Combinational Logic Design Techniques

Introduction to Digital Systems

Lecture #5

Prepared by
Pinaki Mazumder
Professor of Computer Science & Engineering
University of Michigan

Today's Lecture Addresses

1. How to design a digital system when the design specs are given in plain English
2. Other Representations - Equation, Truth Table, & Input-Output Waveforms
3. Minterm Expansion, Maxterm Expansion, Canonical SOP, Canonical POS, Self-Duality
4. Verilog Modeling and Design using Verilog

Reading Assignment: Lecture Slides, Textbook Chapter 2, Sec. 2.6-2.8; pp. 61-83; Chapter 9, Sec. 9.2-9.4; pp. 489-511.

Digital System Design Principles

1. Problem Statement
2. Canonical Implementation
3. Minimization by Boolean Algebra
4. Scaling of Problem Size

Objectives of Today's Lecture

Given a Problem in English Statement

1. How to assign Boolean variables
2. How to obtain Truth Table
3. How to write the
   - canonical sum of products (SOP)
   - canonical product of sums (POS)
4. How to minimize logic expressions using Boolean algebra

Objectives of Today's Lecture

5.How to Implement SOP in
AND-OR, NAND-NAND,
OR-NAND, NOR-OR Gates
6.How to Implement POS in
OR-AND, NOR-NOR,
AND-NOR, NAND-AND Gates

Altogether EIGHT (4 SOP and 4 POS) 2-level logic gate implementations can be done

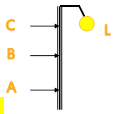
Step #1: ASSIGN BOOLEAN VARIABLES AND DEFINE THEIR VALUES with respect to the PROBLEM

Let L be the Boolean variable denoting the lamp, and A, B, and C are switches on the *first*, the *second* and the *third* floors, respectively.

Let each switch, X ε {A, B, C} has two positions – Up and Down.

Let X = 1 ➔ switch X is in Up position
X = 0 ➔ switch X is in Down position

Let L = 0 ➔ lamp L is Off
L = 1 ➔ lamp L is On



PROBLEM STATEMENT

In a 3-story building, there is a lamp to illuminate a *stairwell.*

The lamp can be *independently* turned **ON** and **OFF** from each floor by flipping an electrical switch on that floor. Design the logic circuit for the problem.

Note that Lamp can be either On or Off
But Switches can be either Up or Down. By flipping switches Up or Down, the Lamp can be turned On or Off.

Step #2: WRITE THE TRUTH TABLE

The TRUTH TABLE Shows the Input-Output Relationships between Boolean variables.

- **Input Boolean Variables: A, B, and C**
- **Output Boolean Variable: L**
- **Boolean Function: L(A,B,C)**

Number of Input Variables = 3 →
Number of Rows = 8.

Switch and Lamp States	Switch A	Switch B	Switch C	Lamp L
A=B=C=DOWN, L is OFF	0	0	0	0
C=UP, A=B=DOWN L is ON	0	0	1	1
A=C=DOWN, B=UP L is ON	0	1	0	1
B=C=UP, A=DOWN L is OFF	0	1	1	0
B=C=DOWN, A=UP L is ON	1	0	0	1
A=C=UP, B=DOWN L is OFF	1	0	1	0
C=DOWN, A=B=UP L is OFF	1	1	0	0
A=B=C=UP L is ON	1	1	1	1

All 3 switches DOWN, Lamp is OFF.

001, 010, 100

2 switches DOWN, 1 UP, Lamp is ON.

011, 101, 110

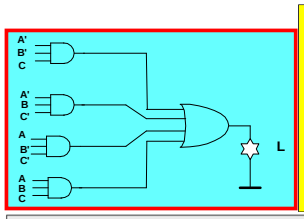
1 switch DOWN, 2 UP Lamp is OFF.

111

All 3 switches UP, Lamp is ON.

Therefore, L is ON (1) iff (A is DOWN (0) & B is DOWN (0) & C is UP (1)) OR (A is DOWN (0) & B is UP (1) & C is DOWN (0)) OR (A is in UP (1) & B is DOWN (0) & C is DOWN (0)) OR (A is UP (1) & B is UP (1) & C is UP (1))

Step #4: DIRECT IMPLEMENTATION OF THE CANONICAL SOP EXPRESSION



Cost of Implementation

No. of Gates = Gate cost= 4 X 3-input AND + 1 X 4-input OR

No. of Literals = Literal Cost = 3 X 4 = 12 (3 literals per gate)

No. of Transistors = 4 x (6+2) + (8+2) = 32 + 10 = 42

Rule: In standard CMOS technology logic gates generally have *negated* outputs. In order to implement an N-input NAND or NOR gate, 2N transistors are required. In order to implement an AND or OR gate, an additional NOT gate using 2 transistors is needed. Hence, NAND and NOR gates are more commonly used (instead of AND and OR gates) in CMOS implementation of Boolean functions.

Hence, L=1 iff (A=B=0,C=1) OR (A=C=0, B=1) OR (A=1, B=C=0) OR (A=B=C=1)

Step #3: WRITE CANONICAL SUM OF PRODUCTS (STANDARD SOP) FOR THE OUTPUT BOOLEAN VARIABLE(s)

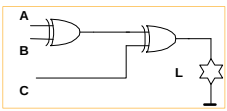
$$L(A, B, C) = A'B'C + A'BC' + AB'C' + ABC$$

Note that each **Product term** has all three inputs in **True** or **Complement** form. These inputs are called **Literals**, and each product term is called a **Minterm**. The output function can be written as a sum of minterms, $L(A,B,C) = \Sigma(1,2,4,7)$ which is called the **Minterm Expansion for L**(A,B,C).

Step #5: MINIMIZATION OF THE CANONICAL SOP

Use Boolean Algebra to Simplify the Canonical Expression

$L(A, B, C) = AB'C' + A'BC' + A'B'C + ABC$
$= (AB'+A'B).C' + (A'B' + AB).C$ (Law of Distribution)
$= (A \oplus B).C' + (A \oplus B)'.C$ (Def. of Ex-Or Equivalence)
$= (A \oplus B) \oplus C = A \oplus B \oplus C.$



Cost of Implementation

Gate cost= 2 X 2-input Ex-OR

Literal Cost = 3

No of Transistors = 6 x 2 = 12

Savings due to Minimization

Gate Cost = (5-2)/5 = 60%
Literal Cost = (12-3)/12 = 9/12 = 75%
No. of Transistors = (42-12)/42 = 30/42 = 75%

How many Minterms appear in the Expression?
(Exponential Growth of the Output Expression)

For *n* = 3, # of minterms or product terms = 4
For *n* = 11, # of minterms or product terms = 1024.

Combinational Logic Design Techniques

Introduction to Digital Systems

Lecture #5 contd.

Prepared by
Pinaki Mazumder
Professor of Computer Science & Engineering
University of Michigan

Exclusive OR and Exclusive NOR (or Equivalence) gates are used in such pathological cases when the minterms cannot be combined to yield smaller product terms.

GENERAL SOLUTION:
For an *n*-story building, the Lamp equation will be given by:

$$L(S_0, S_1, S_2, \ldots, S_{n-1}) = S_0 \oplus S_1 \oplus S_2 \oplus \ldots \oplus S_{n-1}$$

Digital System Design

1. Majority Gate
2. Canonical SOP Implementation
3. Minimization of SOP by Boolean Algebra
4. Canonical POS Implementation
5. Minimization of POS by Boolean Algebra
6. Self-Duality of Majority Function

DESIGN OF A MAJORITY GATE

An *n*-input Majority Gate has an output $Z = 1$ iff *at least* $\lfloor n/2 \rfloor + 1$ inputs are 1. Otherwise, $Z = 0$.

For a 3-input (*A*, *B*, and *C*) Majority Gate, at least 2 inputs must be 1 in order that output $Z(A,B,C) = 1$.

Step #1: IDENTIFY BOOLEAN VARIABLES

Input Variables = A, B, C
Output Variable = Z

Therefore, Z is ON (1) iff (A=0 & B=1 & C =1) OR (A=1 & B=0 & C=1) OR (A=1 & B=1 & C=0) OR (A=1 & B=1 & C=1)

Step #3: WRITE CANONICAL SUM OF PRODUCTS (STANDARD SOP) OR MINTERM EXPANSION FOR THE OUTPUT BOOLEAN VARIABLE

$$Z(A, B, C) = A'BC + AB'C + ABC' + ABC = \Sigma\, m(3,5,6,7)$$

Note that each **Product term** has all three inputs in **true** or **complement** form. These inputs are called **literals,** and each product term is called a **Minterm. The expression Σm(3,5,6,7) is called the Minterm Expansion for Z (A, B, C)**

Step #2: WRITE THE TRUTH TABLE

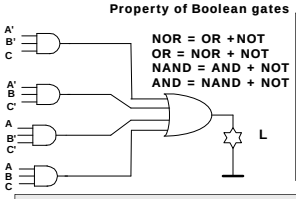
The TRUTH TABLE shows the Input-Output relationships between Boolean variables.

Input Boolean Variables: **A, B,** and **C**;

Output Boolean Variable: **Z**

Minterm and Maxterm	A	B	C	Z
m0 = A'B'C' M0 = A+B+C	0	0	0	0
m1=A'B'C M1 = A+B+C'	0	0	1	0
m2 = A'BC' M2 = A+B'+C	0	1	0	0
m3 = A'BC M3 = A+B'+C'	0	1	1	1
m4 = AB'C' M4 = A'+B+C	1	0	0	0
m5 = AB'C M5 = A'+B+C'	1	0	1	1
m6 = ABC' M6 = A'+B'+C	1	1	0	1
m7 = ABC M7 = A' +B'+C'	1	1	1	1

Step #4: DIRECT IMPLEMENTATION OF THE CANONICAL SOP EXPRESSION



Implementation Cost:

No. of gates = 5

No. of literals = 4 x 3 = 12

No. of transistors = 4 x [3x2 (NAND) + 2 (NOT)] + 1 x [4x2 (NOR) + 2 (NOT)] = 42

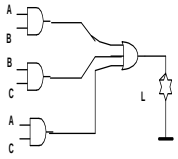
Rule: In standard CMOS technology logic gates generally have *negated* outputs. In order to implement an N-input NAND or NOR gate, 2N transistors are required. In order to implement an AND or OR gate, an additional NOT gate using 2 transistors is needed. Hence, NAND and NOR gates are more commonly used (instead of AND and OR gates) in CMOS implementation of Boolean functions.

MINIMIZATION OF THE CANONICAL SOP

Use Boolean Algebra to Simplify the Canonical Expression

$Z(A, B, C) = A'BC + AB'C + ABC' + ABC$

$= A'BC + ABC + AB'C + ABC + ABC' + ABC$ (Idempotent Laws)
$= (A+A')BC + A(B+B')C + AB(C+C')$ (Distributive Laws)
$= 1.BC + A.1.C + AB.1$ (Laws of Complementarity)
$= BC + CA + AB$ (since, Y.1 = Y; Commutative Laws)

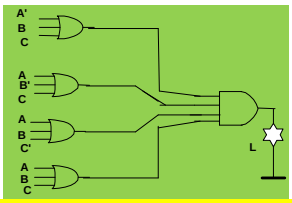


Implementation Cost:

No. of gates = 4 (Savings = 20%)
No. of literals = 3 x 2 = 6 (Savings = 50%)
No. of transistors =
3 x [2x2 (NAND) + 2 (NOT)]
+ 1 x [3x2 (NOR) + 2 (NOT)]
= 26 (Savings = 16%)

The above expression for Z(A,B,C) is called **CANONICAL PRODUCT OF SUMS, or PRODUCT OF MAXTERMS, or MAXTERM EXPANSION FORM.**
Note that $Z(A,B,C) = \Sigma m(3,5,6,7) = \Pi M(0,1,2,4)$
Hence, the SOP and POS implementations are functionally equivalent.

Step #4': DIRECT IMPLEMENTATION OF THE CANONICAL POS EXPRESSION



ALTERNATIVE IMPLEMENTATION

CANONICAL PRODUCT OF SUMS OR MAXTERM EXPANSION

Alternatively, Z is OFF (0), i.e., Z' is ON (1) iff (A=0 & B=0 & C =0) OR (A=0 & B=0 & C=1) OR (A=0 & B=1 & C=0) OR (A=1 & B=0 & C=0)

$Z'(A, B, C) = A'B'C' + A'B'C + A'BC' + AB'C' = \Sigma m(0,1,2,4)$

→ $Z(A,B,C) = [A'B'C' + A'B'C + A'BC' + AB'C']'$

→ $Z(A,B,C) = (A'B'C')' . (A'B'C)' . (A'BC')' . (AB'C')'$
(applying De Morgan's Laws)
→ $Z(A,B,C) = (A+B+C).(A+B+C').(A+B'+C).(A'+B+C)$

→ $Z(A,B.C) = M0.M1.M2.M4 = \Pi M(0,1,2,4)$

Step #5: MINIMIZATION OF THE CANONICAL POS

Canonical POS expression for Z(A,B,C) is given by:

$Z(A,B,C) = (A+B+C).(A+B+C').(A+B'+C).(A'+B+C)$
$Z(A,B,C) = [(A+B+C).(A+B+C')].[(A+B+C).(A+B'+C)].$
$[(A+B+C).(A'+B+C)]$
[X = X .X, Laws of Idempotence]

$Z(A,B,C) = (A+B).(A+C).(B+C)$ [Laws of Complementarity, (X+Y).(X+Y') = X]

Dual of POS= Z_D (A,B,C) = A.B+B.C+C.A = Z(A,B,C) of SOP
→ Majority Function (Z) is Self-Dual.

2-LEVEL LOGIC IMPLEMENTATION STYLES

Sum of Products (SOP)

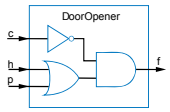
1. AND-OR: $Z = AB + BC + CA$
2. NAND-NAND: $Z = \overline{\overline{Z}} = \overline{\overline{AB + BC + CA}}$
   $= \overline{\overline{AB}.\overline{BC}.\overline{CA}}$

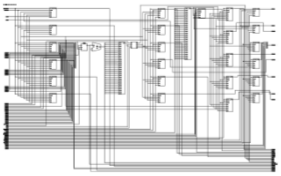
Inverted Inputs

3. OR-NAND: $Z = \overline{(A'+B').(B'+C').(C'+A')}$
4. NOR-OR: $Z = (A'+B')' + (B'+C')' + (C'+A')'$

Large Circuit Diagram Becomes Messy

- A drawing of a circuit, or *schematic*, contains graphical information about a design
  - Inverter is above the OR gate, AND gate is to the right, etc.
- Such graphical information may not be useful for large designs
- Can use textual language instead





Product of Sums (POS)

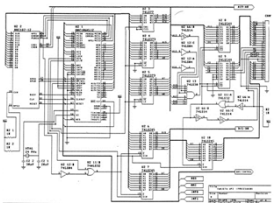
1. OR-AND: $Z = (A+B)(B+C).(C+A)$
2. NOR-NOR: $Z = \overline{\overline{Z}} = \overline{[(A+B)(B+C).(C+A)]'}$

Inverted Inputs $= [(A+B)' + (B+C)' + (C+A)']'$

3. AND-NOR: $Z = \overline{(A.B') + (B.C') + (C'.A)'}$
4. NAND-AND: $Z = \overline{(AB').(BC').(C'A')}$

Therefore, given a Boolean expression, you can implement the expression in 8 different styles.

- **A drawing of a circuit, or *schematic*, contains graphical information about a design**
- **Such graphical information may not be useful for large designs**
- **Can use textual language instead**



Computer-Readable Textual Language for Describing Hardware Circuits: HDLs

- **Hardware description language (HDL)**
  - Intended to describe circuits textually, for a computer to read
  - Evolved starting in the 1970s and 1980s
- **Popular languages today include:**
  - VHDL –Defined in 1980s by U.S. military; Ada-like language
  - Verilog –Defined in 1980s by a company; C-like language
  - SystemC –Defined in 2000s by several companies; consists of libraries in C++


```
/usr/caen/ius-8.2/tools/bin/ncverilog     probs.v

ncsim> source /usr/caen/ius-8.2/tools/inca/files/ncsimrc
ncsim> run

A:0 B:0 C:0     majority:0     lamp:0
A:0 B:0 C:1     majority:0     lamp:1
A:0 B:1 C:0     majority:0     lamp:1
A:0 B:1 C:1     majority:1     lamp:0
A:1 B:0 C:0     majority:0     lamp:1
A:1 B:0 C:1     majority:1     lamp:0
A:1 B:1 C:0     majority:1     lamp:0
A:1 B:1 C:1     majority:1     lamp:1

Simulation complete via $finish(1) at time 39 NS + 0
./probs.v:46     $finish;

ncsim> exit
```

Majority = 1, if at least 2 inputs are 1.

Lamp = 1, if odd number of inputs are 1.

VERILOG CODE

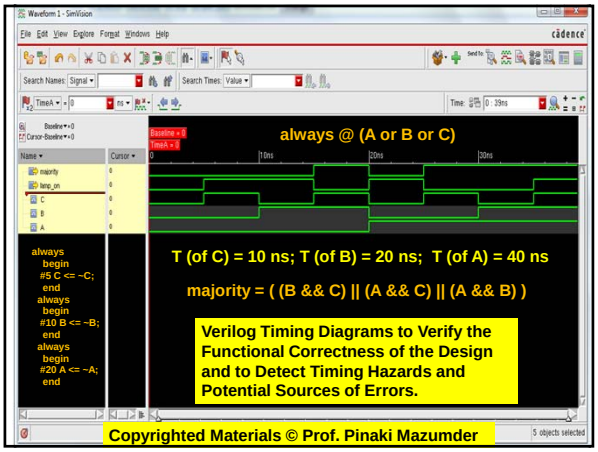

```
`timescale 1ns/1ps
module probs( majority, lamp_on );
  output reg majority;   output reg lamp_on;
reg     A;   reg     B; reg     C;
    always
     begin
    #5 C <= ~C;
     end
    always
     begin
    #10 B <= ~B;
     end
    always
     begin
    #20 A <= ~A;
     end
    always @ (A or B or C)
     begin
    if ( (B && C) || (A && C) || (A && B) )
      majority = 1'b1;
    else
      majority = 1'b 0;

    if ( (~A && ((~B && C) || (B && ~C))) || (A && ((~B && ~C) || (B && C))))
      lamp_on=1'b1;
    else
      lamp_on=1'b0;
     end

   initial
     begin
    $monitor("A:%b B:%b C:%b majority:%b lamp:%b ", A, B, C, majority, lamp_on);
    A <= 0;    B <= 0;    C <= 0;    #39;      $finish;
     end
endmodule
```

Hardware Description Language

VERILOG Simulation Codes are written for Designing Large Digital Systems and are Verified for Functional a and Timing Correctness before Hardware Implementation is Made using FPGA or Discrete Chips.



T (of C) = 10 ns; T (of B) = 20 ns; T (of A) = 40 ns

majority = ((B && C) || (A && C) || (A && B))

Verilog Timing Diagrams to Verify the Functional Correctness of the Design and to Detect Timing Hazards and Potential Sources of Errors.

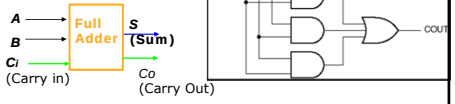
ANOTHER EXAMPLE

Before Digital Gates

- **FULL ADDER that consists of**

3 inputs (A, B, Ci) and two outputs (S, Co)

$$S = A \oplus B \oplus Ci$$
$$Co = AB + BCi + CiA$$



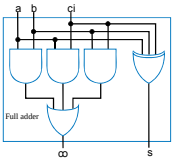
A → Addend
B → Augend
Cin → Carry In

S → Sum
Cout → Carry Out

Describing a Full-Adder in Verilog

- Module
  - Declares inputs/outputs
  - Described behaviorally (could have been described structurally)
  - "always" procedure
    - Sensitive to inputs
  - Computes expressions, sets outputs

s = a xor b xor ci
co = bc + ac + ab



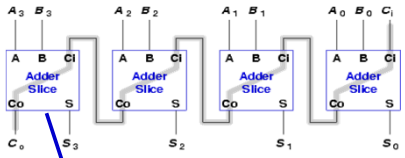
Chapter 9 of Vahid

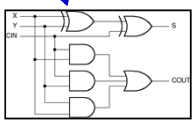

```
module FullAdder(a, b, ci, s, co);
  input a, b, ci;
  output s, co;
  reg s, co;

  always @(a or b or ci)
  begin
    s <= a ^ b ^ ci;
    co <= (b & ci) | (a & ci) | (a & b);
  end
endmodule
```

Hierarchical Design of a 4-bit Ripple Carry Adder (RCA)

A[0:3]+B[0:3] = S[0:3], Co

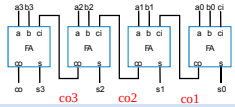




Hierarchical Design Method Allows the Digital System to Scale up Easily and Reduces the Design Complexity

Describing a Carry-Ripple Adder in Verilog

- Module
  - Declares inputs/outputs
  - Uses vectors for 4-bit inputs/outputs
  - Described structurally by composing four full-adders (could have been described behaviorally instead)
  - Instantiates four full-adders, connects
    - Note use of three interna wires for connecting carry-out of one stage to carry-in of next stage



Chapter 9 of Vahid


```
module CarryRippleAdder4(a, b, ci, s, co);
  input [3:0] a;
  input [3:0] b;
  input ci;
  output [3:0] s;
  output co;

  wire co1, co2, co3;

  FullAdder FullAdder1(a[0], b[0], ci,
                       s[0], co1);
  FullAdder FullAdder2(a[1], b[1], co1,
                       s[1], co2);
  FullAdder FullAdder3(a[2], b[2], co2,
                       s[2], co3);
  FullAdder FullAdder4(a[3], b[3], co3,
                       s[3], co);
endmodule
```

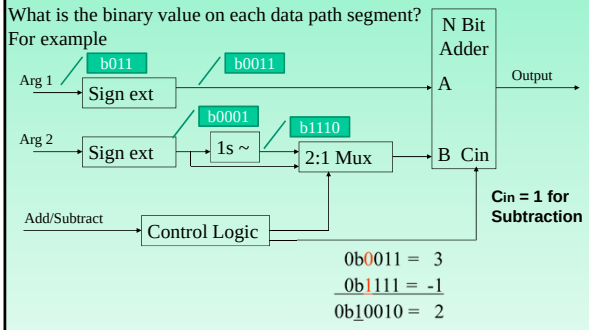
High Level Functional Organization to Implement Addition and Subtraction

Additional functionality will be required to implement absolute value

Consider 3 – 1 = 2

What is the binary value on each data path segment?

For example



0b0011 = 3

0b1111 = -1

0b10010 = 2

Today's Lecture Addresses

1. How to design a digital system when the design specs are given in plain English
2. Other Representations - Equation, Truth Table, & Input-Output Waveforms
3. Minterm Expansion, Maxterm Expansion, Canonical SOP, Canonical POS, Self-Duality
4. Verilog Modeling and Design using Verilog

Reading Assignment: Lecture Slides, Textbook Chapter 2, Sec. 2.6-2.8; pp. 61-83; Chapter 9, Sec. 9.2-9.4; pp. 489-511.