Reminders:
Lab 1 due today by 11:59pm Boston time
Post lecture questions due by 10AM tomorrow
No lecture on Tuesday due to student holiday
No recitation on Wednesday
Lab 1 checkoff due Wednesday

Enjoy your long weekend!
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

Output a “1” if at least 2 out of 3 of my inputs are a “1”. Otherwise, output “0”.

I will generate a valid output in no more than 2 nanoseconds after seeing valid inputs.
Digital Inputs and Outputs

Different specifications for inputs and outputs

- Digital output: “0” ≤ \( V_{OL} \), “1” ≥ \( V_{OH} \)
- Digital input: “0” ≤ \( V_{IL} \), “1” ≥ \( V_{IH} \)
- \( V_{OL} < V_{IL} < V_{IH} < V_{OH} \)

A digital device accepts marginal inputs and provides unquestionable outputs (to leave room for noise).
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 (\(\cdot\)), 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 Specification

Each combinational element has propagation delay $t_{PD}$ measured in (ns).

What is $t_{PD}$ of combinational device?

$t_{PD}$ of longest input to output path

$t_{PD} = 5 + 2 + 3 = 10\text{ns}$

$t_{PD} = 5 + 6 = 11\text{ns}$
Boolean Algebra

How to interpret and manipulate Boolean expressions
Boolean Algebra

- Boolean algebra comprises
  - Two elements, 0 and 1
  - Two binary operators, AND (\(\cdot\)) and OR (\(+\))
  - One unary operator, NOT (\(\overline{a}\))

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

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

March 4, 2021
MIT 6.004 Spring 2021
L06-7
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:

<table>
<thead>
<tr>
<th>Identity</th>
<th>a \cdot 1 = a</th>
<th>a + 0 = a</th>
</tr>
</thead>
<tbody>
<tr>
<td>Null</td>
<td>a \cdot 0 = 0</td>
<td>a + 1 = 1</td>
</tr>
<tr>
<td>Negation</td>
<td>\overline{0} = 1</td>
<td>\overline{1} = 0</td>
</tr>
</tbody>
</table>

- 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:

  - **commutative**
    \[ a \cdot b = b \cdot a \]
    \[ a + b = b + a \]

  - **associative**
    \[ a \cdot (b \cdot c) = (a \cdot b) \cdot c \]
    \[ a + (b + c) = (a + b) + c \]

  - **distributive**
    \[ a \cdot (b + c) = a \cdot b + a \cdot c \]
    \[ a + b \cdot c = (a + b) \cdot (a + c) \]

  - **complements**
    \[ a \cdot \overline{a} = 0 \]
    \[ a + \overline{a} = 1 \]

  - **absorption**
    \[ a \cdot (a + b) = a \]
    \[ a + a \cdot b = a \]

  - **reduction**
    \[ a \cdot b + a \cdot \overline{b} = a \]
    \[ (a + b) \cdot (a + \overline{b}) = a \]

  - **DeMorgan’s Law**
    \[ \overline{a \cdot b} = \overline{a} + \overline{b} \]
    \[ \overline{a + b} = \overline{a} \cdot \overline{b} \]
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 “+”
    \[ a + b \cdot c = (a+b) \cdot (a+c) \]
    does not hold for integer “+”!

- To familiarize yourself with the properties, we recommend that you simply prove them

  - \textit{Example: DeMorgan’s Law}

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>a \cdot b</th>
<th>\overline{a} + \overline{b}</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</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 product terms where each term covers a single 1 in the truth table.

<table>
<thead>
<tr>
<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>
</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>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

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

- This representation is called the function’s **normal form**.
  - It is unique, but there may be simpler expressions.
- Corollary: Boolean expressions can represent any combinational function.
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?
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.

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

### Truth Tables with “Don’t Cares”

<table>
<thead>
<tr>
<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>
</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>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

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

\[
\begin{align*}
F &= (A+B) \cdot C + (A+B) \cdot D \\
X &= A + B \\
F &= X \cdot C + X \cdot D
\end{align*}
\]

- Multi-level simplification has no well-defined optimum
  - Adding levels may reduce gates but increase delay
Logic Optimization

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**
  \[ Z = \bar{A} + \bar{B} \]
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:
\[ t_{PD} = 1 \text{ NAND2} + \text{ NOR2} = 65 \text{ ps}, \text{ size} = 2 \text{ NAND2} + \text{ NOR2} = 46\mu^2 \]
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.,

Area    2        3         4           5

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 Technology