6. Combinational Circuits

Digital Circuits

What is a digital system?
- Digital: signals are 0 or 1.
- Analog: signals vary continuously.

Why digital systems?
- Accuracy and reliability.
- Staggeringly fast and cheap.

Basic abstractions.
- On, off.
- Wire: propagates on/off value.
- Switch: controls propagation of on/off values through wires.

Digital circuits and you.
- Computer microprocessors.
- Antilock brakes, cell phones, iPods, etc.

Wires

Wires.
- On (1): connected to power.
- Off (0): not connected to power.
- If a wire is connected to a wire that is on, that wire is also on.
- Typical drawing convention: "flow" from top, left to bottom, right.

This lecture. Boolean circuits.

Ahead. Putting it all together and building a TOY machine.
**Controlled Switch**

Controlled switch. [relay implementation]

- 3 connections: input, output, control.
- Magnetic force pulls on a contact that cuts electrical flow.
- Control wire affects output wire, but output does not affect control; establishes forward flow of information over time.

**Circuit Anatomy**

Controlled Switch

**Layers of Abstraction**

Layers of abstraction.

- Circuits are build from wires and switches. (implementation)
- A circuit is defined by its inputs and outputs. (interface)
- To control complexity, we encapsulate circuits. (ADT)
Logic Gates: Fundamental Building Blocks

**NOT** = \( x' \)

\[
\begin{array}{c|c|c}
 x & \text{NOT} & x' \\
 0 & 1 & x' \\
 1 & 0 & x' \\
\end{array}
\]

**OR** = \( x + y \)

\[
\begin{array}{c|c|c}
 x & y & \text{OR} \\
 0 & 0 & 0 \\
 0 & 1 & y \\
 1 & 0 & x \\
 1 & 1 & x + y \\
\end{array}
\]

**AND** = \( xy \)

\[
\begin{array}{c|c|c}
 x & y & \text{AND} \\
 0 & 0 & 0 \\
 0 & 1 & 0 \\
 1 & 0 & 0 \\
 1 & 1 & xy \\
\end{array}
\]

**Multiway Gates**

- **OR**: 1 if any input is 1; 0 otherwise.
- **AND**: 1 if all inputs are 1; 0 otherwise.
- **Generalized**: negate some inputs.

Multiway gates.
Boolean Algebra

History.
- Developed by Boole to solve mathematical logic problems (1847).
- Shannon master’s thesis applied it to digital circuits (1937).

Basics.
- Boolean variable: value is 0 or 1.
- Boolean function: function whose inputs and outputs are 0, 1.

Relationship to circuits.
- Boolean variables: signals.
- Boolean functions: circuits.

Truth Table for Functions of 2 Variables

Truth table.
- 16 Boolean functions of 2 variables.

Truth Table

Truth Table for Functions of 3 Variables

Truth table.
- 16 Boolean functions of 2 variables.
- 256 Boolean functions of 3 variables.
- $2^2(2^3)$ Boolean functions of $n$ variables!
Universality of AND, OR, NOT

Fact. Any Boolean function can be expressed using AND, OR, NOT.
- \{ \text{AND}, \text{OR}, \text{NOT} \} are \text{universal}.
- Ex: \text{XOR}(x, y) = xy' + x'y.

<table>
<thead>
<tr>
<th>Notation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>x'y</td>
<td>NOT x</td>
</tr>
<tr>
<td>xy</td>
<td>x AND y</td>
</tr>
<tr>
<td>x + y</td>
<td>x OR y</td>
</tr>
</tbody>
</table>

Expressing XOR Using AND, OR, NOT

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>x'</th>
<th>y'</th>
<th>xy'</th>
<th>x'y'</th>
<th>x OR y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</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>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<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>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Exercise. Show \{ \text{AND, NOT} \}, \{ \text{OR, NOT} \}, \{ \text{NAND} \}, \{ \text{AND, XOR} \} are universal.

Hint. DeMorgan’s law: \((x'y')' = x + y\).

Sum-of-Products

Sum-of-products. Systematic procedure for representing a Boolean function using AND, OR, NOT.
- Form AND term for each 1 in Boolean function.
- OR terms together.

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>z</th>
<th>MAJ</th>
<th>x'y'z</th>
<th>xy'z</th>
<th>xyz</th>
<th>x'y'z + xy'z + xyz' + xyz</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>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<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>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</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>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<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>

Expressing MAJ using sum-of-products

Translate Boolean Formula to Boolean Circuit

Sum-of-products. XOR.

\[
\text{XOR} = x'y + xy'
\]

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>XOR</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>

Truth table  Circuit

Translate Boolean Formula to Boolean Circuit

Sum-of-products. XOR.

\[
\text{XOR} = x'y + xy'
\]

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>XOR</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>

Truth table  Abstract circuit  Circuit
Translate Boolean Formula to Boolean Circuit

**Sum-of-products. XOR.**

\[ \text{XOR} = x'y + xy' \]

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>XOR</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>

**Truth table**  
**Abstract circuit**  
**Circuit**

Translating the formula into a circuit involves designing a circuit that implements the logic of the formula. The XOR function is typically realized using logic gates such as AND, OR, and NOT gates. In the abstract circuit, we can see how the input variables (x, y) are manipulated through these gates to produce the output (XOR).

---

**Sum-of-products. Majority.**

\[ \text{MAJ} = x'y'z + xy'z + xyz' + xyz \]

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>z</th>
<th>MAJ</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>0</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>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</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>

**Truth table**  
**Abstract circuit**  
**Circuit**

Majority functions are often implemented using a combination of AND, OR, and NOT gates. The truth table shows all possible input combinations and their corresponding outputs, which guide the design of the circuit.
ODD Parity Circuit

ODD(x, y, z).
- 1 if odd number of inputs are 1.
- 0 otherwise.

Ex. \( \text{MAJ}(x, y, z) = x'y'z + xy'z + xz' = xy + yz + xz \).

Expressing ODD using sum-of-products:

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>z</th>
<th>ODD</th>
<th>x'y'z</th>
<th>xy'z</th>
<th>xyz</th>
<th>x'y'z + xy'z + xz' + xyz</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>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</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>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Expressing ODD using sum-of-products.
Let's Make an Adder Circuit

**Goal.** \( x + y = z \) for 4-bit integers.

- We build 4-bit adder: 9 inputs, 4 outputs.
- Same idea scales to 128-bit adder.
- Key computer component.

**Step 1.** Represent input and output in binary.

<table>
<thead>
<tr>
<th>Bin</th>
<th>Hex</th>
<th>Dec</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1111</td>
<td>F</td>
<td>15</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bin</th>
<th>Hex</th>
<th>Dec</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1111</td>
<td>F</td>
<td>15</td>
</tr>
</tbody>
</table>

**Step 2.** (do one bit at a time)
- Build truth table for carry bit.
- Build truth table for summand bit.

---

Let's Make an Adder Circuit

**Goal.** \( x + y = z \) for 4-bit integers.

**Step 2.** (first attempt)
- Build truth table.
- Why is this a bad idea?
  - 128-bit adder: \( 2^{256-1} \) rows > # electrons in universe!
Let's Make an Adder Circuit

Goal. \( x + y = z \) for 4-bit integers.

Step 3.
- Derive (simplified) Boolean expression.

<table>
<thead>
<tr>
<th>Carry Bit</th>
<th>Summand Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x_i )</td>
<td>( y_i )</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Derive (simplified) Boolean expression:

\[
\begin{array}{c}
\text{C}_{\text{out}} \\
\begin{array}{cc}
\times_3 & \times_2 \\
\times_1 & \times_0 \\
\end{array} \\
\begin{array}{cc}
+ & \ \\
\times_3 & \times_2 \\
\times_1 & \times_0 \\
\end{array} \\
\begin{array}{c}
z_3 \\
z_2 \\
z_1 \\
z_0 \\
\end{array}
\end{array}
\]

Step 4.
- Transform Boolean expression into circuit.
- Chain together 1-bit adders.

Adder: Interface

Adder: Component Level View
Adder: Switch Level View

carry in

\[ x_3 \quad y_3 \quad x_2 \quad y_2 \quad x_1 \quad y_1 \quad x_0 \quad y_0 \]

carry out

\[ Z_3 \quad Z_2 \quad Z_1 \quad Z_0 \]

Decoder

- [n-bit]
  - n address inputs, \(2^n\) data outputs.
  - Addressed output bit is 1; others are 0 register.

Shifter

Ex. Put in a binary amount to shift.
Arithmetic Logic Unit

Arithmetic logic unit (ALU). Computes all operations in parallel.
- Add and subtract.
- Xor.
- And.
- Shift left or right.

Q. How to select desired answer?

1 Hot OR

1 hot OR.
- All devices compute their answer; we pick one.
- Exactly one select line is on.
- Implies exactly one output line is relevant.

Device Interface Using Buses

Device. Processes a word at a time.
- Input bus. Wires on top.
- Output bus. Wires on bottom.
- Control. Individual wires on side.
Summary

Lessons for software design apply to hardware design!
- Interface describes behavior of circuit.
- Implementation gives details of how to build it.

Layers of abstraction apply with a vengeance!
- On/off.
- Controlled switch (relay or transistor).
- Gates (AND, OR, NOT).
- Boolean circuit (MAJ, ODD).
- Adder.
- Shifter.
- Arithmetic logic unit.
- ...
- TOY machine.