Boolean Algebra and Logic Synthesis

Reminders:
- Lab 2 due Thursday
- Bash tutorial in lab on Thursday 7:30pm
A combinational device is a circuit element that has

- one or more digital inputs
- one or more digital outputs
- a functional specification that details the value of each output for every possible combination of valid input values
- a timing specification consisting (at a minimum) of a propagation delay ($t_{PD}$): an upper bound on the required time to produce valid, stable output values from an arbitrary set of valid, stable input values

Static discipline

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Output a “1” if at least 2 out of 3 of my inputs are a “1”. Otherwise, output “0”.</td>
</tr>
<tr>
<td>B</td>
<td>I will generate a valid output in no more than 2 minutes after seeing valid inputs</td>
</tr>
<tr>
<td>C</td>
<td>output Y</td>
</tr>
</tbody>
</table>
Reminder: Composing Combinational Devices

A set of interconnected elements is a combinational device if

– each circuit element is combinational
– every input is connected to exactly one output or to a constant (0 or 1)
– the circuit contains no directed cycles

Cycles can cause feedback loops that prevent output from reaching a known or stable value!

Example:
Functional Specifications

- There are many ways to specify the function of a combinational device

- We will use two systematic approaches:
  - Truth tables enumerate the output values for all possible combinations of input values
  - Boolean expressions are equations containing binary (0/1) variables and three operations: AND (·), OR (+), and NOT (overbar)

\[ Y = \overline{C} \cdot A + C \cdot B \]

Any combinational function can be specified as a truth table or Boolean expression.
Timing Specifications

- Propagation delay ($t_{PD}$): An upper bound on the delay from valid inputs to valid outputs.

- Contamination delay ($t_{CD}$): A lower bound on the delay from invalid inputs to invalid outputs.
  - Used later (for sequential logic), can ignore for now.

Goal: Minimize $t_{PD}$!
The Combinational Contract

A  B
0  1
1  0

$t_{PD}$ propagation delay
$t_{CD}$ contamination delay

A

B

Must be $\geq t_{CD}$
Must be $\leq t_{PD}$

No promises during $\cdots$
Boolean Algebra

How to interpret and manipulate Boolean expressions
**Boolean Algebra**

- **Boolean algebra comprises**
  - Two elements, 0 and 1
  - Two binary operators, AND (·) and OR (+)
  - One unary operator, NOT (overbar)

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

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

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

- **All of Boolean algebra can be derived from the definitions of AND, OR, and NOT**
Boolean Algebra Axioms

- Instead of using truth tables to define AND, OR, and NOT, we can derive all of Boolean algebra using a small set of axioms:
  
  **identity**  
  \[ a \cdot 1 = a \]  
  \[ a + 0 = a \]

  **null**  
  \[ a \cdot 0 = 0 \]  
  \[ a + 1 = 1 \]

  **negation**  
  \[ \overline{0} = 1 \]  
  \[ \overline{1} = 0 \]

- **Duality principle**: If a Boolean expression is true, then replacing 0 ↔ 1 and AND ↔ OR yields another expression that is true
  
  - This principle holds for the axioms → Holds for all expressions
  - Halves the number of expressions you have to learn 😊
**Useful Boolean Algebra Properties**

- **Using the axioms**, we can derive several useful properties to manipulate and simplify Boolean expressions:

<table>
<thead>
<tr>
<th>Property</th>
<th>Formula 1</th>
<th>Formula 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Commutative</td>
<td>$a \cdot b = b \cdot a$</td>
<td>$a+b = b+a$</td>
</tr>
<tr>
<td>Associative</td>
<td>$a \cdot (b \cdot c) = (a \cdot b) \cdot c$</td>
<td>$a+(b+c) = (a+b)+c$</td>
</tr>
<tr>
<td>Distributive</td>
<td>$a \cdot (b+c) = a \cdot b + a \cdot c$</td>
<td>$a+b \cdot c = (a+b) \cdot (a+c)$</td>
</tr>
<tr>
<td>Complements</td>
<td>$a \cdot \overline{a} = 0$</td>
<td>$a + \overline{a} = 1$</td>
</tr>
<tr>
<td>Absorption</td>
<td>$a \cdot (a+b) = a$</td>
<td>$a + a \cdot b = a$</td>
</tr>
<tr>
<td>Reduction</td>
<td>$a \cdot b + a \cdot \overline{b} = a$</td>
<td>$(a+b) \cdot (a+\overline{b}) = a$</td>
</tr>
<tr>
<td>DeMorgan’s Law</td>
<td>$\overline{a \cdot b} = \overline{a} + \overline{b}$</td>
<td>$\overline{a+b} = \overline{a} \cdot \overline{b}$</td>
</tr>
</tbody>
</table>
Useful Boolean Algebra Properties

- Many of these properties are easy to remember because they match the ones for integer algebra, but be aware of the differences
  - e.g., distributive property for Boolean “+” \( a + b \cdot c = (a+b)(a+c) \) does not hold for integer “+”!

- To familiarize yourself with the properties, we recommend that you simply prove them
  - *Example: DeMorgan’s Law*

<table>
<thead>
<tr>
<th></th>
<th>( a )</th>
<th>( b )</th>
<th>( \overline{a \cdot b} )</th>
<th>( \overline{a} + \overline{b} )</th>
</tr>
</thead>
<tbody>
<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>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Equivalence and Normal Form

- Given a truth table, it is easy to derive an equivalent Boolean expression: write a sum of products where each product term covers a single 1 in the truth table.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th>Y</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>

\[ Y = \overline{C}BA + \overline{C}BA + CB\overline{A} + CBA \]

- This representation is called the function’s normal form.
  - It is unique, but there may be simpler expressions.
Logic Synthesis

Building logic circuits from Boolean expressions
A logic diagram represents a Boolean expression as a circuit schematic with logic gates and wires.

Basic logic gates:

- **AND**
  
  \[ Z = A \cdot B \]

- **OR**
  
  \[ Z = A + B \]

- **Inverter**
  
  \[ Z = \overline{A} \]

We often use AND and OR gates with more than two inputs.

AND, OR, and NOT are universal: They can implement any combinational function.

Why?
Straightforward Logic Synthesis

- We can implement any sum-of-products (SOP) Boolean expression with three levels of gates:
  1. Inverters
  2. ANDs
  3. OR

- However, we can often implement the same function with fewer gates. This requires simplifying its Boolean expression to use fewer operations.

\[ Y = \overline{CBA} + \overline{CBA} + CBA + \overline{CBA} \]
Boolean Simplification of SOPs

- A **minimal sum-of-products** is a sum-of-products expression that has the smallest possible number of AND and OR operators
  - Unlike the normal form, it is not unique (a function may have multiple minimal SOPs)
  - Minimal SOPs can be implemented with fewer gates

- Simple algebraic manipulation (using the properties we’ve seen) is sufficient to minimize small expressions (3-4 variables)

- More sophisticated techniques exist (e.g., K-maps), but we will not need them in this course

\[
Y = \overline{CBA} + CB\overline{A} + CBA + \overline{CBA}
\]

\[
Y = \overline{CBA} + CB + \overline{CBA} 
\]

\[
Y = \overline{CA} + CB
\]
Another way to reveal simplification is to rewrite the truth table using “don’t cares” (--, X, or ?) to indicate when the value of a particular input is irrelevant in determining the value of the output.

<table>
<thead>
<tr>
<th>C</th>
<th>B</th>
<th>A</th>
<th>Y</th>
<th>C</th>
<th>B</th>
<th>A</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>X</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>X</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>X</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>X</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>X</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>X</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note: Some input combinations (e.g., 000) are matched by more than one row in the “don’t care” table. It would be a bug if all matching rows didn’t specify the same output value!
Multi-Level Boolean Simplification

- We can often reduce the number of gates by using more logic levels than an SOP
  - Find common subexpressions and factor them out into independent variables
- Example:  \( F = A \cdot C + B \cdot C + A \cdot D + B \cdot D \) (minimal SOP)
  \[
  F = (A+B) \cdot C + (A+B) \cdot D \\
  \]
  \[
  X = A + B \\
  F = X \cdot C + X \cdot D \\
  \]
- Multi-level simplification has no well-defined optimum
  - Adding levels may reduce gates but increase delay
In practice, tools use Boolean simplification and other techniques to synthesize a circuit that meets certain area, delay, and power goals:

- High-level circuit specification (e.g., Boolean algebra, Minispec)
- Standard cell library (set of gates and their physical characteristics)
- Optimization goals (area/delay/power)
- Synthesis tool
- Optimized circuit implementation (using standard cell library gates)
Other Common Gates

- **XOR (Exclusive-OR)**

  \[ Z = A \oplus B = \bar{A} \cdot B + A \cdot \bar{B} \]

- **Inverting logic**

  - **NAND**
    \[ Z = A \cdot B \]

  - **NOR**
    \[ Z = \overline{A + B} \]
Universal Building Blocks

- NANDs and NORs are universal:

- Any logic function can be implemented using only NANDs (or, equivalently, NORs)
Standard Cell Library

- Library of gates and their physical characteristics
- Example:

<table>
<thead>
<tr>
<th>Gate</th>
<th>Delay (ps)</th>
<th>Area (µ²)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inverter</td>
<td>20</td>
<td>10</td>
</tr>
<tr>
<td>Buffer</td>
<td>40</td>
<td>20</td>
</tr>
<tr>
<td>AND2</td>
<td>50</td>
<td>25</td>
</tr>
<tr>
<td>NAND2</td>
<td>30</td>
<td>15</td>
</tr>
<tr>
<td>OR2</td>
<td>55</td>
<td>26</td>
</tr>
<tr>
<td>NOR2</td>
<td>35</td>
<td>16</td>
</tr>
<tr>
<td>AND4</td>
<td>90</td>
<td>40</td>
</tr>
<tr>
<td>NAND4</td>
<td>70</td>
<td>30</td>
</tr>
<tr>
<td>OR4</td>
<td>100</td>
<td>42</td>
</tr>
<tr>
<td>NOR4</td>
<td>80</td>
<td>32</td>
</tr>
</tbody>
</table>

Observations:
1. In current technology (CMOS), inverting gates are faster and smaller
2. Delay and area grow with number of inputs
**Design Tradeoffs: Delay vs Size**

**AND4:**
\[ t_{PD} = 90 \text{ ps}, \text{ size} = 40\mu^2 \]

**NAND4 + INV:**
\[ t_{PD} = 90 \text{ ps}, \text{ size} = 40\mu^2 \]

**Demorgan’s Laws:**
\[ \overline{A \cdot B} = \overline{A} + \overline{B} \]
\[ \overline{A + B} = \overline{A} \cdot \overline{B} \]

**2*NAND2 + NOR2:**
\[ t_{PD} = 1 \text{ NAND2 + NOR2} = 65 \text{ ps}, \text{ size} = 2 \text{ NAND2 + NOR2} = 46\mu^2 \]
Example: Mapping a Circuit to a Standard Cell Library

Find an implementation of a circuit, e.g.,

Using gates from a standard cell library, e.g.,

That optimizes for some goal, e.g., minimum area
Example: Mapping a Circuit to a Standard Cell Library

Possible implementations:

7 NAND2 (3) = 21
5 INV (2) = 10

Total area cost: 31

2 INV = 4
2 NAND2 = 6
1 NAND3 = 4
1 NAND4 = 5

Total area cost: 19
Logic Optimization Takeaways

- Synthesizing an optimized circuit is a very complex problem
  - Boolean simplification
  - Mapping to cell libraries with many gates
  - Multidimensional tradeoffs (e.g., minimize area-delay-power product)

- Infeasible to do by hand for all but the smallest circuits!

- Instead, hardware designers write circuits in a hardware description language, and use a synthesis tool to derive optimized implementations
Summary

- Any combinational (Boolean) function can be specified by a truth table or a Boolean expression (binary literals and AND, OR, NOT, which form a Boolean algebra)

- Any combinational function can be expressed as a sum-of-products (SOP) and implemented with three levels of logic gates (NOTs, ANDs, OR)

- Boolean simplification (finding a minimal SOP, multi-level simplification) results in simpler circuits

- There are MANY design tradeoffs in mapping Boolean functions to gates. We will use synthesis tools to find optimized circuit implementations
Thank you!

Next lecture: CMOS