the function by applying some of the identities of Boolean algebra:

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



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

Simplify the following Boolean functions to a minimum number of literals.

1. $x(x' + y) = xx' + xy = 0 + xy = xy.$
2. $x + x'y = (x + x')(x + y) = 1(x + y) = x + y.$
3. $(x + y)(x + y') = x + xy + xy' + yy' = x(1 + y + y') = x.$
4. $xy + x'z + yz = xy + x'z + yz(x + x')$
   $= xy + x'z + xyz + x'yz$
   $= xy(1 + z) + x'z(1 + y)$
   $= xy + x'z.$
5. $(x + y)(x' + z)(y + z) = (x + y)(x' + z)$, by duality from function 4.

Find the complement of the functions $F_1 = x'yz' + x'y'z$ and $F_2 = x(y'z' + yz)$. By applying DeMorgan's theorems as many times as necessary, the complements are obtained as follows:

$$F_1' = (x'yz' + x'y'z)' = (x'yz')'(x'y'z)' = (x + y' + z)(x + y + z')$$

$$F_2' = [x(y'z' + yz)]' = x' + (y'z' + yz)' = x' + (y'z')'(yz)'$$

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

$$= x' + yz' + y'z$$

Find the complement of the functions F_1 and F_2 by taking their duals and complementing each literal.

1. $F_1 = x'yz' + x'y'z$.
   The dual of F_1 is $(x' + y + z')(x' + y' + z)$.
   Complement each literal: $(x + y' + z)(x + y + z') = F_1'$.
2. $F_2 = x(y'z' + yz)$.
   The dual of F_2 is $x + (y' + z')(y + z)$.
   Complement each literal: $x' + (y + z)(y' + z') = F_2'$.

CANONICAL AND STANDARD FORMS

- A binary variable may appear either in its normal form (x) or in its complement form (x').
- Now consider two binary variables *x* and *y* combined with an AND operation.
- Since each variable may appear in either form, there are four possible combinations: x'.y', x'.y, x.y' and x.y
- Each of these four AND terms is called a Minterm, or a standard product.
- In a similar manner, *n* variables can be combined to form 2^n Minterms.

- In a similar fashion, *n* variables forming an OR term, provide 2^n possible combinations, called Maxterms, or standard sums.
- Each Maxterm is obtained from an OR term of the *n* variables, with each variable being unprimed if the corresponding bit is a 0 and primed if a 1.
- Each Maxterm is the complement of its corresponding minterm and vice versa.
- A Boolean function can be expressed algebraically from a given truth table by forming a minterm for each combination of the variables that produces a 1 in the function and then taking the OR of all those terms.

Minterms and Maxterms for Three Binary Variables

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

Example – Write the Boolean Function for this Truth Table

Functions of Three Variables

x	y	z	Function f_1	Function f_2
0	0	0	0	0
0	0	1	1	0
0	1	0	0	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

$$f_1 = x'y'z + xy'z' + xyz = m_1 + m_4 + m_7$$
$$f_2 = x'yz + xy'z + xyz' + xyz = m_3 + m_5 + m_6 + m_7$$

- Any Boolean function can be expressed as a Sum of Minterms (with "Sum" meaning the ORing of terms).
- This is called **Sum of Products (SOP)**

- Consider the complement of a Boolean function.
- It may be read from the truth table by forming a minterm for each combination that produces a 0 in the function and then ORing those terms.
- The complement of f_1 is read as:

$$f_1' = x'y'z' + x'yz' + x'yz + xy'z + xyz'$$

- Take the complement of f_1', to obtain the function $f1$: (Use DeMorgan's rule)

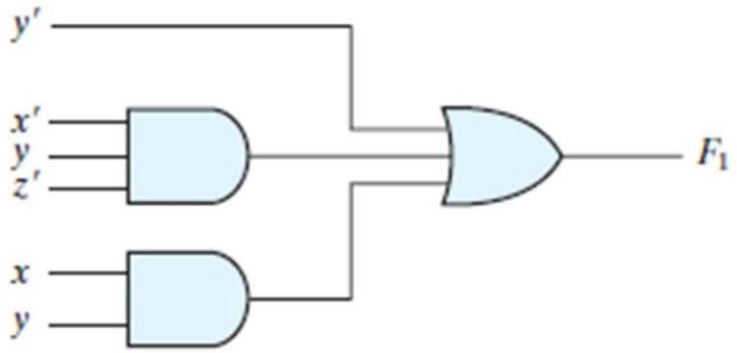
$$f_1 = (x + y + z)(x + y' + z)(x' + y + z')(x' + y' + z)$$
$$= M_0 \cdot M_2 \cdot M_3 \cdot M_5 \cdot M_6$$

- Any Boolean function can be expressed as a product of Maxterms (with "product" meaning the ANDing of terms). **Product of Sums (POS).**
- The procedure for obtaining the product of Maxterms directly from the truth table is as follows:
- Form a Maxterm for each combination of the variables that produces a 0 in the function, and then form the AND of all those Maxterms.
- **Boolean functions expressed as a sum of Minterms or product of Maxterms are said to be in *Canonical form.***

Standard Form - SOP

- The *sum of products* is a Boolean expression containing AND terms, called *product terms,* with one or more literals each.
- The *sum* denotes the ORing of these terms.
- An example of a function expressed as a sum of products is : $F_1 = y' + xy + x'yz'$
- The expression has three product terms, with one, two, and three literals.
- Their sum is, in effect, an OR operation.
- The logic diagram of a sum-of-products expression consists of a group of AND gates followed by a single OR gate.

Two-level implementation

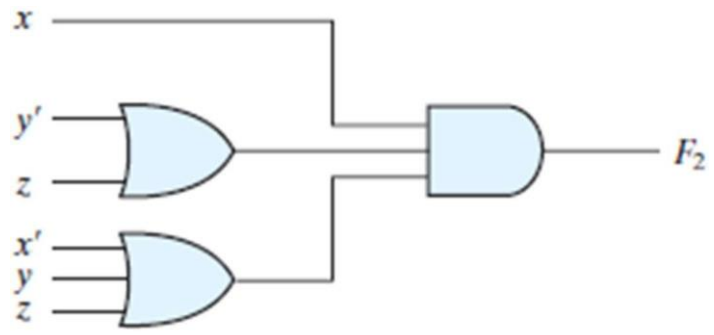


(a) Sum of Products

Standard Form - POS

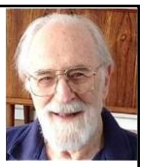
- A *product of sums* is a Boolean expression containing OR terms, called *sum terms*.
- Each term may have any number of literals.
- The *product* denotes the ANDing of these terms.
- An example of a function expressed as a product of sums is: $F_2 = x(y' + z)(x' + y + z')$
- This expression has three sum terms, with one, two, and three literals.
- The product is an AND operation.
- The gate structure of the POS expression consists of a group of OR gates for the sum terms followed by an AND gate.

Two-level implementation



(b) Product of Sums

Karnaugh Map-Minimization of Boolean Functions

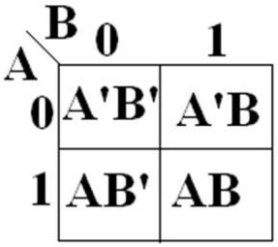


Maurice Karnaugh
4 October 1924 (age 95)

- The **Karnaugh map** (**KM** or **K-map**) is a method of simplifying Boolean expressions introduced by Maurice Karnaugh in 1953.
- Karnaugh maps are used to simplify real-world logic requirements so that they can be implemented using a minimum number of physical logic gates.

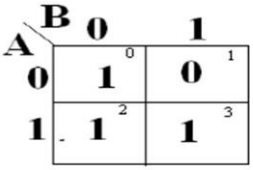
Two-variable Karnaugh(K)-map

Mapping of SOP Expressions



The minterms of a two-variable k-map

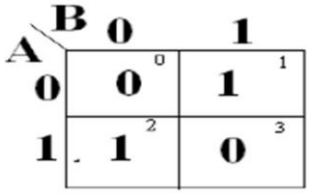
The mapping of the expressions =Σ m (0,2,3) is:



Mapping of SOP expressions

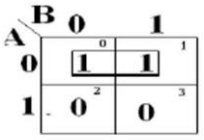
- Each sum term in the standard SOP expression is called a Minterm.
- A function in two variables (A, B) has four possible Minterms, A'B', A'B, AB', AB
- They are represented as m_0, m_1, m_2, and m_3.
- The lowercase letter m stands for Minterm and its subscript denotes the decimal designation of that Minterm.
- Treat the non-complemented variable as a 1 and the complemented variable as a 0 and put them side by side for reading the decimal equivalent of the binary number so formed.
- For mapping a SOP expression on to the K-map, 1s are placed in the squares corresponding to the Minterms which are presented in the expression.

Draw the K-Map the expressions F= A'B+AB'



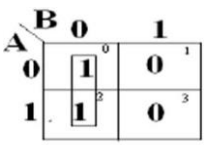
Minimizations of SOP expressions

f_1=Σ m (0,1)



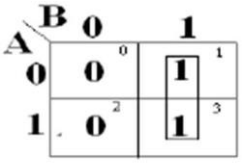
f_1= A'

f_2=Σ m (0,2)



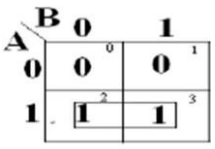
f_2= B'

f_3=Σ m (1,3)



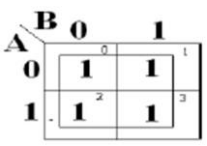
f_3= B

f4=Σ m (2,4)



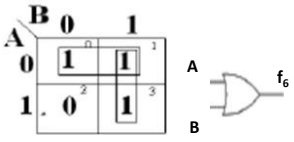
f_4= A

f_5=Σ m (0,1,2,3)



f_5= 1

f_6=Σ m (0,1,3)

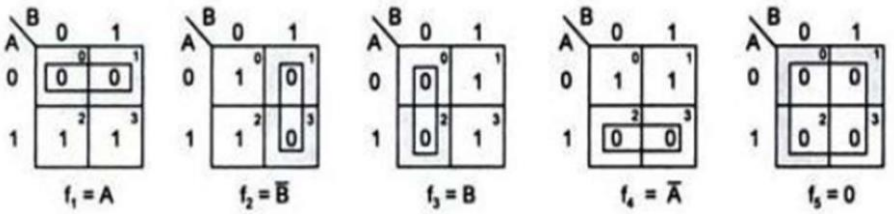


f_6= A'+B

Mapping of POS expressions

- Each sum term in the standard POS expression is called a Maxterm.
- A function in two variables (A, B) has four possible Maxterms, (A'+B'),(A'+B),(A+B'),(A+B)
- They are represented as M_0, M_1, M_2, and M_3.
- The uppercase letter M stands for Maxterm and its subscript denotes the decimal designation of that Maxterm.
- Treat the non-complemented variable as a 0 and the complemented variable as a 1 and put them side by side for reading the decimal equivalent of the binary number so formed.
- For mapping a POS expression on to the K-map, 0s are placed in the squares corresponding to the Maxterms which are presented in the expression.

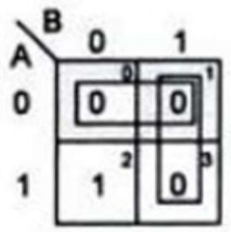
The possible Maxterm groupings in a two-variable K-map



Minimization of POS Expressions

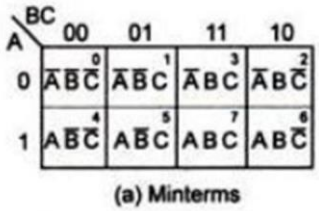
Ex: Reduce the expression f=(A+B).(A+B').(A'+B') using mapping.

The given expression in terms of maxterms is f=πM(0,1,3).

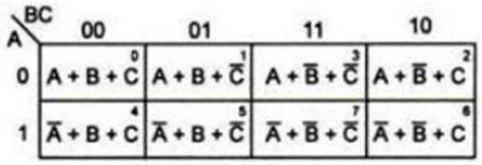


f=A.B'

Three-variable K-map



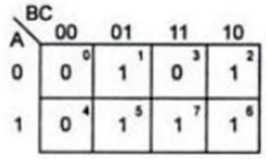
(a) Minterms



(b) Maxterms

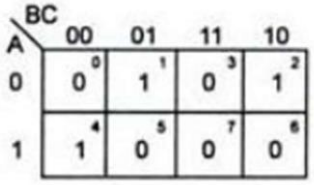
Example: Map the expression $\sum m(1,2,5,6,7)$.

The corresponding k-map is :



K-map in SOP form

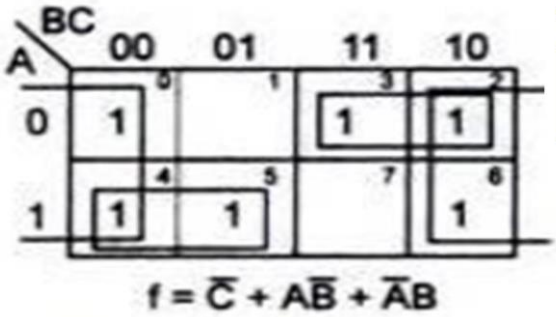
Example: Map the expression $\pi M\ (0,3,5,6,7)$.



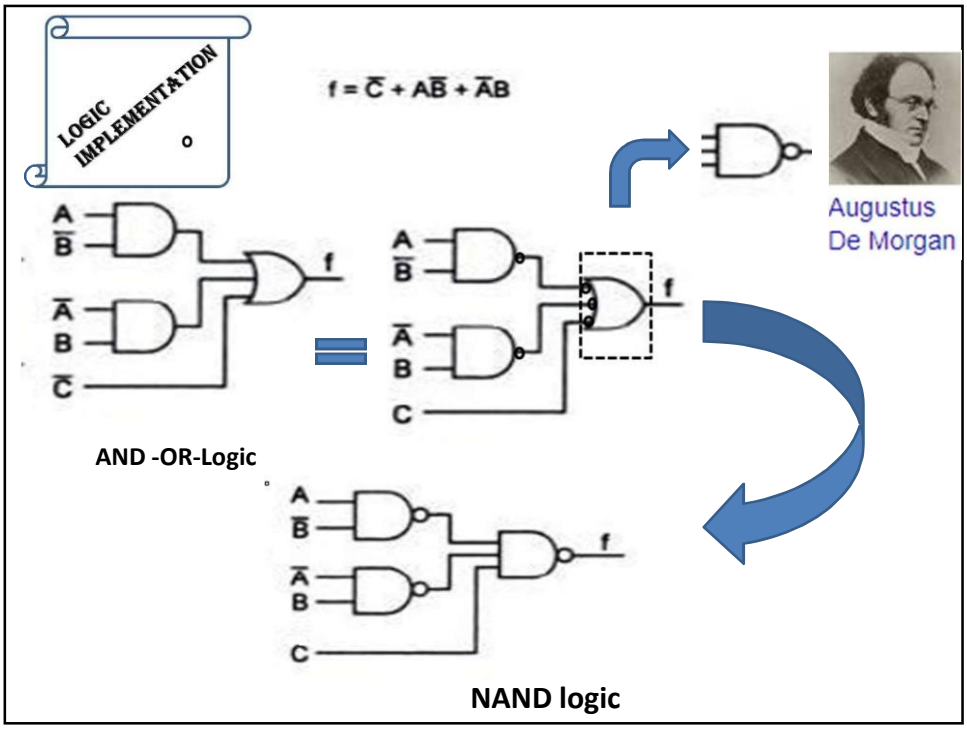
K-map in POS form.

Minimization of SOP expressions

Ex:Reduce the expression f=$\sum m(0,2,3,4,5,6)$

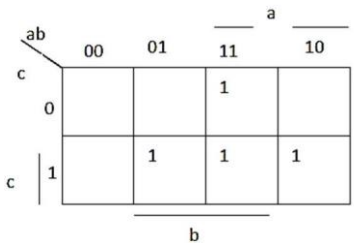


$f = \overline{C} + A\overline{B} + \overline{A}B$

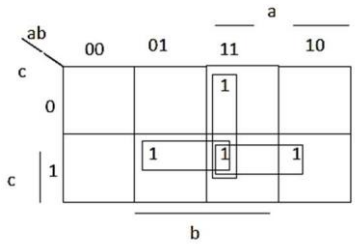


Minimize the Boolean expr $m = a'bc + ab'c + abc' + abc.$

Use 3-var K-Map



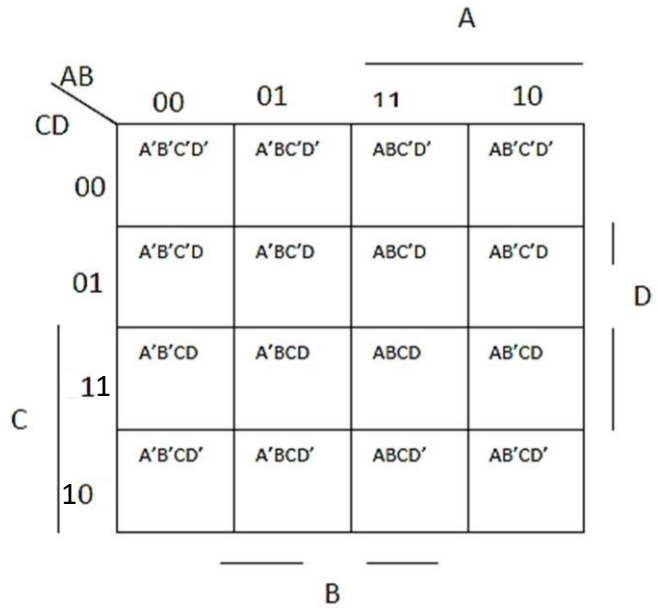
Mapping



Grouping and reduction

Minimized Boolean expr $bc + ac + ab$

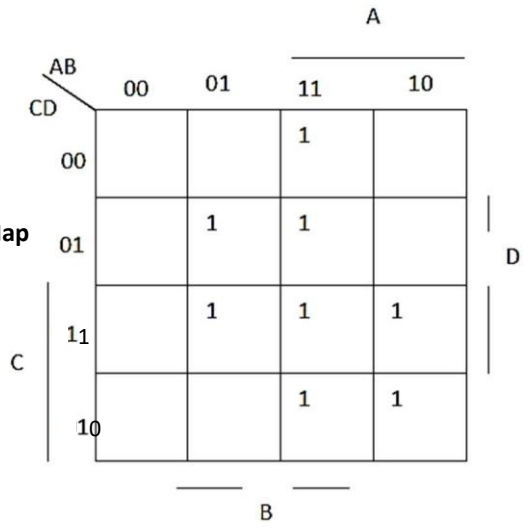
Four-variable K-map



Minimize the Boolean expr

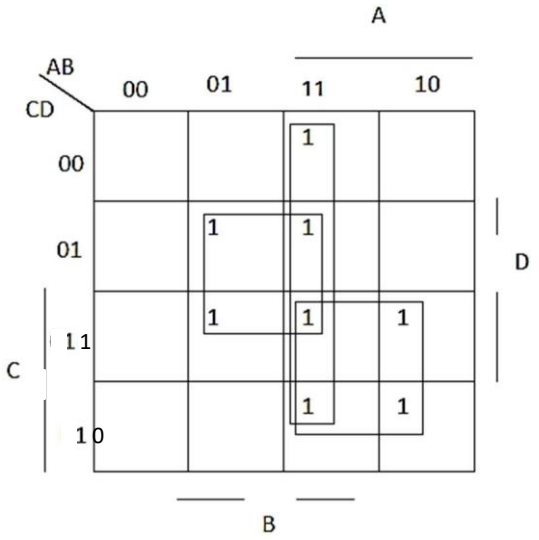
$$Q = A'BC'D + A'BCD + ABC'D' + ABC'D + ABCD + ABCD' + AB'CD + AB'CD'$$

Use 4-var K-Map



Mapping

Grouping and reduction



Minimized Boolean expr $Q = BD + AC + AB$

Structural Specification of Logic Gates

- Verilog includes a set of Gate-level Primitives that correspond to commonly-used logic gates.
- A gate is represented by indicating its functional name, output, and inputs.
- For example, a two-input AND gate, with output *y* and inputs x_1 and x_2, is denoted as:

```
and (y, x1, x2);
```

- A four-input OR gate is specified as:

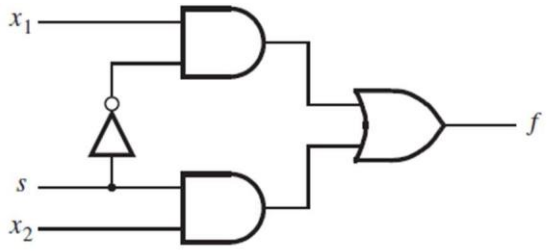
```
or (y, x1, x2, x3, x4);
```

- The keywords **nand** and **nor** are used to define the NAND and NOR gates in the same way.
- The NOT gate given by:

```
not (y, x); // implements y = x'.
```

- A logic circuit is specified in the form of a *module* that contains the statements that define the circuit.
- A module has inputs and outputs, which are referred to as its *ports*.
- The word port is a commonly-used term that refers to an input or output connection to an electronic circuit.

Example code: Multiplexer circuit



Logic Circuit

```
module example1 (x1, x2, s, f);
    input x1, x2, s;
    output f;

    not (k, s);
    and (g, k, x1);
    and (h, s, x2);
    or (f, g, h);

endmodule
```

Verilog Code

- The first statement gives the module a name, *example1*, and indicates that there are four port signals.
- The next two statements declare that x_1, x_2, and *s* are to be treated as **input** signals, while *f* is the **output**.
- The actual structure of the circuit is specified in the four statements that follow.
- The NOT gate gives $k = s'$
- The AND gates produce $g = s'.x_1$ and $h = s.x_2$.
- The outputs of AND gates are combined in the OR gate to form: $f = g+h$
  $= s'.x_1+s.x_2$
- The module ends with the **endmodule** statement.

Second example of Verilog code

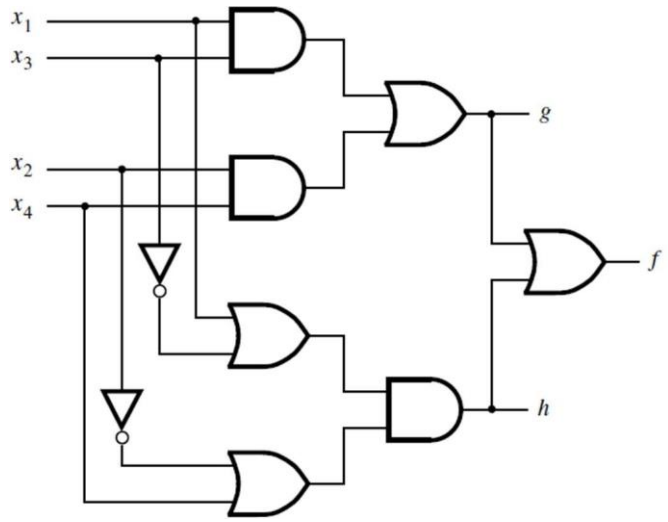
- It defines a circuit that has four input signals, x_1, x_2, x_3, and x_4, and three output signals, f, g, and h.
- It implements the logic functions:

$g = x_1.x_3 + x_2.x_4$

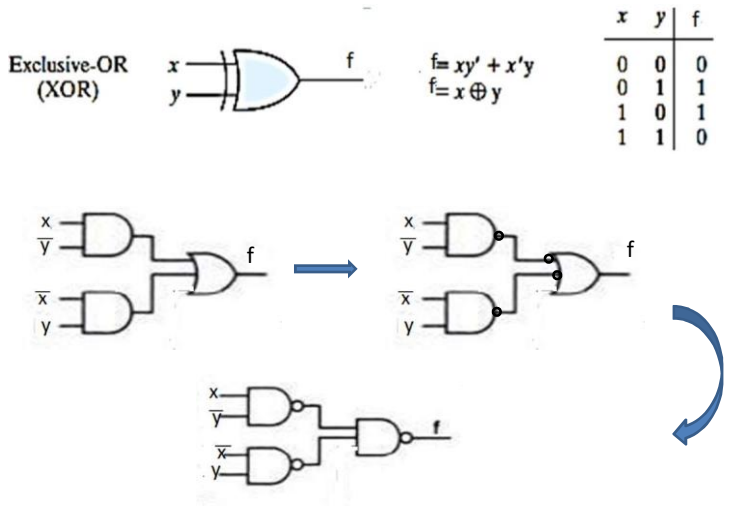
$h = (x_1 + x_3')(x_2' + x_4)$

$f = g + h$

```
module  example2 (x1, x2, x3, x4, f, g, h);
    input x1, x2, x3, x4;
    output f, g, h;

    and (z1, x1, x3);
    and (z2, x2, x4);
    or (g, z1, z2);
    or (z3, x1, ~x3);
    or (z4, ~x2, x4);
    and (h, z3, z4);
    or (f, g, h);

endmodule
```



EX-OR Gate using NAND Gate



Verilog code for EX-OR Gate

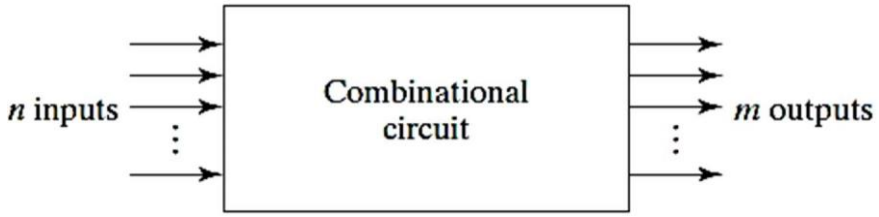
```
module xor (x, y, f);
    input x, y;
    output f;
    not (j, x);
    not (k, x);
    nand (g, x, j);
    nand (h, y, k);
    nand (f, g, h);
endmodule
```

Combinatorial and Arithmetic Circuits

ECT 203 Module -3

Combinational Logic Systems

- A Combinational circuit consists of logic gates whose outputs at any time are determined from only the present combination of inputs.
- A Combinational circuit performs an operation that can be specified logically by a set of Boolean functions.
- A Combinational circuit consists of an inter connection of logic gates.
- Combinational logic gates react to the values of the signals at their inputs and produce the value of the output signal.
- Transforms binary information from the given input data to a required output data.



Block Diagram of a Combinational Circuit

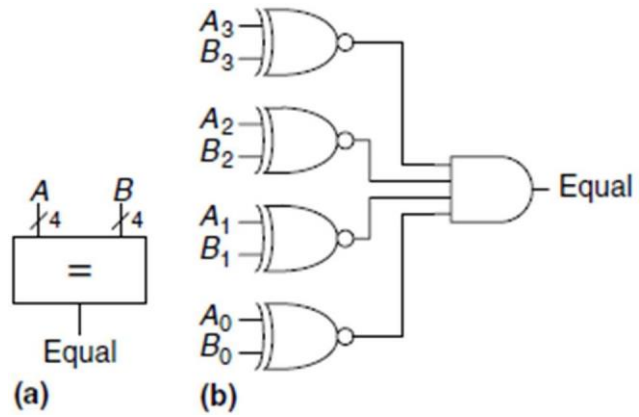
- The *n* input binary variables come from an external source.
- the *m* output variables are produced by the internal combinational logic circuit and go to an external destination.
- Each input and output variable exists physically as an analog signal whose values are interpreted to be a binary signal that represents logic 1 and logic 0.
- For *n* input variables, there are 2^n possible combinations of the binary inputs.
- For each possible input combination, there is one possible value for each output variable.

- A combinational circuit can be specified with a truth table that lists the output values for each combination of input variables.
- A combinational circuit also can be described by *m* Boolean functions, one for each output variable.
- Each output function is expressed in terms of the *n* input variables.
- The most important standard combinational circuits, such as multiplexers, adders, subtractors, comparators, demultiplexers, decoders, encoders will be studied.
- These components are available in integrated circuits as medium-scale integration (MSI) circuits.
- They are also used as *standard cells* in complex very large scale integrated (VLSI) circuits such as Application-Specific Integrated Circuits (ASICs).
- The standard cell functions are interconnected within the VLSI circuit in the same way that they are used in multiple-IC MSI design.

Comparators

- A *Comparator* determines whether two binary numbers are equal or if one is greater or less than the other.
- A Comparator receives two *N*-bit binary numbers, *A* and *B.*
- There are two common types of Comparators.
- They are Equality Comparator and Magnitude Comparator.
- An *equality comparator* produces a single output indicating whether *A* is equal to *B* (*A== B*).
- A *magnitude comparator* produces one or more outputs indicating the relative values of *A* and *B.*

Equality Comparator

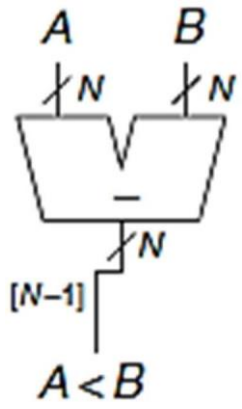


4-bit equality comparator: (a) symbol, (b) implementation

- The equality comparator is the simpler piece of hardware.
- The diagram shows the symbol and implementation of a 4-bit equality comparator.
- It first checks to determine whether the corresponding bits in each column of *A* and *B* are equal, using XNOR gates.
- The numbers are equal if all of the columns are equal.

Magnitude comparison

- Magnitude comparison is usually done by computing *A- B* and looking at the sign (most significant bit) of the result, as shown in diagram.
- If the result is negative (i.e., the sign bit is 1), then *A* is less than *B.*
- Otherwise *A* is greater than or equal to *B.*

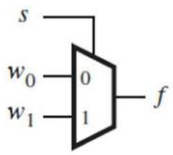


N-bit Magnitude Comparator

Multiplexers

- A multiplexer circuit has a number of data inputs, one or more select inputs, and one output.
- It passes the signal value on one of the data inputs to the output.
- The data input is selected by the values of the select inputs.

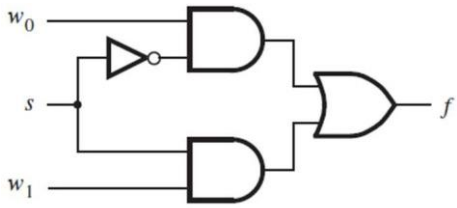
2-to-1 Multiplexer



(a) Graphical symbol

s	f
0	w_0
1	w_1

(b) Truth table



(c) Sum-of-products circuit

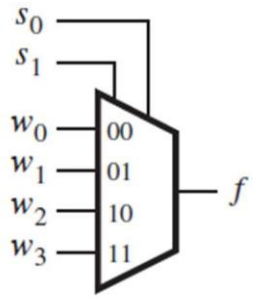
- Part *(a)* gives the symbol commonly used.
- The *select* input, *s*, chooses as the output of the multiplexer either input w_0 or w_1.
- The multiplexer's functionality can be described in the form of a truth table as shown in part *(b)* of the figure.
- Part*(c)* gives a sum-of-products implementation of the 2-to-1 multiplexer.

4-to-1 Multiplexer

- This is a larger multiplexer with four data inputs, $w_0, \ldots, w_3$, and two select inputs, s_1 and s_0.
- As shown in the truth table in part *(b)* of the figure, the two-bit number represented by s_1s_0 selects one of the data inputs as the output of the multiplexer.
- A sum-of-products implementation of the 4-to-1 multiplexer appears in part (c).
- It realizes the multiplexer function

$$f = \overline{s}_1\overline{s}_0w_0 + \overline{s}_1s_0w_1 + s_1\overline{s}_0w_2 + s_1s_0w_3$$

4-to-1 Multiplexer

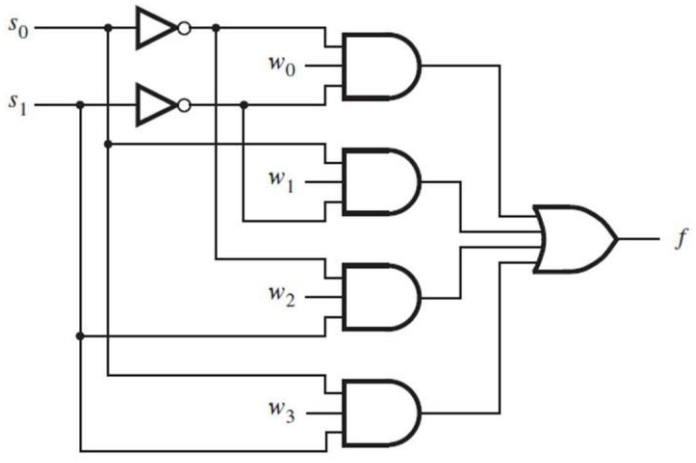


(a) Graphical symbol

s_1	s_0	f
0	0	w_0
0	1	w_1
1	0	w_2
1	1	w_3

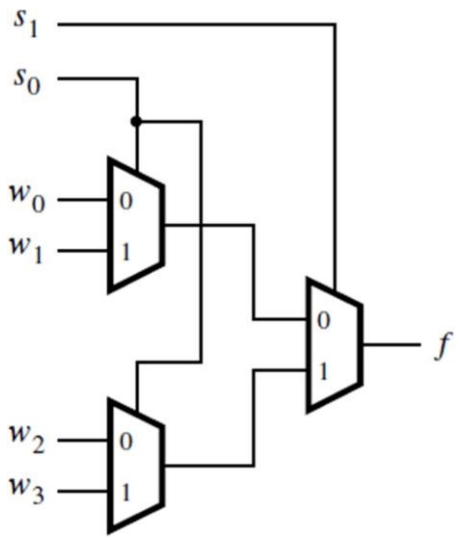
(b) Truth table

4-to-1 Multiplexer

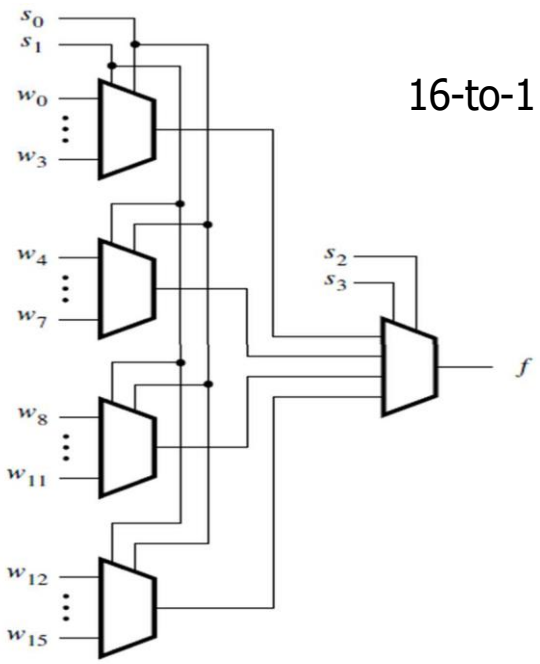


(c) Circuit

- It is possible to build larger multiplexers using the same approach.
- Usually, the number of data inputs, n, is an integer power of two.
- A multiplexer that has n data inputs, $w_0, \ldots, w_{n-1}$, requires $\log_2 n$ select inputs.
- Larger multiplexers can also be constructed from smaller multiplexers.
- For example, the 4-to-1 multiplexer can be built using three 2-to-1 multiplexers
- A 16-to-1 multiplexer is constructed with five 4-to-1 multiplexers.



Using 2-to-1 multiplexers to build a 4-to-1 multiplexer.



16-to-1 multiplexer.

Synthesis of Logic Functions Using Multiplexers

- Multiplexers can also be used in a more general way to synthesize logic functions.
- Example: 1: Implement the Ex-OR function using multiplexer.

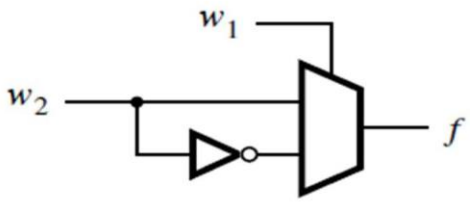
- Truth table defines function f = w1 ⊕ w2.
- Implement with a 4-to-1 multiplexer where values of *f* in each row of truth table are connected to multiplexer data inputs.
- The multiplexer select inputs are *w*1 and *w*2.
- Thus for each valuation of *w*1*w*2, output *f* is equal to function value in corresponding row of truth table.
- The above implementation is straightforward, but it is not very efficient.

- A better implementation can be got by modifying truth table as shown in Figure *b*, which allows *f* to be implemented by a single 2-to-1 multiplexer.
- One of the input signals, *w*1, is chosen as select input of the 2-to-1 multiplexer.
- The truth table is redrawn to indicate the value of *f* for each value of *w*1.
- When *w*1 = 0, *f* has the same value as input *w*2, and when *w*1 = 1, *f* has the value of *w*2'.

w_1	w_2	f
0	0	0
0	1	1
1	0	1
1	1	0

w_1	f
0	w_2
1	$\overline{w}_2$

(b) Modified truth table



(c) Circuit

Two-input XOR implemented with 2-to-1 multiplexer

Example: 2

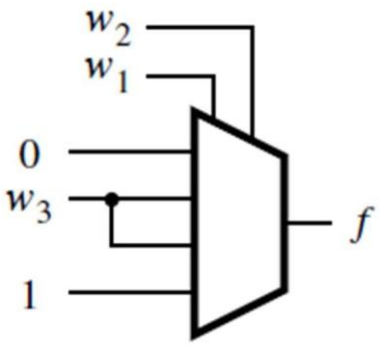
Implement the truth table using appropriate Multiplexer.

w_1	w_2	w_3	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

w_1	w_2	w_3	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

w_1	w_2	f
0	0	0
0	1	w_3
1	0	w_3
1	1	1

(a) Modified truth table



(b) Circuit

Implementation with 4-to-1 multiplexer

Example: 3

Implement the function *f* using appropriate Multiplexer.

f = w1 ⊕ w2 ⊕ w3

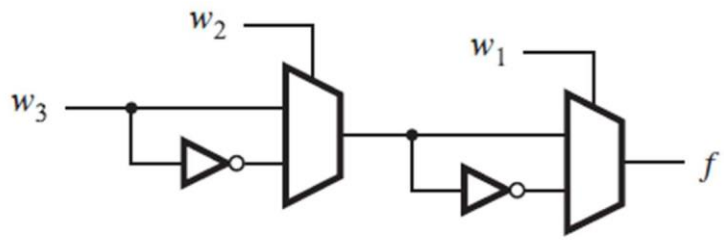
w_1	w_2	w_3	f
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

w_1	w_2	w_3	f	
0	0	0	0	
0	0	1	1	$w_2 \oplus w_3$
0	1	0	1	
0	1	1	0	
1	0	0	1	
1	0	1	0	$\overline{w_2 \oplus w_3}$
1	1	0	0	
1	1	1	1	

(a) Truth table

- The function can be implemented using 2-to-1 multiplexers.
- When $w1 = 0$, f is equal to XOR of $w2$ and $w3$, and when $w1 = 1$, f is the XNOR of $w2$ and $w3$.
- Part (*b*) gives a corresponding circuit.
- The left multiplexer in the circuit produces $w2 \oplus w3$
- The right multiplexer uses the value of $w1$ to select either $w2 \oplus w3$ or its complement.
- This circuit can be got directly by writing the function as :

  $f = (w2 \oplus w3) \oplus w1$



(b) Circuit

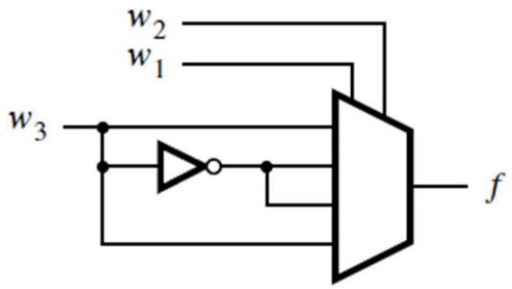
Three-input XOR implemented with 2-to-1 multiplexers

Alternative implementation

f = w1 ⊕ w2 ⊕ w3

w_1	w_2	w_3	f		w_1	w_2	w_3	f	
0	0	0	0		0	0	0	0	w_3
0	0	1	1		0	0	1	1	
0	1	0	1		0	1	0	1	$\overline{w}_3$
0	1	1	0	→	0	1	1	0	
1	0	0	1		1	0	0	1	$\overline{w}_3$
1	0	1	0		1	0	1	0	
1	1	0	0		1	1	0	0	w_3
1	1	1	1		1	1	1	1	

(a) Truth table



(b) Circuit

Three-input XOR implemented with a 4-to-1 multiplexer

Magnitude Comparator

- The comparison of two numbers is an operation that determines whether one number is greater than, less than, or equal to the other number.
- A *magnitude comparator* is a combinational circuit that compares two numbers *A* and *B* and determines their relative magnitudes.
- The outcome of the comparison is specified by three binary variables that indicate whether $A > B$, $A = B$, or $A < B$.

4-bit Magnitude Comparator

- Consider two numbers, *A* and *B* , with four digits each.
- Write the coefficients of the numbers in descending order of significance:

$$A = A_3\, A_2\, A_1\, A_0 \qquad B = B_3\, B_2\, B_1\, B_0$$

- The two numbers are equal if all pairs of significant digits are equal: $A_3 = B_3, A_2 = B_2, A_1 = B_1,$ and $A_0 = B_0$.
- When the numbers are binary, the digits are either 1 or 0, and the equality of each pair of bits can be expressed logically with an exclusive-NOR function as:

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

- where $x_i = 1$ only if the pair of bits in position *i* are equal (i.e., if both are 1 or both are 0).

- The equality of the two numbers *A* and *B* is displayed in a combinational circuit by an output binary variable that we designate by the symbol (A = B)
- This binary variable is equal to 1 if the input numbers, *A* and *B* , are equal, and is equal to 0 otherwise.
- For equality to exist, all x_i variables must be equal to 1, a condition that dictates an AND operation of all variables:

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

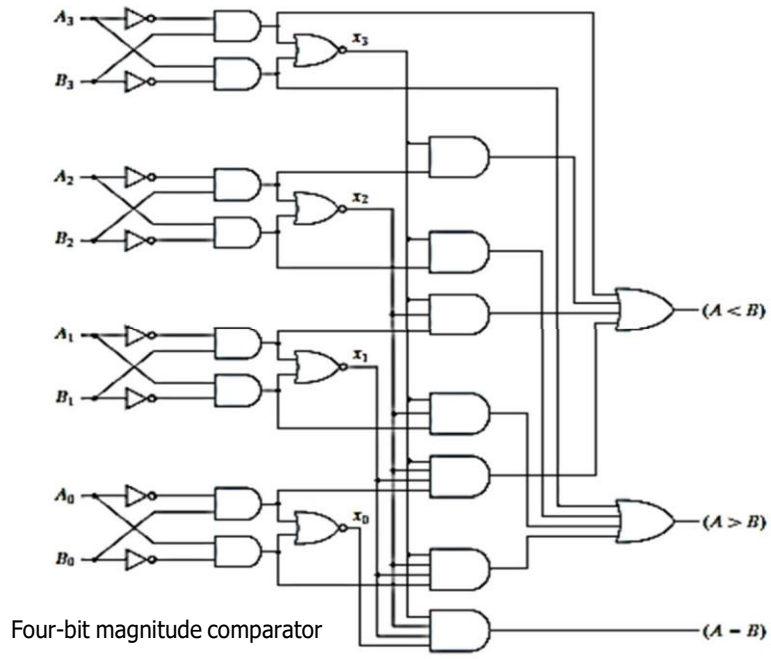
- The *binary* variable (A = B) is equal to 1 only if all pairs of digits of the two numbers are equal.

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

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

$$(A > B) = A_3B_3' + x_3A_2B_2' + x_3x_2A_1B_1' + x_3x_2x_1A_0B_0'$$
$$(A < B) = A_3'B_3 + x_3A_2'B_2 + x_3x_2A_1'B_1 + x_3x_2x_1A_0'B_0$$

- The symbols (A > B) and (A < B) are *binary* output variables that are equal to 1 when A > and A < B, respectively.



Four-bit magnitude comparator

Decoder

- A binary code of n bits is capable of representing up to 2^n distinct elements of coded information.
- A *decoder* is a combinational circuit that converts binary information from n input lines to a maximum of 2^n unique output lines.
- If the n-bit coded information has unused combinations, the decoder may have fewer than 2^n outputs.

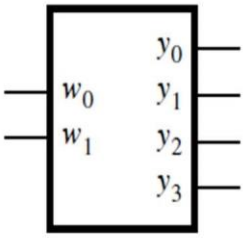
- The decoders presented here are called n -to-m -line decoders, where $m \leq 2^n$.
- Their purpose is to generate the 2^n (or fewer) minterms of n input variables.
- Each combination of inputs will assert a unique output.
- The name *decoder* is also used in conjunction with other code converters, such as a BCD-to-seven-segment decoder.

Consider a 2-to-4 decoder.

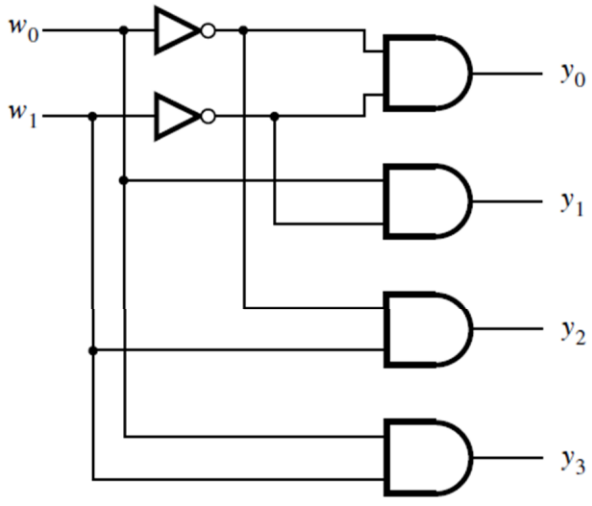
Here n =2 , m=2^2 = 4

w_1	w_0	y_0	y_1	y_2	y_3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

(a) Truth table



(b) Graphical symbol



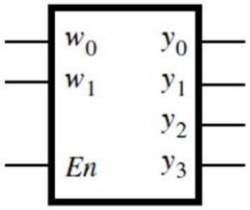
(c) Logic circuit

2-to-4 decoder

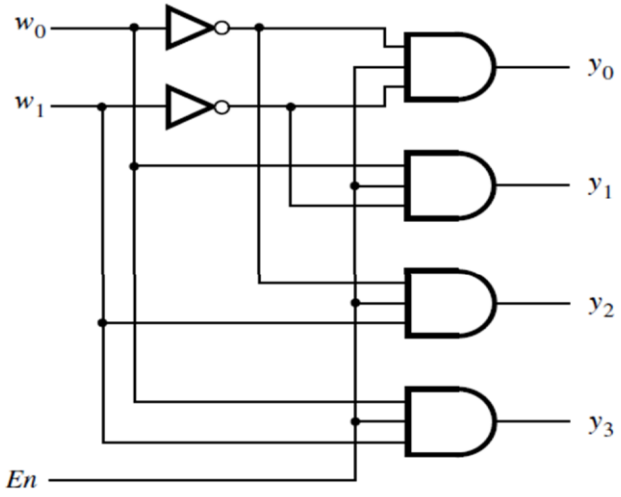
It is useful to include an *enable* input, E_n in a decoder circuit.

En	w_1	w_0	y_0	y_1	y_2	y_3
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1
0	x	x	0	0	0	0

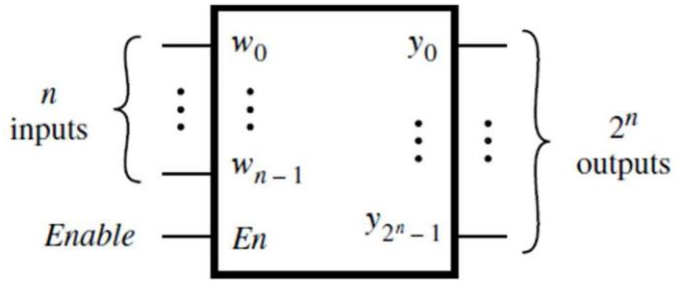
(a) Truth table



(b) Graphical symbol



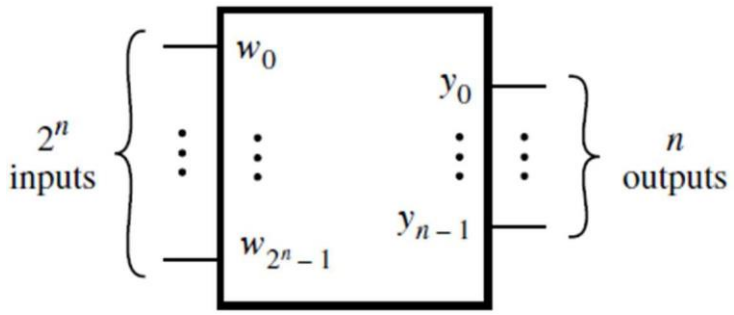
(c) Logic circuit 2-to-4 decoder with enable



An n-to-2^n decoder

Encoders

- An encoder performs the opposite function of a decoder.
- It encodes given information into a more compact form.
- A Binary encoder encodes information from 2^n inputs into an *n*-bit code, as shown.
- Exactly one of the input signals should have a value of 1, and the outputs present the binary number that identifies which input is equal to 1.



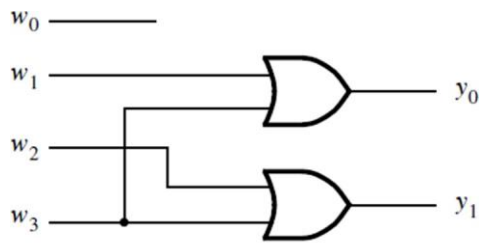
A 2^n-to-n binary encoder

4-2 Encoder

- The truth table for a 4-to-2 encoder is shown.
- Observe that the output *y*0 is 1 when either input *w*1 or *w*3 is 1, and output *y*1 is 1 when input *w*2 or *w*3 is 1.
- Hence these outputs can be generated by the circuit shown.
- Assume that the inputs are one-hot encoded.
- All input patterns that have multiple inputs set to 1 are not shown in the truth table, and they are treated as don't-care conditions.

w_3	w_2	w_1	w_0	y_1	y_0
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1

(a) Truth table



(b) Circuit

Applications

- Encoders are used to reduce the number of bits needed to represent given information.
- A practical use of encoders is for transmitting information in a digital system.
- Encoding the information allows the transmission link to be built using fewer wires.
- Encoding is also useful if information is to be stored for later use because fewer bits need to be stored.

Priority Encoders

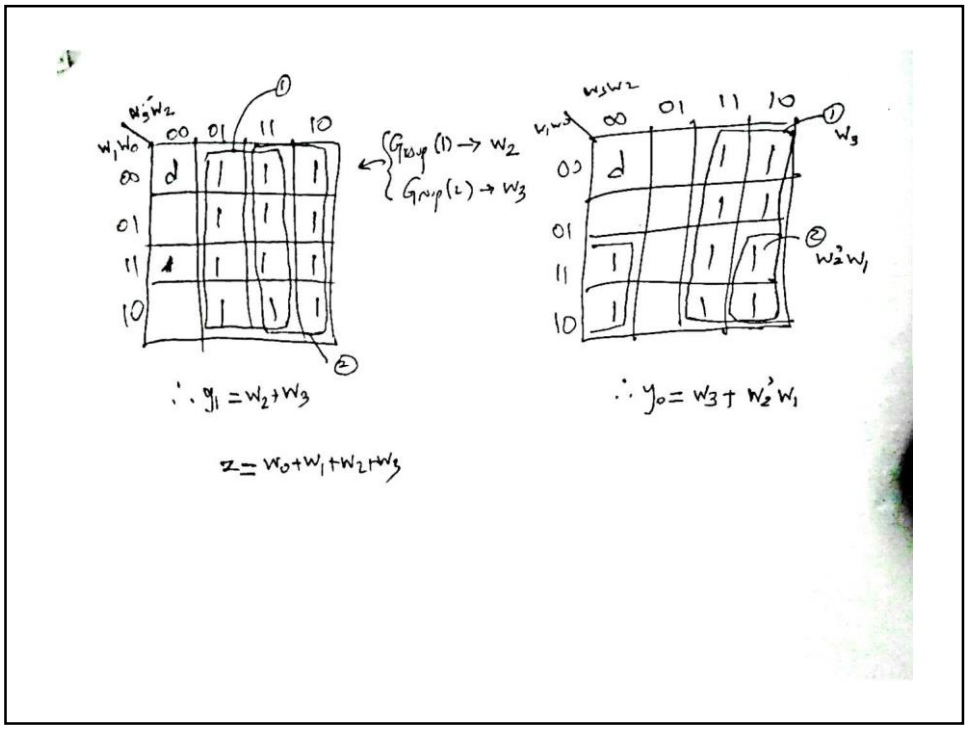
- Another useful class of encoders is based on the priority of input signals.
- In a *priority encoder* each input has a priority level associated with it.
- The encoder outputs indicate the active input that has the highest priority.
- When an input with a high priority is asserted, the other inputs with lower priority are ignored.
- The truth table for a 4-to-2 priority encoder is shown in diagram.

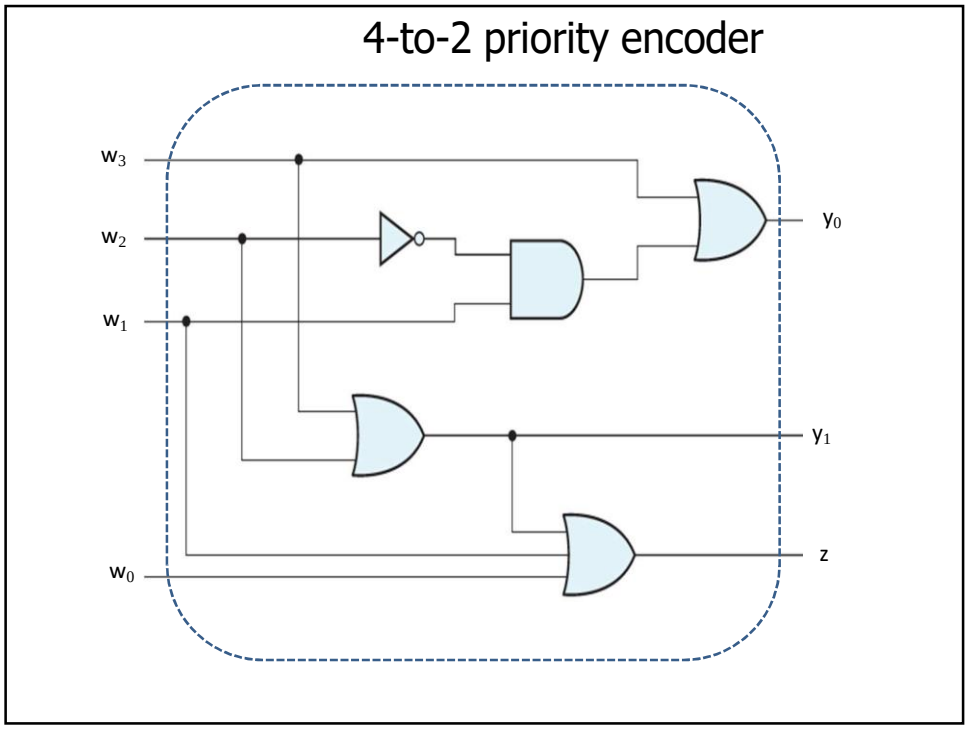
- It assumes that $w0$ has the lowest priority and $w3$ the highest.
- The outputs $y1$ and $y0$ represent the binary number that identifies the highest priority input set to 1.
- Since it is possible that none of the inputs is equal to 1, an output, z, is provided to indicate this condition.
- It is set to 1 when at least one of the inputs is equal to 1.
- It is set to 0 when all inputs are equal to 0.
- The outputs $y1$ and $y0$ are not meaningful in this case, and hence the first row of the truth table can be treated as a don't-care condition for $y1$ and $y0$.

w_3	w_2	w_1	w_0	y_1	y_0	z
0	0	0	0	d	d	0
0	0	0	1	0	0	1
0	0	1	x	0	1	1
0	1	x	x	1	0	1
1	x	x	x	1	1	1

Truth table for a 4-to-2 priority encoder.

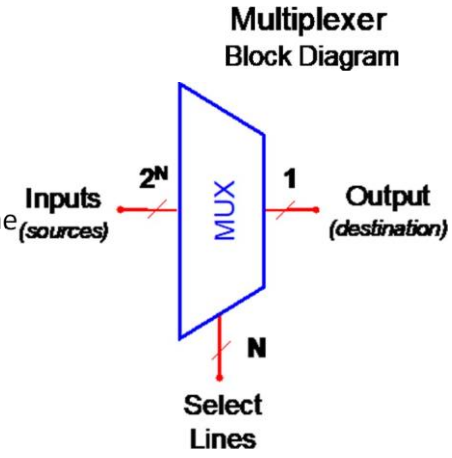
- The behavior of the priority encoder is most easily understood by first considering the last row in the truth table.
- It specifies that if input *w*3 is 1, then the outputs are set to *y*1*y*0 = 11.
- Because *w*3 has the highest priority level, the values of inputs *w*2, *w*1, and *w*0 do not matter.
- To reflect the fact that their values are irrelevant, *w*2, *w*1, and *w*0 are denoted by the symbol x in the truth table.
- The second-last row in the truth table stipulates that
- if *w*2 = 1, then the outputs are set to *y*1*y*0 = 10, but only if *w*3 = 0.
- Similarly, input *w*1 causes the outputs to be set to *y*1*y*0 = 01 only if both *w*3 and *w*2 are 0.
- Input *w*0 produces the outputs *y*1*y*0 = 00 only if *w*0 is the only input that is asserted.


$G_{Group}(1) \rightarrow w_2$
$G_{Group}(2) \rightarrow w_3$
$\therefore y_1 = w_2 + w_3$
$\therefore y_0 = w_3 + w_2' w_1$
$z = w_0 + w_1 + w_2 + w_3$

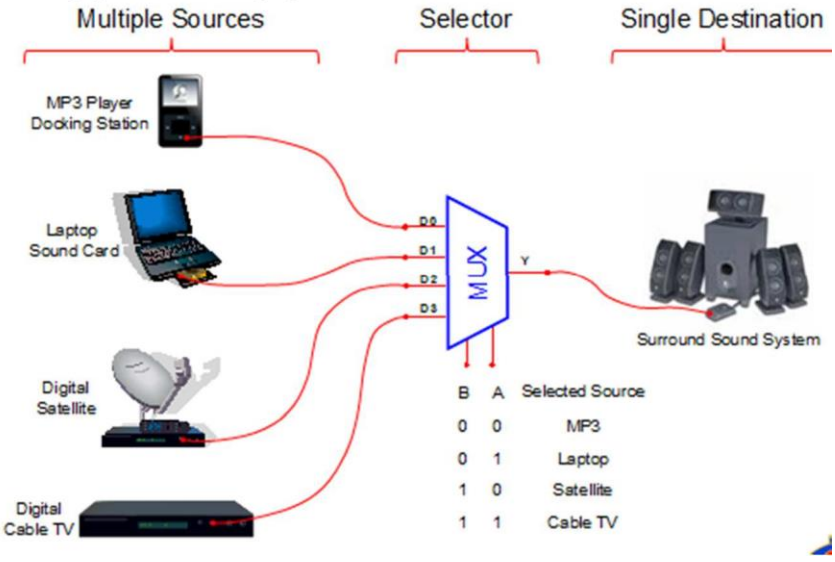

4-to-2 priority encoder
w_3
w_2
w_1
w_0
y_0
y_1
z

Multiplexer (MUX)

- A MUX is a digital switch that has multiple inputs (sources) and a single output (destination).
- The select lines determine which input is connected to the output.
- MUX Types
- 2-to-1 (1 select line)
- 4-to-1 (2 select lines)
- 8-to-1 (3 select lines)
- 16-to-1 (4 select lines)

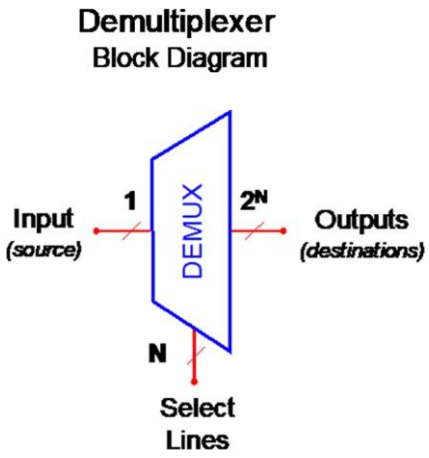


Typical Application of a MUX

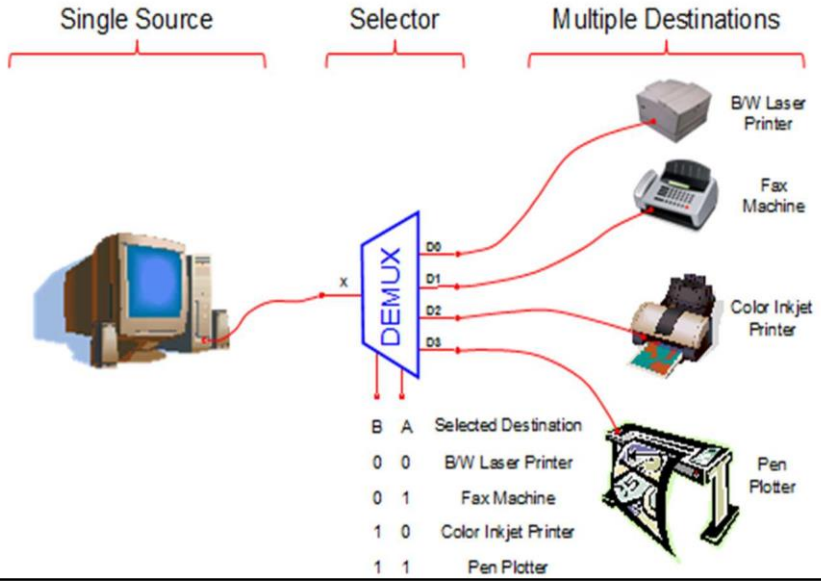


Demultiplexer (DEMUX)

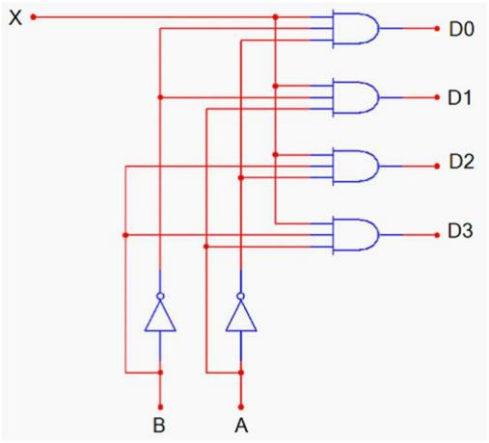
- A DEMUX is a digital switch with a single input (source) and a multiple outputs (destinations).
- The select lines determine which output the input is connected to.
- DEMUX Types
- 1-to-2 (1 select line)
- 1-to-4 (2 select lines)
- 1-to-8 (3 select lines)
- 1-to-16 (4 select lines)

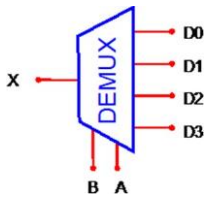


Typical Application of a DEMUX



1-to-4 De-Multiplexer (DEMUX)

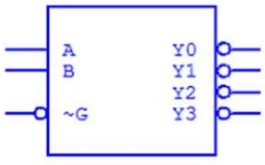




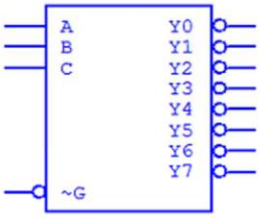
B	A	D0	D1	D2	D3
0	0	X	0	0	0
0	1	0	X	0	0
1	0	0	0	X	0
1	1	0	0	0	X

Medium Scale Integration DEMUX

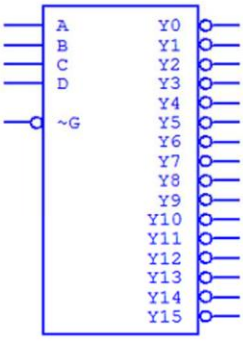
1-to-4 DEMUX



1-to-8 DEMUX



16-to-1 MUX



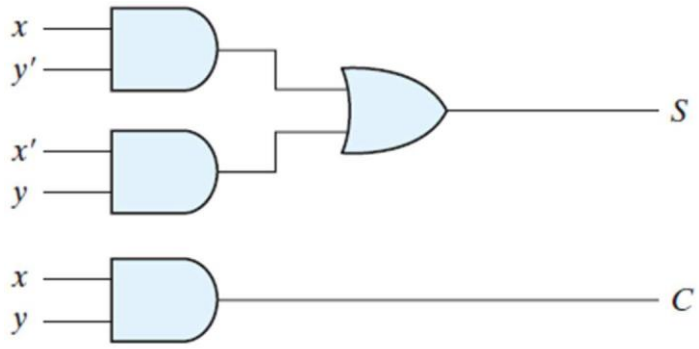
Note : Most Medium Scale Integrated (MSI) DEMUXs , like the three shown, have outputs that are inverted. This is done because it requires few logic gates to implement DEMUXs with inverted outputs rather than no-inverted outputs.

Half Adder

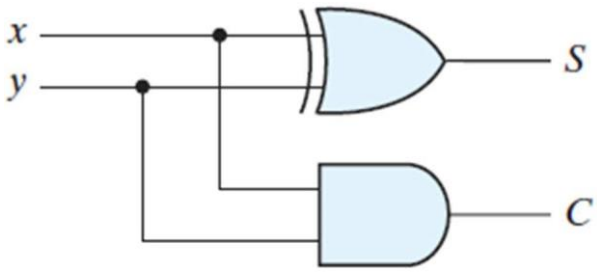
- This circuit needs two binary inputs and two binary outputs.
- The input variables designate the augend and addend bits;
- the output variables produce the sum and carry.
- We assign symbols *x* and *y* to the two inputs and *S* (for sum) and *C* (for carry) to the outputs.

Half Adder

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



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



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

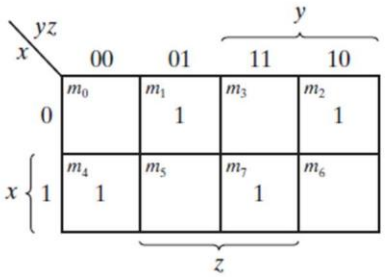
Full Adder

Full Adder

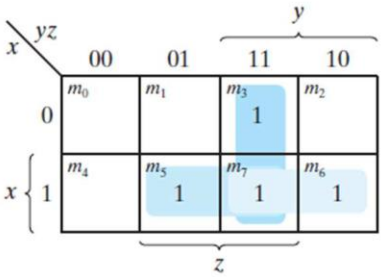
x	y	z	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

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

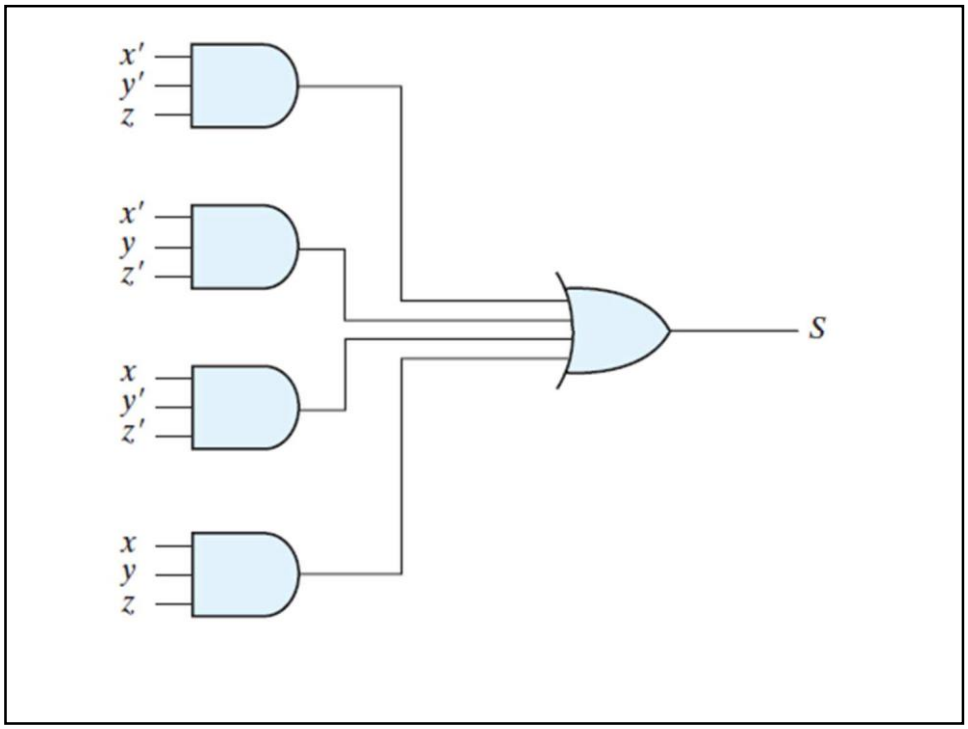
$$C = xy + xz + yz$$



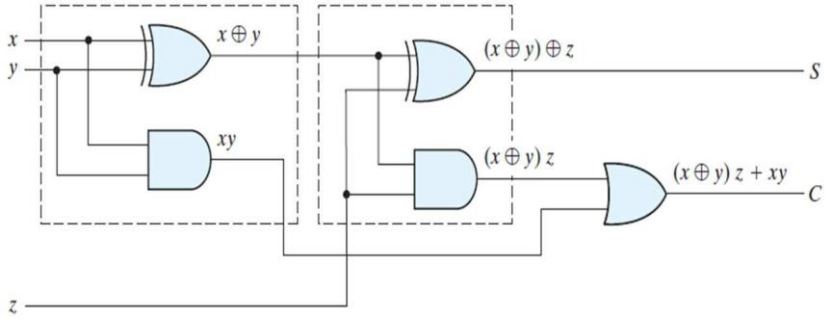
(a) $S = x'y'z + x'yz' + xy'z' + xyz$



(b) $C = xy + xz + yz$



It can also be implemented with <u>two half adders and one OR gate</u>, as shown,



The *S* output from the second half adder is the exclusive-OR of *z* and the output of the first half adder, giving,

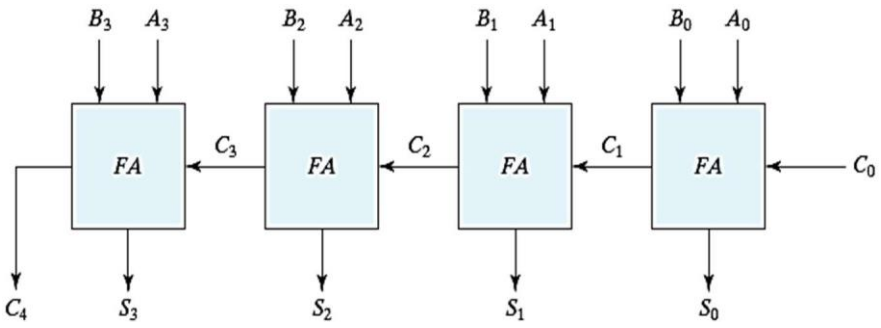
$$\begin{aligned} S &= z \oplus (x \oplus y) \\ &= z'(xy' + x'y) + z(xy' + x'y)' \\ &= z'(xy' + x'y) + z(xy + x'y') \\ &= xy'z' + x'yz' + xyz + x'y'z \end{aligned}$$

The carry output is,

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

Binary Adder

- A binary adder is a digital circuit that produces the arithmetic sum of two binary numbers.
- Full adders are connected in cascade, with the o/p carry from each full adder connected to the i/p carry of the next full adder in the chain.
- Addition of *n*-bit numbers requires a chain of *n* full adders.
- The input carry to lsb position is fixed at 0.



Four-bit adder

- Interconnection of four full-adder (FA) circuits to provide a four-bit binary ripple carry adder.
- The augend bits of *A* and the addend bits of *B* are designated by subscript numbers from right to left, with subscript 0 denoting the Lsb.
- The carries are connected in a chain through the full adders.
- The input carry to the adder is C_0, and it ripples through the full adders to the output carry C_4.
- The *S* outputs generate the required sum bits.
- An *n* -bit adder requires *n* full adders, with each output carry connected to the input carry of the next higher order full adder.

Consider the two binary numbers *A* = 1011,*B* = 0011. Their sum *S* = 1110 needs four-bit adder as follows:

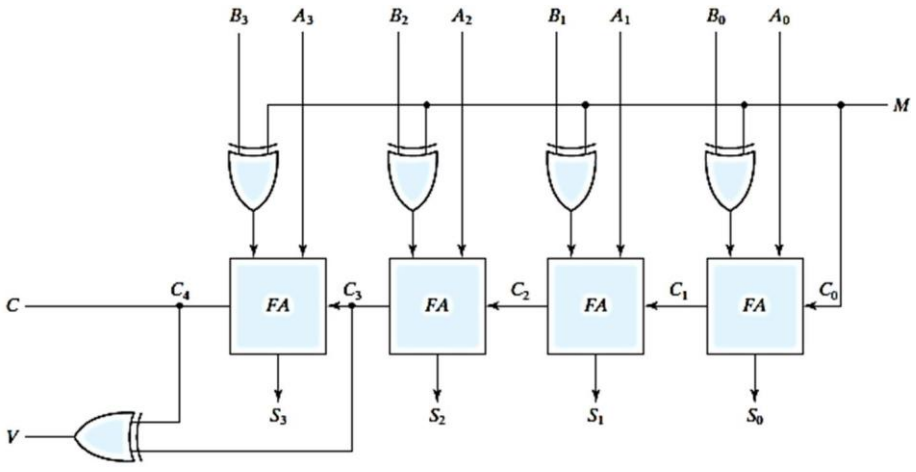
Subscript *i*:	**3**	**2**	**1**	**0**	
Input carry	0	1	1	0	C_i
Augend	1	0	1	1	A_i
Addend	0	0	1	1	B_i
Sum	1	1	1	0	S_i
Output carry	0	0	1	1	C_{i+1}

- The bits are added with full adders, starting from lsb, to form the sum bit and carry bit.
- The input carry C_0 in the lsb must be 0.
- The value of C_{i+1} in a given significant position is the output carry of the full adder.
- This value is transferred into the input carry of the full adder that adds the bits one higher significant position to the left.
- The sum bits are thus generated starting from the rightmost position and are available as soon as the corresponding previous carry bit is generated.
- All the carries must be generated for the correct sum bits to appear at the outputs.
- The four-bit adder is a typical example of a standard component.
- It can be used in many applications involving arithmetic operations.

Binary Subtractor

- The subtraction of unsigned binary numbers can be done by complements.
- subtraction *A* - *B* can be done by taking the 2's complement of *B* and adding it to *A* .
- The 2's complement can be obtained by taking the 1's complement and adding 1 to the lsb pair of bits.
- The 1's complement can be implemented with inverters, and a 1 can be added to the sum through the input carry.

- The circuit for subtracting *A* - *B* consists of an adder with inverters placed between each data input *B* and the corresponding input of the full adder.
- The input carry *C*0 must be equal to 1 when subtraction is performed.
- The operation thus performed becomes *A*, plus the 1's complement of *B* , plus 1.
- This is equal to *A* plus the 2's complement of *B*
- For unsigned numbers, that gives *A* - *B* if $A \geq B$ or the 2's complement of $B - A$ if $A < B$.
- For signed numbers, the result is *A* - *B*, provided that there is no overflow.



Four-bit adder–subtractor (with overflow detection)

- The add and sub operations can be combined into one circuit with one common binary adder by including an ex-OR gate with each full adder.
- A four-bit adder–subtractor circuit is shown.
- The mode input *M* controls the operation.
- When *M* = 0, circuit is an adder, and when *M* = 1, circuit becomes a subtractor.
- Each ex-OR gate receives input *M* and one of i/ps of *B* .
- When *M* = 0, we have $B \oplus 0 = B$.
- The full adders receive the value of *B* , the i/p carry is 0, and the circuit performs *A* plus *B* .
- When *M* = 1, we have $B \oplus 1 = B$ and $C_0 = 1$.
- The *B* i/ps are all inverted and a 1 is added thro' i/p carry.
- The circuit performs the operation *A* plus the 2's complement of *B* .
- The ex-OR with output *V* is for detecting an overflow.

Overflow

- When two numbers with *n* digits each are added and the sum is a number occupying *n* + 1 digits, we say that an overflow occurred.
- This is true for binary or decimal numbers, signed or unsigned.
- Overflow is a problem since the bits that hold the number is finite and a result that contains *n* + 1 bits cannot be put up by an *n* -bit word.
- So, it is needed to detect the occurrence of an overflow.

- The detection of an overflow after addition of numbers depends on whether the numbers are signed or unsigned.
- When two unsigned numbers are added, an overflow is detected from the end carry out of the msb.
- In the case of signed numbers, two details are important:
  - the leftmost bit always represents the sign, and
  - negative numbers are in 2's-complement form.
- When two signed numbers are added, the sign bit is treated as part of the number and the end carry does not indicate an overflow.
- An overflow may occur if the two numbers added are both positive or both negative.

- An overflow condition can be detected by observing the carry into the sign bit position and the carry out of the sign bit position.
- If these two carries are not equal, an overflow has occurred.
- If the two carries are applied to an ex-OR gate, an overflow is detected, when the output of the gate is equal to 1.
- For this to work correctly, the 2's complement of a negative number must be computed by taking the 1's complement and adding 1.

BCD Adder

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

Derivation of BCD Adder

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

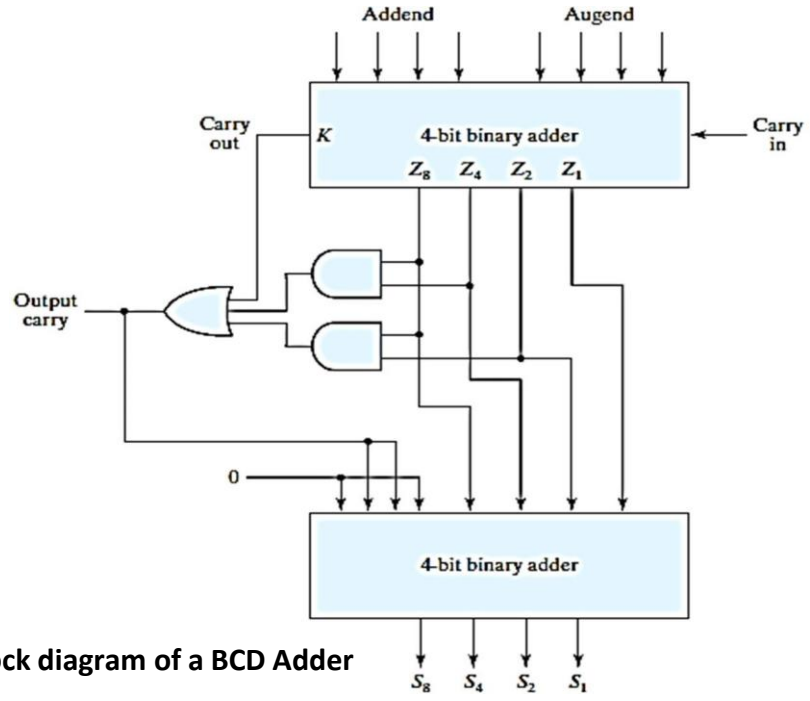
- These binary numbers are listed and are labeled by symbols K, Z_8, Z_4, Z_2, and Z_1.
- K is the carry, and the subscripts under the letter Z represent the weights 8, 4, 2, and 1 that can be assigned to four bits in BCD code.
- The columns under binary sum list the binary value that appears in the o/ps of the four-bit binary adder.
- The output sum of two decimal digits must be represented in BCD and should appear in the form listed in the columns under "BCD Sum."

- When the binary sum is equal to or less than 1001, the corresponding BCD number is identical, and so no conversion is needed.
- When the binary sum is greater than 1001, we obtain an invalid BCD representation.
- The addition of binary 6 (0110) to the binary sum converts it to the correct BCD value and also produces an output carry as required.
- The logic circuit that detects the necessary correction can be derived from the entries in the table.
- It is obvious that a correction is needed when the binary sum has an output carry $K = 1$.

- The other six combinations from 1010 through 1111 that need a correction have a 1 in position Z_8.
- To distinguish them from binary 1000 and 1001, which also have a 1 in position Z_8, specify that either Z_4 or Z_2 must have a 1.
- The condition for a correction and an output carry can be expressed by the Boolean function,

$$C = K + Z_8Z_4 + Z_8Z_2$$

- When C = 1, it is needed to add 0110 to binary sum and provide an o/p carry for next stage.



Block diagram of a BCD Adder

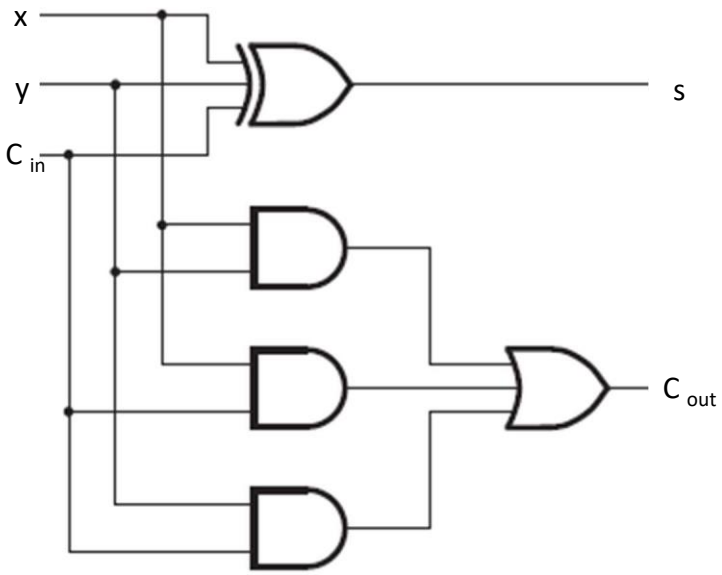
- The two decimal digits, together with the input carry, are first added in the top four-bit adder to produce the binary sum.
- When the output carry is equal to 0, nothing is added to the binary sum.
- When it is equal to 1, binary 0110 is added to the binary sum thro' the bottom four-bit adder.
- The output carry generated from bottom adder can be ignored, since it supplies information already available at the output carry terminal.
- A decimal parallel adder that adds *n* decimal digits needs *n* BCD adder stages.
- The o/p carry from one stage must be connected to the i/p carry of the next higher order stage.

Gate Level Modelling of Combinational Logic in Verilog

ECT 203 Module – III (Cont…)

Design of Arithmetic Circuits Using Verilog

- To implement the full-adder circuit that has the inputs C_{in}, x and y and produces the outputs s and C_{out}.
- One way of specifying this circuit in Verilog is to use the gate-level primitives
- Each of the three AND gates in the circuit is defined by a separate statement.
- Verilog allows combining such statements into a single statement.
- In this case, commas are used to separate the definition of each AND gate.



Logic Circuit of Full Adder

```
module fulladd (Cin, x, y, s, Cout);
   input Cin, x, y;  // declare Cin, x and y as system inputs
   output s, Cout;   // declare Cout and s as system outputs

   xor (s, x, y, Cin);
   and (z1, x, y);
   and (z2, x, Cin);
   and (z3, y, Cin);
   or (Cout, z1, z2, z3);

endmodule
```

Verilog code for the full-adder using gate-level primitives.

```
module fulladd (Cin, x, y, s, Cout);
   input Cin, x, y;
   output s, Cout;

   xor (s, x, y, Cin);
   and (z1, x, y),
       (z2, x, Cin),
       (z3, y, Cin);
   or (Cout, z1, z2, z3);

endmodule
```

Another version of Verilog code

- Another possibility is to use functional expressions.
- The XOR operation is denoted by the ^ sign.
- Again, it is possible to combine the two continuous assignment statements into a single statement.
- Both of the above approaches result in the same full-adder circuit being synthesized.
- Use Full Adder code to build a Multi-bit Adder.

```
module fulladd (Cin, x, y, s, Cout);
   input Cin, x, y;
   output s, Cout;

   assign s = x ^ y ^ Cin;
   assign Cout = (x & y) | (x & Cin) | (y & Cin);

endmodule
```

Verilog code for the full-adder using continuous assignment.

```
module fulladd (Cin, x, y, s, Cout);
   input Cin, x, y;
   output s, Cout;

   assign s = x ^ y ^ Cin,
          Cout = (x & y) | (x & Cin) | (y & Cin);

endmodule
```

Another version of Verilog code

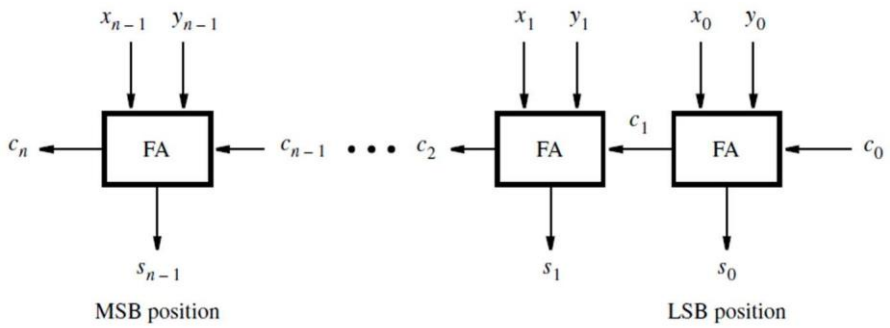
Ripple Carry Adder (RCA)

- Create a separate Verilog module for the ripple-carry adder, which instantiates the *fulladd* module as a sub circuit.
- One method of doing this is shown.
- The module comprises the code for a four-bit ripple-carry adder, named *adder4*.
- One of the 4-bit numbers to be added is represented by the four signals *x*3, *x*2, *x*1, *x*0, and the other is represented by *y*3, *y*2, *y*1, *y*0.
- The sum is represented by *s*3, *s*2, *s*1, *s*0.
- The circuit incorporates a carry input, *carryin*, into the lsb position and a carry output, *carryout*, from the msb position.

```
module adder4 (carryin, x3, x2, x1, x0, y3, y2, y1, y0, s3, s2, s1, s0, carryout);
   input carryin, x3, x2, x1, x0, y3, y2, y1, y0;
   output s3, s2, s1, s0, carryout;

   fulladd stage0 (carryin, x0, y0, s0, c1);
   fulladd stage1 (c1, x1, y1, s1, c2);
   fulladd stage2 (c2, x2, y2, s2, c3);
   fulladd stage3 (c3, x3, y3, s3, carryout);

endmodule

module fulladd (Cin, x, y, s, Cout);
   input Cin, x, y;
   output s, Cout;

   assign s = x ^ y ^ Cin;
   assign Cout = (x & y) | (x & Cin) | (y & Cin);

endmodule
```

Verilog code for a four-bit adder.

- The four-bit adder is described using four *instantiation* statements.
- Each statement begins with the name of the module, *fulladd*, that is being instantiated, followed by an *instance name*.
- The instance names must be unique.
- The least-significant stage in the adder is named *stage0* and the most-significant stage is *stage3*.
- The signal names in the *adder4* module that are to be connected to each input and output port on the *fulladd* module are then listed.
- These signals are listed in the same order as in the *fulladd* module, namely the order *Cin*, *x*, *y*, *s*, *Cout*.



An n-bit ripple-carry adder.

- The signal names associated with each instance of the *fulladd* module implicitly specify how the full-adders are connected together.
- For example, the carry-out of the *stage0* instance is connected to the carry-in of the *stage1* instance.
- The synthesized circuit has the same structure as the original block diagram.
- The *fulladd* module may be included in the same Verilog source code file as the *adder4* module.
- If compiler needs the function, create another file *fulladd* and the location of the file *fulladd* has to be indicated to the compiler.

Using Vectored Signals

- Each of the four-bit inputs and the four-bit output of the adder is represented using single-bit signals.
- A more convenient approach is to use multi-bit signals, called *vectors*, to represent the numbers.
- Just as a number is represented in a logic circuit as signals on multiple wires, it can be represented in Verilog code as a multi-bit vector.

An example of an input vector is,

input [3:0] X;

This statement defines *X* to be a four-bit vector.

Its individual bits can be referred to by using an index value in square brackets.

The (MSB) is referred to as *X* [3] and the (LSB) is *X* [0].

A two-bit vector that consists of the two middle bits of *X* is denoted as *X* [2:1].

The symbol *X* refers to the entire vector.

- Use vectors to specify the four-bit adder.
- In addition to the input vectors *X* and *Y* , and output vector *S*, chose to define the carry signals between the full-adder stages as a three-bit vector *C*[3:1].
- Note that the carry into *stage0* is still called *carryin*, while the carry from *stage3* is called *carryout*.
- The internal carry signals are defined in the statement,

wire [3:1] C;

- Signal C[1] is used to connect the carry output of the full-adder in stage 0 to the carry input of the full-adder in stage 1.
- Similarly, C[2] and C[3] are used to connect the other stages of the adder.
- The vector specification gives the bit width in square brackets, as in *X* [3:0].
- The bit width is specified using the index of the MSB first and the LSB last.
- Hence, *X* [3] is the MSB and *X* [0] is the LSB.

```
module adder4 (carryin, X, Y, S, carryout);
   input carryin;
   input [3:0] X, Y;
   output [3:0] S;
   output carryout;
   wire [3:1] C;

   fulladd stage0 (carryin, X[0], Y[0], S[0], C[1]);
   fulladd stage1 (C[1], X[1], Y[1], S[1], C[2]);
   fulladd stage2 (C[2], X[2], Y[2], S[2], C[3]);
   fulladd stage3 (C[3], X[3], Y[3], S[3], carryout);

endmodule
```

A four-bit adder using vectors.

The Conditional Operator

- In a logic circuit it is needed to choose between several possible signals or values.
- It's based on the state of some condition.
- An example is a multiplexer circuit in which the output is equal to the data input signal chosen by the valuation of the select inputs.
- For simple implementation of such choices Verilog provides a *conditional* operator (?:) which assigns one of two values depending on a conditional expression.
- It involves three operands used in the syntax
- `conditional_expression ? true_expression : false_expression`

- If the conditional expression evaluates to 1 (true), then the value of true_expression is chosen;
- otherwise, the value of false_expression is chosen.
- For example, the statement
- A= (B < C) ? (D + 5) : (D + 2);
- means that if *B* is less than *C*, the value of *A* will be *D* + 5, or else *A* will have value *D* + 2.
- The conditional operator can be used both in continuous assignment statements and in procedural statements inside an **always** block.

- A 2-to-1 multiplexer can be defined using the conditional operator in an **assign** statement.
- The module, named *mux2to1*, has the inputs *w*0, *w*1, and *s*, and the output *f* .
- The signal *s* is used for the selection criterion.
- The output *f* is equal to *w*1 if the select input *s* has the value 1;
- otherwise, *f* is equal to *w*0.

```
module mux2to1 (w0, w1, s, f);
   input w0, w1, s;
   output f;

   assign f = s ? w1 : w0;

endmodule
```

A 2-to-1 multiplexer specified using the conditional operator.

```
module mux2to1 (w0, w1, s, f);
   input w0, w1, s;
   output reg f;

   always @(w0, w1, s)
      f = s ? w1 : w0;

endmodule
```

An alternative specification of a 2-to-1 multiplexer using the conditional operator.

```
module mux4to1 (w0, w1, w2, w3, S, f);
   input w0, w1, w2, w3;
   input [1:0] S;
   output f;

   assign f = S[1] ? (S[0] ? w3 : w2) : (S[0] ? w1 : w0);

endmodule
```

4-to-1 multiplexer specified using the conditional operator.

The If-Else Statement

```
if(conditional_expression)
statement;
else statement;
```

- The conditional expression may use the relational operators.
- If the expression is evaluated to true then the first statement (or a block of statements delineated by **begin** and **end** keywords) is executed, or else the second statement (or a block of statements) is executed.

Example

- The **if-else** statement can be used to describe a 2-to-1 multiplexer.
- The **if** clause states that *f* is assigned the value of *w*0 when *s* = 0.
- Else, *f* is assigned the value of *w*1.

```
module mux2to1 (w0, w1, s, f);
   input w0, w1, s;
   output reg f;

   always @(w0, w1, s)
      if (s == 0)
         f = w0;
      else
         f = w1;

endmodule
```

Code for a 2-to-1 multiplexer using the **if-else** statement.

- The **if-else** statement can be used to implement larger multiplexers.
- A 4-to-1 multiplexer is shown.
- The **if-else** clauses set *f* to the value of one of the inputs *w*0*, . . . ,w*3, depending on valuation of *S*.

```
module mux4to1 (w0, w1, w2, w3, S, f);
    input w0, w1, w2, w3;
    input [1:0] S;
    output reg f;

    always @(*)
        if (S == 2'b00)
            f = w0;
        else if (S == 2'b01)
            f = w1;
        else if (S == 2'b10)
            f = w2;
        else
            f = w3;

endmodule
```

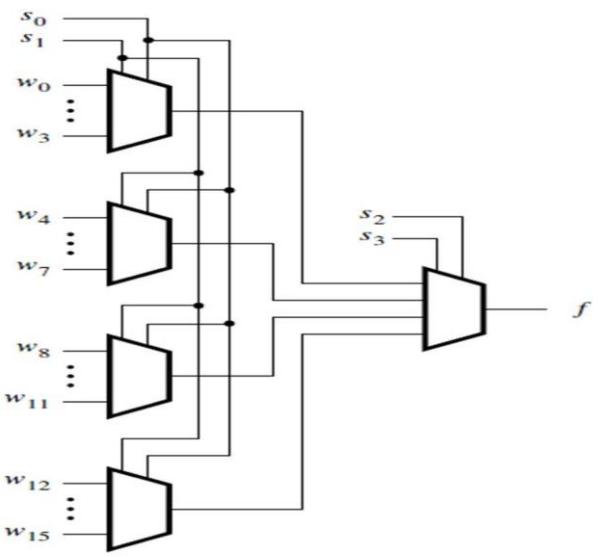
Alternate code

- Another way of defining the same circuit is shown.
- In this case, a four-bit vector *W* is defined instead of single-bit signals *w*0, *w*1, *w*2, and *w*3.
- Also, the four different values of *S* are specified as decimal rather than binary numbers.

```
module mux4to1 (W, S, f);
    input [0:3] W;
    input [1:0] S;
    output reg f;

    always @(W, S)
        if (S == 0)
            f = W[0];
        else if (S == 1)
            f = W[1];
        else if (S == 2)
            f = W[2];
        else
            f = W[3];

endmodule
```

16-to-1 Multiplexer

- A 16-to-1 multiplexer can be built by using five 4-to-1 multiplexers.
- Verilog code for this circuit uses 5 instantiations of *mux4to1* module.
- The data inputs to the *mux16to1* module are the 16-bit vector *W*, and the select inputs are the four-bit vector *S*.
- In the Verilog code signal names are needed for the outputs of the four 4-to-1 multiplexers on the left.
- A four-bit signal named *M* is used for this purpose.
- The first multiplexer instantiated, *Mux1*, corresponds to the multiplexer at the top left.
- Its first four ports are driven by the signals *W*[0], . . . , *W*[3].



A 16-to-1 multiplexer.

$$f = \bar{s}_3\bar{s}_2\bar{s}_1\bar{s}_0w_0 + \bar{s}_3\bar{s}_2\bar{s}_1s_0w_1 + \bar{s}_3\bar{s}_2s_1\bar{s}_0w_2 + \cdots + s_3s_2s_1\bar{s}_0w_{14} + s_3s_2s_1s_0w_{15}$$

- The syntax *S*[1:0] is used to attach the signals *S*[1] and *S*[0] to the two-bit *S* port of *mux4to1* module.
- The *M*[0] signal is connected to the multiplexer's output port.
- Similarly, *Mux2, Mux3*, and *Mux4* are instantiations of the next three multiplexers on the left.
- The Multiplexer on the right is instantiated as *Mux5*.
- The signals *M*[0], . . . , *M*[3] are connected to its data inputs, and bits *S*[3] and *S*[2] are attached to the select inputs.
- The output port generates the *mux16to1* output *f* .
- Compiling the code results in the multiplexer function

- Since *mux4to1* module is being instantiated in the code, it is necessary to either include the code in same file as the *mux16to1* module or place the *mux4to1* module in a separate file in same directory.

- Observe that if the scalar code were used as the required *mux4to1* module, then need to list the ports separately, as in *W*[0], *W*[1], *W*[2], *W*[3], rather than as the vector *W*[0:3].

```
module mux16to1 (W, S, f);
   input [0:15] W;
   input [3:0] S;
   output f;
   wire [0:3] M;

   mux4to1 Mux1 (W[0:3], S[1:0], M[0]);
   mux4to1 Mux2 (W[4:7], S[1:0], M[1]);
   mux4to1 Mux3 (W[8:11], S[1:0], M[2]);
   mux4to1 Mux4 (W[12:15], S[1:0], M[3]);
   mux4to1 Mux5 (M[0:3], S[3:2], f);

endmodule
```

Hierarchical code for a 16-to-1 multiplexer.

The Case Statement

```
case (expression)
    alternative1: statement;
    alternative2: statement;
    .
    .
    .
    alternativej: statement;
    [default: statement;]
endcase
```

- The value of the controlling expression and each alternative are compared bit by bit.
- When there is one or more matching alternative, the statement(s) associated with the first match (only) is executed.
- When the specified alternatives do not cover all possible valuations of the controlling expression, the optional **default** clause should be included.

4:1 MUX using case statement

- **T**he **case** statement can be used to define a 4-to-1 multiplexer.
- The four values that the select vector *S* can have are given as decimal numbers, but they could also be given as binary numbers.

```
module  mux4to1 (W, S, f);
   input  [0:3] W;
   input  [1:0] S;
   output  reg  f;

   always @(W, S)
      case (S)
         0: f = W[0];
         1: f = W[1];
         2: f = W[2];
         3: f = W[3];
      endcase

endmodule
```

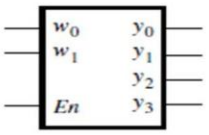
A 4-to-1 multiplexer defined using the **case** statement.

2-to-4 Binary Decoder

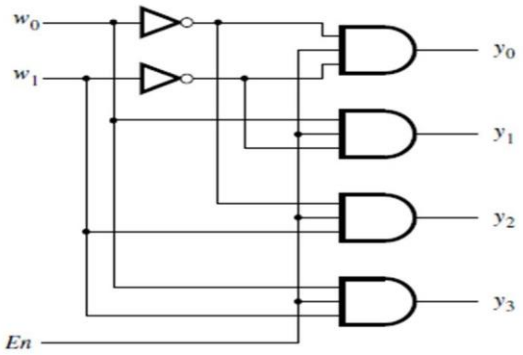
- A **case** statement can be used to describe the truth table for a 2-to-4 binary decoder.
- The data inputs are two-bit vector *W*, and the enable input is *En*.
- The four outputs are represented by the four-bit vector *Y* .
- In truth table, the inputs are listed in the order *En, w*1, *w*0.
- To represent these three signals in the controlling expression, Verilog code uses concatenate operator to combine the *En* and *W* signals into a three-bit vector.
- The four alternatives in the **case** statement correspond to the truth table in where *En* = 1, and the decoder outputs have the same patterns as in the first four rows of the truth table.
- The last clause uses the **default** keyword and sets the decoder outputs to 0000, because it represents all other cases, namely those where *En* = 0.

En	w_1	w_0	y_0	y_1	y_2	y_3
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1
0	x	x	0	0	0	0

(a) Truth table



(b) Graphical symbol



(c) Logic circuit

```
module dec2to4 (W, En, Y);
   input [1:0] W;
   input En;
   output reg [0:3] Y;

   always @(W, En)
      case ({En, W})
         3'b100: Y = 4'b1000;
         3'b101: Y = 4'b0100;
         3'b110: Y = 4'b0010;
         3'b111: Y = 4'b0001;
         default: Y = 4'b0000;
      endcase

endmodule
```

Verilog code for a 2-to-4 binary decoder.

Alternative way

- **T**he 2-to-4 decoder can be specified using a combination of **if-else** and **case** statements.
- If *En* = 0, then all four bits of the output *Y* are set to the value 0, else
- the **case** alternatives are evaluated if *En* = 1.

```
module dec2to4 (W, En, Y);
   input [1:0] W;
   input En;
   output reg [0:3] Y;

   always @(W, En)
   begin
      if (En == 0)
         Y = 4'b0000;
      else
         case (W)
            0: Y = 4'b1000;
            1: Y = 4'b0100;
            2: Y = 4'b0010;
            3: Y = 4'b0001;
         endcase
   end

endmodule
```

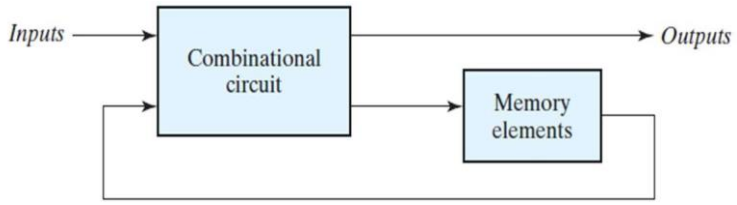
Alternative code for a 2-to-4 binary decoder.

Sequential Logic Circuits

ECT Module IV

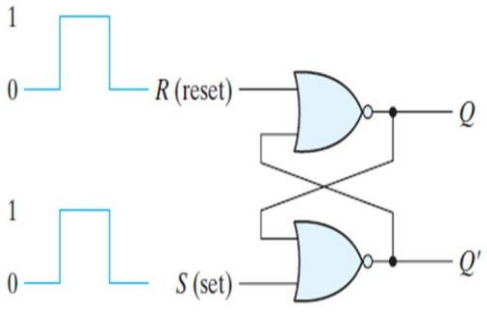
SEQUENTIAL CIRCUITS

- It consists of a combinational circuit to which storage elements are connected to form a feedback path.
- The binary information stored in these elements at any given time defines the *State* of the sequential circuit at that time.
- The sequential circuit receives information from external i/ps that, together with the present state of the storage elements, determine the binary value of the outputs.
- These external i/ps also determine the condition for changing the state.



Block diagram of sequential circuit

S R Latch



(a) Logic diagram

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

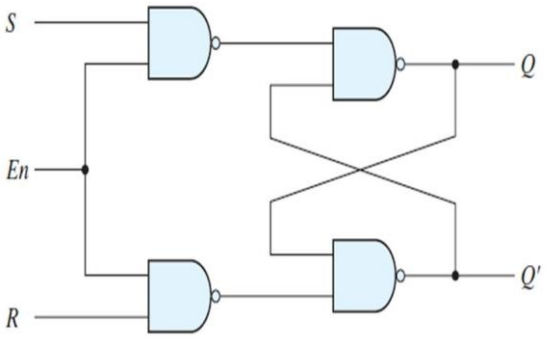
(b) Function table

SR latch with NOR gates

- The *SR* latch is a circuit with two cross-coupled NOR gates or two cross-coupled NAND gates, and two i/ps labeled *S* for set and *R* for reset.
- The latch has two useful States.
- When output *Q* = 1 and *Q'* = 0, the latch is said to be in the *set state* .
- When *Q* = 0 and *Q'* = 1, it is in the *reset state* .
- Outputs *Q* and *Q* are normally the complement of each other.
- However, when both inputs are equal to 1 at the same time, a condition in which both outputs are equal to 0 occurs.
- If both inputs are then switched to 0 simultaneously, the device will enter an unpredictable or undefined state or a metastable state.
- So, in practical applications, setting both inputs to 1 is forbidden.

- Under normal conditions, both I /ps of the latch remain at 0 unless the state has to be changed.
- The application of a 1 to the *S* input causes the latch to go to the set state.
- The *S* input must go back to 0 before any other changes take place, in order to avoid the occurrence of an undefined next state that results from the forbidden input condition.
- As shown in the function table, two i/p conditions cause the circuit to be in set state.

- The first condition ($S = 1, R = 0$) is the action that must be taken by input *S* to bring the circuit to the set state.
- Removing the active i/p from *S* leaves the circuit in same state.
- After both i/ps return to 0, it is then possible to shift to the reset state by applying a 1 to the *R* input.
- The 1 can then be removed from *R*, whereupon the circuit remains in the reset state.
- Thus, when both i/ps *S* and *R* are equal to 0, the latch can be in either the set or the reset state, depending on which input was most recently a 1.
- If a 1 is applied to both the *S* and *R* inputs of the latch, both outputs go to 0.
- In normal operation, this condition is avoided by making sure that 1's are not applied to both inputs simultaneously.



(a) Logic diagram

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

(b) Function table

SR latch with control input

SR latch with a control input

- It consists of the basic *SR* latch and two additional NAND gates.
- The control input *En* acts as an *enable* signal for the other two inputs.
- **The outputs of the NAND gates stay at the logic-1 level as long as the enable signal remains at 0.**
- This is the quiescent condition for the *SR* latch.
- When the enable input goes to 1, information from the *S* or *R* input is allowed to affect the latch.
- The set state is reached with $S = 1$, $R = 0$, $En = 1$

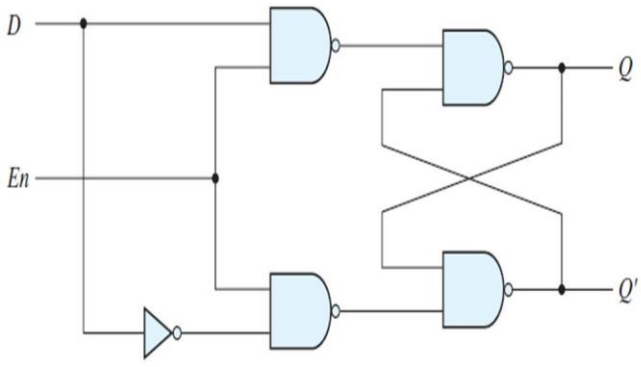
- To change to the reset state, the inputs must be $S = 0$, $R = 1$, and $En = 1$.
- In either case, when *En* returns to 0, the circuit remains in its current state.
- The control input disables the circuit by applying 0 to *En*, so that the state of the output does not change regardless of the values of *S* and *R* .
- Moreover, when $En = 1$ and both the *S* and *R* inputs are equal to 0, the state of the circuit does not change.

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

D Latch (Transparent Latch)

- To remove the undesirable condition of the indeterminate state in the *SR* latch ensure that i/p *S* and *R* are never equal to 1 at the same time.
- This is done in the *D* latch.
- This latch has only two i/p: *D* (data) and *En* (enable).
- The *D* i/p goes directly to *S* input, its complement is applied to the *R* i/p.
- With enable i/p is at 0, the cross-coupled *SR* latch has both i/p at the 1 level and the circuit cannot change state regardless of the value of *D* .

- The *D* i/p is sampled when *En* = 1.
- If *D* = 1, the *Q* output goes to 1, placing the circuit in the set state.
- If *D* = 0, output *Q* goes to 0, placing the circuit in the reset state.



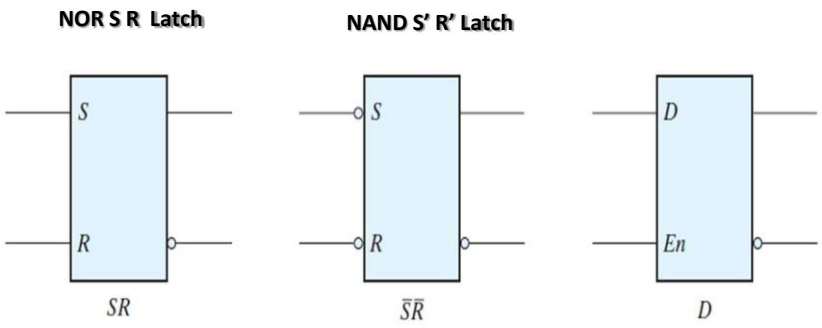
(a) Logic diagram

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

(b) Function table

D latch

So, Class, tell me how to convert a S R latch to D latch …

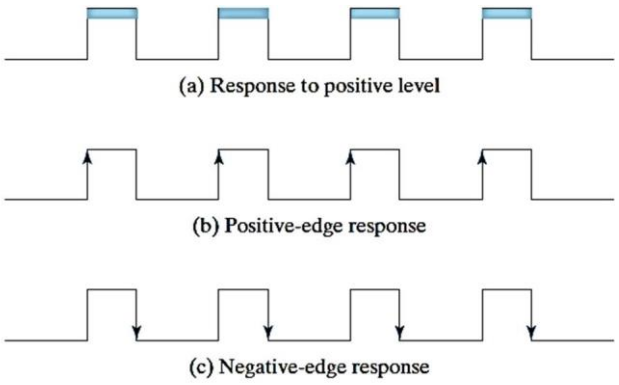


Graphic symbols for latches

- The *D* latch is suited for use as a temporary storage for binary information.
- The binary information present at data i/p of *D* latch is transferred to *Q* output when enable i/p is asserted.
- The o/p follows changes in the data input as long as the enable i/p is asserted.
- This situation provides a path from i/p *D* to the o/p, and for this reason, the circuit is often called a *transparent* latch.
- When the enable i/p signal is de-asserted, the binary info present at the data i/p at the time the transition occurred is retained (i.e., stored) at the *Q* o/p until the enable i/p is asserted again.

Points to Ponder: Latch vs Flip-Flop

- Flip-flop circuits are built to make them operate properly when they are part of a sequential circuit that employs a common clock.
- The problem with the latch is that it responds to a change in the level of a clock pulse.
- The key to the proper operation of a flip-flop is to trigger it only during a signal transition .
- A clock pulse goes through two transitions: from
  - 0 to 1 · Positive Edge response
  - from 1 to 0 · Negative Edge response



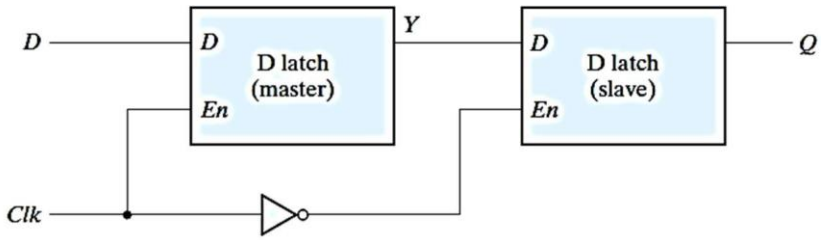
Clock response in latch and flip-flop

Latch Flip-Flop

- Modify a latch to form a flip-flop.
  - Use two latches in a special configuration that isolates o/p of flip-flop, prevents it from being affected while i/p to flip-flop is changing.
  - To produce a flip-flop that triggers only during a signal transition (from 0 to 1 or from 1 to 0) of synchronizing signal (clock) and is disabled during rest of the clock pulse.

Edge-Triggered D Flip-Flop

- A *D* flip-flop is built with two *D* latches and an inverter.
- The first latch is called the Master and the second latch is the Slave.
- The circuit samples the *D* i/p and changes its o/p *Q* only at the -ve edge of clock *Clk* .
- When the clock is 0, o/p of the inverter is 1.
- The slave latch is enabled, and its o/p *Q* is equal to the master o/p *Y* .
- The master latch is disabled because *Clk* = 0.

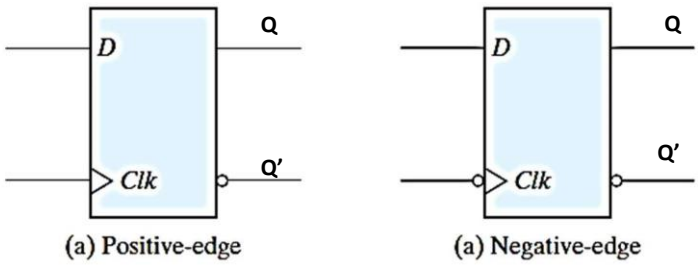


Master–slave *D* flip-flop

- When the i/p pulse changes to logic-1 level, data from ext *D* i/p are transferred to Master.
- The 'poor' slave is disabled as long as the clock remains at the 1 level, as its *enable* input is 0.
- Any change in i/p changes the Master o/p at *Y*, but cannot affect the slave o/p.
- When the clock pulse returns to 0, the Master is disabled and is isolated from the *D* input.
- At same time, slave is enabled and value of *Y* is transferred to o/p of flip-flop at *Q* .
- Thus, *a change in the o/p of the flip-flop can be triggered only by transition of clock from 1* to 0.
- Similarly, *a change in the o/p of the flip-flop can be triggered only by transition of clock from 0 to 1 by use of addnl inverter at Clk i/p.*

Graphic Symbol for ET or M-S D FF

- It is similar to the symbol used for the *D* latch, except for the arrowhead-like symbol in front of the letter *Clk*, designating a *dynamic* input.
- The *dynamic indicator* (>) indicates that flip-flop responds to the edge transition of clock.
- A bubble besides dynamic indicator indicates a negative edge for triggering the circuit.
- Absence of a bubble designates a positive-edge response.

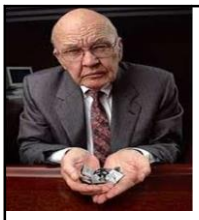


Graphic symbol for edge-triggered *D* flip-flop

Flip Flop changes State with every Clock edge
Flip Flop at Clock edge at t=1, changes to next State at Clock edge at t=2, FF at Clock edge at t=2 changes to next state at Clock edge at t=3 and so on ….
Hence given state Q(t) the next state is Q(t+1)

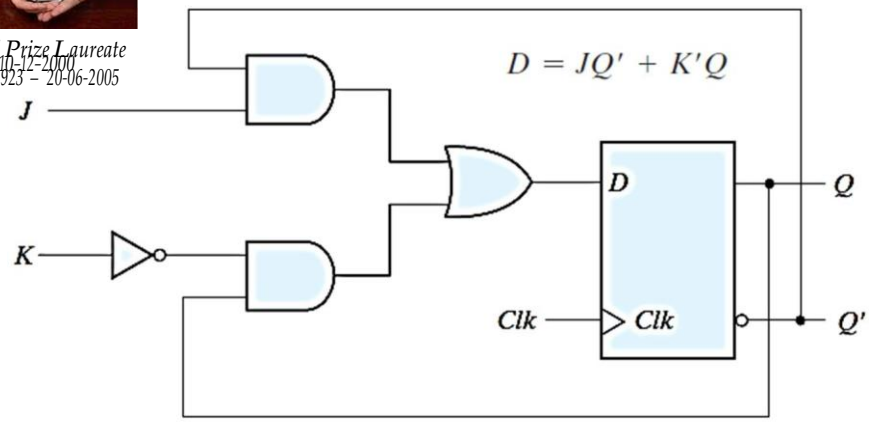
Clk	D	$Q(t+1)$
0	x	$Q(t)$
1	0	0
1	1	1

Characteristic table



Nobel Prize Laureate
10-12-2000
08-11-1923 – 20-06-2005

Jack Kilby (J-K) Flip Flop



(a) Circuit diagram

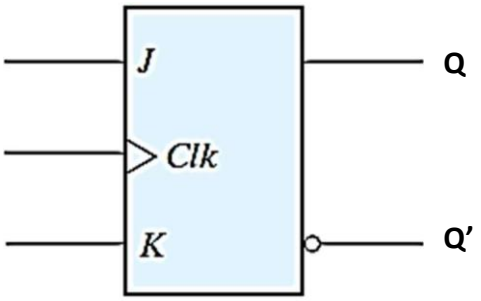
So, Class, tell me how to convert a D flip flop to J K flip flop…

J-K Flip Flop

Certified as Universal Flip Flop by ANSI and IEEE

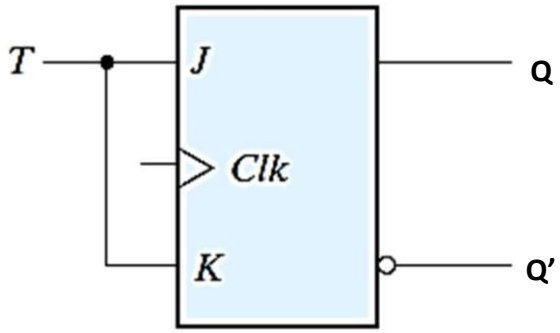
J	K	Q(t + 1)
0	0	Q(t)
0	1	0
1	0	1
1	1	$\overline{Q}(t)$

Characteristic table



Graphic symbol

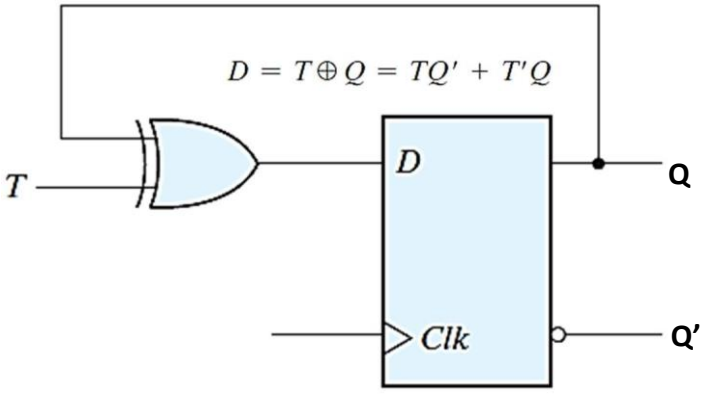
T Flip Flop from J-K Flip Flop



From *JK* flip-flop

So, Class, tell me how to convert a J K flip flop to T flip flop…

T Flip Flop from D Flip Flop

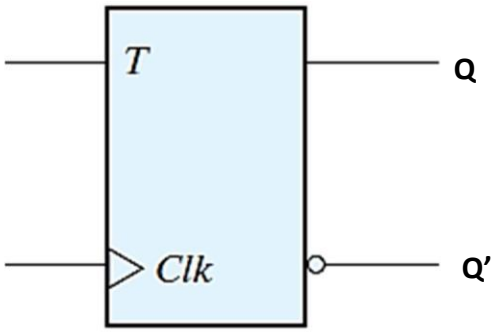


From *D* flip-flop

So, Class, tell me how to convert a D flip flop to T flip flop…

T	Q(t + 1)
0	Q(t)
1	$\overline{Q}(t)$

Characteristic table



Graphic symbol

Characteristic Tables of Flip Flops

***JK* Flip-Flop**

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

***D* Flip-Flop**

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

***T* Flip-Flop**

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

Characteristic Equations

D flip Flop · $Q(t+1) = D$

JK flip Flop · $Q(t+1) = JQ' + K'Q$

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

Excitation Tables

SR Flip-flop			
Q(t)	Q(t+1)	S	R
0	0	0	X
0	1	1	0
1	0	0	1
1	1	X	0

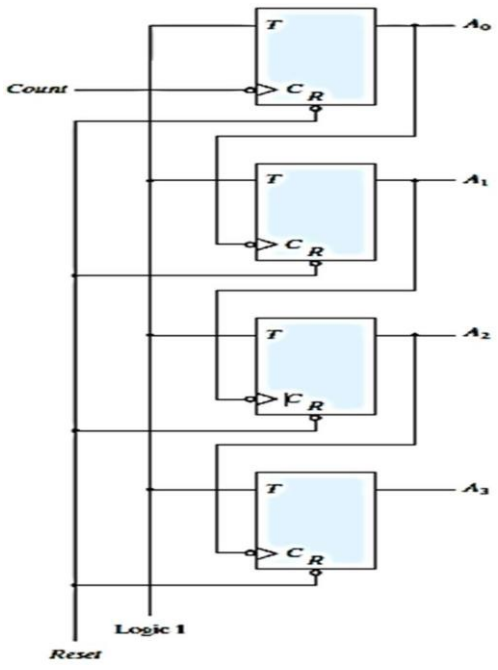
D Flip-flop		
Q(t)	Q(t+1)	DR
0	0	0
0	1	1
1	0	0
1	1	1

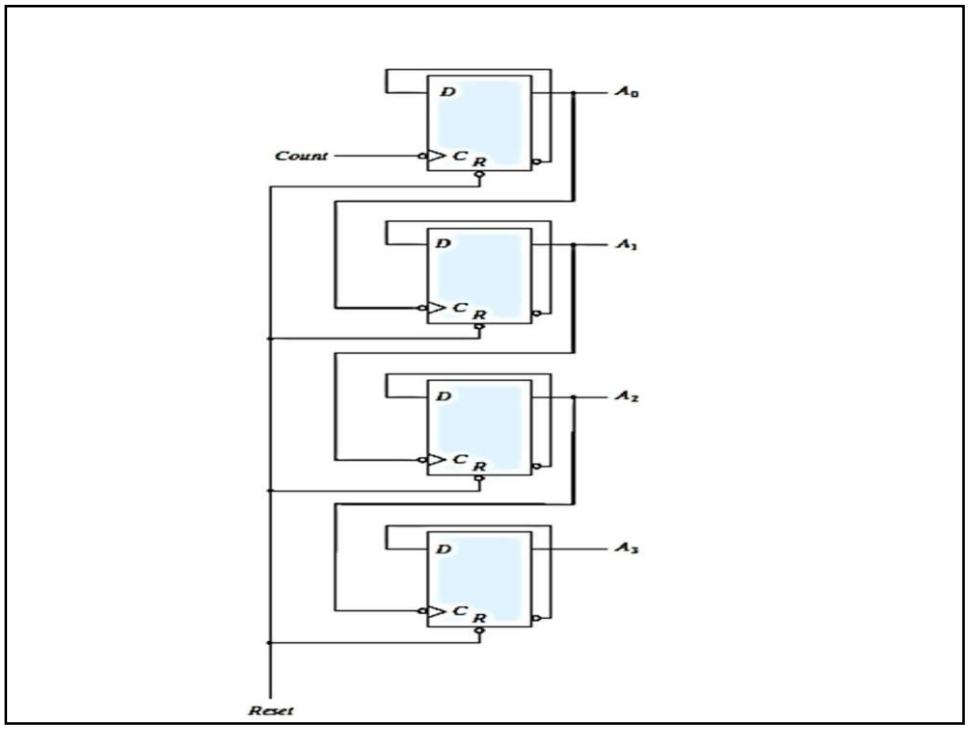
JK flip-flop			
Q(t)	Q(t+1)	J	K
0	0	0	x
0	1	1	x
1	0	x	1
1	1	x	0

T flip-flop		
Q(t)	Q(t+1)	DR
0	0	0
0	1	1
1	0	1
1	1	0

Binary Ripple Counter

- A binary ripple counter consists of a series connection of T flip-flops, with output of each flip-flop connected to *Clk* input of next higher order flip-flop.
- The flip-flop holding the *l*sb receives incoming count pulses.
- Also can use *D* flip-flop with complement output connected to the *D* input.
- So, the *D* input is always the complement of the present state, and the next clock pulse will cause the flip-flop to complement.





Binary Count Sequence

A_3	A_2	A_1	A_0
0	0	0	0
0	0	0	1
0	0	1	0
0	0	1	1
0	1	0	0
0	1	0	1
0	1	1	0
0	1	1	1
1	0	0	0

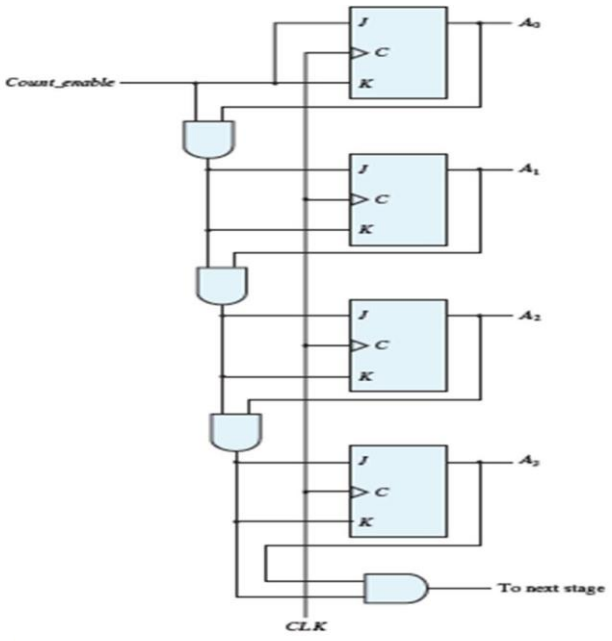
Operation of Ripple Counter

- The lsb, *A*0, is complemented with each count pulse input.
- Every time that *A*0 goes from 1 to 0, it complements *A*1.
- Every time that *A*1 goes from 1 to 0, it complements *A*2.
- Every time that *A*2 goes from 1 to 0, it complements *A*3, and so on for any other higher order bits of a ripple counter.
- For example, consider transition from count 0011 to 0100.
- *A*0 is complemented with the count pulse.
- Since *A*0 goes from 1 to 0, it triggers *A*1, complements it.
- As a result, *A*1 goes from 1 to 0, which in turn complements *A*2, changing it from 0 to 1.
- *A*2 does not trigger *A*3, as *A*2 produces a positive transition and the flip-flop responds only to negative transitions.

- Thus, the count from 0011 to 0100 is achieved by changing the bits one at a time, so the count goes from 0011 to 0010, then to 0000, and finally to 0100.
- The flip-flops change one at a time in succession, and the signal propagates through the counter in a ripple fashion from one stage to the next.

Synchronous counters

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



Four-bit synchronous binary counter

- In a synchronous binary counter, the flip-flop in the least significant position is complemented with every pulse.
- *A flip-flop in any other position is complemented when all the bits in the lower significant positions are equal to 1* .
- For example, if the present state of a four-bit counter is *A*3*A*2*A*1*A*0 = 0011, the next count is 0100.
- *A*0 is always complemented.
- *A*1 is complemented as the present state of *A*0 = 1.
- *A*2 is complemented as present state of *A*1*A*0 = 11.
- But, *A*3 is not complemented, as present state of *A*2*A*1*A*0 = 011, which is not an all-1's condition.

- Synchronous binary counters have a regular pattern and can be built with J K Flip Flops and Gates.
- *C* inputs of all flip-flops are connected to a common clock.
- The counter is enabled by *Count_enable*.
- If the enable input is 0, all *J* and *K* inputs are equal to 0 and the clock does not change the state of the counter.
- First stage, *A*0, has its *J=1* and *K* =1 if counter is enabled.
- The other *J* and *K* inputs are equal to 1 if all previous least significant stages are equal to 1 and the count is enabled.
- The chain of AND gates generates the required logic for the *J* and *K* inputs in each stage.

- In general, Counter can have any no: of stages, with each stage having an addnl flip-flop and an AND gate that gives an output of 1 if all previous flip-flop outputs are 1.

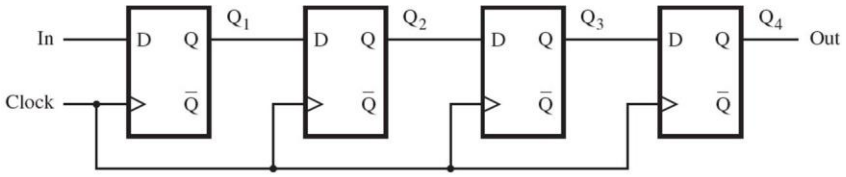
- Note that the flip-flops trigger on the positive edge of the clock.
- The polarity of the clock is not essential here, but it is with the ripple counter.
- The synchronous counter can be triggered with either the positive or the negative clock edge.

Registers

- A flip-flop stores one bit of information.
- When a set of *n* flip-flops is used to store *n* bits of information, such as an *n*-bit number, we refer to these flip-flops as a *register*.
- A common clock is used for each flip-flop in a register, and each flip-flop operates in a synchronous manner.
- A register that provides the ability to shift its contents is called a *shift register*.

Four-bit Shift Register

- A four-bit shift register that is used to shift its contents one bit position to the right.
- The data bits are loaded into the shift register in a serial manner using the *In* input.
- The contents of each flip-flop are transferred to the next flip-flop at each positive edge of the clock.



(a) Circuit

	In	Q_1	Q_2	Q_3	Q_4 = Out
t_0	1	0	0	0	0
t_1	0	1	0	0	0
t_2	1	0	1	0	0
t_3	1	1	0	1	0
t_4	1	1	1	0	1
t_5	0	1	1	1	0
t_6	0	0	1	1	1
t_7	0	0	0	1	1

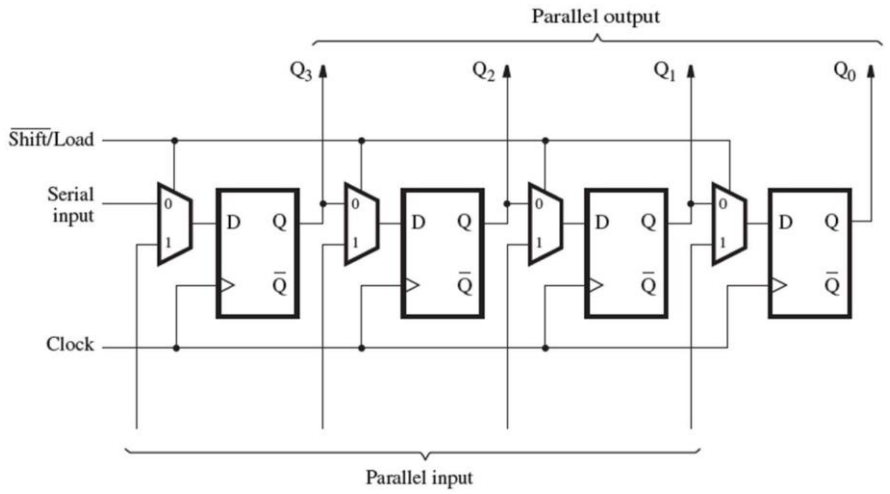
(b) A sample sequence

- A four-bit shift register is used to shift its contents one bit position to the right.
- The data bits are loaded into the shift register in a serial fashion using the *In* input.
- The contents of each flip-flop are transferred to the next flip-flop at each positive edge of the clock.
- Figure *b*, shows what happens when the signal values at *In* during eight consecutive clock cycles are 1, 0, 1, 1, 1, 0, 0, and 0, assuming that the initial state of all flip-flops is 0.

Parallel-Access Shift Register

- In computer systems it is needed to transfer *n*-bit data items.
- This may be done by transmitting all bits at once using *n* separate wires, in which case the transfer is performed in parallel.
- But it is also possible to transfer all bits using a single wire, by performing the transfer one bit at a time, in *n* consecutive clock cycles. This scheme is serial transfer.
- To transfer an *n*-bit data item serially, we can use a shift register that can be loaded with all *n* bits in parallel (in one clock cycle).
- Then during the next *n* clock cycles, the contents of the register can be shifted out for serial transfer.
- The reverse operation is also needed.
- If bits are received serially, then after *n* clock cycles the contents of the register can be accessed in parallel as an *n*-bit item.

- A four-bit shift register provides the parallel access.
- A 2-to-1 MUX on its *D* input allows each flip-flop to be connected to two different sources.
- One source is the preceding flip-flop, which is needed for the shift-register operation.
- The other source is the external i/p that corresponds to the bit that is to be loaded into the flip-flop as a part of the parallel-load operation.
- The control signal *Shift*/*Load* is used to select the mode of operation.
- If *Shift*/*Load* = 0, then circuit operates as a shift register.
- If *Shift*/*Load* = 1, then the parallel input data are loaded into the register.
- In both cases the action takes place on the positive edge of the clock



Parallel-access shift register.

- Label the flip-flops' outputs as Q3, . . . ,Q0 as shift registers are often used to hold binary numbers.
- The contents of the register can be accessed in parallel by observing the outputs of all flip-flops.
- The flip-flops can also be accessed serially, by observing the values of Q0 during consecutive clock cycles while the contents are being shifted.
- A circuit in which data can be loaded in series and then accessed in parallel is called a **series-to-parallel converter**.
- Similarly, the opposite type of circuit is a **parallel-to-series converter**.

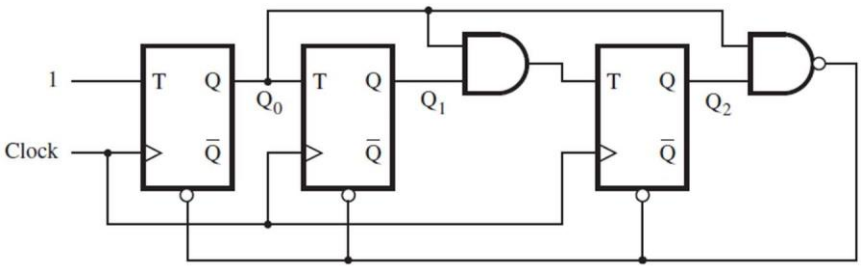
Mod N Counter

- Need to be able to clear, or *reset*, the contents of a counter before a counting operation.
- This can be done using the clear i/p of the individual flip-flops.
- Also can reset count to 0 during normal counting process.
- An *n*-bit up-counter functions as a modulo-2^n counter.
- Consider a counter that counts modulo some base that is not a power of 2.
- For ex, design a modulo-6 counter, for which the counting sequence is 0, 1, 2, 3, 4, 5, 0, 1, and so on.

Modulo-6 counter

- To recognize when the count reaches 5 and then reset the counter.
- An AND gate can be used to detect occurrence of the count of 5.
- Actually, it is sufficient to ascertain that Q2 = Q0 = 1, which is true only for 5 in the desired counting sequence.

Modulo-6 counter

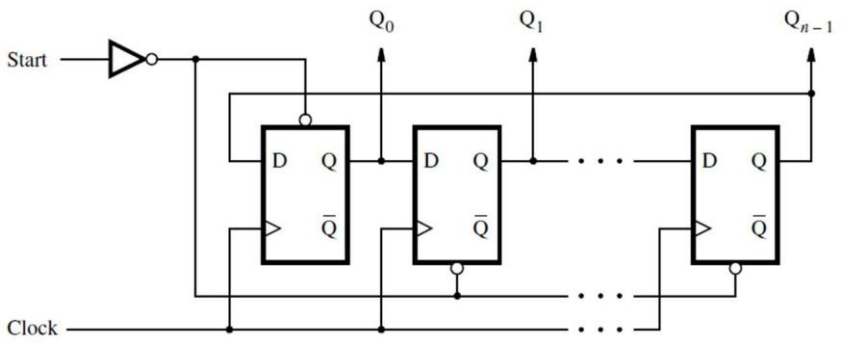


(a) Circuit

- Since the clear inputs are active when low, a NAND gate is used to detect the occurrence of the count of 5 and cause the clearing of all three flip-flops.
- As soon as the count reaches this value, the NAND gate triggers the resetting action.
- The flip-flops are cleared to 0 a short time after NAND gate has detected the count of 5.

Ring Counter

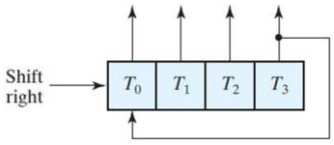
- Design a Counter in which each flip-flop reaches the state $Q_i = 1$ for exactly one count, for all other counts $Q_i = 0$.
- Q_i indicates directly occurrence of corresponding count.
- Such a circuit can be built from a simple shift register.
- The Q output of the last stage in the shift register is fed back as the input to the first stage, which creates a ring -like structure.
- If a single 1 is injected into the ring, this 1 will be shifted through the ring at successive clock cycles.
- For example, in a four-bit structure, the possible codes $Q_0Q_1Q_2Q_3$ will be 1000, 0100, 0010, and 0001.
- Such encoding, where there is a single 1 and the rest of the code variables are 0, is called a *One-Hot Code*.
- Such a circuit is referred to as a Ring Counter.



An n-bit Ring Counter

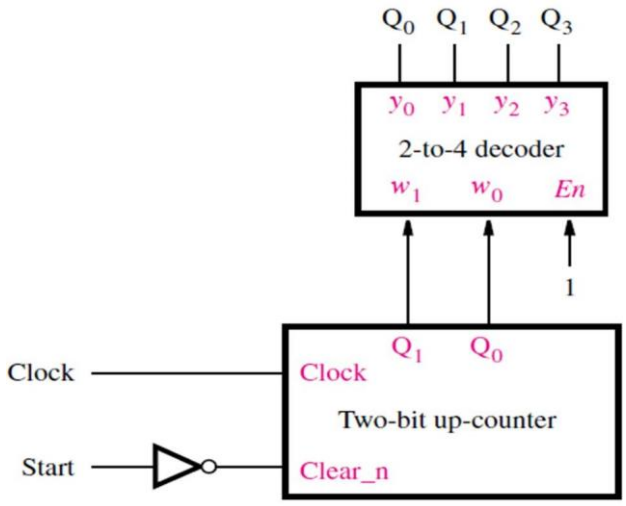
Operation of Ring Counter

- Its operation has to be initialized by injecting a 1 into the first stage.
- This is achieved by using the *Start* control signal, which presets the left-most flip-flop to 1 and clears the others to 0.
- Assume that all changes in value of the *Start* signal occur shortly after an active clock edge so that the flip-flop timing parameters are not violated.



4-bit Ring Counter

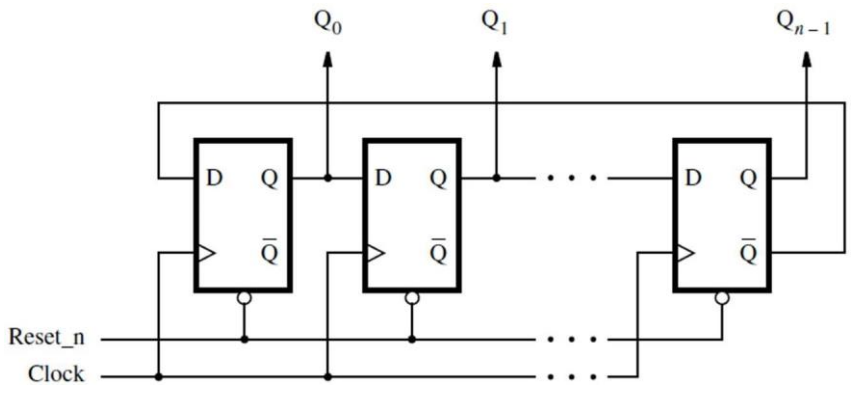
- A ring counter can be built with *'n' no: bits*.
- For *n* = 4, a ring counter can be constructed using a two-bit up-counter and a decoder.
- When *Start* is set to 1, the counter is reset to 00.
- After *Start* changes back to 0, the counter increments its value in the normal way.
- The 2-to-4 decoder, changes the counter output into a one-hot code.
- For the count values 00, 01, 10, 11, 00, and so on, the decoder produces $Q_0Q_1Q_2Q_3$ = 1000, 0100, 0010, 0001, 1000, and so on.
- This circuit structure can be used for larger ring counters, as long as the number of bits is a power of 2.



A four-bit ring counter

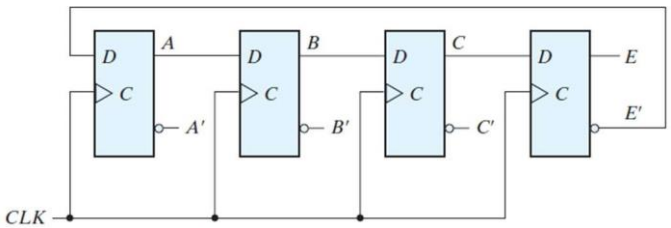
Johnson Counter

- An interesting variation of the Ring Counter results if, instead of Q output, Q' output of last stage is taken and feed it back to first stage.
- This circuit is known as a Johnson Counter, in honour of pioneering computer scientist and UCLA professor, Dr. Robert Royce Johnson.
- An *n*-bit counter of this type generates a counting sequence of length 2*n*.
- For example, a four-bit counter produces the sequence 0000, 1000, 1100, 1110, 1111, 0111, 0011, 0001, 0000, and so on.
- Note that in this sequence, only a single bit has a different value for two consecutive codes.
- To initialize the operation of the Johnson counter, it is necessary to reset all flip-flops.
- Observe that neither the Johnson nor the Ring Counter will generate the desired counting sequence if not initialized properly.



Johnson Counter

4-bit Johnson Counter



(a) Four-stage switch-tail ring counter

Sequence number	Flip-flop outputs A	B	C	E	AND gate required for output
1	0	0	0	0	$A'E'$
2	1	0	0	0	AB'
3	1	1	0	0	BC'
4	1	1	1	0	CE'
5	1	1	1	1	AE
6	0	1	1	1	$A'B$
7	0	0	1	1	$B'C$
8	0	0	0	1	$C'E$

(b) Count sequence and required decoding

Using Verilog Constructs for Storage Elements

- A simple way of specifying a storage element is by using **if-else** statement to describe the desired behavior responding to changes in the levels of data and clock inputs.
- Consider the **always** block,

```
always @(Control, B)
  if (Control)
    A= B;
```

- where *A* is a variable of **reg** type.
- This code specifies that the value of *A* should be made equal to value of *B* when *Control* = 1.
- But the statement does not indicate an action that should occur when *Control* = 0.
- In the absence of an assigned value, the Verilog compiler assumes that the value of *A* caused by the **if** statement must be maintained when *Control* is not equal to 1.
- This notion of *implied memory* is realized by instantiating a latch in the circuit.

```
module D_latch (D, Clk, Q);
   input D, Clk;
   output reg Q;

   always @(D, Clk)
      if (Clk)
         Q = D;

endmodule
```

Code for a gated D latch.

- The code defines a module named *D_latch*, which has inputs *D* and *Clk* and the output Q.
- The **if** clause defines that the Q output must take the value of *D* when *Clk* = 1.
- Since no **else** clause is given, a latch will be synthesized to maintain value of Q when *Clk*=0.
- Therefore, the code describes a gated D latch.
- The sensitivity list includes *Clk* and *D* because both of these signals can cause a change in the value of the Q output.

- An **always** construct is used to define a circuit that responds to changes in the signals that appear in the sensitivity list.
- The **always** blocks are sensitive to the *levels* of signals, it is also possible to specify that a response should take place only at a particular edge of a signal.
- The desired edge is specified by using the Verilog keywords **posedge** and **negedge**, which are used to implement edge-triggered circuits.

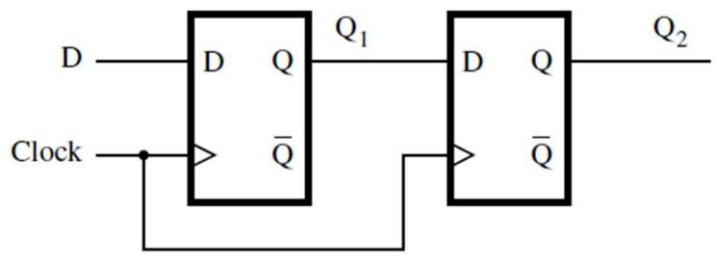
```
module flipflop (D, Clock, Q);
   input D, Clock;
   output reg Q;

   always @(posedge Clock)
      Q = D;

endmodule
```

Code for a D flip-flop.

- It defines a module named *flipflop*, which is a positive-edge-triggered D flip-flop.
- The sensitivity list contains only the clock signal because it is the only signal that can cause a change in the Q output.
- The keyword **posedge** specifies that a change may occur only on the positive edge of *Clock*.
- At this time output Q is set to value of input *D*.
- Since **posedge** appears in sensitivity list, Q will be implemented as the output of a flip-flop.

```
module example5_4 (D, Clock, Q1, Q2);
   input D, Clock;
   output reg Q1, Q2;

   always @(posedge Clock)
   begin
      Q1 <= D;
      Q2 <= Q1;
   end

endmodule
```

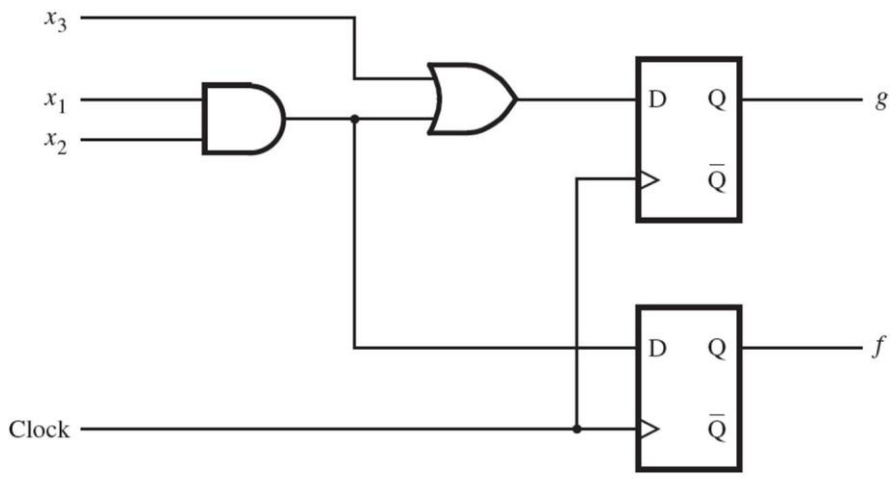
Code for two cascaded flip-flops.



Circuit defined

- Using non-blocking assignments.
- In the two statements
- Q1 <= D;
- Q2 <= Q1;
- The variables Q1 and Q2 have some value at the start of evaluating the **always** block, and
- Then they change to a new value concurrently at the end of the **always** block.
- This code generates a cascaded connection between flip-flops, which implements the shift register.

```
module example5_5 (x1, x2, x3, Clock, f, g);
    input x1, x2, x3, Clock;
    output reg f, g;

    always @(posedge Clock)
    begin
        f = x1 & x2;
        g = f | x3;
    end

endmodule
```



Circuit for Example 5.5.

ASYNCHRONOUS CLEAR

```
module  flipflop (D, Clock, Resetn, Q);
    input  D, Clock, Resetn;
    output  reg  Q;

    always @(negedge Resetn, posedge Clock)
        if (!Resetn)
            Q <= 0;
        else
            Q <= D;

endmodule
```

D flip-flop with asynchronous reset.

- This is a module that defines a D flip-flop with an asynchronous active-low reset (clear) input.
- When *Resetn*, the reset input, is equal to 0, the flip-flop's Q output is set to 0.
- Note that the sensitivity list specifies the negative edge of *Resetn* as an event trigger along with the positive edge of the clock.
- It's not possible to omit the keyword **negedge** because the sensitivity list cannot have both edge-triggered and level sensitive signals.

SYNCHRONOUS CLEAR

```
module flipflop (D, Clock, Resetn, Q);
   input D, Clock, Resetn;
   output reg Q;

   always @(posedge Clock)
      if (!Resetn)
         Q <= 0;
      else
         Q <= D;

endmodule
```

D flip-flop with synchronous reset.

- This shows a D flip-flop with a synchronous reset input.
- In this case the reset signal is acted upon only when a positive clock edge arrives.
- This code generates the circuit which has an AND gate connected to the flip-flop's D input.

AN N-BIT REGISTER

- Since registers of different sizes are often needed in logic circuits, it is useful to define a register module for which the number of flip-flops can be easily changed.
- The code for an *n*-bit register is given.
- The parameter *n* specifies the number of flip-flops in the register.
- By changing this parameter, the code can represent a register of any size.

```
module regn (D, Clock, Resetn, Q);
    parameter n = 16;
    input [n – 1:0] D;
    input Clock, Resetn;
    output reg [n – 1:0] Q;

    always @(negedge Resetn, posedge Clock)
        if (!Resetn)
            Q <= 0;
        else
            Q <= D;

endmodule
```

Code for an *n*-bit register with asynchronous clear.

A FOUR-BIT SHIFT REGISTER

- Verilog code for four-bit parallel-access shift register.
- Write hierarchical code that uses four sub circuits.
- Each sub circuit consists of a D flip-flop with a 2-to-1 multiplexer connected to the *D* input.
- The module named *muxdff*, which represents this sub circuit.
- The two data inputs are *D*0 and *D*1, and they are selected using the *Sel* input.
- The **if-else** statement specifies that on the positive clock edge if *Sel* = 0, then Q is assigned the value of *D*0; otherwise, Q is assigned the value of *D*1.
- An alternative way of defining the same circuit is using the conditional assignment statement specifies a 2-to-1 multiplexer with the output *D*, which is then connected to the flip-flop in the **always** block.

```
module muxdff (D0, D1, Sel, Clock, Q);
   input D0, D1, Sel, Clock;
   output reg Q;

   always @(posedge Clock)
      if (!Sel)
         Q <= D0;
      else
         Q <= D1;

endmodule
```

Code for a D flip-flop with a 2-to-1 multiplexer on the D input.

- The code defines the four-bit shift register.
- The module *Stage3* instantiates the leftmost flip-flop, which has the output Q3, and the module *Stage0* instantiates the right-most flip-flop, Q0.
- When *L* = 1, the register is loaded in parallel from the *R* input; and when *L* = 0, shifting takes place in the left to right direction.
- Serial data is shifted into the most significant
- bit, Q3, from the *w* input.

```
module muxdff (D0, D1, Sel, Clock, Q);
    input D0, D1, Sel, Clock;
    output reg Q;

    wire D;
    assign D = Sel ? D1 : D0;

    always @(posedge Clock)
        Q <= D;

endmodule
```

Alternative code for a D flip-flop with a 2-to-1 multiplexer on the D input.

```
module shift4 (R, L, w, Clock, Q);
    input [3:0] R;
    input L, w, Clock;
    output wire [3:0] Q;

    muxdff Stage3 (w, R[3], L, Clock, Q[3]);
    muxdff Stage2 (Q[3], R[2], L, Clock, Q[2]);
    muxdff Stage1 (Q[2], R[1], L, Clock, Q[1]);
    muxdff Stage0 (Q[1], R[0], L, Clock, Q[0]);

endr
```

Hierarchical code for a four-bit shift register.

```
module shift4 (R, L, w, Clock, Q);
    input [3:0] R;
    input L, w, Clock;
    output reg [3:0] Q;

    always @(posedge Clock)
        if (L)
            Q <= R;
        else
        begin
            Q[0] <= Q[1];
            Q[1] <= Q[2];
            Q[2] <= Q[3];
            Q[3] <= w;
        end

endmodule
```

Alternative code for a four-bit shift register.

```
module shiftn (R, L, w, Clock, Q);
    parameter n = 16;
    input [n-1:0] R;
    input L, w, Clock;
    output reg [n-1:0] Q;
    integer k;

    always @(posedge Clock)
        if (L)
            Q <= R;
        else
        begin
            for (k = 0; k < n-1; k = k+1)
                Q[k] <= Q[k+1];
            Q[n-1] <= w;
        end

endmodule
```

An *n*-bit shift register.

AN N-BIT SHIFT REGISTER

- This is the code that can be used to represent shift registers of any size.
- The parameter *n*, which has the default value 16, sets the number of flip-flops.
- First, *R* and Q are defined in terms of *n*.
- Second, the **else** clause that describes the shifting operation is generalized to work for any number of flip-flops by using a **for** loop.

UP-COUNTER

- This is a a four-bit up-counter with a reset input, *Resetn*, and an enable input, *E*.
- The outputs of the flip-flops in the counter are represented by the vector named Q.
- The **if** statement specifies an asynchronous reset of the counter if *Resetn* = 0.
- The **else if** clause specifies that if *E* = 1 the count is incremented on the positive clock edge.

```
module  upcount (Resetn, Clock, E, Q);
    input  Resetn, Clock, E;
    output  reg  [3:0] Q;

    always @(negedge Resetn, posedge Clock)
        if (!Resetn)
            Q < = 0;
        else if (E)
            Q < = Q + 1;

endmodule
```

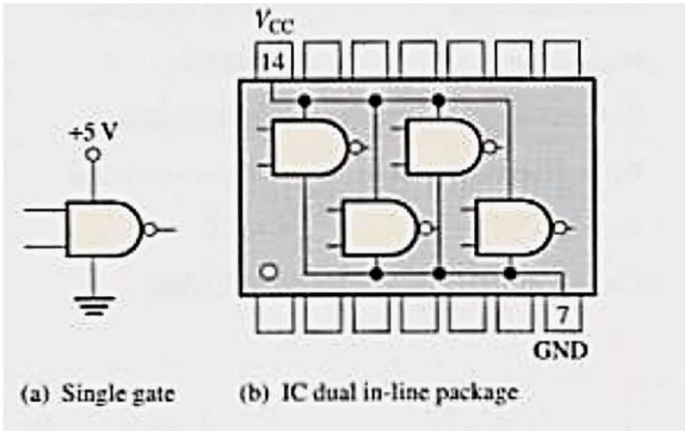
Code for a four-bit up-counter.

Logic Families and its Characteristics

ECT 203 Module V

BASIC OPERATIONAL CHARACTERISTICS AND PARAMETERS

- Operational properties include voltage levels, noise immunity, power dissipation, fan-out, and propagation delay time.
- **DC Supply Voltage**
- The nominal value of the dc supply voltage for TTL(Transistor-Transistor Logic) devices is +5 V.
- TTL is also designated T^2L.
- CMOS (Complementary Metal-Oxide Semiconductor) devices are available in different supply voltage categories: +5 V, 3.3 V, 2.5 V and 1.2 V.
- Although omitted from logic diagrams for simplicity, the dc supply voltage is connected to the Vn pin of an IC package, and ground is connected to the GND pin.
- Both voltage and ground are distributed internally to all elements within the package, as illustrated as shown below for a 14-pin package.



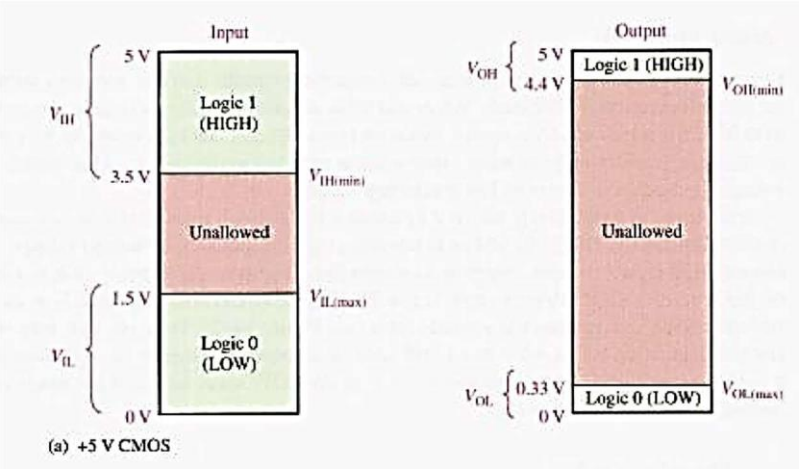
Example of V_{cc} and ground in an IC package. Other pin connections are omitted for simplicity.

CMOS Logic Levels

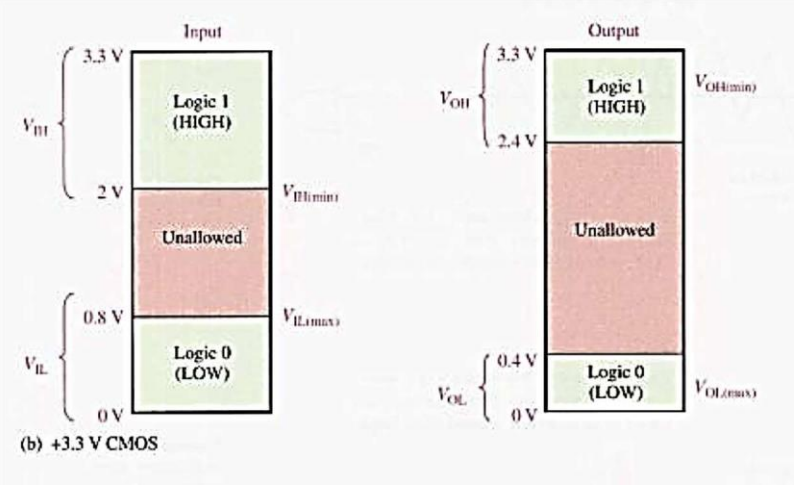
- There are 4 different logic-level specifications: V_{IL} V_{IH} V_{OL} and V_{OH}.
- For CMOS circuits, the ranges of input voltages (V_{IL}) that can represent a valid LOW (logic 0) are from 0V to 1 .5 V for the 5 V logic and 0 V to 0.8 V for the 3.3 V logic.
- The ranges of input voltages (V_{IH}) that can represent a valid HIGH (Logic 1) are from 3.5 V to 5V for the 5V logic and 2 V to 3.3V for the 3.3V logic.
- The ranges of values from 1.5V to 3.5 V for 5V logic and 0.8 V to 2 V for 3.3 V logic are regions of unpredictable performance, and values in these ranges are unallowed.
- When an input voltage is in one of these ranges, it can be interpreted as either a HIGH or a LOW by the logic circuit.
- Therefore, CMOS gates cannot be operated reliably when the input voltages are in these unallowed ranges.

CMOS Output Voltages

- The ranges of CMOS output voltages V_{OL} and V_{OH} for both 5V and 3.3V logic are shown.
- Notice that the minimum HIGH output voltage $V_{OH(min)}$ is greater than the minimum HIGH input voltage $V_{IH(min)}$.
- Also, notice that the maximum LOW output voltage $V_{OL(max)}$, is less than the maximum LOW input voltage $V_{IL(max)}$.



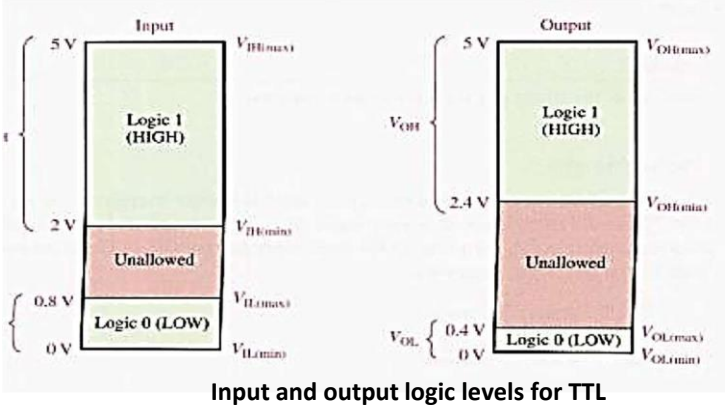
Input and output logic levels for CMOS operating at +5V



Input and output logic levels for CMOS operating at +3.3 V

TTL Logic Levels

- The input and output logic levels for TTL are in terms of 4 different logic level specifications.
- They are: V_{IL}, V_{IH}, V_{OL}, and V_{OH} .



Input and output logic levels for TTL

Noise Immunity

- Noise is unwanted voltage that is induced in electrical circuits and can present a threat to the proper operation of the circuit.
- Wires and other conductors within a system can pick up stray high-frequency electromagnetic radiation from adjacent conductors in which currents are changing rapidly or from many other sources external to the system.
- Also, power-line voltage fluctuation is a form of low-frequency noise.

- In order not to be adversely affected by noise, a logic circuit must have a certain amount of noise immunity.
- This is the ability to tolerate a certain amount of unwanted voltage fluctuation on its inputs without changing its output state.
- For example, if noise voltage causes the input of a 5 V CMOS gate to drop below 3.5V in the HIGH state, the input is in the unallowed region and operation is unpredictable.
- Thus, the gate may interpret the fluctuation below 3.5 V as a LOW level.
- Similarly, if noise causes a gate input to go above 1.5 V in the LOW state, an uncertain condition is created.

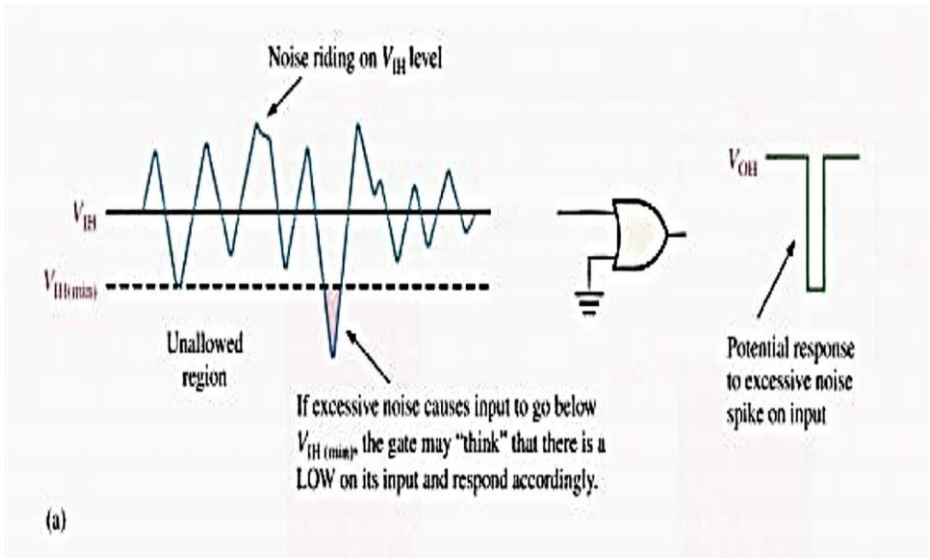


Illustration of the effects of input noise on gate operation.

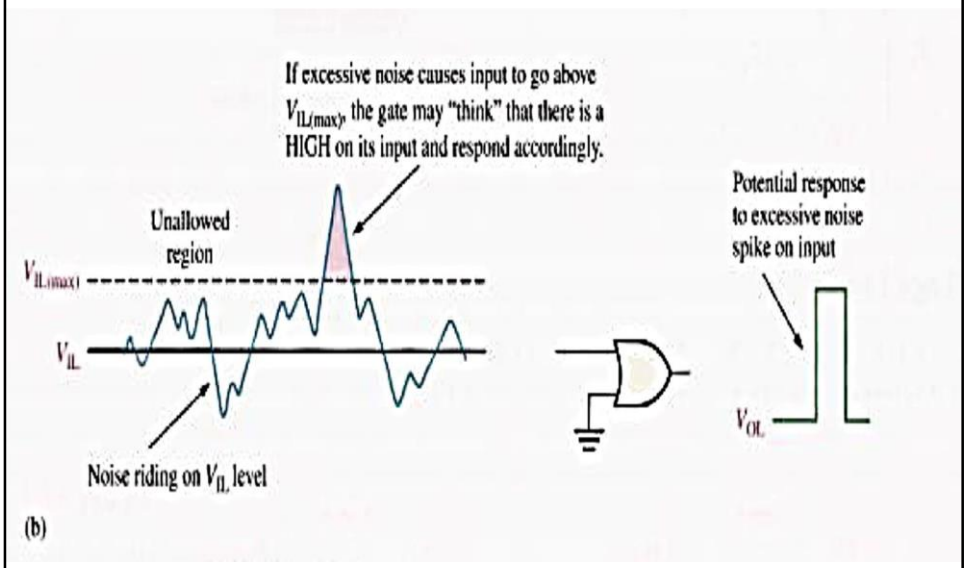


Illustration of the effects of input noise on gate operation.

Noise Margin

- A measure of a circuit's noise immunity is called Noise Margin, expressed in Volts.
- There are two values of noise margin specified for a given logic circuit: HIGH-level noise margin (V_{NH}) and LOW-level noise margin (V_{NL}).
- These parameters are defined by the following equations:

$$V_{NH} = V_{OH(min)} - V_{IH(min)} \quad \text{.....(1)}$$

$$V_{NL} = V_{IL(max)} - V_{OL(max)} \quad \text{.....(2)}$$

- The noise margin is expressed as a percentage of Vcc.
- From equations. V_{NH} is the difference between the lowest possible HIGH output from a driving gate ($V_{OH(min)}$) and the lowest possible HIGH input that the load gate can tolerate ($V_{IH(min)}$).
- Noise margin, V_{NL} is the difference between the maximum possible LOW input that a gate can tolerate ($V_{IL(max)}$) and the maximum possible LOW output of the driving gate ($V_{OL(max)}$) .

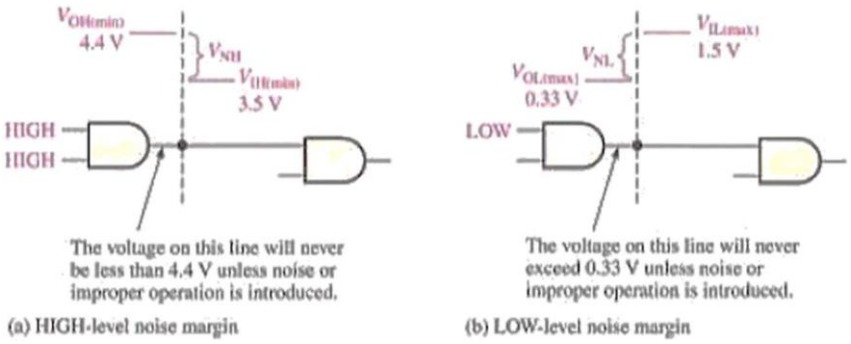


Illustration of Noise Margin, values are for 5 V CMOS, but it applies to any logic family

EXAMPLE -1

Determine the High-level and LOW-level noise margins for CMOS and for TTL using their logic level voltage ranges.

Sol

For 5 V CMOS,

$$V_{IH(min)} = 3.5\ V$$
$$V_{IL(max)} = 1.5\ V$$
$$V_{OH(min)} = 4.4\ V$$
$$V_{OL(max)} = 0.33\ V$$
$$V_{NH} = V_{OH(min)} - V_{IH(min)} = 4.4\ V - 3.5\ V = \mathbf{0.9\ V}$$
$$V_{NL} = V_{IL(max)} - V_{OL(max)} = 1.5\ V - 0.33\ V = \mathbf{1.17\ V}$$

For TTL,

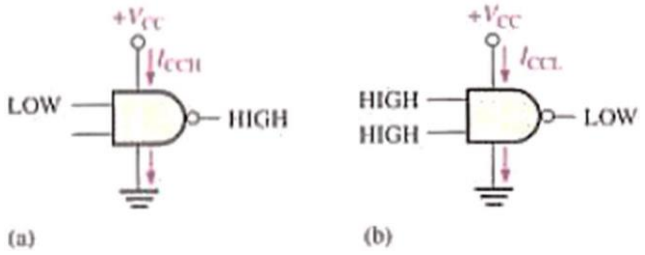
$$V_{IH(min)} = 2\ V$$
$$V_{IL(max)} = 0.8\ V$$
$$V_{OH(min)} = 2.4\ V$$
$$V_{OL(max)} = 0.4\ V$$
$$V_{NH} = V_{OH(min)} - V_{IH(min)} = 2.4\ V - 2\ V = \mathbf{0.4\ V}$$
$$V_{NL} = V_{IL(max)} - V_{OL(max)} = 0.8\ V - 0.4\ V = \mathbf{0.4\ V}$$

A TTL gate is immune to up to 0.4 V of noise for both the HIGH and LOW input states.

Based on the preceding noise margin calculations, which family of devices, 5 V CMOS or TTL, should be used in a high-noise environment?

Power Dissipation

- A logic gate draws current from the dc supply voltage source.
- When the gate is in the HIGH output state, an amount of current designated by I_{CCH} is drawn.
- And in the LOW output state, a different amount of current I_{CCL} is drawn.



Currents drawn from the dc source

As an example, if I_{CCH} is specified as 1.5mA when V_{cc} is 5 V and if the gate is in a static (non changing) HIGH output state, the power dissipation P_D of the gate is ,

$$P_D = V_{CC}I_{CCH} = (5\text{ V})(1.5\text{ mA}) = 7.5\text{ mW}$$

When a gate is pulsed, its output switches back and forth between HIGH and LOW and the amount of supply current varies between I_{CCH} and I_{CCL} .

The average power dissipation depends on the duty cycle and is usually specified for a duty cycle of 50%.

When the duty cycle is 50%. the output is HIGH half the time and LOW the other half.

The average supply current is,

$$I_{CC} = \frac{I_{CCH} + I_{CCL}}{2} \quad \text{.....(3)}$$

The average power dissipation is,

$$P_D = V_{CC}I_{CC} \quad \text{.....(4)}$$

EXAMPLE -2

A certain gate draws 2 µA when its output is HIGH and 3.6 µA when its output is LOW. What is its average power dissipation if V_{cc} is 5 V and the gate is operated on a 50% duty cycle?

Sol

The average I_{CC} is

$$I_{CC} = \frac{I_{CCH} + I_{CCL}}{2} = \frac{2.0\,\mu A + 3.6\,\mu A}{2} = 2.8\,\mu A$$

The average power dissipation is

$$P_D = V_{CC} I_{CC} = (5\text{ V})(2.8\,\mu A) = \mathbf{14\,\mu W}$$

Homework!

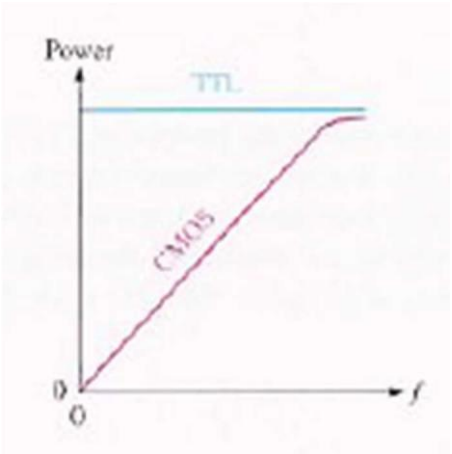
1. A certain IC gate has an I_{CCH} = 1.5µA and I_{CCL} = 2.8 µA. Determine the average Power dissipation for 50% duty cycle operation if V_{cc} is 5 V.

2.

Based on the preceding noise margin calculations, which family of devices, 5 V CMOS or TTL, should be used in a high-noise environment?

Power dissipation TTL vs CMOS

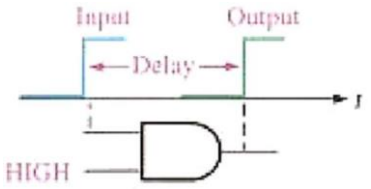
- Power dissipation in a TTL circuit is essentially constant over its range of operating frequencies.
- Power dissipation in CMOS, however, is frequency dependent.
- It is extremely low under static (dc) conditions and increases as the frequency increases.



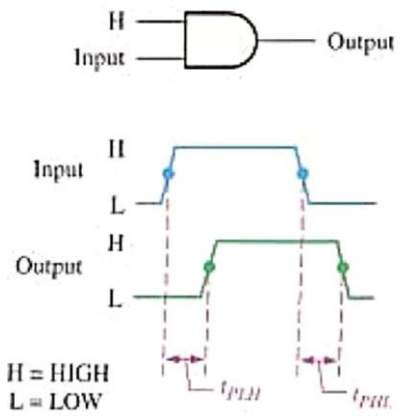
Power-vs-frequency curve for TTL and CMOS.

Propagation Delay Time

- When a signal passes (propagates) through a logic circuit, it always experiences a time delay.
- A change in the output level always occurs a short time, called the propagation delay time, later than the change in the input level that caused it.



- There are two propagation delay times specified for logic gates:
- t_{PHL}: The time between a designated point on the input pulse and the corresponding point on the output pulse when the output is changing from HIGH to LOW.
- t_{PLH} : The time between a designated point on the input pulse and the corresponding point on the output pulse when the output is changing from LOW to HIGH.
- These propagation delay times are illustrated below with the 50% points on the pulse edges used as references.



Propagation delay times.

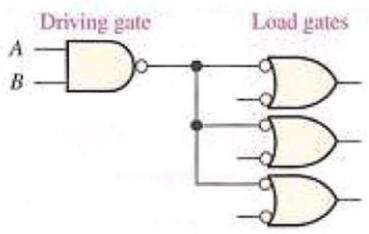
- The propagation delay time of a gate limits the frequency at which it can be operated.
- The greater the propagation delay time, the lower the maximum frequency.
- Thus, a higher speed circuit is one that has a smaller propagation delay time.
- For example, a gate with a delay of 3 ns is faster than one with a 10 ns delay.

Speed-Power Product

- The speed-power product provides a basis for the comparison of logic circuits when both propagation delay time and power dissipation are important considerations in the selection of the type of logic to be used in a certain application.
- The lower the speed-power product, the better. The unit of speed-power product is the pico-joule (pJ).
- For ex, HCMOS has a speed-power product of 1.2 pJ at I00 kHz while LS TTL has a value of 22 pJ.

Loading and Fan-Out

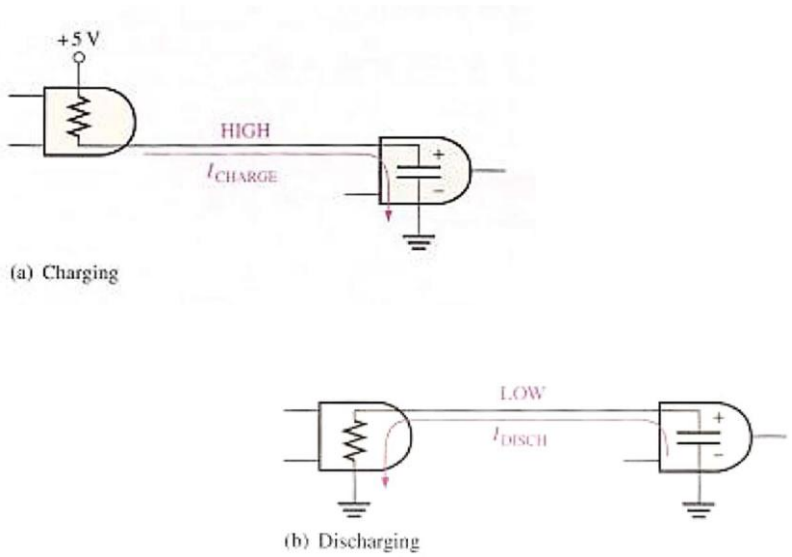
- When the output of a logic gate is connected to one or more inputs of other gates, a load on the driving gate is created, as shown.
- There is a limit to the number of load gate inputs that a given gate can drive.
- This limit is called the fan-out of the gate.



Loading a gate output with gate inputs.

CMOS Loading

- Loading in CMOS differs from that in TTL because the type of transistors used in CMOS logic present a predominantly capacitive load to driving gate.
- In this case, the limitations are the charging and discharging times associated with the output resistance of the driving gate and the input capacitance of the load gates.
- When the output of the driving gate is HIGH, the input capacitance of the load gate is charging through the output resistance of the driving gate.
- When the output of the driving gate is LOW, the capacitance is discharging.

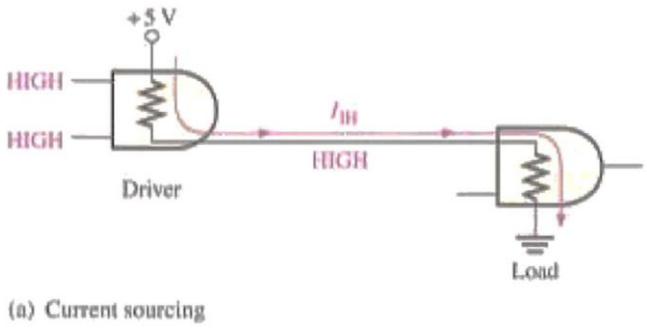


Capacitive loading of a CMOS gate

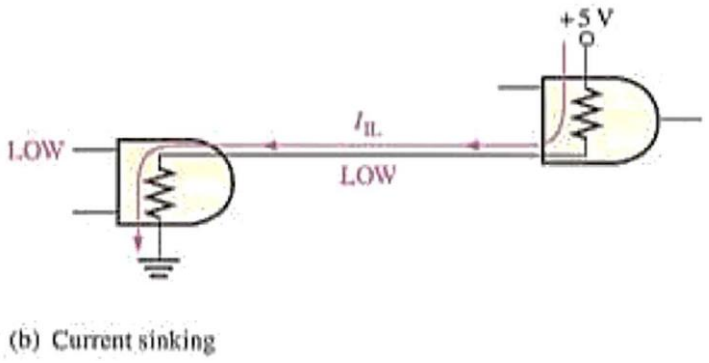
- When more load gate inputs are added to the driving gate output, the total capacitance increases because the input capacitances effectively appear in parallel.
- This increase in capacitance increases the charging and discharging limes, thus reducing the maximum frequency at which the gate can be operated.
- Therefore, the fan-out of a CMOS gate depends on the frequency of operation.
- The fewer the load gate inputs, the greater the maximum frequency.

TTL Loading

- A TTL driving gate sources current to a load gate input in the HIGH state(I_{IH}) and sinks current from the load gate in LOW state (I_{IL}).
- Current sourcing and current sinking are illustrated in simplified form.
- Where the resistors represent the internal input and output resistance of the gate for the two conditions.



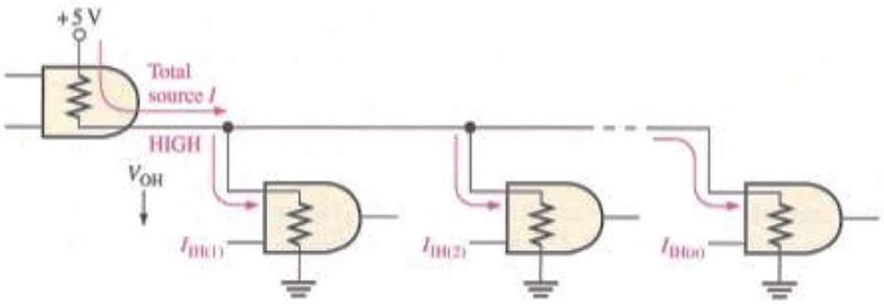
Basic illustration of current sourcing in logic gates



(b) Current sinking

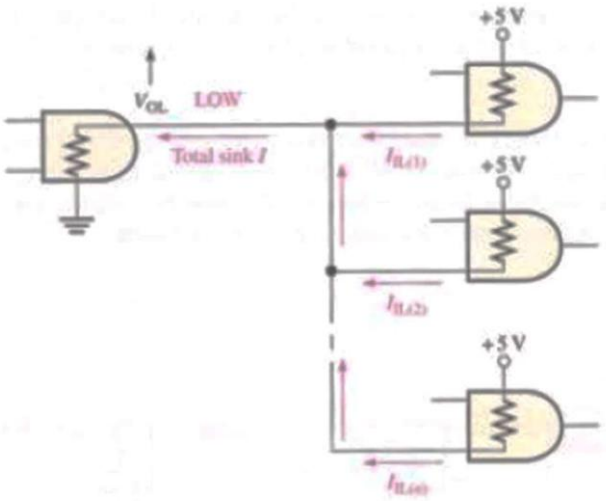
Basic illustration of current sinking in logic gates

- As more load gates are connected to the driving gate, the loading on the driving gate increases.
- The total source current increases with each load gate input that is added, as illustrated.
- As this current increases, the internal voltage drop of the driving gate increases, causing the output V_{OH} to decrease.
- If an excessive number of load gate inputs are connected, V_{OH} drops below $V_{OH(min)}$ and the HIGH-level noise margin is reduced thus compromising the circuit operation.
- Also, as the total source current increases, the power dissipation of the driving gate increases.



HIGH-state TTL loading

- The fan-out is the max no: of load gate inputs that can be connected without adversely affecting the specified operational characteristics of the gate.
- For ex, low power Schottky (LS) TTL has a fan-out of 20 unit loads.
- One input of the same logic family as the driving gate is called a unit load.
- The total sink current also increases with each load gate input that is added, as shown.
- As this current increases, the internal voltage drop of the driving gale increases. causing V_{OL} increase.
- If an excessive number of loads are added V_{OL} exceeds $V_{OL(max)}$ and the LOW-level noise margin is reduced.
- In TTL the current-sinking capability (LOW output state) is the limiting factor in determining the fan-out.



Low Stage TTL Loading

Check your Understanding!

1. Define V_{IH}, V_{IL}, V_{OH}, and V_{OL}.

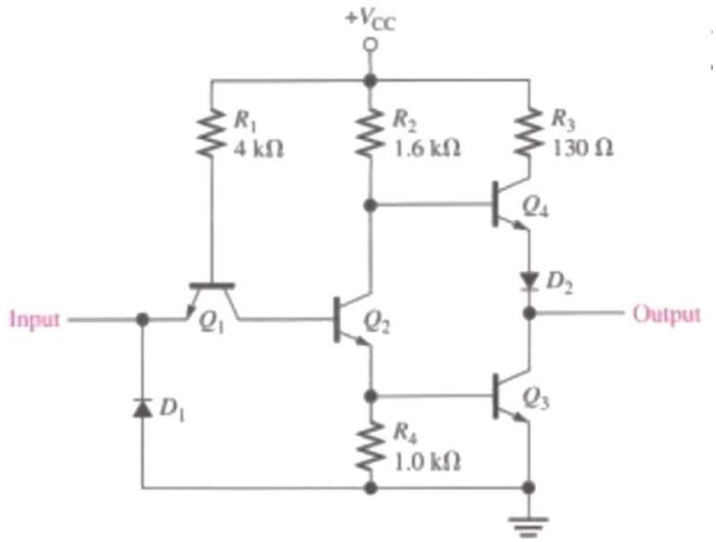
2. Is it better to have a lower value of noise margin or a higher value?

3. Gate *A* has a greater propagation delay time than gate *B*. Which gate can operate at a higher frequency?

4. How does excessive loading affect the noise margin of a gate?

TTL Inverter

- The logic function of an inverter or any type of gate is always the same, regardless of the type of circuit technology that is used.
- A standard TTL circuit for an inverter is studied.
- In this diagram Q_1 is the input coupling transistor, and D, is the input clamp diode.
- Transistor Q_2 is called a phase splitter, and the combination of Q_3 and Q_4 forms the output circuit often referred to as a totem-pole arrangement.



A standard TTL inverter circuit

Operation

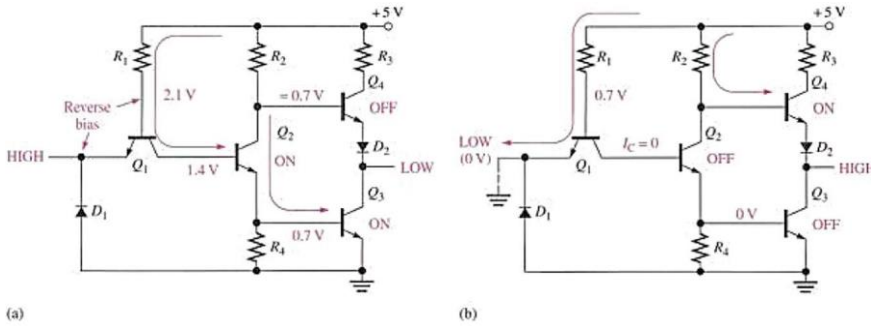
- When the input is a HIGH, the base-emitter junction of Q_1 is reverse biased, and the base-collector junction is forward biased.
- This condition permits current through R_1 and the base-collector junction of Q_1, into the base of Q_2, thus driving Q_2 into saturation.
- As a result Q_3 is turned on by Q_2 and its collector voltage, which is the output, is near ground potential.
- We therefore have a LOW output for a HIGH input.
- At the same time, the col lector of Q_2 is at a sufficiently low voltage level to keep Q_4 off.
- When the input is LOW the base-emitter junction of Q_1 is forward biased, and the base collector junction is reverse biased.
- There is current through R_1 and the base-emitter junction of Q1 to the LOW input.

Operation

- A LOW provides a path to ground for the current.
- There is no current into base of Q_2 so it is off.
- The collector of Q_2 is HIGH, thus turning Q_4 ON.
- A saturated Q4 provides a low-resistance path from V_{cc} to the output.
- There is a HIGH on the output for a LOW on the input.
- At the same time, the emitter of Q_2 is at ground potential, keeping Q_3 off.

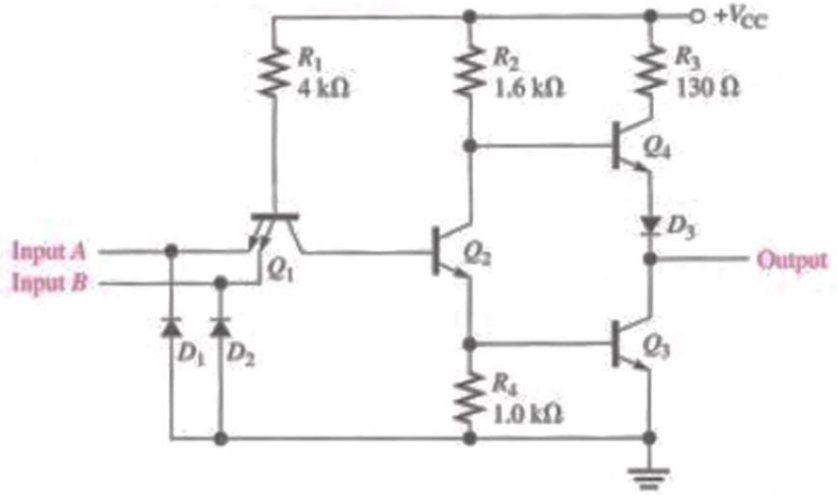
- Diode D_1 in the TTL circuit prevents negative spikes of voltage on the input from damaging Q1.
- Diode D_2 ensures that Q4, will turn off when Q2 is on (HIGH input).
- In this condition the collector voltage of Q_2 is equal to the base-to-emitter voltage V_{BE} of Q_3 plus the collector-to-emitter voltage V_{CE} of Q_2.
- Diode D_2 provides an additional V_{BE} equivalent drop in series with the base-emitter junction of Q_4 to ensure its turn-off when Q_2 is on.

- The operation of the TTL inverter for the two input states is illustrated in diagram.
- In the circuit in part (a), the base of Q_1 is 2. 1 V above ground, so Q_2 and Q_3 are ON.
- In the circuit in part (b) the base of Q_1 is about 0.7 V above ground—not enough to turn Q_2 and Q_3 on.

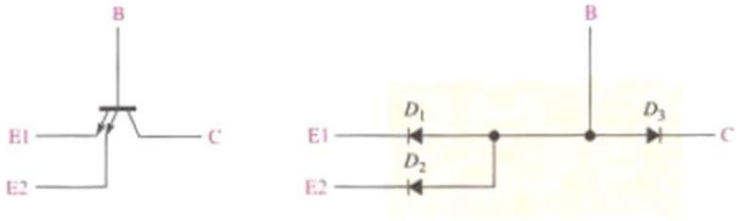


Operation of a TTL inverter

TTL NAND Gate



A 2-input TTL NAND gate is shown.
Basically, it is the same as the inverter circuit except for the additional input emitter of Q_1.
In TTL technology multiple-emitter transistors are used for the input devices.
These multiple-emitter transistors can be compared to the diode arrangement, as shown.



Diode equivalent of a TTL multiple-emitter transistor.

Q_1 is replaced by the diode arrangement.
A LOW on either input A or input H forward-biases the respective diode and reverse-biases D_3 (Q_1 base collector junction).

This action keeps Q_2 off and results in a HIGH output.
A LOW on both inputs will do the same thing.
A HIGH on both inputs reverse-biases both input diodes and forward-biases D_3 (Q_1 base collector junction).

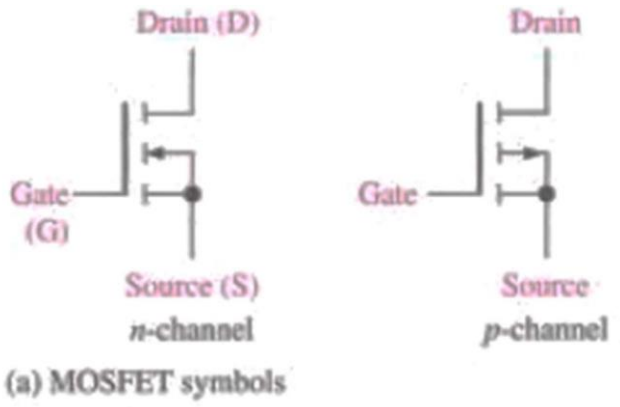
This action turns Q_2 on and results in a LOW output.
The operation is that of the NAND function.
The output is LOW only if all inputs are HIGH.

CMOS CIRCUITS -The MOSFET

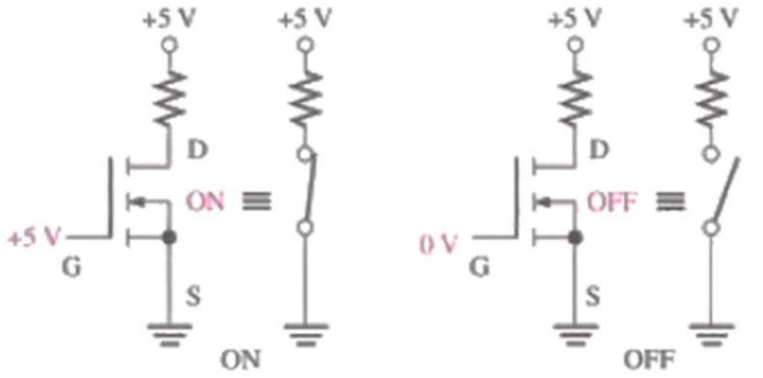
- Metal-oxide semiconductor field-effect transistors (MOSFETs) are the active switching elements in CMOS circuits.
- These devices differ greatly in construction and internal operation from bipolar junction transistors used in TTL circuits, but the switching action is basically the same.
- They function ideally as open or closed switches, depending on the input.

The MOSFET

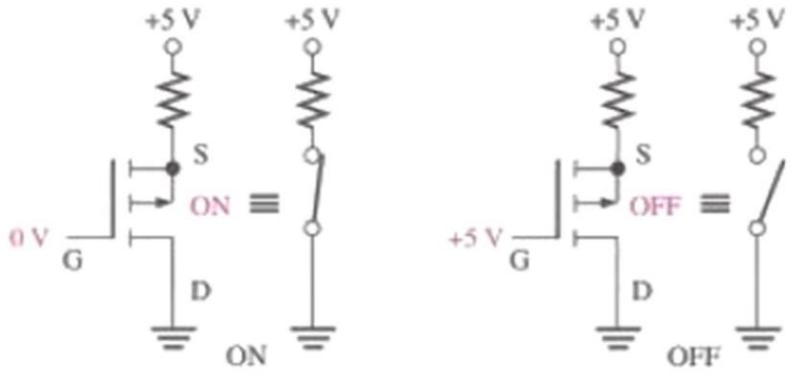
- The three terminals of a MOSFET are gate, drain, and source.
- When the gate voltage of an n-channel MOSFET is more positive than the source, the MOSFET is ON (Saturation), and there is ideally, a closed switch between the drain and the source.
- When the gate-to-source voltage is zero, the MOSFET is OFF (cutoff), and there is ideally, an open switch between the drain and the source.



(a) MOSFET symbols



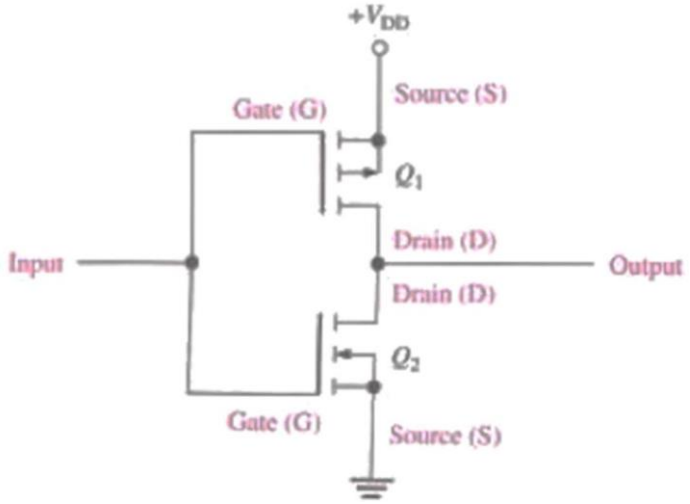
(b) *n*-channel switch



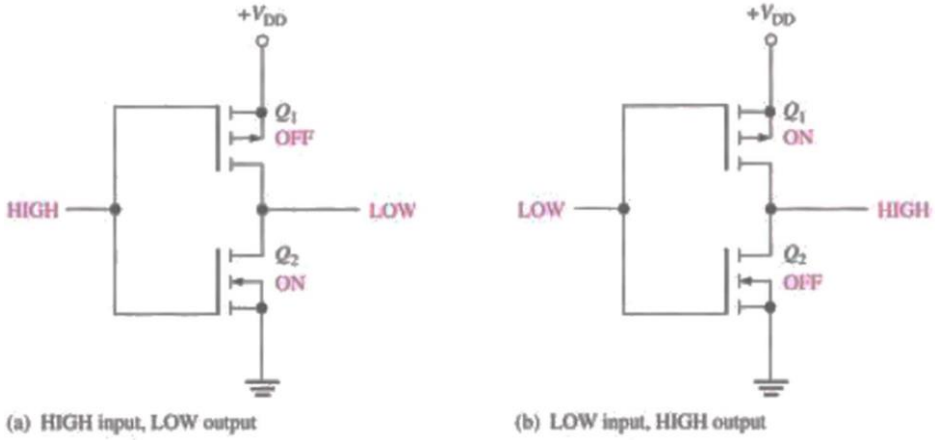
(c) *p*-channel switch

Simplified MOSFET symbol.

CMOS Inverter

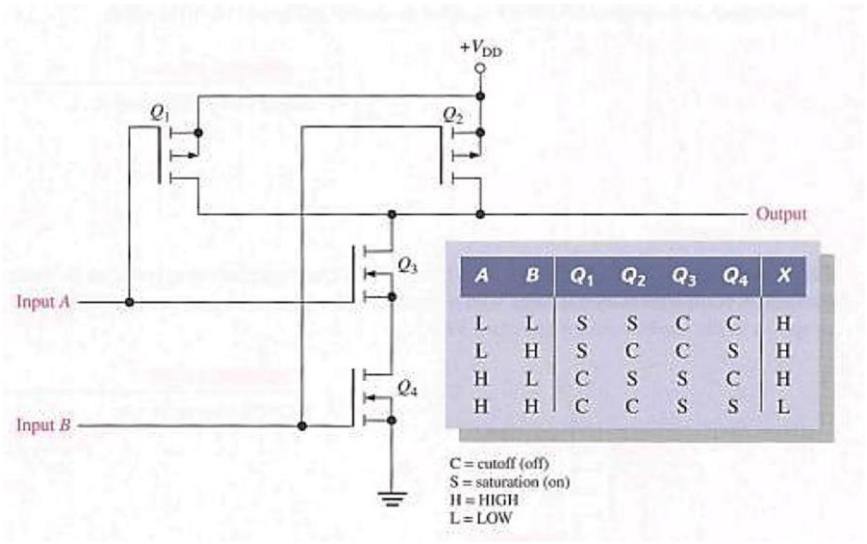


- Complementary MOS (CMOS) logic uses the MOSFET in complementary pairs as its basic element.
- A complementary pair uses both n -channel and p-channel enhancement MOSFETs.
- When a HIGH is applied lo the input, the p-channel MOSFET Q1 is off and the n-channel MOSFET Q2 is on.
- This condition connects the output to ground through the on resistance of Q2, resulting in a LOW output.
- When a LOW is applied to the input, Q1 is on and Q2 is off.
- This condition connects the output to +VDD (dc supply voltage) through the on resistance of Q1 resulting in a HIGH output.



Operation of a CMOS inverter

CMOS NAND Gate



A	B	Q_1	Q_2	Q_3	Q_4	X
L	L	S	S	C	C	H
L	H	S	C	C	S	H
H	L	C	S	S	C	H
H	H	C	C	S	S	L

CMOS NAND

- The operation of a CMOS NAND gate is as follows:

When both inputs are LOW Q1 and Q2 are on and Q3 and Q4 are off.

The output is pulled HIGH through the on resistance of Q1 and Q2 in parallel.

When input A is LOW and input B is HIGH Q1 and Q4 are on and Q2 and Q3, are off.
The output is pulled HIGH through the low on resistance of Q1.

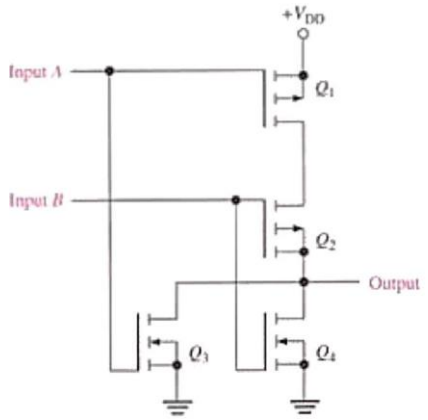
When input A is HIGH and input B is LOW Q1and Q4 are off and Q2 and Q3 are on.

The output is pulled HIGH through the low on resistance of Q2.

Finally, when both inputs are HIGH Q1 and Q2 are off and Q3 and Q4 are on.

In this case, the output is pulled LOW through the on resistance of Q3 and Q4 in series to ground.

CMOS NOR Gate



A	B	Q_1	Q_2	Q_3	Q_4	X
L	L	S	S	C	C	H
L	H	S	C	C	S	L
H	L	C	S	S	C	L
H	H	C	C	S	S	L

C = cutoff (off)
S = saturation (on)
H = HIGH
L = LOW

The operation of a CMOS NOR gate is as follows:

When both inputs are LOW Q1 and Q2 are on and Q3 and Q4 are off.

As a result, the output is pulled HIGH through the on resistance of Q1 and Q2 in series.

When input A is LOW and input B is HIGH Q1 and Q4 are on and Q2 and Q3 are off.

The output is pulled LOW through the low on resistance of Q4 to ground.

When input A is HIGH and input B is LOW Q1 and Q4 are off and Q2 and Q3 are on.

The output is pulled LOW through the on resistance of Q3 to ground.

When both inputs are HIGH Q1 and Q4 are off , and Q2 and Q3 are on.

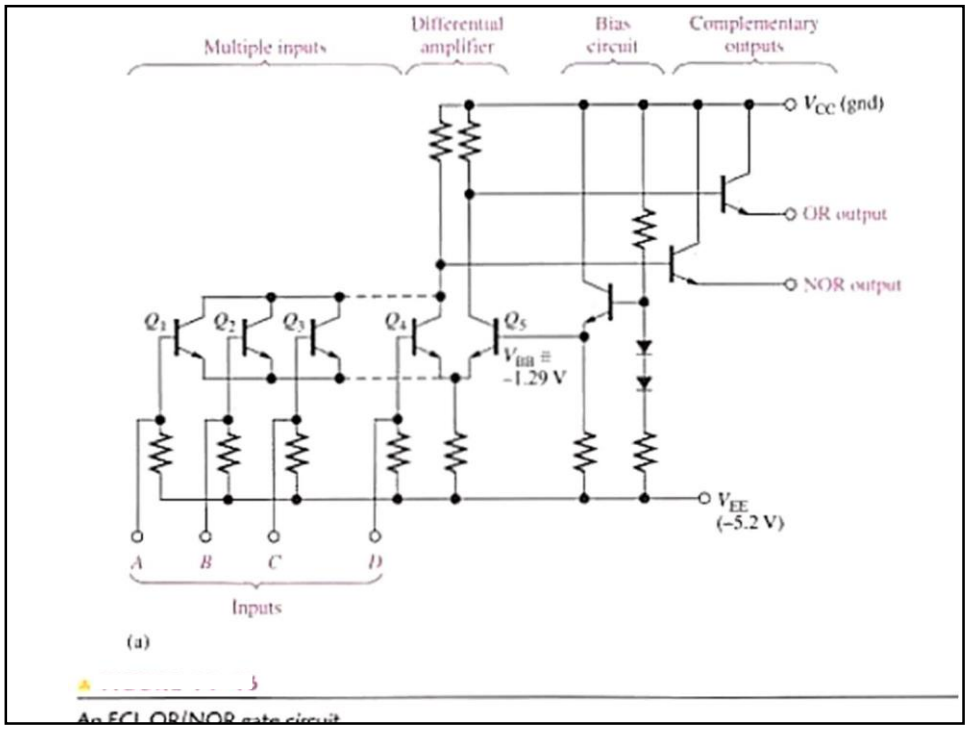
The output is pulled LOW through the on resistance of Q3 and Q4 in parallel lo ground.

EMITTER-COUPLED LOGIC (ECL) CIRCUITS

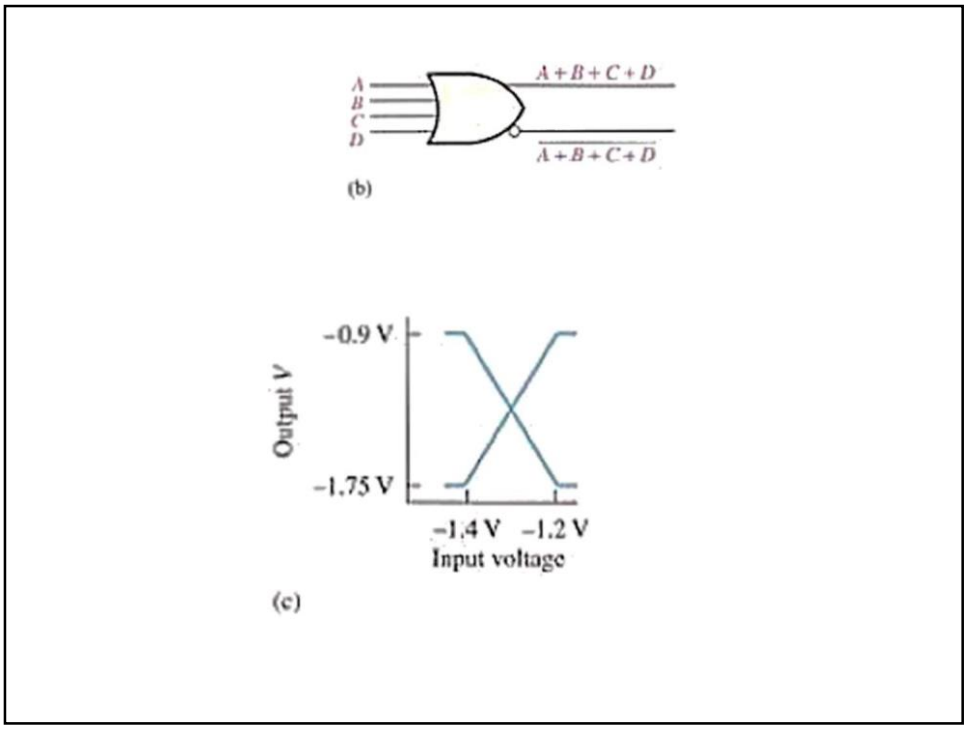
- Emitter-coupled logic, like TTI is a bipolar technology.
- The typical ECL circuit consists of a differential amplifier input circuit, a bias circuit, and emitter-follower outputs.
- ECL is much faster than TTL because the transistors do not operate in saturation and is used in more specialized high-speed applications.

OR/NOR Gate

- An ECL OR/NOR gate is considered.
- The emitter-follower outputs provide the OR logic function and its NOR complement, as indicated.



An ECL OR/NOR gate circuit



Operation

- Due to the low output impedance of the emitter-follower and the high input impedance of the differential amplifier input, high fan-out operation is possible.
- In this type of circuit. saturation is not possible. The lack of saturation results in higher power consumption and limited voltage swing (less than IV).
- But it permits high-frequency switching.
- The Vcc pin is normally connected to ground, and the V_{EE} pin is connected to -5.2 V from the power supply for best operation.
- Notice that the output varies from a LOW level of 1.75 V to a HIGH level of -0.9 V with respect to ground.
- In positive logic a 1 is the HIGH level (less negative), and a 0 is the LOW level (more negative).

Noise Margin

- The noise margin of a gate is the measure of its immunity to undesired voltage fluctuations (noise).
- Typical ECL circuits have noise margins from about 0.2V to 0.25 V.
- These are less than for TTL and make ECL less suitable in high-noise environments.

Comparison of ECL with TTL and CMOS

	BIPOLAR (TTL) F	CMOS AHC	BIPOLAR (ECL)
Speed			
Gate propagation delay, t_p (ns)	3.3	3.7	0.22–1
FF maximum clock freq. (MHz)	145	170	330–2800
Power Dissipation Per Gate			
Bipolar: 50% dc	8.9 mW		25 mW–73 mW
CMOS: quiescent		2.5 μW	

F- Fast. AHC- Advanced High Speed CMOS