Logic Design II

CSci 2021: Machine Architecture and Organization
Lecture #37, April 27th, 2015

Your instructor: Stephen McCamant

Truth Tables

- **Combinational circuit = Boolean function**
  - Combinational: no cycles or memory
  - Outputs are determined just by inputs
- **Finite size**
  - A Boolean function has a finite representation
  - If \( i \) input bits, \( 2^i \) possible input combinations
  - Can study by just writing the output for all possible inputs
- **Truth table**
  - Standard way to write a function
  - \( 2^i \) rows, input combinations in increasing order
  - One column per intermediate or output

Truth Table Example

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>(a &amp; b)</th>
<th>(a &amp; b)</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<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>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</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>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Equivalences with a Truth Table

- **Check whether two Boolean formulas are equal**
  - Write truth table covering both
  - Check two columns have all the same entries
- **Advantages**
  - Straightforward
  - No algebraic insight needed
- **Disadvantages**
  - Effort exponential in number of input bits

Equivalence Example

| a | b | c | (b & c) | e | (a | b) | (e | c) | (a | b) & (a | c) |
|---|---|---|---------|---|-------|-------|----------------|
| 0 | 0 | 0 | 0       | 0 | 0     | 0     | 0              |
| 0 | 0 | 1 | 0       | 0 | 1     | 0     | 0              |
| 0 | 1 | 0 | 0       | 1 | 0     | 0     | 0              |
| 0 | 1 | 1 | 1       | 1 | 1     | 1     | 1              |
| 1 | 0 | 0 | 0       | 1 | 1     | 1     | 1              |
| 1 | 0 | 1 | 0       | 1 | 1     | 1     | 1              |
| 1 | 1 | 0 | 0       | 1 | 1     | 1     | 1              |
| 1 | 1 | 1 | 1       | 1 | 1     | 1     | 1              |

Combinational Logic Design

- **Given:** description of circuit behavior
  - Word problem, or truth table
- **Goal:** efficient circuit implementation
  - Usually most important: fewest gates and wires
  - Secondarily: reduce number of levels (propagation delay)
- **Kinds of techniques**
  - Up to 6 inputs: pencil and paper approaches
  - Large but structured: split into repeated pieces
  - Large and unstructured: computer algorithm
DNF / SOP

- An input or its negation is called a literal
  - E.g.: a, !b

- An AND of literals is a product term or cube
  - E.g.: (a & c), (a & !b), (!a & !b & !c), c

- An OR of product terms is a sum of products (SOP), or in disjunctive normal form (DNF)
  - E.g.: (a & b) | (a & c)
  - (Dual: product of sums (POS), or conjunctive normal form (CNF))

Truth Table → SOP

- Simple but not very efficient
- Create a product term for each 1 entry
- Example with XOR:

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>a ^ b</th>
<th>(a &amp; b)</th>
<th>(a &amp; !b)</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>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Result: (!a & b) | (a & !b)

(Also possible: dual with 0s and CNF)

Inefficiency of Straight DNF

- Consider another example:

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

Result: (!a & b) | (a & b)

- By algebra, can simplify back to “b”
  - Factor: (!a | a) = 1, 1 & b = b
- Can we recognize these patterns earlier?

Logistics Intermission

- Sorry, no quiz 2s today
  - Good chance of grades by tomorrow and papers Wednesday
- Cache Lab due tonight
  - Moodle has been having some slowness
  - Suggest you allow a little extra time for final submission
- Assignment V out on Wednesday
  - Mostly logic design

Karnaugh Map Idea

- Write truth table entries in an array
- Product terms represented by certain rectangles
- Visually, find small number of rectangles to cover 1 bits
  - OK to cover more than once, combine with OR
  - Fewer rectangles = smaller circuit

2-variable “Karnaugh Map”
2-variable “Karnaugh Map” example

Result: \( \overline{a} \lor b \)

\[
\begin{array}{cc}
0 & 1 \\
0 & 1 \\
1 & 0 \\
1 & 1 \\
\end{array}
\]

Extending to 3 and 4 Variables

- Put two variables on a side
  - Weird order: 00 01 11 10
  - “Gray Code”: change only one bit at a time
- Rectangles can enclose 1, 2, 4, or 8 entries
  - Bigger is better
- Rectangles can wrap around the edges
  - 00 is adjacent to 10

4-variable Karnaugh Map Example

\[
\begin{array}{cccc}
00 & 10 & 11 & 01 \\
00 & 0 & 1 & 0 \\
10 & 0 & 1 & 0 \\
11 & 0 & 1 & 1 \\
01 & 0 & 1 & 1 \\
\end{array}
\]

(a \& b) | (a \& d) | (\overline{a} \& \overline{b} \& \overline{c} \& \overline{d})

Extending to 5 and 6 Variables

- 2D is no longer enough
  - No way to order 3 variables to capture 12 adjacencies
- Approach: stacking
  - Make 2 (for 5 inputs) or 4 (for 6 inputs) 4-input Karnaugh maps
  - Corresponding entries are “on top of” each other
  - Rectangles become 3D
  - Usually still drawn as 2D
  - With 6, more possibilities for wrapping too

5-variable Karnaugh Map Example

Karnaugh Map Tips: Overlap is Good

\[
\begin{array}{cccc}
00 & 10 & 11 & 01 \\
00 & 0 & 1 & 0 \\
10 & 0 & 1 & 0 \\
11 & 0 & 1 & 1 \\
01 & 0 & 1 & 1 \\
\end{array}
\]
Don’t Cares

- Some results don’t matter
  - Domain of function is a subset of all n-bit strings
  - Unused bit patterns in encodings
  - Bits sometimes ignored by other circuits
- “Don’t care” value could be 0 or 1
  - Usually denoted by X
- Don’t-cares allow designs to be simpler
  - Choose the value that allows a simpler circuit
- In early CPUs, led to undocumented instructions
  - Example: x86 ASL vs. SHL
  - On modern CPUs, more error checking

Dual (POS) Karnaugh Maps

- Pretend 0s are 1s
  - And vice-versa
- Negate final result

Dual (POS) Karnaugh Maps

- Pretend 0s are 1s
  - And vice-versa
- Negate final result

Dual (POS) Karnaugh Maps

- Pretend 0s are 1s
  - And vice-versa
- Negate final result

Dual (POS) Karnaugh Maps

- Pretend 0s are 1s
  - And vice-versa
- Negate final result
Automated Methods

- Karnaugh maps don’t scale well beyond 6 inputs
- Good job for a computer!
- Quine-McCluskey algorithm
  - Tabular analog to Karnaugh maps
  - Optimal, but suffers from exponential blowup
- Heuristic methods like “espresso”
  - First, greedily achieve coverage
  - Then, opportunistically improve
  - No optimality guarantee, but good scalability
- Now a standard part of CAD systems
  - Like compilers for software