Lecture A3: Boolean Circuits

George Boole
(1815 – 1864)

Claude Shannon
(1916 – 2001)

Digital Circuits

What is a digital system?

Why digital systems?

Basic abstraction.

- On, off.
- Switch that can turn something on or off.

Digital circuits and you.

- Computer microprocessors.
- Antilock brakes.
- VCR.
- Cell phone.

Abstract Switch

- Electrical (transistor).
- Electro-mechanical (relay).
- Hydraulic (water valve).

Hydraulic valve.

- 3 connections: input, output, control.
- Pressure on control pushes on a piston that turns on water flow from input to output.
  - valve is always entirely open or closed (digital)
  - amplification, restoring logic design
- Control pipe affects output pipe, but output does not affect control.
  - establishes forward flow of information over time

Computer Architecture

Lecture A1 – A2.
- TOY machine.

Lecture A3 – A4.
- Digital circuits.

Lecture A5.
- Putting it all together.
Hydraulic computer. *(The Pattern on the Stone)*
- Build computer by connecting hydraulic valves and pipe.

### Hydraulic Computer

- **Pipe:** Water pressure
- **Hydraulic valve:**
- **Connector:**
- **Signal:** Hydraulic valve
- **Switch:**

### Electrical Computer

- **Wire:** Voltage
- **Metal-oxide transistor:**

### Wires

- Wires.
  - Propagate logical values from place to place.
  - Signals "flow" from left to right and top to bottom.

### Logic Gates

- **Logical gates.** Fundamental building blocks.

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

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>xy</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</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>x</th>
<th>y</th>
<th>x + y</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>
**Multiway AND Gates**

\[ \text{AND}(x_0, x_1, x_2, x_3, x_4, x_5, x_6, x_7). \]

- 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 first applied 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**

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

**AND Truth Table**

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

**Example AND circuit diagram**
### Truth Table

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

#### Truth Table for All Boolean Functions of 2 Variables

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

#### Truth Table for All Boolean Functions of 2 Variables

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>NOR</th>
<th>EQ</th>
<th>y'</th>
<th>x'</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>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</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>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."
- XOR(x, y) = xy' + x'y

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

**Exercise:** \{AND, NOT\}, \{OR, NOT\}, \{NAND\}, \{AND, XOR\} are universal.

### 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 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>xyz</th>
<th>xy'z</th>
<th>xyz</th>
<th>x'y'z + xy'z' + 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>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>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>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</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>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>1</td>
<td>1</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.

Translate Boolean Formula to Boolean Circuit

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

Simplification Using Boolean Algebra

Many possible circuits for each Boolean function.
- Sum-of-products not necessarily optimal in:
  - number of gates
  - depth of circuit
- MAJ(x, y, z) = x'yz + xy'z + xyz' + xyz = xy + yz + xz.

Recipe for Making Combinational Circuit

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.
**ODD Parity Circuit**

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

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

**Let's Make an Adder Circuit**

**Goal:** \( x + y = z \).

- We build 4-bit adder: 8 inputs, 4 outputs.
  - (Same idea scales to 128-bit adder.)
- Key computer component.

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

**Step 2.**
- Build truth table.
- Why is this a bad idea?

**4-Bit Adder Truth Table**

<table>
<thead>
<tr>
<th>( c_0 )</th>
<th>( c_3 )</th>
<th>( c_2 )</th>
<th>( c_1 )</th>
<th>( c_0 )</th>
<th>( x_3 )</th>
<th>( x_2 )</th>
<th>( x_1 )</th>
<th>( x_0 )</th>
<th>( y_3 )</th>
<th>( y_2 )</th>
<th>( y_1 )</th>
<th>( y_0 )</th>
<th>( z_3 )</th>
<th>( z_2 )</th>
<th>( z_1 )</th>
<th>( z_0 )</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>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<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>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<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>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<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>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<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>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<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>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<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>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<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>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<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>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

\( 2^8 = 512 \) rows!
Let's Make an Adder Circuit

Goal:  \( x + y = z \).

Step 2.
  - Build truth table for carry bit.
  - Build truth table for summand bit.

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

Step 3.
  - Derive (simplified) Boolean expression.

```
Carry Bit          Summand Bit
\hline
\( x_3 \) & \( x_2 \) & \( x_1 \) & \( x_0 \) \\
\( + \) & \( y_3 \) & \( y_2 \) & \( y_1 \) & \( y_0 \) \\
\hline
\( z_3 \) & \( z_2 \) & \( z_1 \) & \( z_0 \) \\
\hline
```

Step 4.
  - Transform Boolean expression into circuit.
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

N-Bit Decoder

N-bit decoder.
- \( N \) address inputs, \( 2^N \) data outputs.
- Addressed output bit is 1; all others are 0.

Application.
- Convert from binary to “unary.”
- Decode opcode into instruction type.
8-to-1 Multiplexer

**2^N-to-1 multiplexer.**
- N select inputs, 2^N data inputs, 1 output.
- Copies "selected" data input bit to output.

8-to-1 Mux Interface

8-to-1 Mux Implementation

---

Bus

- **16-bit bus.**
  - Bundle of 16 wires.
  - Memory transfer, register transfer.

  16

- **8-bit bus.**
  - Bundle of 8 wires.
  - TOY memory address.

  8

- **4-bit bus.**
  - Bundle of 4 wires.
  - TOY register address.

  4

---

4-Wide 2-to-1 Multiplexer

**Goal:** select from one of two 4-bit buses.

- Implement by layering 4 2-to-1 multiplexers.

4-Wide 2-to-1 Mux Interface

4-Wide 2-to-1 Mux Implementation

---

4 copies of same bit
**k-Wide n-to-1 Multiplexer**

Goal: select from one of n k-bit buses.
- Implement by layering k n-bit muxes.

**Multiplexer: Application**

Program counter
- Result of jump or branch instruction.
- Adding 1 to old program counter.

Use 8-wide 2-to-1 mux to route appropriate 8-bit address to PC.
- Unspecified detail: how to set control wire?

**Arithmetic Logic Unit: 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

**Arithmetic Logic Unit: Implementation**

<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>
**Bitwise AND, XOR, NOT**

Bitwise logical operations.
- Inputs x and y: n-bits each.
- Output z: n-bits.
- Apply logical operation to each corresponding pair of bits.

**Abstraction and Encapsulation**

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

Layers of abstraction apply with a vengeance!
- TOY ALU is made of:
  - multiplexer, which is made of:
  - adder, which is made of:
  - and some other things also made of gates
- TOY ALU will itself be a component of TOY computer (Lecture A5).