Understanding Logic Design

Appendix A of your Textbook does not have the needed background information. This document supplements it.

When you write add ADD R0, R1, R2, you imagine something like this:

What kind of hardware can ADD two binary integers?
We need to learn about GATES and BOOLEAN ALGEBRA that are foundations of logic design.
AND gate

<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>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>

OR gate

<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 gate

<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>

Typically, logical 1 = +3.5 volt, and logical 0 = 0 volt. Other representations are possible.
Analysis of logical circuits

What is the value of F when X=0 and Y=1?

Draw a truth table.

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>F</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>

This is the *exclusive or* (XOR) function. In algebraic form

\[ F = \overline{X}.Y + X.\overline{Y} \]
More practice

1. Let \( \bar{A} \cdot B + A \cdot C = 0 \). What are the values of \( A, B, C \)?

2. Let \( (A + B + C).\bar{A} \cdot \bar{B} \cdot \bar{C} = 0 \). What are the possible values of \( A, B, C \)?

- Draw truth tables.
- Draw the logic circuits for the above two functions.
**Boolean Algebra**

\[
\begin{align*}
A + 0 &= A & A + A' &= 1 \\
A \cdot 1 &= A & A \cdot A' &= 0 \\
1 + A &= 1 & A + B &= B + A \\
0 \cdot A &= 0 & A \cdot B &= B \cdot A
\end{align*}
\]

\[
\begin{align*}
A + (B + C) &= (A + B) + C \\
A \cdot (B \cdot C) &= (A \cdot B) \cdot C
\end{align*}
\]

\[
\begin{align*}
A + A &= A \\
A \cdot A &= A
\end{align*}
\]

\[
\begin{align*}
A \cdot (B + C) &= A.B + A.C & \text{Distributive Law} \\
A + B.C &= (A+B). (A+C)
\end{align*}
\]

\[
\begin{align*}
\overline{A \cdot B} &= A + B & \text{De Morgan's theorem} \\
\overline{A + B} &= A \cdot B
\end{align*}
\]
De Morgan’s theorem

\[
\begin{align*}
A \cdot B &= A + B \\
A + B &= A \cdot B
\end{align*}
\]

Thus, \( \text{AND} \) is equivalent to \( \text{OR} \)

Verify it using truth tables. Similarly,

\( \text{OR} \) is equivalent to \( \text{AND} \)

These can be generalized to more than two variables: to

\[
\begin{align*}
A \cdot B \cdot C &= A + B + C \\
A + B + C &= A \cdot B \cdot C
\end{align*}
\]
Synthesis of logic circuits

Many problems of logic design can be specified using a truth table. Give such a table, can you design the logic circuit?

Design a logic circuit with three inputs A, B, C and one output F such that F=1 only when a majority of the inputs is equal to 1.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>F</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>

Sum of product form

F = \overline{A}.B.C + A.\overline{B}.C + A.B.\overline{C} + A.B.C

Draw a logic circuit to generate F
Simplification of Boolean functions

Using the theorems of Boolean Algebra, the algebraic forms of functions can often be simplified, which leads to simpler (and cheaper) implementations.

Example 1

\[ F = A \cdot B + A \cdot B + B \cdot C \]
\[ = A \cdot (B + B) + B \cdot C \]
\[ = A \cdot 1 + B \cdot C \]
\[ = A + B \cdot C \]

How many gates do you save from this simplification?
Example 2

\[ F = \overline{A}.B.C + A\overline{B}.C + A.B.C + A.B.C + A.B.C \]

\[ = \overline{A}.B.C + A\overline{B}.C + A.B.C + A.B.C + A.B.C + A.B.C \]

\[ = (A.B.C + A.B.C) + (A.B.C + A.B.C) + (A.B.C + A.B.C) \]

\[ = (A + A).B.C + (B + B).C.A + (C + C).A.B \]

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

Example 3  
Show that \( A + A.B = A \)

\[ A + A.B \]

\[ = A.1 + A.B \]

\[ = A. (1 + B) \]

\[ = A. 1 \]

\[ = A \]
Other types of gates

NAND gate

\[ A \land B \]

NOR gate

\[ A \lor B \]

Be familiar with the truth tables of these gates.

Exclusive OR (XOR) gate

\[ A \oplus B = A.B + A.B \]
NAND and NOR are universal gates

Any function can be implemented using only NAND or only NOR gates. How can we prove this?

(Proof for NAND gates) Any boolean function can be implemented using AND, OR and NOT gates. So if AND, OR and NOT gates can be implemented using NAND gates only, then we prove our point.

1. Implement NOT using NAND
2. Implementation of AND using NAND

![AND using NAND circuit diagram]

3. Implementation of OR using NAND

![OR using NAND circuit diagram]

Exercise. Prove that NOR is a universal gate.
Logic Design (continued)

XOR Revisited

XOR is also called modulo-2 addition.

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

A ⊕ B = 1 only when there are an odd number of 1's in (A,B). The same is true for A ⊕ B ⊕ C also.

\[
1 \oplus A = \overline{A} \quad \text{Why?}
\]

\[
0 \oplus A = A
\]
Logic Design Examples

Half Adder

\[ S = A \oplus B \]
\[ C = A \cdot B \]

\[
\begin{array}{c|c|c}
A & B & S & C \\
0 & 0 & 0 & 0 \\
0 & 1 & 1 & 0 \\
1 & 0 & 1 & 0 \\
1 & 1 & 0 & 1 \\
\end{array}
\]
Full Adder

\[ S = A \oplus B \oplus C_{\text{in}} \]
\[ C_{\text{out}} = A.B + B.C_{\text{in}} + A.C_{\text{in}} \]

Design a full adder using two half-adders (and a few gates if necessary)

Can you design a 1-bit subtracter?
Decoders

A typical decoder has $n$ inputs and $2^n$ outputs.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>D3</th>
<th>D2</th>
<th>D1</th>
<th>D0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</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>
</tr>
<tr>
<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>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

A 2-to-4 decoder and its truth table

D3 = $A \cdot B$
D2 = $A \cdot \overline{B}$
D1 = $\overline{A} \cdot B$
D0 = $\overline{A} \cdot \overline{B}$

Draw the circuit of this decoder.

The decoder works per specs when (Enable = 1). When Enable = 0, all the outputs are 0.

Exercise. Design a 3-to-8 decoder.

Question. Where are decoders used?

Can you design a 2-4 decoder using 1-2 decoders?
Encoders

A typical encoder has $2^n$ inputs and $n$ outputs.

A 4-to-2 encoder and its truth table

\[
\begin{align*}
A &= D1 \lor D3 \\
B &= D2 \lor D3
\end{align*}
\]
**Multiplexor**

It is a *many-to-one switch*, also called a *selector*.

\[
\begin{align*}
A &\rightarrow 0 & F & S = 0, F = A \\
B &\rightarrow 1 & & S = 1, F = B
\end{align*}
\]

*Specifications of the mux*

A 2-to-1 mux

\[F = \overline{S} \cdot A + S \cdot B\]

Exercise. Design a 4-to-1 multiplexor using two 2-to-1 multiplexors.
Demultiplexors

A demux is a one-to-many switch.


So, $X = S \cdot A$, and $Y = S \cdot A$

Exercise. Design a 1-4 demux using 1-2 demux.
A 1-bit ALU

Operation = 00 implies AND
Operation = 01 implies OR
Operation = 10 implies ADD

♦ Understand how this circuit works.
- **Converting an adder into a subtractor**

  \[ A - B \text{ (here - means arithmetic subtraction)} = A + 2\text{'s complement of } B \]

  \[ = A + 1\text{'s complement of } B + 1 \]

  1-bit adder/subtractor

  For subtraction, \( B \text{ invert} = 1 \) and \( \text{Carry in} = 1 \)
A 32-bit ALU

B invert

C in

operation

A0

ALU

ALU

ALU

Result 0

B0

Cout

Cout

Cout

Result 1

A1

.. 

B1

Result 31

A31

ALU

ALU

ALU

overflow

B31
Combinational vs. Sequential Circuits

**Combinational circuits**

The output depends only on the current values of the inputs and not on the past values. Examples are adders, subtractors, and all the circuits that we have studied so far.

**Sequential circuits**

The output depends not only on the current values of the inputs, but also on their past values. These hold the secret of how to memorize information. *We will study sequential circuits later.*