Basics of Logic Design: Boolean Algebra, Logic Gates

CPS 104
Lecture 8

Today’s Lecture

Outline
• Building the building blocks…
• Logic Design
  ➢ Truth tables, Boolean functions, Gates and Circuits

Reading
  Appendix B, Chapter 4
The Big Picture

- The Five Classic Components of a Computer

What We’ve Done, Where We’re Going

Software

Interface Between HW and SW

Instruction Set Architecture, Memory, I/O

Hardware

CPU Memory I/O system

Digital Design

Circuit Design

Top Down

Bottom UP to CPU
Digital Design

- Logic Design, Switching Circuits, Digital Logic

Recall: Everything is built from transistors
- A transistor is a switch
- It is either on or off
- On or off can represent True or False

Given a bunch of bits (0 or 1)...
- Is this instruction a lw or a beq?
- What register do I read?
- How do I add two numbers?
- Need a method to reason about complex expressions

Boolean Algebra

- Boolean functions have arguments that take two values (\{T,F\} or \{1,0\}) and they return a single or a set of (\{T,F\} or \{1,0\}) value(s).
- Boolean functions can always be represented by a table called a “Truth Table”
- Example: \( F: \{0,1\}^3 \rightarrow \{0,1\}^2 \)

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>f_1</th>
<th>f_2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
**Boolean Functions**

- **Example** Boolean Functions: NOT, AND, OR, XOR, . . .

<table>
<thead>
<tr>
<th>a</th>
<th>NOT (a)</th>
<th>b</th>
<th>AND (a, b)</th>
<th>OR (a, b)</th>
<th>XOR (a, b)</th>
<th>XNOR (a, b)</th>
<th>NOR (a, b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<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>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Boolean Functions and Expressions**

- **Boolean algebra notation**: Use * for AND, + for OR, ~ for NOT.
  - Not is also written as $A'$ and $\overline{A}$
- Using the above notation we can write Boolean expressions for functions
  
  \[ F(A, B, C) = (A * B) + (\sim A * C) \]

- We can evaluate the Boolean expression with all possible argument values to construct a truth table.

- What is truth table for $F$?
Boolean Function Simplification

- Boolean expressions can be simplified by using the following rules:
  - $A \cdot A = A$
  - $A \cdot 0 = 0$
  - $A \cdot 1 = A$
  - $A \cdot \overline{A} = 0$
  - $A + A = A$
  - $A + 0 = A$
  - $A + 1 = 1$
  - $A + \overline{A} = 1$
  - $A \cdot B = B \cdot A$
  - $A \cdot (B + C) = (B + C) \cdot A = A \cdot B + A \cdot C$

Boolean Function Simplification

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>$f_1$</th>
<th>$f_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</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>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

$f_1 = \overline{a} \cdot \overline{b} \cdot c + \overline{a} \cdot b \cdot c + a \cdot \overline{b} \cdot c + a \cdot b \cdot c$

$f_2 = \overline{a} \cdot b \cdot \overline{c} + \overline{a} \cdot \overline{b} \cdot c + a \cdot b \cdot \overline{c} + a \cdot b \cdot c$

Simplify these functions...
Boolean Functions and Expressions

- The Fundamental Theorem of Boolean Algebra: Every Boolean function can be written in disjunctive normal form as an OR of ANDs (Sum-of-products) of its arguments or their complements.

“Proof:” Write the truth table, construct sum-of-product from the table.

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>XNOR(a, b)</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>

XNOR = (~a * ~b) + (a * b)

Boolean Functions and Expressions

- Example-2:

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>f1</th>
<th>f2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

f1 = ~a+~b+c + ~a+b~c + a~b~c + a+b+c

f2 = ~a+~b~c + ~a+b~c + a~b~c + a+b+c
DeMorgan’s Laws

• \( \neg(A+B) = \neg A \times \neg B \)
• \( \neg(A\times B) = \neg A + \neg B \)

Example:

• \( \neg C \times \neg A \times B + \neg C \times A \times \neg B + C \times A \times B + C \times \neg A \times \neg B \)
• Use only XOR to represent this function

Applying the Theory

• Lots of good theory
• Can reason about complex boolean expressions
• Now we have to make it real...
**Boolean Gates**

- **Gates** are electronic devices that implement simple Boolean functions

**Examples**

- **AND** $(a, b)$
- **OR** $(a, b)$
- **NAND** $(a, b)$
- **NOR** $(a, b)$
- **XOR** $(a, b)$
- **XNOR** $(a, b)$
- **NOT** $(a)$

---

**Reality Check**

- Basic 1 or 2 Input Boolean Gate 1-4 Transistors

**Pentium III**

- Processor Core 9.5 Million Transistors
- Total: 28 Million Transistors

**Pentium 4**

- Total: 42 Million Transistors
### Boolean Functions, Gates and Circuits

- **Circuits** are made from a network of gates. (function compositions).

\[
F = \neg a \cdot b + \neg b \cdot a
\]

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>XOR(a, b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</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>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

![Gate Diagram](image)

### Digital Design Examples

Input: 2 bits representing an unsigned number (n)
Output: \(n^2\) as 4-bit unsigned binary number

Input: 2 bits representing an unsigned number (n)
Output: \(3-n\) as unsigned binary number
Design Example

• Consider machine with 4 registers
• Given 2-bit input (register specifier, \(I_1, I_0\))
• Want one of 4 output bits (\(O_3-O_0\)) to be 1
  - E.g., allows a single register to be accessed
• What is the circuit for this?

Circuit Example: Decoder

<table>
<thead>
<tr>
<th>(I_1)</th>
<th>(I_0)</th>
<th>(Q_0)</th>
<th>(Q_1)</th>
<th>(Q_2)</th>
<th>(Q_3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>1 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1</td>
<td>0 1 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0</td>
<td>0 0 1 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 1</td>
<td>0 0 0 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Circuit Example: 2x1 MUX**

Multiplexer (MUX) selects from one of many inputs

\[ Y = (A \cdot S) + (B \cdot \sim S) \]

**Example 4x1 MUX**
Arithmetic and Logical Operations in ISA

- What operations are there?
- How do we implement them?
  - Consider a 1-bit Adder

Summary

- Boolean Algebra & functions
- Logic gates (AND, OR, NOT, etc)
- Multiplexors

Reading
- Appendix B, Chapter 4
Example: 4-bit adder

Subtraction

- How do we perform integer subtraction?
- What is the HW?
Overflow

Example 1:

\[
\begin{array}{c}
0100000 \\
0110101_2 \quad (= 53_{10}) \\
+0101010_2 \quad (= 42_{10}) \\
1011111_2 \quad (= -33_{10})
\end{array}
\]

Example 2:

\[
\begin{array}{c}
1000000 \\
1010101_2 \quad (= -43_{10}) \\
+1001010_2 \quad (= -54_{10}) \\
0011111_2 \quad (= 31_{10})
\end{array}
\]

Example 3:

\[
\begin{array}{c}
1100000 \\
0110101_2 \quad (= 53_{10}) \\
+1101010_2 \quad (= -22_{10}) \\
0011111_2 \quad (= 31_{10})
\end{array}
\]

Example 4:

\[
\begin{array}{c}
0000000 \\
0010101_2 \quad (= 21_{10}) \\
+0101010_2 \quad (= 42_{10}) \\
0111111_2 \quad (= 63_{10})
\end{array}
\]

ALU Slice

\[
\begin{array}{c|c|c}
A & F & Q \\
0 & 0 & a + b \\
1 & 0 & a - b \\
-1 & 1 & \text{N} \text{OT b} \\
-2 & 2 & a \text{ OR b} \\
-3 & 3 & a \text{ AND b}
\end{array}
\]
Summary

- Boolean Algebra & functions
- Logic gates (AND, OR, NOT, etc)
- Multiplexors
- Adder
- Arithmetic Logic Unit (ALU)
- Reading
- Appendix B, Chapter 4