Logic Design

Goal: to become literate in most common concepts and terminology of digital electronics

Important concepts:
- use abstraction and composition to implement complicated functionality with very simple digital electronics
- keep things as simple, regular, and small as possible

Things we will not explore:
- physics
- chip fabrication
- layout
- tools for chip specification and design

Motivation

Consider the external view of addition:

\[
x \xrightarrow{\text{Adder}} x + y
\]

What kind of circuitry would go into the "black box" adder to produce the correct results?

How would it be designed? What modular components might be used?
Basic Logic Gates

Fundamental building blocks of circuits; mirror the standard logical operations:

- **NOT gate**
- **AND gate**
- **OR gate**

<table>
<thead>
<tr>
<th>A</th>
<th>Out</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>A</th>
<th>B</th>
<th>Out</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>A</th>
<th>B</th>
<th>Out</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>

Note the outputs of the AND and OR gates are commutative with respect to the inputs. Multi-way versions of the AND and OR gates are commonly assumed in design.

Multi-way Gates

Multi-way versions of the AND and OR gates are commonly assumed in design. They may be trivially implemented using the basic 2-input versions:

- **5-way AND gate**

We will generally assume the availability of n-input AND and OR gates for arbitrary values of n.
### Combinational and Sequential Circuits

A **combinational circuit** is one with no "memory". That is, its output depends only upon the current state of its inputs, and not at all on the current state of the circuit itself.

A **sequential circuit** is one whose output depends not only upon the current state of its inputs, but also on the current state of the circuit itself.

For now, we will consider only combinational circuits.

### From Function to Combinational Circuit

Given a simple Boolean function, it is relatively easy to design a circuit composed of the basic logic gates to implement the function:  

\[
z = x_0 \cdot \overline{y_0} + x_0 \cdot y_0
\]

This circuit implements the *exclusive or* (XOR) function. This is often represented as a single logic gate:
### Additional Common Logic Gates

#### XOR gate

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Out</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>

#### NAND gate

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

#### NOR gate

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

#### XNOR gate

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

### Multiplexor

An $2^n \times 1$ multiplexor receives $2^n$ input bits and $n$ selector bits, and outputs exactly one of the input bits, determined by the pattern of the selector bits.

#### 2 x 1 multiplexor

\[ C = A \cdot \bar{S} + B \cdot S \]

#### 4 x 1 multiplexor
**Decoder**

An n x $2^n$ decoder takes n inputs and sets exactly one of its $2^n$ outputs, based upon the pattern of its inputs.

**2 x 4 decoder**

---

**Encoders and Demultiplexors**

A $2^n$ x n encoder takes $2^n$ inputs and sets each of its n outputs, based upon the pattern of its inputs.

Essentially, an encoder is the inverse of a decoder.

Similarly, a 1 x $2^n$ demultiplexor is the inverse of a multiplexor.

It takes 1 input and transmits that input on exactly one of its outputs, determined by the pattern of its n selector bits.
Efficiency of Expression

While the sum-of-products form is arguably natural, it is not necessarily the simplest way form, either in:
- number of gates (space)
- depth of circuit (time)

$$F(x, y, z) = \overline{x \cdot y \cdot \overline{z}} + \overline{x \cdot y \cdot z} + x \cdot y \cdot \overline{z} + x \cdot y \cdot z$$

$$F(x, y, z) = x \cdot y + y \cdot \overline{z} + x \cdot \overline{z}$$

1-bit Half Adder

Let's make a 1-bit adder (half adder)… we can think of it as a Boolean function with two inputs and the following truth table:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Sum</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>

Here's the resulting circuit.

It's equivalent to the XOR circuit seen earlier.

But… in the final row of the truth table above, we've ignored the fact that there's a carry-out bit.
Dealing with the Carry

The carry-out value from the 1-bit sum can also be expressed via a truth table. However, the result won't be terribly useful unless we also take into account a carry-in.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C&lt;sub&gt;i&lt;/sub&gt;</th>
<th>Sum</th>
<th>C&lt;sub&gt;o&lt;/sub&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<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>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</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>
</tr>
<tr>
<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>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

The resulting sum-of-products expressions are:

\[
\text{Sum} = \overline{A} \cdot \overline{B} \cdot \overline{C_i} + \overline{A} \cdot B \cdot \overline{C_o} + A \cdot \overline{B} \cdot \overline{C_o} + A \cdot B \cdot C_i
\]

\[
\text{Carry} = \overline{A} \cdot B \cdot C_i + A \cdot \overline{B} \cdot C_i + A \cdot B \cdot \overline{C_o} + A \cdot B \cdot C_i
\]

\[
= \overline{A} \cdot B \cdot \overline{C_o} + A \cdot \overline{B} \cdot C_i + A \cdot B \cdot (\overline{C_o} + C_i)
\]

\[
= \overline{A} \cdot B \cdot C_i + A \cdot \overline{B} \cdot C_i + A \cdot B
\]

\[
= B \cdot C + A \cdot C + A \cdot B
\]

1-bit Full Adder

The expressions for the sum and carry lead to the following unified implementation:

\[
\text{Sum} = \overline{A} \cdot \overline{B} \cdot \overline{C_i} + \overline{A} \cdot B \cdot \overline{C_o} + A \cdot \overline{B} \cdot C_o + A \cdot B \cdot C_i
\]

\[
\text{Carry} = B \cdot C + A \cdot C + A \cdot B
\]

This implementation requires only two levels of logic (ignoring the inverters as customary).

Is there an alternative design that requires fewer AND/OR gates? If so, how many levels does it require?
The previous circuit is easily implemented in LogiSim.

The Project menu contains an option to analyze the circuit.

You may find these useful in verifying the correctness of a circuit.
1-bit Full Adder as a Module

When building more complex circuits, it is useful to consider sub-circuits as individual, "black-box" modules. For example:

\[
\begin{align*}
\text{Sum} &= \overline{A} \cdot B \cdot C_i + A \cdot B \cdot \overline{C_i} \\
&\quad + A \cdot \overline{B} \cdot \overline{C_i} + \overline{A} \cdot B \cdot C_i \\
\text{Carry} &= B \cdot C + A \cdot C + A \cdot B
\end{align*}
\]

Chaining an 8-bit Adder

An 8-bit adder build by chaining 1-bit adders:

This has one serious shortcoming. The carry bits must ripple from top to bottom, creating a lag before the result will be obtained for the final sum bit and carry.
We know that every Boolean function can be expressed in sum-of-products form, and it is obvious that a sum-of-products form can be implemented with two levels of logic, the first consisting of AND gates and the second of OR gates.

A **programmable logic array** (PLA) is a structured logic element consisting of a set of inputs (and corresponding input complements), and two stages of logic the first generating product terms of the inputs and the second generating sums of the product terms.

A PLA can be used to implement a set of Boolean functions in a compact form. Any set of $k$ functions on $n$ variables can be implemented via an $m \times n$ array of AND gates connected to a $k \times m$ array of OR gates, where $m$ is the number of different product terms that are needed.

**PLA Example**

Consider the collection of Boolean functions defined by the following truth table:

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</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>

Since there are 7 rows in which at least one function is 1, we will need 7 AND gates. We will need an OR gate to form the output for each function.
Here's the PLA:

This is advantageous because the functions in question have a significant number of product terms in common.

A read-only-memory (ROM) is a decoder-like circuit that takes n bits as input and selects one of $2^n$ k-bit sequences as its output.

The shape of the ROM is its height, which is $2^n$, and its width, which is k.

A ROM exactly encodes a truth table.

A ROM is generally larger than the equivalent PLA.

At right is a modular representation of a ROM with height 16, and width 32.
Creating a Basic 1-bit ALU

Here's a basic 1-bit ALU:

- user-selectable output via the MUX
- 01: A AND B
- 10: A OR B
- 11: A + B
- carry-in and carry-out bits

The multiplexor is used to select the desired function the ALU will perform. Additional features can be incorporated fairly easily.

Extending the 1-bit ALU

Let's add some features:

- negation of A and/or B
- A + B, A + -B (hence, subtraction)
- A NOR B
- user-selectable output via the MUX
- 00: A AND B
- 01: A OR B
- 10: A + B
- 11: SLT
- carry-in and carry-out bits
- output bit for SLT*
- overflow detection*

* implemented ONLY for high-order bit
Creating a Multi-bit ALU

As we saw earlier with the adder, we can chain 1-bit ALUs together to produce an ALU for any width input we desire.

There are a few considerations:
- only the ALU for the high-order bit would include the Set output and the Overflow detection logic and output.
- the Set output from the high-order ALU will be connected to the Less input of the low-order ALU in order to implement the SLT instruction.
- the addition module still uses a ripple-carry; that should be remedied eventually.

A 4-bit ALU

Here's a 4-bit ALU created from the extended 1-bit ALU shown earlier:

Here, the ALU is being used to evaluate the `slt` instruction on its operands, setting the result in Result.

Note that the InvertB input is also connected to the CarryIn port of the low-order 1-bit ALU module. Why?
To support conditional branch we need the \texttt{slt} instruction, and a way to test whether the inputs are equal (or non-equal). This can be added fairly trivially:

Effectively, this ALU has a 4-bits control mechanism for selecting the desired function.

<table>
<thead>
<tr>
<th>InvA</th>
<th>InvB</th>
<th>FnSel</th>
<th>ALU Fn</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td>AND</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>OR</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>10</td>
<td>add</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>10</td>
<td>sub</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11</td>
<td>slt</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>00</td>
<td>NOR</td>
</tr>
</tbody>
</table>