6.1 Combinational Circuits

Signals and Wires

Digital signals
- Binary (or "logical") values: 1 or 0, on or off, high or low voltage

Wires
- Propagate digital signals from place to place.
- Signals "flow" from left to right.
  - A drawing convention, sometimes violated
  - Actually: flow from producer to consumer(s) of signal

Logic Gates

Logical gates.
- Fundamental building blocks.

NOT

AND

OR

Multiway AND Gates

AND(x₀, x₁, x₂, x₃, x₄, x₅, x₆, x₇).
- 1 if all inputs are 1.
- 0 otherwise.
Multiway OR Gates

\[ \text{OR}(x_0, x_1, x_2, x_3, x_4, x_5, x_6, x_7). \]
- 1 if at least one input is 1.
- 0 otherwise.

Boolean Algebra

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

"possibly the most important, and also the most famous, master’s thesis of the [20th] century" -- Howard Gardner

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

Truth table.
- Systematic method to describe Boolean function.
- One row for each possible input combination.
- N inputs \( \Rightarrow \) \( 2^N \) rows.

Truth Table for Functions of 2 Variables

Truth table.
- 16 Boolean functions of 2 variables.
- every 4-bit value represents one

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>ZERO</th>
<th>AND</th>
<th>x'</th>
<th>y'</th>
<th>XOR</th>
<th>OR</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</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>0</td>
<td>0</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>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>NOR</th>
<th>EQ</th>
<th>x'</th>
<th>y'</th>
<th>NAND</th>
<th>ONE</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</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>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>EQ</th>
<th>x'</th>
<th>y'</th>
<th>EQ'</th>
<th>XOR</th>
<th>OR</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</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>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</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>
<td>0</td>
</tr>
</tbody>
</table>
Truth Table for Functions of 3 Variables

- 16 Boolean functions of 2 variables.
  - every 4-bit value represents one
- 256 Boolean functions of 3 variables.
  - every 8-bit value represents one
- $2^{2^N}$ Boolean functions of $N$ variables!

### Some Functions of 3 Variables

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>z</th>
<th>AND</th>
<th>OR</th>
<th>MAJ</th>
<th>ODD</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>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<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>1</td>
<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>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<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>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

### Universality of AND, OR, NOT

Any Boolean function can be expressed using AND, OR, NOT.
- "Universal"
- Example: XOR($x, y$) = $xy' + x'y$

<table>
<thead>
<tr>
<th>Notation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x'$</td>
<td>NOT $x$</td>
</tr>
<tr>
<td>$x \cdot y$</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>$x'y'$</th>
<th>$xy'$</th>
<th>$x'y + xy'$</th>
<th>XOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</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>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</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>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

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

Hint. Use DeMorgan’s Law: $(xy)' = (x' + y')$ and $(x + y)' = (xy')$

### Sum-of-Products

Any Boolean function can be expressed using AND, OR, NOT.
- Sum-of-products is systematic procedure.
  - form AND term for each 1 in truth table of Boolean function
  - OR terms together

#### Expressing MAJ Using Sum-of-Products

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

### Translate Boolean Formula to Boolean Circuit

#### Use sum-of-products form.
- XOR($x, y$) = $xy' + x'y$. 

- 0 0 0 0 0
- 0 0 1 0 0
- 0 1 0 0 0
- 0 1 1 1 0
- 1 0 0 0 0
- 1 0 1 0 1
- 1 1 0 1 0
- 1 1 1 1 1
Translate Boolean Formula to Boolean Circuit

Use sum-of-products form.
- \( \text{MAJ}(x, y, z) = x'yz + xy'z + xyz' + xyz. \)

![Boolean Circuit Diagram]

Expressing a Boolean Function Using AND, OR, NOT

Ingredients.
- AND gates.
- OR gates.
- NOT gates.
- Wire.

Instructions.
- Step 1: represent input and output signals with Boolean variables.
- Step 2: construct truth table to carry out computation.
- Step 3: derive (simplified) Boolean expression using sum-of-products.
- Step 4: transform Boolean expression into circuit.

Simplification Using Boolean Algebra

Many possible circuits for each Boolean function.
- Sum-of-products not necessarily optimal in:
  - number of gates (space)
  - depth of circuit (time)

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

![Simplified Circuit Diagram]

ODD Parity Circuit

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

![ODD Parity Circuit Diagram]

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 + x'yz' + xy'z' + xyzz</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>
**ODD Parity Circuit**

**ODD**($x, y, z$).

- 1 if odd number of inputs are 1.
- 0 otherwise.

---

**Let's Make an Adder Circuit**

**Goal:** $x + y = z$ for 4-bit integers.

**Step 1.**

- Represent input and output in binary.

**Step 2. (first attempt)**

- Build truth table.
- Why is this a bad idea?
  - 128-bit adder: $2^{256+1}$ rows > # electrons in universe!

**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 3.
- Derive (simplified) Boolean expression.

### Carry Bit

<table>
<thead>
<tr>
<th>( x_i )</th>
<th>( y_i )</th>
<th>( c_i )</th>
<th>( c_{i+1} )</th>
<th>MAJ</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0</td>
<td>0 0 0 0</td>
<td>0 0 0 0</td>
<td>0 0 0 0</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td>0 0 1 0</td>
<td>0 0 1 0</td>
<td>0 0 1 0</td>
<td>0 0 1 0</td>
<td>0 0 1 0</td>
</tr>
<tr>
<td>0 1 0 0</td>
<td>0 1 0 0</td>
<td>0 1 0 0</td>
<td>0 1 0 0</td>
<td>0 1 0 0</td>
</tr>
<tr>
<td>0 1 1 1</td>
<td>0 1 1 1</td>
<td>0 1 1 1</td>
<td>0 1 1 1</td>
<td>0 1 1 1</td>
</tr>
<tr>
<td>1 0 0 0</td>
<td>1 0 0 0</td>
<td>1 0 0 0</td>
<td>1 0 0 0</td>
<td>1 0 0 0</td>
</tr>
<tr>
<td>1 0 1 1</td>
<td>1 0 1 1</td>
<td>1 0 1 1</td>
<td>1 0 1 1</td>
<td>1 0 1 1</td>
</tr>
<tr>
<td>1 1 0 1</td>
<td>1 1 0 1</td>
<td>1 1 0 1</td>
<td>1 1 0 1</td>
<td>1 1 0 1</td>
</tr>
<tr>
<td>1 1 1 1</td>
<td>1 1 1 1</td>
<td>1 1 1 1</td>
<td>1 1 1 1</td>
<td>1 1 1 1</td>
</tr>
</tbody>
</table>

### Summand Bit

<table>
<thead>
<tr>
<th>( x_i )</th>
<th>( y_i )</th>
<th>( z_i )</th>
<th>ODD</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0</td>
<td>0 0 0 0</td>
<td>0 0 0 0</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td>0 0 1 1</td>
<td>0 0 1 1</td>
<td>0 0 1 1</td>
<td>0 0 1 1</td>
</tr>
<tr>
<td>0 1 0 1</td>
<td>0 1 0 1</td>
<td>0 1 0 1</td>
<td>0 1 0 1</td>
</tr>
<tr>
<td>0 1 1 0</td>
<td>0 1 1 0</td>
<td>0 1 1 0</td>
<td>0 1 1 0</td>
</tr>
<tr>
<td>1 0 0 1</td>
<td>1 0 0 1</td>
<td>1 0 0 1</td>
<td>1 0 0 1</td>
</tr>
<tr>
<td>1 0 1 0</td>
<td>1 0 1 0</td>
<td>1 0 1 0</td>
<td>1 0 1 0</td>
</tr>
<tr>
<td>1 1 0 0</td>
<td>1 1 0 0</td>
<td>1 1 0 0</td>
<td>1 1 0 0</td>
</tr>
<tr>
<td>1 1 1 1</td>
<td>1 1 1 1</td>
<td>1 1 1 1</td>
<td>1 1 1 1</td>
</tr>
</tbody>
</table>

Let’s Make an Adder Circuit

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

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

Subtractor

Subtractor circuit: \( z = x - y \).
- One approach: design like adder circuit.
- Better idea: reuse adder circuit.
- 2’s complement: to negate an integer, flip bits, then add 1

4-Bit Subtractor Interface

4-Bit Subtractor Implementation
Arithmetic Logic Unit: Interface

ALU Interface.
- Add, subtract, bitwise and, bitwise xor, shift left, shift right, copy.
- Associate 3-bit integer with 5 primary ALU operations.
  - ALU performs operations in parallel
  - Control wires select which result ALU outputs

<table>
<thead>
<tr>
<th>op</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>+, -</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>&amp;</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>^</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>&lt;&lt;, &gt;&gt;</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>input 2</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

2^n-to-1 Multiplexer

- n = 8 for main memory
- n select inputs, 2^n data inputs, 1 output.
- Copies "selected" data input bit to output.

6.2: Sequential Circuits
### Sequential vs. Combinational Circuits

**Combinational circuits.**
- Output determined solely by inputs.
- Can draw solely with left-to-right signal paths.

**Sequential circuits.**
- Output determined by inputs AND previous outputs.
- Feedback loop.

### SR Flip-Flop

**SR Flip-Flop.**
- \( S = 1, R = 0 \) (set) \( \Rightarrow \) "Flips" bit on.
- \( S = 0, R = 1 \) (reset) \( \Rightarrow \) "Flops" bit off.
- \( S = R = 0 \) \( \Rightarrow \) Status quo.
- \( S = R = 1 \) \( \Rightarrow \) Not allowed.

**What is the value of Q if:**
- \( S = 1 \) and \( R = 0 \) ? \( \Rightarrow \) Q is surely 1.
- \( S = 0 \) and \( R = 1 \) ? \( \Rightarrow \) Q is surely 0.
- \( S = 0 \) and \( R = 0 \) ? \( \Rightarrow \) Q is possibly 0 . . . or possibly 1.

While \( S = R = 0 \), Q remembers what it was the last time S or R was 1.

### Clock

**Clock.**
- Fundamental abstraction.
  - regular on-off pulse
- External analog device.
- Synchronizes operations of different circuit elements.
- 1 GHz clock means 1 billion pulses per second.
**Clocked SR Flip-Flop**

- Same as SR flip-flop except S and R only active when clock is 1.

![Clocked SR Flip-Flop Diagram](image)

**Clocked D Flip-Flop**

- Output follows D input while clock is 1.
- Output is remembered while clock is 0.

![Clocked D Flip-Flop Diagram](image)

**Stand-Alone Register**

- k-bit register.
  - Stores k bits.
  - Register contents always available on output.
  - If write enable is asserted, k input bits get copied into register.

**Registers**

- TOY registers: fancy 16 x 16-bit register file.
  - Want to be able to read two registers, and write to a third in the same instructions: \( R_1 \leftarrow R_2 + R_3 \).
  - 3 address inputs, 1 data input, 2 data outputs.
  - Add decoders and muxes for additional ports.
Main Memory

TOY main memory: 256 x 16-bit register file.

The TOY Datapath

Control: controls components, enables connections.
- Input: opcode, clock, conditional evaluation. (green)
- Output: control wires. (orange)
Summary

Combinational circuits: how to compute things
- And, Or, Not primitives sufficient for any Boolean function
- Systematic method: truth tables and sum-of-products
- Examples
  - Majority
  - Binary adder
  - Multiplexor

Sequential circuits: where to put things
- Flip flop primitive holds one bit
- Many flip flops make a register (16 for TOY)
- Many registers make a register file
- Lots and lot of registers make a memory (256 for TOY)

A whole computer
- Uses combinational circuits to perform computations
- Uses sequential circuits to store results
- Uses a little of each for control

The final secret

All three of our logic primitives can be made using a single* type of electronic primitive: the transistor!

*not counting the passive resistors