Combinational Circuits

http://smbc-comics.com/comic/logic-gates
The final frontier

- Java programs running on Java VM
  - Or Python programs running in Python interpreter
- C programs compiled on Linux
- Assembly / machine code on CPU + memory
- ???
- Electricity?
Aside: Relays

- From “Code” recommended reading:

![Relay (off)](image1)

Relay (off)

Light is on when switch is on

![Relay (on)](image2)

Relay (on)

Question: what happens if we connect the light bulb to the other contact?
Aside: Relays

- From “Code” recommended reading:

  Regular relay

  Inverted relay (NOT)
Aside: Relays

• From “Code” recommended reading:

Relays in series (AND)

Relays in parallel (OR)
Digital signals are transmitted via electric signals by varying voltages

- 1.0 V (high) = binary 1
- 0.0 V (low) = binary 0
- Use a threshold to distinguish
Transistors

- **Transistors** are the fundamental hardware component of computing
  - Similar to relays; replaced vacuum tubes
    - Smaller, more reliable, and use less energy
    - Primary functions: switching and amplification
  - Mostly silicon-based semiconductors now
    - Metal–Oxide–Semiconductor Field-Effect Transistor (MOSFET)
    - n-channel ("on" when $V_{\text{gate}} = 1\text{V}$) vs. p-channel ("off" when $V_{\text{gate}} = 1\text{V}$)
    - Mass-produced on integrated circuit chips
  - For convenience, we abstract their behavior using logic gates
Logic gates

- Primary gates:

  - AND
    - Truth table:
      - $0 \& 0 = 0$
      - $0 \& 1 = 0$
      - $1 \& 0 = 0$
      - $1 \& 1 = 1$

  - OR
    - Truth table:
      - $0 \lor 0 = 0$
      - $0 \lor 1 = 1$
      - $1 \lor 0 = 1$
      - $1 \lor 1 = 1$

  - NOT
    - Truth table:
      - $\neg 0 = 1$
      - $\neg 1 = 0$

  - NAND
    - Truth table:
      - $(0 \& 0) = 1$
      - $(0 \& 1) = 0$
      - $(1 \& 0) = 0$
      - $(1 \& 1) = 0$

  - NOR
    - Truth table:
      - $(0 \lor 0) = 0$
      - $(0 \lor 1) = 1$
      - $(1 \lor 0) = 1$
      - $(1 \lor 1) = 1$

  - XOR
    - Truth table:
      - $(0 \oplus 0) = 0$
      - $(0 \oplus 1) = 1$
      - $(1 \oplus 0) = 1$
      - $(1 \oplus 1) = 0$
Fluid-based gate visualization

- [https://i.imgur.com/wUhtCgL.gifv](https://i.imgur.com/wUhtCgL.gifv)
- [https://i.imgur.com/UJyNd9T.gifv](https://i.imgur.com/UJyNd9T.gifv)
Basic circuits

- **Circuits** are formed by connecting gates together
  - Inputs and outputs
    - Link output of one gate to input of another
    - Some circuits have multiple inputs and/or outputs
  - Textbook uses Hardware Description Language (HDL)
  - Equivalent to boolean formulas or functions
    - $f(g(x, y))$ means “apply $f$ to the result of applying $g$ to $x$ and $y$”
    - In a diagram: $x, y \rightarrow g \rightarrow f$ (i.e., ordering is $g$ first, then $f$)
Basic circuits

- \( f(g(x, y)) \) means “apply \( f \) to the result of applying \( g \) to \( x \) and \( y \)”
  - In a diagram: \( x, y \rightarrow g \rightarrow f \) (i.e., ordering is \( g \) first, then \( f \))
- NAND example: (similarly for NOR)
  - Infix/boolean notation: \( a \text{ NAND } b = \text{NOT}(a \text{ AND } b) = !(a \& B) \)
  - Function notation: \( \text{NAND}(a, b) = \text{NOT}(\text{AND}(a, b)) \)
Basic circuits

- Circuits are **equivalent** if the truth tables are the same
  - $a \text{ XOR } b = (a \text{ OR } b) \text{ AND } (a \text{ NAND } b)$
  - $\text{XOR}(a, b) = \text{AND}(\text{OR}(a,b), \text{NAND}(a,b))$
Basic circuits

- Circuits are equivalent if the truth tables are the same
  - $a \ XOR \ b = (a \ OR \ b) \ AND \ (a \ NAND \ b)$
  - $XOR(a, b) = AND(OR(a, b), NAND(a, b))$
Basic circuits

- Circuits are **equivalent** if the truth tables are the same
  - $a \text{ XOR } b = (a \text{ OR } b) \text{ AND } (a \text{ NAND } b)$
  - $\text{XOR}(a, b) = \text{AND}(\text{OR}(a,b), \text{NAND}(a,b))$
Basic circuits

- Circuits are **equivalent** if the truth tables are the same
  - \( a \text{ XOR } b = (a \text{ OR } b) \text{ AND } (a \text{ NAND } b) \)
  - \( \text{XOR}(a, b) = \text{AND}(\text{OR}(a,b), \text{NAND}(a,b)) \)

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>f(a, b)</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>

\( f(a,b) \)
Important properties

- **Identity:** $a \text{ AND } 1 = a$ \hspace{1cm} $(a \text{ OR } 0) = a$

- **Constants:** $a \text{ AND } 0 = 0$ \hspace{1cm} $(a \text{ OR } 1) = 1$
  - Also: $a \text{ NAND } 0 = 1$ \hspace{1cm} $(a \text{ NOR } 1) = 0$

- **Inverses:** $a \text{ NAND } 1 = !a$ \hspace{1cm} $(a \text{ NOR } 0) = !a$
  - Also: $a \text{ NAND } a = !a$ \hspace{1cm} $a \text{ NOR } a = !a$

- **Double inverse:** $!!a = a$
  - Or: $\text{NOT(NOT}(a)) = a$

- **De Morgan’s law:** $!(a \text{ & } b) = !a \text{ | } !b$ \hspace{1cm} \text{(remember this from CS 227!)}
  - Alternatively: $!(a \text{ | } b) = !a \text{ & } !b$
Universal gates

- NAND and NOR gates are universal
  - Each one alone can reproduce all other gates
  - Example: $a \text{ AND } b = a \& b = !(!(a \& b)) = !(a \text{ NAND } b) = (a \text{ NAND } b) \text{ NAND } (a \text{ NAND } b)$
Universal gates

- NAND and NOR gates are **universal**
  - Each one alone can reproduce all other gates
  - Example: \( a \text{ AND } b = a \& b = !(a \& b) = !(a \text{ NAND } b) = (a \text{ NAND } b) \text{ NAND } (a \text{ NAND } b) \)
  - Similarly: \( a \text{ AND } b = !(a \& b) = !(a \text{ NOR } b) = (a \text{ NOR } a) \text{ NOR } (b \text{ NOR } b) \)
Circuit types

• Two main kinds of circuits:
  - **Combinational** circuits: outputs are a boolean function of inputs
    • Not time-dependent
    • Used for **computation**
  - **Sequential** circuits: output is dependent on previous outputs
    • Time-dependent
    • Used for **memory**
Goal: identify circuits that perform useful computation

- Testing bits to see if they’re equal
- Selecting between multiple inputs
- Adding or subtracting bits
- Bitwise operations (AND, OR, XOR)
- Make them work on bytes instead of bits
Equality

\[ a \text{ EQ } b = (a \& b) \mid (\neg a \& \neg b) \]
Multiplexor ("selector")

\[ \text{MUX}(a, b, s) = (s \& a) \lor (!s \& b) \]
Half adders

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>

Half Adder
Half adders

A + B = A \oplus B + A \cdot B

**Half Adder**

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<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>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

sum carry
Abstraction

- Name circuits, then use them to build more complex circuits
  - E.g., use bit-level EQ to build a word-level equality circuit:
Word-level 2-way multiplexer

A). Bit-level implementation

B). Word-level abstraction
How many selector inputs would be required for eight data inputs?

How many data inputs could be supported using four selector inputs?
Connect full adders to build a **ripple-carry adder** that can handle multi-bit addition:
Adder/subtractor

In two's complement: \( B - A = B + \sim A + 1 \)

(invert carry-out for CF if D=1)
ALUs

- Combine **adders and multiplexors** to make arithmetic/logic units

Basic Arithmetic Logic Unit (ALU)
CPUs

- Combine ALU with registers and memory to make CPUs
Computers

- Combine **CPU** with other electronic components and devices (similarly constructed) communicating via buses to make a computer.
Big picture

• Basic systems design approach: exploit abstraction
  - Start with simple components
  - Combine to make more complex components
  - Repeat using the new components as black box “simple components”

• This is true of most areas in systems
  - **CS 261**: transistors → gates → circuits → adders/flip-flops → ALUs/registers → CPUs/memory → computers
  - **CS 261**: machine code → assembly → C code → Java/Python code
  - **CS 361/470**: threads → processes → nodes → networks/clusters
  - **CS 432**: scanner → parser → analyzer → code generator → optimizer
  - **CS 450**: files + processes + I/O → kernel → operating system
  - **CS 455**: byte stream → frames → packets → datagrams → messages
  - **CS 456**: multiplexers → primitives → modules → CPUs (on FPGAs)
Course status

- We’ve hit the bottom
  - Or at least as far down as we’re going to go (logic gates); from here we go back up!

- Next up:
  - Sequential circuits
  - CPU architecture

Suggestion: download Logisim Evolution and play around with some circuits! (.circ file on Canvas)