A “logic circuit” is composed of switching functions

- “or” shape
- “and” shapes
- “not” bubbles

Think of this as an application of functional programming.

**Truth Table**

<table>
<thead>
<tr>
<th>u</th>
<th>v</th>
<th>w</th>
<th>x</th>
<th>f</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>0</td>
<td>1</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>0</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>0</td>
</tr>
<tr>
<td>1</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>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<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>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**SOP and Minterm forms**

\[ f(u, v, w, x) = uv'w'x' + u'vw'x + u'vwx' \]

- This is called a **SOP** (sum-of-products) form. In this case, a “product” means product of variables or complemented variables.
- It is also a **minterm** form. A minterm is a product that uses *all* of the variables.
- Not every SOP is a minterm form.

**Example**

\[ f(u, v, w, x) = u v'w'x' + u'vw'x + u'vwx' \]
Note the Connection

\[ f(u, v, w, x) = u \cdot v'w'x' + u'v \cdot w'x + u'v'w \cdot x \]

The "1" rows of the truth table correspond exactly to the minterms!

Ways to Specify Boolean Functions

- Logic circuit
- Functional expression
  - SOP form
    - Minterm expansion
- Truth table
- Compressed truth table
- Index number
- Karnaugh map

Shorthands

Show only the "1" rows (be careful)

Represent whole table by a set of "minterm numbers":
\{2, 5, 8\}

Represent whole table by a single numeral: 0010010010000000

Counting Functions

- The following are all equal:
  - The number of boolean functions of \( n \) variables.
  - The number of ways to assign 0 or 1 to the \( 2^n \) rows of the truth table.
  - The number of subsets of \( \{0, 1, 2, \ldots, 2^n - 1\} \)
Number of Switching Functions

- n = 1: 4
- n = 2: 16
- n = 3: 256
- n = 4: 65,536
- n = 5: 4,294,967,296
- n = 6: 18,446,744,073,709,551,616

Each level squares the previous.

The 16 switching functions of 2 variables

<table>
<thead>
<tr>
<th>b</th>
<th>c</th>
<th>Function number</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0 0 0 0 0 0 1 1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0 0 0 1 0 0 0 1 1 1 0 0 0 1 1 1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0 0 1 1 0 1 1 0 0 1 1 0 0 1 1 1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0 0 0 1 0 1 0 1 0 1 0 1 0 1 0 1</td>
</tr>
</tbody>
</table>

and xor or equiv implies

Logic Synthesis:
Abstraction to Implementation

- From: Verbal problem description
- To: Implementation as a network of basic switching functions

Logic Synthesis: Stages

1. Provide verbal problem description.
2. Tabulate description as function on finite sets.
3. Encode finite sets into bits.
4. Transcribe the encoded tables.
5. Split into individual switching functions.
6. Realize as a network of basic gates.
Example

• Verbal description:
  Implement a “mod 3 adder using logic gates”

• A definition of mod3 addition:
  \[ f(a, b) = (a+b) \mod 3 \]
  where \( a, b \in \{0,1,2\} \)

Tabulate definition of function

<table>
<thead>
<tr>
<th>((x+y) \mod 3)</th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Encode sets into bits

• Set to be encoded: \( \{0,1,2\} \)

• Chosen encoding (among many):
  
  \[
  \begin{align*}
  0 & \rightarrow 00 \\
  1 & \rightarrow 01 \\
  2 & \rightarrow 10
  \end{align*}
  \]

Transcribe the Function to the Encoded Values

<table>
<thead>
<tr>
<th>Function</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>((x+y) \mod 3)</td>
<td>0 → 00 1 → 01 2 → 10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Encoded Function uv</th>
<th>xy</th>
<th>uv</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>01</td>
<td>10</td>
</tr>
<tr>
<td>01</td>
<td>01</td>
<td>10</td>
</tr>
<tr>
<td>10</td>
<td>00</td>
<td>01</td>
</tr>
</tbody>
</table>
**Split** the Encoded Function into individual switching functions

The resulting switching functions generally will only be **partially** specified.

As the unspecified values will never occur, we can give the function either value 0 or 1.

For the time being, let's just make them all 0. Now we can "read off" an expression for each function.

Exercise: "read off" the expression for \( f_2 \).
Realize each function by gates

\[ f_1(u, v, w, x) = u'v'w'x' + u'vw'x + uv'w'x' \]

Check by Simulating Circuit
(Try all combinations; one combination is shown)

Rex Implementation

```plaintext
encode(0) => [0, 0];
encode(1) => [0, 1];
encode(2) => [1, 0];

decode([0, 0]) => 0;
decode([0, 1]) => 1;
decode([1, 0]) => 2;
decode([x, y]) => "error, should not occur";

add3(i, j) = (i+j)%3;

addByCircuit(i, j) = decode(circuit(encode(i), encode(j)));
```

Testing Code

```plaintext
test(addByCircuit(0, 0), add3(0, 0));
test(addByCircuit(0, 1), add3(0, 1));
test(addByCircuit(0, 2), add3(0, 2));
test(addByCircuit(1, 0), add3(1, 0));
test(addByCircuit(1, 1), add3(1, 1));
test(addByCircuit(1, 2), add3(1, 2));
test(addByCircuit(2, 0), add3(2, 0));
test(addByCircuit(2, 1), add3(2, 1));
test(addByCircuit(2, 2), add3(2, 2));
```