Realizing Behaviors

Two sections ago we learned to analyze logical expressions by using truth tables. We can, of course, use the same process to analyze logic networks. For instance, if we have the logic network

\[
\begin{align*}
a \\
b \\
c \\
d
\end{align*}
\]

and we want to understand what outputs we will get when we set the input variables a, b, and c to either 0 or 1, we can understand this behavior by first writing the equivalent logical expression - \(ab + \overline{cb}\) - and analyzing the expression. We already analyzed this expression two sections ago and found the following truth table (omitting the helper columns).

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>(ab + \overline{cb})</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</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>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>

This truth table completely describes the behavior of the logic network. It says that the output of the network will always be 1 unless a has an input of 0 while b and c have inputs of 1 (the fourth row of the table).

We call this process of starting with a logic network and getting a truth table that describes the behavior analysis. But we often want to go in the other direction; to start with a behavior that we want and build a logic network that has that behavior. We call this opposite process design, and we say that the logic network is the realization of the behavior.

![Logic Network Diagram]

<table>
<thead>
<tr>
<th>Logic Network</th>
<th>Analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

Realization

Behavior

---

Truth Table

Design
For example, suppose we want to build a logic network that can realize the behavior of majority rule. That is, we want a logic network whose behavior looks like:

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

Here the output $Y$ is 0 if the majority of the inputs are 0 and the output is 1 if the majority of the inputs is 1. So the output will be 1 for the fourth row or the last three rows. In the fourth row, we have $a = 0$, $b = 1$, and $c = 1$; which is to say that we have $(\text{NOT } a) \text{ AND } b \text{ AND } c$.

We can build an AND gate with this behavior:

4th row (0 1 1):

Note that this AND gate will produce an output of 1 exactly when $a = 0$, $b = 1$, and $c = 1$. Similarly, we can build AND gates for the last three rows:

6th row (1 0 1):

7th row (1 1 0):

8th row (1 1 1):
Finally, since the truth table has an output of 1 for the 4th row OR 6th row OR 7th row OR 8th row we will hook the outputs of these four AND gates into an OR gate:

You can double-check that this logic network does indeed have the desired behavior by analyzing the corresponding logical expression - \( \overline{abc} + \overline{ab}c + abc + ab\overline{c} \) - to see that its truth table is the majority rule truth table.

We can use a similar procedure to easily design a logic network for any desired truth table.

**Method for easily designing a logic network**

Starting with a truth table that describes a given behavior,

1) Create an AND gate for each row where the output is 1. If an input is 0, insert a NOT gate between the input and the AND gate.

2) Hook the outputs of the AND gates from 1) as inputs for a single OR gate.

We can use this method for any truth table; that is, any direct relationship between inputs of 0’s and 1’s and outputs of 0’s and 1’s. The 1’s and 0’s can stand for any pair of things – not just true and false (for the majority rule network, 0 might stand for Candidate A and 1 might stand for Candidate B).

**Example**  Design a logic network for the following input-output behavior:
**Answer**  We have four input variables \(a, b, c,\) and \(d\) but we have just three rows where the output is 1. Constructing three AND gates for these rows (0000, 1001, and 1100) and hooking them together in an OR gate, we get

\[
\begin{array}{cccc|c}
  a & b & c & d & Y \\
  0 & 0 & 0 & 0 & 1 \\
  0 & 0 & 0 & 1 & 0 \\
  0 & 0 & 1 & 0 & 0 \\
  0 & 0 & 1 & 1 & 0 \\
  0 & 1 & 0 & 0 & 0 \\
  0 & 1 & 0 & 1 & 0 \\
  0 & 1 & 1 & 0 & 0 \\
  0 & 1 & 1 & 1 & 0 \\
  1 & 0 & 0 & 0 & 0 \\
  1 & 0 & 0 & 1 & 1 \\
  1 & 0 & 1 & 0 & 0 \\
  1 & 0 & 1 & 1 & 0 \\
  1 & 1 & 0 & 0 & 1 \\
  1 & 1 & 0 & 1 & 0 \\
  1 & 1 & 1 & 0 & 0 \\
  1 & 1 & 1 & 1 & 0 \\
\end{array}
\]

Now that we know how to design logic networks with a given behavior, we can turn our attention toward starting to design a machine that can do arithmetic calculations.

Recall that the essence of addition and multiplication involves a reset and carry. To design a logic network that can realize a reset, recall how numbers 0 and 1 add together. We can write the addition table, where \(Y = a + b\), as follows:
Our method for designing logic networks requires two AND gates – one for each of the two rows where the output is 1. Hooking these two AND gates into an OR gate gives:

**Reset:**

We do a carry only when we are adding 1 to 1 (so $a$ and $b$ are both 1). The truth table for this is:

<table>
<thead>
<tr>
<th>$a$</th>
<th>$b$</th>
<th>$Y$</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>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Which is realized by just a single AND gate:

**Carry:**

Putting these two networks together we get a logic network that can do basic addition:
This logic network is known as a **half-adder** since it only handles single-digit binary numbers. A full adder can handle addition of two binary numbers with any number of digits, but the idea of how to design it is similar.

These logic networks are the very simplest versions of a digital computer.

Our method for designing networks is easy to understand, but unfortunately it tends to produce logic networks which are not very efficient. Consider the following input-output behavior:

<table>
<thead>
<tr>
<th>$a$</th>
<th>$b$</th>
<th>$Y$</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>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Our method requires two AND gates hooked together to an OR gate like so:

The corresponding logical expression is

$$\overline{a}b + ab$$

By the Distributive Law, we can rewrite this as

$$a(\overline{b} + b)$$

By the Cancellation Law for OR, this is the same as

$$a1$$

And by the Law of Identity for AND this is just

$$a$$

So the complicated network above can be replaced by just

$$a$$
We have eliminated several gates and the variable $b$ entirely. In the real world, when building gates out of hardware, the fewer gates there are in a logic network the less expensive it will be to build the network. So we would really like some way to design logic networks which is more efficient.

We’ll see how to do this in the next section.