Where We Are At

- We have learned the abstract interface presented by a machine: the instruction set architecture
- What we will learn: the implementation behind the interface:
  - Start with switching devices (such as transistors)
  - Build logic gates with transistors
  - Build combinational circuit (memory-less) devices using gates
  - Next lecture: build sequential circuit (memory) devices
  - The one after: glue these devices into a computer

Digital Systems

- ... however, the application of digital logic extends way beyond just computers.
- Today, digital systems are replacing all kinds of analog systems in life (data processing, control systems, communications, measurement, ...)
- What is a digital system?
  - Digital: quantities or signals only assume discrete values
  - Analog: quantities or signals can vary continuously
- Why digital systems?
  - Greater accuracy and reliability
The heart of a digital system is usually a digital logic circuit.

A smallest useful circuit is a logic gate.
We will connect these small gates into larger circuits.
Building Circuits Using Gates

• Can implement any circuit using only AND, OR, and NOT gates
• But things get complicated when we have lots of inputs and outputs...

Problems

• Many different ways of implementing a circuit (the two above circuits turn out to be the same!)
• How do we find the best implementation? Need better formalism
• Also need more compact representation
• This leads to the study of boolean algebra

Outline

• Introduction
• Logic gates
• Boolean algebra
• Implementing gates with switching devices
• Common combinational devices
• Conclusions

Boolean Algebra

• History
  • Developed in 1847 by Boole to solve mathematic logic problems
  • Shannon first applied it to digital logic circuits in 1939
• Basics
  • Boolean variables: variables whose values can be 0 or 1
  • Boolean functions: functions whose inputs and outputs are boolean variables
• Relationship with logic circuits
  • Boolean variables correspond to signals
  • Boolean functions correspond to circuits
Defining a Boolean Function with a Truth Table

<table>
<thead>
<tr>
<th>x</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>AND (x, y)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

A systematic way of specifying a function value for all possible combination of input values

A function that takes 2 inputs has 2x2 columns

A function that takes n inputs has 2^n columns

This particular example is the AND-function

OR and NOT Truth Tables

<table>
<thead>
<tr>
<th>x</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>OR (x, y)</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>x</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOT (x)</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Defining a General Boolean Function Using Three Basic Boolean Functions

\[ \text{AND} (x, y) = xy = x*y \]
\[ \text{OR} (x, y) = x+y \]
\[ \text{NOT} (x) = x' \]

The three basic functions have short-hand notations

Can compose the three basic boolean functions to form arbitrary boolean functions [such as \( g(x, y) = xy + z' \)]

Two Ways of Defining a Boolean Function

<table>
<thead>
<tr>
<th>x</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>XOR (x, y) = x^y</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

\[ \text{XOR} (x, y) = x^y = x' y + xy' \]

We have learned that any function can be defined in these two ways: truth table and composition of basic functions

Why do we need all these different representations?

- Some are easier than others to begin with to design a circuit
- Usually start with truth table (or variants of it)
- Derive a boolean expression from it (perhaps including simplification)
- Straightforward transformation from boolean expression to circuit
More Examples of Boolean Functions

Gluing the truth tables of all functions of two variables into one table

For n variables, there are a total of $2^n$ functions!

CS126 11-16 Randy Wang

So How to Translate a Truth Table to a Boolean Expression (Sum-of-Products)?

CS126 11-17 Randy Wang

Another Example

Parity Function Construction Demo

CS126 11-18 Randy Wang

CS126 11-19 Randy Wang
Transform a Boolean Expression into a Boolean Circuit

Use sum-of-products form of function

Example: majority

\[ m = x'y'z + x'y'z + xyz' + xyz \]

Simplification Using Boolean Algebra

- Large body of boolean algebra laws can be employed to simplify circuits
- The previous example:

\[ xy + xy' = x(y+y') = x*1 = x \]

- Much more, but you don’t have to know any of this...

Mini-Summary:
How Do We Make a Combinational Circuit

- Represent input signals with input boolean variables, represent output signals with output boolean variables
- Construct truth table based on what we want the circuit to do
- Derive (simplified) boolean expression from the truth table
- Transform boolean expression into a circuit by replacing basic boolean functions with primitive gates

Outline

- Introduction
- Logic gates
- Boolean algebra
  - Implementing gates with switching devices
- Common combinational devices
- Conclusions
Switching Devices

- Any two-state device can be a switching device, examples are relays, diodes, transistors, and magnetic cores
- A transistor example
- Any boolean function can be implemented by wiring together transistors

Make a NOT-gate Using a Transistor

\[ O = MC' = 1 \times x' = x' \]

Make an OR-gate Using Transistors

\[(x' y')' = x + y \quad \text{(DeMorgan's Law)}\]

Make an AND-gate Using Transistors

\[ y(x'') = xy \]
Outline

• Introduction
• Logic gates
• Boolean algebra
• Implementing gates with switching devices
• **Common combinational devices**
• Conclusions

Decoder Interface

```
3-8 decoder
x y z
```

Decoder Boolean Expressions

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>z</th>
<th>d0</th>
<th>d1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</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>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>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>z</th>
<th>d2</th>
<th>d3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</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>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

\[ d_0 = x'y'z' \]
\[ d_1 = x'y'z \]

Decoder Implementation
Decoder Demo

Multiplexer Interface

- I₀ - I₇ are the “data inputs”, x, y, z form the “control” inputs and are interpreted together as one binary number.
- One data input is selected by the control and becomes output.
- For example, if x, y, z are 1, 0, 1, then M = I₅

Multiplexer Boolean Expression

\[ M = x'y'z'I₀ + x'y'zI₁ + x'y'zI₂ + x'y'zI₃ + x'y'zI₄ + x'y'zI₅ + xy'zI₆ + xyzI₇ \]

- A lot easier in this case to directly derive the boolean expression instead of starting with a truth table.

Multiplexer Implementation
**An Adder Bit-Slice Interface**

- Add three 1-bit numbers x, y, z
- s is the 1-bit sum
- c is the 1-bit carry

**An Adder Bit-Slice Implementation**

- See slides 11-16, 11-17, and 11-18 for details of the odd parity circuit and majority circuit

**An N-bit Adder Made with Bit-Slices**

**Outline**

- Introduction
- Logic gates
- Boolean algebra
- Implementing gates with switching devices
- Common combinational devices
- Conclusions
Abstractions and Encapsulation

All the lessons we learned for ADT apply here to hardware as well!

Building a Computer Bottom Up

• **Circuit design**: specifying the interconnection of components such as resistors, diodes, and transistors to form logic building blocks

• **Logic design**: determining how to interconnect logic building blocks such as logic gates and flip-flops to form subsystems

• **System design** (or computer architecture): specifying the number, type, and interconnection of subsystems such as memory units, ALUs, and I/O devices

What We Have Learned

• How to build basic gates using transistors

• How to build a combinational circuit
  - Truth table
  - Sum-of-product boolean expression
  - Transform a boolean expression into a circuit of basic gates

• The functionality of some common devices and how they are made
  - Decoder
  - Multiplexer
  - Bit-slice adder

• You’re **not** responsible for
  - Boolean algebra laws, or circuit simplification