Gates and Combinational Logic

1) Combinational Timing Analysis
2) Lenient Gate Specifications
3) Universal Gates
4) Large Fan-in Gates
5) Logic Design Methodologies
6) Minimal SOP Realizations
7) Glitches & Static Hazards
8) Multiplexer Logic

A Quick Review

- A **combinational device** is a circuit element that has
  - one or more digital inputs
  - one or more digital outputs
  - a functional specification that details the value of each output for every possible combination of valid input values
  - a timing specification consisting (at minimum) of an upper bound $t_{PD}$ on the required time for the device to compute the specified output values from an arbitrary set of stable, valid input values

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

I will generate a valid output in no more than 2 weeks after seeing valid inputs
Delays, They’re a Fact of Life

Propagation delay (t_{PD}): An UPPER BOUND on the delay from valid inputs to valid outputs.

A common design goal is to minimize propagation delay!

Contamination Delay

INVALID inputs take time to propagate, too...

Do we really need t_{CD}? Usually not... it'll be important when we design circuits with registers (coming soon!). Sometimes you will see it spec'ed with the dubious name "minimum propagation delay".

If t_{CD} is not specified, safe to assume it's 0.
The Combinational Contract

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>( t_{PD} ) propagation delay</th>
<th>( t_{CD} ) contamination delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Hey, that's nota bean! Opps… nota bene

N.B:
1. No Promises during
2. Default (conservative) spec: \( t_{CD} = 0 \)

Example: Timing Analysis

If NAND gates have a \( t_{PD} = 4nS \) and \( t_{CD} = 1nS \)

\( t_{PD} = _____ nS \)

\( t_{CD} = _____ nS \)

\( t_{PD} \) is the maximum cumulative propagation delay over all paths from inputs to outputs.

\( t_{CD} \) is the minimum cumulative contamination delay over all paths from inputs to outputs.
One Last Issue...

Recall the rules for combinational devices:

Output guaranteed to be valid when all inputs have been valid for at least $t_{PD}$ and, outputs may become invalid no earlier than $t_{CD}$ after an input changes!

Many gate implementations--e.g., CMOS—adhere to even tighter restrictions.

What Happens In This Case?

LENIENT Combinational Device:
Output guaranteed to be valid when any combination of inputs sufficient to determine output value has been valid for at least $t_{PD}$. Tolerates transitions -- and invalid levels -- on irrelevant inputs!
Now We're Ready to Design Stuff!

We need to start somewhere -- usually it's the functional specification

```
<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

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

Argh... I'm tired of word games

Truth Table

<table>
<thead>
<tr>
<th>C</th>
<th>B</th>
<th>A</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 you are like most scientist you'd rather see a formula than parse a logic puzzle. The fact is, any combinational function can be expressed as a table.

These "truth tables" are a concise description of the combinational system's function. Conversely, any computation performed by a combinational system can expressed as a truth table.

Where Do We Start?

We have a bag of gates.

We need a systematic approach for designing logic
A Slight Diversion

Are we sure we have all the gates we need?
Just how many two-input gates are there?

<table>
<thead>
<tr>
<th>AND</th>
<th>OR</th>
<th>NAND</th>
<th>NOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>AB</td>
<td>Y</td>
<td>AB</td>
<td>Y</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>00</td>
<td>1</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>01</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>11</td>
<td>0</td>
</tr>
</tbody>
</table>

Hum... all of these have 2-inputs (no surprise)
... each with 4 permutations, giving $2^2$ output cases
How many permutations of 4 outputs are there? ___

There Are Only So Many Gates

There are only 16 possible 2-input gates
... some we know already, others are just silly

<table>
<thead>
<tr>
<th>I</th>
<th>N</th>
<th>P</th>
<th>Z</th>
<th>X</th>
<th>N</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>O</td>
<td>A</td>
<td>B</td>
<td>X</td>
<td>N</td>
<td>O</td>
</tr>
<tr>
<td>O</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td>O</td>
<td>B</td>
<td>O</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>AB</th>
<th>O</th>
<th>D</th>
<th>B</th>
<th>A</th>
<th>B</th>
<th>R</th>
<th>R</th>
<th>R</th>
<th>B</th>
<th>B</th>
<th>A</th>
<th>A</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Do we need all of these gates?
Nope. After all, we describe them all using AND, OR, and NOT.
We Can Make Most Gates Out of Others

<table>
<thead>
<tr>
<th>B&gt;A</th>
<th>Y</th>
<th>XOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>00</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>01</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>11</td>
</tr>
</tbody>
</table>

How many different gates do we really need?

One Will Do!

NANDs and NORs are universal

Ahl!, but what if we want more than 2-inputs
Stupid Gate Tricks

Suppose we have some 2-input XOR gates:

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

\[t_{pd} = 1\]
\[t_{cd} = 0\]

And we want an N-input XOR:

output = 1
iff number of 1s input is ODD
("ODD PARITY")

\[t_{pd} = O(\_\_\_\_\_) -- WORST CASE.\]

Can we compute N-input XOR faster?

I Think That I Shall Never See
a Gate Lovely as a ...

N-input TREE has \(O(\_\_\_\_\_)\) levels...

Signal propagation takes \(O(\_\_\_\_\_)\) gate delays.

Question: Can EVERY N-Input Boolean function be implemented as a tree of 2-input gates?
Are Trees Always Best?

Alternate Plan: Large Fan-in gates
- N pulldowns with complementary pullups
- Output HIGH if any input is HIGH = “OR”
- Propagation delay: \(O(\_\_\_\_\_\_\_)\) since each additional MOSFET adds \(C\)

Don’t be misled by the “big O” stuff... the constants in this case can be much smaller... so for small \(N\) this plan might be the best.

Here’s a Design Approach

1) Write out our functional spec as a truth table
2) Write down a Boolean expression for every ‘1’ in the output

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

3) Wire up the gates, call it a day, and go home!

This approach will always give us logic expressions in a particular form: SUM-OF-PRODUCTS

- it’s systematic!
- it works!
- it’s easy!
- we get to go home!
Straightforward Synthesis

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

INVERTERS/AND/OR

Propagation delay --
No more than 3 gate delays
(ignoring fan-in)

Logic Simplification

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

RULES 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: \( a + \bar{a} = 1, \ \bar{a}a = 0 \)
Absorption: \( a + ab = a, \ a + \bar{a}b = a + b \)
\( a(a+b) = a, \ a(\bar{a} + b) = ab \)
Reduction: \( ab + \bar{a}b = b, \ (a + b)(\bar{a} + b) = b \)
DeMorgan's Law: \( \bar{a} + \bar{b} = \bar{ab}, \ \bar{ab} = a + b \)
Minimal Sum-of-Products

Once a logic expression is in Sum-of-Product (SOP) form, it is easy to minimize it further. In fact, we can reduce it to minimal set of products.

Here’s the trick:

Factor out common terms and collapse groups of “don’t care” bits.

\[ Y = \overline{CBA} + \overline{CBA} + CBA + CBA \]
\[ Y = \overline{CA(B + B)} + CB(A + A) \]
\[ Y = \overline{CA} + CB \]

How do we find these “don’t care” groups

A Geometric Approach

It is easy to identify clusters of “don’t care” bits geometrically.

Products with these clusters, have terms that differ in exactly one bit position. So if we could arrange all possible products so that those which differ by exactly one bit are adjacent to one another then we could see these clusters easily.

We’ve seen such an arrangement of codings before when we computed Hamming distances.

I just knew they’d bring me back for a sequel
Karnaugh Maps

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

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

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

Jump to Hyperspace

4-variable K-map for a multipurpose logic gate:

\[ Y = \begin{cases} 
A \cdot B & \text{if } CD = 00 \\
A + B & \text{if } CD = 01 \\
B & \text{if } CD = 10 \\
A \oplus B & \text{if } CD = 11 
\end{cases} \]

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

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

We can identify clusters of “don’t care” bits by circling adjacent subcubes of 1s. A subcube is just a lower dimensional cube.

The best strategy is generally a greedy one. Find, the largest subcube and circle it first. Continue circling the largest subcubes possible (even if it has some overlap with a previous one). Then proceed onto smaller and smaller subcubes until no 1s are left.

Write Down Equations

Write down a product term for the portion of each cluster/subcube that is invariant.

This must be all. Let’s go home!
Review: K-map minimization

1) Copy truth table into K-Map
2) Identify subcubes,
   selecting the largest available at each step (even if it involves some
   overlap) until all ones are covered
3) Write down the answer

Does it always work? Not really...
There's still a bit of art to it

The circled terms are called implicants. An
implicant not completely contained in another
implicant is called a prime implicant.

Are K-maps Really All That Useful?

- They are only manageable for small circuits
  (4-5 inputs at most)
- Sometimes you pick the wrong set of subcubes
- There are better techniques
  (better for computer's that is)
- SOP realizations aren't all that relevant
- We've got gates to burn
- Low fan-in gates are better suited to current technologies
  that SOP (FPGAs, Standard Cells)
- Sometimes minimal circuits are glitchy
  (more about this later)
- Some important circuits aren't amenable to minimal SOP
  realizations
That Gate has a Name!

The gate we've been designing for this entire lecture is a relatively important one.

<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>

A Case for Non-Minimal SOP

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

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

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

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

Lenient Mux

<table>
<thead>
<tr>
<th>C</th>
<th>B</th>
<th>A</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>X</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>X</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>X</td>
<td>1</td>
</tr>
<tr>
<td>X</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>X</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Useful Gate Structures

NAND-NAND

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

NOR-NOR

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

"Pushing Bubbles"

More Useful Gate Structures

AOI (AND-OR-INVERT)

OAI (OR-AND-INVERT)

An OAI's DeMorgan equivalent is usually easier to think about.

AOI and OAI structures can be realized using a single CMOS gate. However, their function is equivalent to 3 levels of logic.
Logic That Defies SOP Realization

<table>
<thead>
<tr>
<th>C</th>
<th>A</th>
<th>B</th>
<th>S</th>
<th>C^2</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>1</td>
<td>1</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>1</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>1</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>

Full Adder

S = ABC + ABC + ABC + ABC

C_o = ABC + ABC + ABC + ABC

Can simplify the carry out easy enough, eg...

C_o = BC + AB + AC

But, the sum, S, doesn't have a simple sum-of-products implementation even though it can be implemented using only two 2-input gates.

Logic Synthesis Using MUXes

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

2-input Multiplexer

Truth Table

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

A 4-input Mux implemented as a tree
Systematic Implementation of Combinational Logic

Consider implementation of some arbitrary Boolean function, $F(A,B)$ ... 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

Small Improvements

We can also apply certain optimizations to MUX Logic

- Largely by inspection or exhaustive search
- N-input gate with N-1 input MUX & one inverter

There's something interesting going on in those MUXs
Summary

- Timing specs
  - $t_{PD}$: upper bound on time from valid inputs to valid outputs
  - $t_{CD}$: lower bound on time from invalid inputs to invalid outputs
  - If not specified, assume $t_{CD} = 0$

- Combinational logic
  - Any function that can be specified by a truth table
  - Expressed in terms of AND/OR/NOT
  - Minimally, we can get away with just 2-input NANDs or NORs
  - Max number of inputs = 4? Then use multiple levels of logic

- Sum-of-products canonical form
  - Comes directly from truth table
  - “3-level” implementation of any logic function
  - Limitations on number of inputs (fan-in) increases depth

Now we go home!