6.1 Combinational Circuits

George Boole (1815 – 1864)  Claude Shannon (1916 – 2001)
Signals and Wires

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

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

```
0 ______________________ 0

1

1

1

1

Input  Output
```
Logic Gates

Logical gates.
- Fundamental building blocks.

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

NOT  AND  OR
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.

![Multiway OR Gates Diagram](image)
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.

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

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.

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

AND Truth Table

AND

0
0 0
0 0
0 0
0 0
0 0
1 0
0 1
1 0
1 0
1 0
1 1
1 1

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

<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>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</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>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</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>0</td>
<td>1</td>
<td>0</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>1</td>
<td>1</td>
</tr>
</tbody>
</table>
### Truth Table for Functions of 3 Variables

**Truth table.**
- 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>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</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>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<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>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</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>
<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>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>1</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>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>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

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

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

<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'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>
<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>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>0</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>1</td>
<td>0</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>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Translate Boolean Formula to Boolean Circuit

Use sum-of-products form.

- \( \text{XOR}(x, y) = xy' + x'y \).
Translate Boolean Formula to Boolean Circuit

Use sum-of-products form.

- \( \text{MAJ}(x, y, z) = x'y'z + xy'z + xyz' + xyz. \)
Many possible circuits for each Boolean function.

- Sum-of-products not necessarily optimal in:
  - number of gates (space)
  - depth of circuit (time)

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

\[
\begin{align*}
\text{size} = 8, & \quad \text{depth} = 3 \\
\text{size} = 4, & \quad \text{depth} = 2
\end{align*}
\]
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.
**ODD Parity Circuit**

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

<table>
<thead>
<tr>
<th>(x)</th>
<th>(y)</th>
<th>(z)</th>
<th><strong>ODD</strong></th>
<th>(x'y'z)</th>
<th>(x'yz')</th>
<th>(xy'z')</th>
<th>(xyz)</th>
<th>(x'y'z + x'yz' + 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>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>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>
<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>0</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>
<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>
<td>0</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>

Expressing ODD Using Sum-of-Products
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.
- 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.

\[
\begin{array}{cccc}
  x_3 & x_2 & x_1 & x_0 \\
  y_3 & y_2 & y_1 & y_0 \\
\end{array}
\]

\[
\begin{array}{cccc}
  z_3 & z_2 & z_1 & z_0 \\
  c_0 & & & \\
\end{array}
\]

\[
\begin{array}{cccc}
  1 & 1 & 1 & 0 \\
  2 & 4 & 8 & 7 \\
\end{array}
\]

\[
+ \begin{array}{cccc}
  3 & 5 & 7 & 9 \\
\end{array}
\]

\[
\begin{array}{cccc}
  6 & 0 & 6 & 6 \\
\end{array}
\]

\[
\begin{array}{cccc}
  1 & 1 & 0 & 0 \\
  0 & 0 & 1 & 0 \\
\end{array}
\]

\[
+ \begin{array}{cccc}
  0 & 1 & 1 & 1 \\
\end{array}
\]

\[
\begin{array}{cccc}
  1 & 0 & 0 & 1 \\
\end{array}
\]

\[
\begin{array}{cccccccc}
  x_3 & x_2 & x_1 & x_0 \\
  y_3 & y_2 & y_1 & y_0 \\
\end{array}
\]

\[
+ \begin{array}{cccccccc}
  y_3 & y_2 & y_1 & y_0 \\
\end{array}
\]

\[
\begin{array}{cccccccc}
  z_3 & z_2 & z_1 & z_0 \\
\end{array}
\]
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!

<table>
<thead>
<tr>
<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>
</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>
</tr>
<tr>
<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>1</td>
<td>0</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>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<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>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<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>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>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Let's Make an Adder Circuit

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

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

<table>
<thead>
<tr>
<th>( x )</th>
<th>( y )</th>
<th>( c )</th>
<th>( c_{i+1} )</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>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>1</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>( x )</th>
<th>( y )</th>
<th>( z )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

\( c_3 \) \( c_2 \) \( c_1 \) \( c_0 = 0 \)
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></th>
<th>Summand Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x _i )</td>
<td>( y _i )</td>
<td>( c _i )</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

\[ c_3 \quad c_2 \quad c_1 \quad c_0 = 0 \]

\[ x_3 \quad x_2 \quad x_1 \quad x_0 \]

\[ + \]

\[ y_3 \quad y_2 \quad y_1 \quad y_0 \]

\[ z_3 \quad z_2 \quad z_1 \quad z_0 \]
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.
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: new design, like adder circuit.
- Better idea: reuse adder circuit.
  - 2's complement: to negate an integer, flip bits, then add 1
TOY 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

- \(n\) 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
TOY Arithmetic Logic Unit: Implementation

Input 1

16

Input 2

16

Multiplexer (MUX)

ALU control

16

000

001

010

011

100

3

&

^ ±

<<, >>

subtract

shift direction

op  

2 1 0

+, - 0 0 0

& 0 0 1

^ 0 1 0

<<, >> 0 1 1

input 2 1 0 0
6.2: Sequential Circuits

\[ S \rightarrow Q \]

\[ R \rightarrow (Q) \rightarrow Q \]
**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.
Flip-Flop

Flip-flop.
- A small and useful sequential circuit.
- Abstraction that "remembers" one bit.
- Basis of important computer components:
  - memory
  - counter

We will consider several flavors.
SR Flip-Flop

What is the value of $Q$ if:
- $S = 1$ and $R = 0$ ?  

$\implies$  

$Q$ is surely 1
SR Flip-Flop

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
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
SR Flip-Flop

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 !
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.
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.

Implementation

Interface
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.
How much does it Hert?

Frequency is inverse of cycle time.
- Expressed in hertz.
- Frequency of 1 Hz means that there is 1 cycle per second.
- Hence:
  - 1 kilohertz (kHz) means 1000 cycles/sec.
  - 1 megahertz (MHz) means 1 million cycles/sec.
  - 1 gigahertz (GHz) means 1 billion cycles/sec.
  - 1 terahertz (THz) means 1 trillion cycles/sec.

By the way, no such thing as 1 “hert”!

Heinrich Rudolf Hertz (1857-1894)
Clocked SR Flip-Flop

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

![Implementation diagram](image)

![Interface diagram](image)
Clocked D Flip-Flop

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

**Combinational circuits implement Boolean functions**
- Gates and wires: Fundamental building blocks.
- Truth tables: Describe Boolean functions.
- Sum-of-products: Systematic method to implement functions.

**Sequential circuits add "state" to digital hardware.**
- Flip-flop: Represents 1 bit.
- TOY register: 16 D flip-flops.
- TOY main memory: 256 registers.

Next time: we build a complete TOY computer (oh yes).
George Boole (1815 - 1864)  
Claude Shannon (1916 - 2001)