4. Combinational Logic

6.004x Computation Structures
Part 1 – Digital Circuits

Copyright © 2015 MIT EECS
Functional Specifications

There are many ways of specifying the function of a combinational device, for example:

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

If C is 1 then copy B to Y, otherwise copy A to Y

Concise alternatives:

• *truth tables* are a concise description of the combinational system’s function.

• *Boolean expressions* form an algebra whose operations are AND (multiplication), OR (addition), and inversion (overbar).

\[ Y = \overline{C} \cdot \overline{B} \cdot A + \overline{C}BA + CB\overline{A} + CBA \]

Any combinational (Boolean) function can be specified as a truth table or an equivalent **sum-of-products** Boolean expression!
Here’s a Design Approach

1. Write out our functional spec as a truth table
2. Write down a Boolean expression with terms covering each ‘1’ in the output:
   \[ Y = \bar{C}\bar{B}A + \bar{C}BA + C\bar{B}A + CBA \]
3. We’ll show how to build a circuit using this equation in the next two slides.

This approach will always give us Boolean expressions in a particular form: \text{SUM-OF-PRODUCTS}
Sum-of-products Building Blocks

**INVERTER:**

\[ Z = \overline{A} \]

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

**AND:**

\[ Z = A \cdot B \]

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Z</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:**

\[ Z = A + B \]

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Z</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>
Straightforward Synthesis

We can implement SUM-OF-PRODUCTS with just three levels of logic:

1. Inverters
2. ANDs
3. OR

Propagation delay --
No more than 3 gate delays?*

*assuming gates with an arbitrary number of inputs, which, as we’ll see, isn’t a good assumption!
**ANDs and ORs with > 2 Inputs**

Replace 2-input AND gates with 2-input OR gates to create large fan-in OR gates.

\[ Z = A \cdot B \cdot C = (A \cdot B) \cdot C \]

**Chain:** Propagation delay increases linearly with number of inputs

\[ Z = ((A \cdot B) \cdot C) \cdot D \]

**Tree:** Propagation delay increases logarithmically with number of inputs

\[ Z = (A \cdot B) \cdot (C \cdot D) \]

Which one should I use?
More Building Blocks

NAND (not AND)  \[ Z = \overline{A \cdot B} \]  
\[
\begin{array}{ccc}
A & B & Z \\
0 & 0 & 1 \\
0 & 1 & 1 \\
1 & 0 & 1 \\
1 & 1 & 0 \\
\end{array}
\]

NOR (not OR)  \[ Z = \overline{A + B} \]  
\[
\begin{array}{ccc}
A & B & Z \\
0 & 0 & 1 \\
0 & 1 & 0 \\
1 & 0 & 0 \\
1 & 1 & 0 \\
\end{array}
\]

In a CMOS gate, rising inputs lead to falling outputs and vice-versa, so CMOS gates are naturally inverting. Want to use NANDs and NORs in CMOS designs... But NAND and NOR operations are not associative, so wide NAND and NOR gate can’t use a chain or tree strategy. Stay tuned for more on this!

XOR (exclusive OR)  \[ Z = A \oplus B \]  
\[
\begin{array}{ccc}
A & B & Z \\
0 & 0 & 0 \\
0 & 1 & 1 \\
1 & 0 & 1 \\
1 & 1 & 0 \\
\end{array}
\]

XOR is very useful when implementing parity and arithmetic logic. Also used as a “programmable inverter”: if A=0, Z=B; if A=1, Z=¬B

Wide fan-in XORs can be created with chains or trees of 2-input XORs.
Universal Building Blocks

NANDs and NORs are universal:

Any logic function can be implemented using only NANDs (or, equivalently, NORs). Good news for CMOS technologies!
CMOS ❤️ Inverting Logic

See “The Standard Cell Library” handout in Updates & Handouts

AND4:
\[ t_{PD} = 160 \text{ ps}, \text{ size} = 20 \mu^2 \]

NAND4 + INV:
\[ t_{PD} = 90 \text{ ps}, \text{ size} = 27 \mu^2 \]

Demorgan’s Laws:
\[ \overline{A \cdot B} = \overline{A} + \overline{B} \]
\[ \overline{A + B} = \overline{A} \cdot \overline{B} \]

2*NAND2 + NOR2:
\[ t_{PD} = 80 \text{ ps}, \text{ size} = 30 \mu^2 \]
Wide NANDs and NORs

Most logic libraries include 2-, 3- and 4-input devices:

![NAND circuits](image1)

But for a large number of inputs, the series connections of too many MOSFETs can lead to very large effective R. Design note: use trees of smaller devices...

![NAND and NOR circuits](image2)
CMOS Sum-of-products Implementation

NAND-NAND

\[ \overline{AB} = \overline{A} + \overline{B} \]

"Pushing Bubbles"

You might think all these extra inverters would make this structure less attractive. However, quite the opposite is true.

NOR-NOR

\[ \overline{AB} = \overline{A} + \overline{B} \]

\[ \overline{A} \overline{C} + AB + BC \]

\[ \overline{A} \overline{C} + AB + BC \]
Logic Simplification

Can we implement the same function with fewer gates? Before trying we’ll add a few more tricks in our bag.

**BOOLEAN ALGEBRA:**

**OR rules:**  \( a + 1 = 1, \ a + 0 = a, \ a + a = a \)

**AND rules:**  \( a1 = a, \ a0 = 0, \ aa = a \)

**Commutative:**  \( a + b = b + a, \ ab = ba \)

**Associative:**  \( (a + b) + c = a + (b + c), \ (ab)c = a(bc) \)

**Distributive:**  \( a(b+c) = ab + ac, \ a + bc = (a+b)(a+c) \)

**Complements:**  \( a + \bar{a} = 1, \ \bar{a}\bar{a} = 0 \)

**Absorption:**  \( a + ab = a, \ a + \bar{a}b = a + b \quad a(a + b) = a, \ a(\bar{a} + b) = ab \)

**Reduction:**  \( ab + \bar{a}b = b, \quad (a + b)(\bar{a} + b) = b \)

**DeMorgan’s Law:**  \( \bar{a} + \bar{b} = \bar{ab}, \ \bar{ab} = a + b \)
Boolean Minimization

Let’s (again!) simplify

\[ Y = \overline{CBA} + C\overline{BA} + CBA + \overline{CBA} \]

Using the identity

\[ \alpha A + \alpha \overline{A} = \alpha(A + \overline{A}) = \alpha \cdot 1 = \alpha \]

For any expression \( \alpha \) and variable A:

\[ Y = \overline{CBA} + C\overline{BA} + CBA + \overline{CBA} \]

\[ Y = \overline{CBA} + C + \overline{CBA} \]

\[ Y = \overline{C} A + C\overline{B} \]
Truth Tables with “Don’t Cares”

One way to reveal the opportunities for a more compact implementation is to rewrite the truth table using “don’t cares” (— or X) to indicate when the value of a particular input is irrelevant in determining the value of the output.

Note: Some input combinations (e.g., 000) are matched by more than one row in the “don’t care” table. It would be a bug if all the matching rows didn’t specify the same output value!
The Case for a Non-minimal SOP

$Y = \overline{CA} + CB$

NOTE: The steady state behavior of these circuits is identical. They differ in their transient behavior.

That's what we call a "glitch" or "hazard"
Karnaugh Maps: A Geometric Approach

K-Map: a truth table arranged so that terms which differ by exactly one variable are adjacent to one another so we can see potential reductions easily.

Here's the layout of a 3-variable K-map filled in with the values from our truth table:

<table>
<thead>
<tr>
<th>C \ AB</th>
<th>00</th>
<th>01</th>
<th>11</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></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>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>

It's cyclic. The left edge is adjacent to the right edge. (It's really just a flattened out cube).

Why did he shade that row Gray?
Extending K-maps to 4-variable Tables

4-variable K-map $F(A,B,C,D)$:

<table>
<thead>
<tr>
<th>$\overline{AB}$ \ $\overline{CD}$</th>
<th>00</th>
<th>01</th>
<th>11</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Again it’s cyclic. The left edge is adjacent to the right edge, and the top is adjacent to the bottom.

For functions of 5 or 6 variables, we’d need to use the 3$^{rd}$ dimension to build a 4x4x4 K-map. But then we’re out of dimensions...
Finding Implicants

An implicant

- is a rectangular region of the K-map where the function has the value 1 (i.e., a region that will need to be described by one or more product terms in the sum-of-products)
- has a width and length that must be a power of 2: 1, 2, 4
- can overlap other implicants
- is a prime implicant if it is not completely contained in any other implicant.

- can be uniquely identified by a single product term. The larger the implicant, the smaller the product term.
Finding Prime Implicants

We want to find all the prime implicants. The right strategy is a greedy one.

- Find the uncircled prime implicant with the greatest area
  - Order: $4 \times 4 \Rightarrow 2 \times 4$ or $4 \times 2 \Rightarrow 4 \times 1$ or $1 \times 4$ or $2 \times 2 \Rightarrow 2 \times 1$ or $1 \times 2 \Rightarrow 1 \times 1$
  - Overlap is okay
- Circle it
- Repeat until all prime implicants are circled

\[
\begin{array}{c|cc|cc|c|c|c}
\backslash(AB) & 00 & 01 & 11 & 10 \\
CD & \hline
00 & 0 & 1 & 1 & 1 \\
01 & 1 & 1 & 1 & 1 \\
11 & 1 & 1 & 1 & 1 \\
10 & 1 & 0 & 0 & 1 \\
\end{array}
\]
Write Down Equations

Picking just enough prime implicants to cover all the 1’s in the KMap, combine equations to form minimal sum-of-products.

\[
\begin{array}{c|cccc}
C \backslash AB & 00 & 01 & 11 & 10 \\
\hline
0 & 0 & 0 & 1 & 1 \\
1 & 0 & 1 & 1 & 0 \\
\end{array}
\]

\[Y = A\overline{C} + BC\]

\[
\begin{array}{c|cccc}
\backslash AB \backslash CD & 00 & 01 & 11 & 10 \\
\hline
00 & 0 & 1 & 1 & 1 \\
01 & 1 & 1 & 1 & 1 \\
11 & 1 & 1 & 1 & 1 \\
10 & 1 & 0 & 0 & 1 \\
\end{array}
\]

\[Y = D + BC + AC + BC\]

We're done!

Minimal SOP is not necessarily unique!
Prime Implicants, Glitches & Leniency

This circuit produces a glitch on Y when \( A=1, B=1, C: 1 \rightarrow 0 \)

\[
Y = \overline{C}A + CB
\]

<table>
<thead>
<tr>
<th>C ( \backslash ) AB</th>
<th>00</th>
<th>01</th>
<th>11</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

To make the circuit lenient, include product terms for ALL prime implicants.

\[
Y = \overline{C}A + CB + AB
\]
We’ve Been Designing a Mux

MUXes can be generalized to $2^k$ data inputs and $k$ select inputs …

2-input Multiplexer

<table>
<thead>
<tr>
<th>S</th>
<th>D₁</th>
<th>D₀</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>1</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>0</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>

… and implemented as a tree of smaller MUXes:
Systematic Implementation Strategies

Consider implementing some arbitrary Boolean function, \( F(A,B,C) \) ... using a MULTIPLEXER as the only circuit element:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C(_{in})</th>
<th>C(_{out})</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>

Full-Adder
Carry Out Logic

\( A, B, C_{in} \)
Generalizing:
In theory, we can build any 1-output combinational logic block with multiplexers.

For an N-input function we need a \(2^N\) input mux.

Is this practical for BIG truth tables?
How about 10-input function? 20-input?
A New Combinational Device

DECODER:
- k SELECT inputs,
- \( N = 2^k \) DATA OUTPUTs.

Select inputs choose one of the \( D_j \) to assert HIGH, all others will be LOW.

NOW, we are well on our way to building a general purpose table-lookup device.

We can build a 2-dimensional ARRAY of decoders and selectors as follows ...
Read-only Memory (ROM)

Full Adder

\[
\begin{array}{ccc}
A & B & S \\
0 & 0 & 0 \\
0 & 1 & 0 \\
1 & 0 & 0 \\
1 & 1 & 0 \\
\end{array}
\]

For K inputs, decoder produces \(2^K\) signals, only 1 of which is asserted at a time -- think of it as one signal for each possible product term.

Each column is large fan-in “NOR.” Note location of pulldowns correspond to a “1” output in the truth table!

One column for each output
Read-only Memory (ROM)

Full Adder

\[
\begin{array}{c|c|c|c}
\text{A} & \text{B} & \text{C}_i & \text{S} & \text{C}_o \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 & 0 \\
0 & 1 & 0 & 1 & 0 \\
0 & 1 & 1 & 0 & 1 \\
1 & 0 & 0 & 1 & 0 \\
1 & 0 & 1 & 0 & 1 \\
1 & 1 & 0 & 0 & 1 \\
1 & 1 & 1 & 1 & 1 \\
\end{array}
\]

For K inputs, decoder produces \(2^K\) signals, only 1 of which is asserted at a time -- think of it as one signal for each possible product term.

Each column is large fan-in “NOR.” Note location of pulldowns correspond to a “1” output in the truth table!

One column for each output

Shared decoder

Read-only Memory (ROM)
Read-only Memory (ROM)

Full Adder

\[
\begin{array}{c|c|c|c|c}
A & B & C_i & S & C_o \\
\hline
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 & 0 \\
0 & 1 & 0 & 1 & 0 \\
0 & 1 & 1 & 0 & 1 \\
1 & 0 & 0 & 1 & 0 \\
1 & 0 & 1 & 0 & 1 \\
1 & 1 & 0 & 0 & 1 \\
1 & 1 & 1 & 1 & 1 \\
\end{array}
\]

\text{LONG LINES slow down propagation times...}

The best way to improve this is to build square arrays, using some inputs to drive output selectors (MUXes):

2D Addressing: Standard for ROMs, RAMs, logic arrays...
Logic According to ROMs

ROMs *ignore* the structure of combinational functions ...
- Size, layout, and design are independent of function
- Any Truth table can be “programmed” by minor reconfiguration:
  - Metal layer (masked ROMs)
  - Fuses (Field-programmable PROMs)
  - Charge on floating gates (EPROMs)
  ... etc.

Model: LOOK UP value of function in truth table...
  Inputs: “ADDRESS” of a T.T. entry
  ROM SIZE = # TT entries...
  ... for an N-input boolean function, size $\approx 2^N \times \#\text{outputs}$
Summary

• **Sum of products**
  - Any function that can be specified by a truth table or, equivalently, in terms of AND/OR/NOT (Boolean expression)
  - “3-level” implementation of any logic function
    - Limitations on number of inputs (fan-in) increases depth
  - SOP implementation methods
    - NAND-NAND, NOR-NOR

• **Muxes used to build table-lookup implementations**
  - Easy to change implemented function -- just change constants

• **ROMs**
  - Decoder logic generates all possible product terms
  - Selector logic determines which terms are ORed together