EECS150 - Digital Design

Lecture 4 - Boolean Algebra I
(Representations of Combinational Logic Circuits)

January 30, 2003

John Wawrzynek
Outline

- Review of three representations for combinational logic:
  - truth tables,
  - graphical (logic gates), and
  - algebraic equations
- Relationship among the three
- Adder example
- Formal description of Boolean algebra
- Laws of Boolean algebra
Combinational Logic (CL) Defined

\[ y_i = f_i(x_0, \ldots, x_{n-1}) \], where x, y are \{0,1\}.

- Y is a function of only X.

- If we change X, Y will change immediately (well almost!).
- There is an implementation dependent delay from X to Y.
CL Block Example #1

Boolean Equation:

\[ y_0 = (x_0 \text{ AND } \neg(x_1)) \]
\[ \quad \text{OR} \quad (\neg(x_0) \text{ AND } x_1) \]
\[ y_0 = x_0x_1' + x_0'x_1 \]

Truth Table Description:

<table>
<thead>
<tr>
<th>x0</th>
<th>x1</th>
<th>y</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>

Gate Representation:

How would we prove that all three representations are equivalent?
Boolean Algebra/Logic Circuits

• Why are they called “logic circuits”?  
  Logic: The study of the principles of reasoning.  
  The 19th Century Mathematician, George Boole, developed a math. system (algebra) involving logic, Boolean Algebra.  
  His variables took on TRUE, FALSE  
  Later Claude Shannon (father of information theory) showed (in his Master’s thesis!) how to map Boolean Algebra to digital circuits:  
• Primitive functions of Boolean Algebra:

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>AND</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>OR</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</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>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>a</th>
<th>NOT</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Spring 2003  
EECS150 – Lec4-bool1  
Page 5
Relationship Among Representations

* Theorem: Any Boolean function that can be expressed as a truth table can be written as an expression in Boolean Algebra using AND, OR, NOT.

How do we convert from one to the other?
Notes on Example #1

- The example is the standard function called exclusive-or (XOR, EXOR)
- Has a standard algebraic symbol:

<table>
<thead>
<tr>
<th>x0</th>
<th>x1</th>
<th>y</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>

- And a standard gate symbol:
### CL Block Example #2

- **4-bit adder:**

  \[ R = A + B, \]

  \( c \) is carry out

\[ \text{Diagram of a 4-bit adder with inputs A and B, and outputs R and c.} \]

- **Truth Table Representation:**

<table>
<thead>
<tr>
<th>a3</th>
<th>a2</th>
<th>a1</th>
<th>a0</th>
<th>b3</th>
<th>b2</th>
<th>b1</th>
<th>b0</th>
<th>r3</th>
<th>r2</th>
<th>r1</th>
<th>r0</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

In general: \( 2^n \) rows for \( n \) inputs.

256 rows!

Is there a more efficient (compact) way to specify this function?
4-bit Adder Example

- Motivate the adder circuit design by \textit{hand addition}:

\[
\begin{array}{cccc}
\text{a3} & \text{a2} & \text{a1} & \text{a0} \\
+ \text{b3} & \text{b2} & \text{b1} & \text{b0} \\
\hline
\text{c} & \text{r3} & \text{r2} & \text{r1} & \text{r0}
\end{array}
\]

- Add \(a_0\) and \(b_0\) as follows:

\[
\begin{array}{ccc|c|c}
\text{a} & \text{b} & \text{r} & \text{c} \\
0 & 0 & 0 & 0 \\
0 & 1 & 1 & 0 \\
1 & 0 & 1 & 0 \\
1 & 1 & 0 & 1 \\
\end{array}
\]

\(r = a \text{ XOR } b\)
\(c = a \text{ AND } b = ab\)

- Add \(a_1\) and \(b_1\) as follows:

\[
\begin{array}{ccc|c|c}
\text{ci} & \text{a} & \text{b} & \text{r} & \text{co} \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 & 0 \\
0 & 1 & 0 & 1 & 0 \\
0 & 1 & 1 & 0 & 1 \\
1 & 0 & 0 & 1 & 0 \\
1 & 0 & 1 & 0 & 1 \\
1 & 1 & 0 & 0 & 1 \\
1 & 1 & 1 & 1 & 1 \\
\end{array}
\]

\(r = a \text{ XOR } b \text{ XOR } ci\)
\(co = ab + aci + bci\)
4-bit Adder Example

- In general:
  \[ r_i = a_i \oplus b_i \oplus \text{cin} \]
  \[ c_{out} = a_i \text{cin} + a_i b_i + b_i \text{cin} = \text{cin}(a_i + b_i) + a_i b_i \]

- Now, the 4-bit adder:
  "Full adder cell"
  "ripple" adder
4-bit Adder Example

- Graphical Representation of FA-cell
  \[ r_i = a_i \text{ XOR } b_i \text{ XOR } c_{in} \]
  \[ c_{out} = a_i c_{in} + a_i b_i + b_i c_{in} \]

- Alternative Implementation (with 2-input gates):
  \[ r_i = (a_i \text{ XOR } b_i) \text{ XOR } c_{in} \]
  \[ c_{out} = c_{in} (a_i + b_i) + a_i b_i \]
Boolean Algebra

Defined as: Set of elements $B$, binary operators $\{+, \cdot\}$, unary operation $\{\}'\}$, such that the following axioms hold:

1. $B$ contains at least two elements $a, b$ such that $a \neq b$.
2. Closure : $a, b$ in $B$,
   $a + b$ in $B$, $a \cdot b$ in $B$, $a'$ in $B$.
3. Communitve laws :
   $a + b = b + a$, $a \cdot b = b \cdot a$.
4. Identities : $0, 1$ in $B$
   $a + 0 = a$, $a \cdot 1 = a$.
5. Distributive laws :
   $a + (b \cdot c) = (a + b) \cdot (a + c)$, $a \cdot (b + c) = a \cdot b + a \cdot c$.
6. Complement :
   $a + a' = 1$, $a \cdot a' = 0$. 
Logic Functions

\[ B = \{0,1\}, + = \text{OR}, \bullet = \text{AND}, \ ' = \text{NOT} \]

is a valid Boolean Algebra.

\[
\begin{array}{c|c|c|c}
00 & 0 & 00 & 0 \\
01 & 1 & 01 & 0 \\
10 & 1 & 10 & 0 \\
11 & 1 & 11 & 1 \\
\end{array}
\]

Do the axioms hold?

- Ex: communitive law: \(0 + 1 = 1 + 0\)?
Other logic functions of 2 variables (x,y)

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>f0</th>
<th>f1</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 1 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 1 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 1 1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Look at NOR and NAND:

- **Theorem**: Any Boolean function that can be expressed as a truth table can be expressed using NAND and NOR.
  - Proof sketch:
    - = NOT
    - = AND
    - = OR
  - How would you show that either NAND or NOR is sufficient?
Laws of Boolean Algebra

Duality: A dual of a Boolean expression is derived by interchanging OR and AND operations, and 0s and 1s (literals are left unchanged).

\[ \{ F ( x_1, x_2, ..., x_n, 0, 1, +, \cdot ) \}^D = \{ F ( x_1, x_2, ..., x_n, 1, 0, \cdot, + ) \} \]

Any law that is true for an expression is also true for its dual.

Operations with 0 and 1:
1. \( x + 0 = x \) \( x * 1 = x \)
2. \( x + 1 = 1 \) \( x * 0 = 0 \)

Idempotent Law:
3. \( x + x = x \) \( x * x = x \)

Involution Law:
4. \( (x')' = x \)

Laws of Complementarity:
5. \( x + x' = 1 \) \( x * x' = 0 \)

Commutative Law:
6. \( x + y = y + x \) \( x * y = y * x \)
Laws of Boolean Algebra (cont.)

Associative Laws:
\[(x + y) + z = x + (y + z)\]  \[x \cdot y \cdot z = x \cdot (y \cdot z)\]

Distributive Laws:
\[x \cdot (y + z) = (x \cdot y) + (x \cdot z)\]  \[x + (y \cdot z) = (x + y) \cdot (x + z)\]

“Simplification” Theorems:
\[x \cdot y + x \cdot y' = x\]  \[(x + y) \cdot (x + y') = x\]
\[x + x \cdot y = x\]  \[x \cdot (x + y) = x\]

DeMorgan’s Law:
\[(x + y + z + \ldots)' = x' \cdot y' \cdot z'\]  \[(x \cdot y \cdot z \cdot \ldots)' = x' + y' + z'\]

Theorem for Multiplying and Factoring:
\[(x + y) \cdot (x' + z) = x \cdot z + x' \cdot y\]

Consensus Theorem:
\[x \cdot y \cdot z + x' \cdot z = (x + y) \cdot (y + z) \cdot (x' + z)\]
\[x \cdot y + x' \cdot z = (x + y) \cdot (x' + z)\]
Proving Theorems via axioms of Boolean Algebra

Ex: prove the theorem: \( x y + x y' = x \)
\[ x y + x y' = x (y + y') \text{ distributive law} \]
\[ x (y + y') = x (1) \text{ complementary law} \]
\[ x (1) = x \text{ identity} \]

Ex: prove the theorem: \( x + x y = x \)
\[ x + x y = x 1 + x y \text{ identity} \]
\[ x 1 + x y = x (1 + y) \text{ distributive law} \]
\[ x (1 + y) = x (1) \text{ identity} \]
\[ x (1) = x \text{ identity} \]
**DeMorgan’s Law**

\[
(x + y)' = x' \cdot y'
\]

\[
(x \cdot y)' = x' + y'
\]

*DeMorgan’s Law can be used to convert AND/OR expressions to OR/AND expressions:*

Example: \( z = a'b'c + a'bc + ab'c + abc' \)

\[ z' = a'b'c' + a'b'c + ab'c' + abc \]
Algebraic Simplification

Ex: full adder (FA) carry out function
Cout = a’bc + ab’c + abc’ + abc
Algebraic Simplification

Ex: full adder (FA) carry out function

\[ \text{Cout} = a'bc + ab'c + abc' + abc \]

\[ = a'bc + ab'c + abc' + \text{abc} + \text{abc} \]
\[ = a'bc + \text{abc} + ab'c + abc' + \text{abc} \]
\[ = (a' + a)bc + ab'c + abc' + \text{abc} \]
\[ = (1)bc + ab'c + abc' + \text{abc} \]
\[ = bc + ab'c + abc' + \text{abc} + \text{abc} \]
\[ = bc + ab'c + abc + abc' + abc \]
\[ = bc + a(b' + b)c + abc' + abc \]
\[ = bc + a(1)c + abc' + abc \]
\[ = bc + ac + ab(c' + c) \]
\[ = bc + ac + ab(1) \]
\[ = bc + ac + ab \]