Representations of Boolean Functions

- Readings: 2.5, 2.5.2-2.10.4

- Boolean Function: $F = \overline{X} + YZ$

Truth Table:

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>Z</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Circuit Diagram:
Why Boolean Algebra/Logic Minimization?

\[
\bar{A}BC_{in} + A\bar{B}C_{in} + AB\bar{C}_{in} + ABC_{in} \quad vs. \quad AB + AC_{in} + BC_{in}
\]

Logic Minimization: reduce complexity of the gate level implementation

- reduce number of literals (gate inputs)
- reduce number of gates
- reduce number of levels of gates

fewer inputs implies faster gates in some technologies
fan-ins (number of gate inputs) are limited in some technologies
fewer levels of gates implies reduced signal propagation delays
number of gates (or gate packages) influences manufacturing costs
Basic Boolean Identities:

\[
\begin{align*}
X + 0 &= X \\
X + 1 &= X \\
X + X &= X \\
X + \overline{X} &= X \\
\overline{X} &= X
\end{align*}
\]
Basic Laws

Commutative Law:
\[ X + Y = Y + X \quad \text{and} \quad XY = YX \]

Associative Law:
\[ X + (Y + Z) = (X + Y) + Z \quad \text{and} \quad X(YZ) = (XY)Z \]

Distributive Law:
\[ X(Y + Z) = XY + XZ \quad \text{and} \quad X + YZ = (X + Y)(X + Z) \]
Boolean Manipulations

- Boolean Function: $F = XYZ + \overline{XY} + XYZ\overline{Z}$

Truth Table:

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>Z</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Reduce Function:
Advanced Laws (Absorbtion)

- $X + XY =$
- $XY + X\bar{Y} =$
- $X + \bar{X}Y =$
- $X(X + Y) =$
- $(X + Y)(X + \bar{Y}) =$
- $X(\bar{X} + Y) =$
Boolean Manipulations (cont.)

- **Boolean Function**: $F = \overline{X}YZ + XZ$

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>Z</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Reduce Function:
Boolean Manipulations (cont.)

- Boolean Function: $F = (X+\overline{Y}+X\overline{Y})(XY+\overline{X}Z+YZ)$

Truth Table:

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>Z</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
DeMorgan’s Law

\[(X + Y) = \overline{X} \cdot \overline{Y}\]

\[(X \cdot Y) = \overline{X} + \overline{Y}\]

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>Y</td>
<td>\overline{X}</td>
<td>\overline{Y}</td>
<td>\overline{X} + \overline{Y}</td>
<td>\overline{X} \cdot \overline{Y}</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

DeMorgan's Law can be used to convert AND/OR expressions to OR/AND expressions

Example:

\[Z = \overline{A} \overline{B} C + \overline{A} B C + A \overline{B} C + A B \overline{C}\]

\[\overline{Z} = (A + B + \overline{C}) \cdot (A + \overline{B} + \overline{C}) \cdot (\overline{A} + B + \overline{C}) \cdot (\overline{A} + \overline{B} + C)\]
DeMorgan’s Law example

- If $F = (XY+Z)(\bar{Y}+\bar{X}Z)(X\bar{Y}+\bar{Z})$,

  $\bar{F} =$
Boolean Equations to Circuit Diagrams

- $F = \overline{XYZ} + \overline{XY} + XYZ$

- $F = XY + X(WZ + W\overline{Z})$
Circuit Timing Behavior

- Simple model: gates react after fixed delay

![Circuit Diagram]

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Hazards/Glitches

- Circuit can temporarily go to incorrect states

Diagram:
- Copilot Autopilot Request
- Pilot in Charge?
- Pilot Autopilot Request
- Autopilot Engaged

Time Graph:
- CAR
- PIC
- PAR
- A
- B
- C
- AE
Field Programmable Gate Arrays (FPGAs)

Logic cells imbedded in a general routing structure

Logic cells usually contain:

- 6-input Boolean function calculator
- Flip-flop (1-bit memory)

All features electronically (re)programmable
Verilog

// Verilog code for 2-input multiplexer
module AOI (F, A, B, C, D);
output F;
input A, B, C, D;
assign F = ~(A & B) | (C & D);
endmodule

module MUX2 (V, SEL, I, J);
output V;
input SEL, I, J;
wire SELB, VB;
not G1 (SELB, SEL);
AOI G2 (VB, I, SEL, SELB, J);
not G3 (V, VB);
endmodule

FPGA CAD Tools

Bitstream

Simulation
Verilog

- Programming language for describing hardware
  - Simulate behavior before (wasting time) implementing
  - Find bugs early
  - Enable tools to automatically create implementation

- Similar to C/C++/Java
  - VHDL similar to ADA

- Modern version is “System Verilog”
  - Superset of previous; cleaner and more efficient
// Verilog code for AND-OR-INVERT gate

module AOI (F, A, B, C, D);
  output logic  F;
  input  logic  A, B, C, D;

  assign F = ~((A & B) | (C & D));
endmodule

// end of Verilog code
Verilog Wires/Variables

// Verilog code for AND-OR-INVERT gate

module AOI (F, A, B, C, D);
    output logic F;
    input logic A, B, C, D;
    logic AB, CD, O;

    assign AB = A & B;
    assign CD = C & D;
    assign O = AB | CD;
    assign F = ~O;
endmodule
// Verilog code for AND-OR-INVERT gate

module AOI (F, A, B, C, D);
   output logic F;
   input  logic A, B, C, D;
   logic   AB, CD, O;

   and a1(AB, A, B);
   and a2(CD, C, D);
   or  o1(O, AB, CD);
   not n1(F, O);
endmodule
Verilog Hierarchy

// Verilog code for 2-input multiplexer

module AOI (F, A, B, C, D);
    output logic F;
    input logic A, B, C, D;

    assign F = ~((A & B) | (C & D));
endmodule

module MUX2 (V, SEL, I, J); // 2:1 multiplexer
    output logic V;
    input logic SEL, I, J;
    logic SELB, VB;

    not G1 (SELB, SEL);
    AOI G2 (.F(VB), .A(I), .B(SEL), .C(SELB), .D(J));
    not G3 (V, VB);
endmodule
module MUX2TEST;  // No ports!
    logic SEL, I, J, V;

    initial  // Stimulus
    begin
        SEL = 1; I = 0; J = 1;
        #10 I = 1;
        #10 SEL = 0; J = 0;
        #10 J = 1;
    end

    MUX2 M (.V, .SEL, .I, .J);
endmodule
NAND and NOR Gates

- NAND Gate: NOT(AND(A, B))

- NOR Gate: NOT(OR(A, B))
Bubble Manipulation

- DeMorgan’s Law
- Simplification: \( \overline{AB} + \overline{CD} \)
## NAND and NOR Gate Universality

- NAND and NOR gates are universal
  - can implement all the basic gates (AND, OR, NOT)

<table>
<thead>
<tr>
<th>NAND</th>
<th>NOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOT</td>
<td></td>
</tr>
<tr>
<td>AND</td>
<td></td>
</tr>
<tr>
<td>OR</td>
<td></td>
</tr>
</tbody>
</table>
Converting Circuits to NAND/NOR Form

- Group gates into levels, insert double inversions on alternating levels

- Alternating AND/OR becomes all NAND or NOR
Converting to NAND/NOR Form (cont.)

- Some circuits may require internal inverters

```
Level 1 | Level 2 | Level 3
---|---|---
A       |       |   
B       |       |   
C       |       |   
A       |       |   
B       |       |   
C       |       |   
```
XOR and XNOR Gates

- **XOR Gate**: $Z=1$ if odd # of inputs are true

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

- **XNOR Gate**: $Z=1$ if even # of inputs are true

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>