Lecture 5:

Gate Logic
Logic Optimization

Overview

Reading

McCluskey, Logic Design Principles- or any text in boolean algebra

Introduction

We could design at the level of irsim

- Think about transistors as switches
- Build collections of switches that do useful stuff
- Don't much care whether the collection of transistors is a gate, switch logic, or some combination. It is a collection of switches.

But this is pretty complicated

- Switches are bidirectional, charge-sharing,
- Need to worry about series resistance…

Logic Gates

Constrain the problem to simplify it.

- Constrain how one can connect transistors
  - Create a collection of transistors where the

    Output is always driven by a switch-network to a supply (not an input)

    And the inputs to this unit only connect the gate of the transistors
- Model this collection of transistors by a simpler abstraction

    Units are unidirectional

    Function is modelled by boolean operations

    Capacitance only affects speed and not functionality

    Delay through network is sum of delays of elements

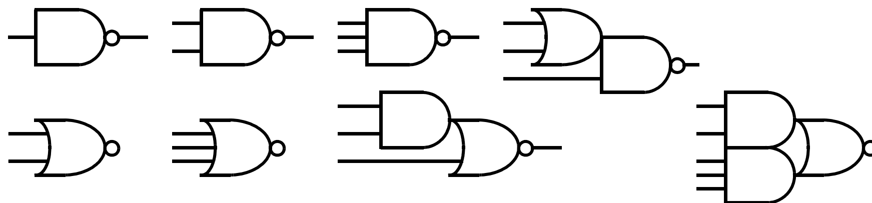
This abstract model is one we have used already.

- It is a logic gate

Logic Gates

Come in various forms and sizes

In CMOS, all of the primitive gates[1] have one inversion from each input to the output.

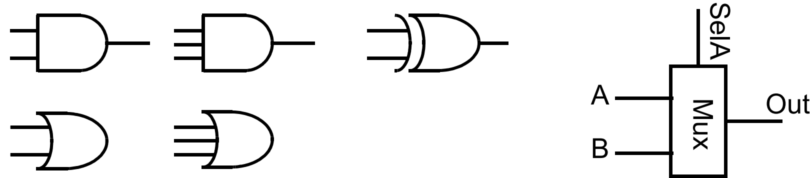


There are many versions of primitive gates. Different libraries have different collections. In general, most libraries have all 3 input gates (NAND, NOR, AOI, and OAI gates) and some 4 input gates. Most libraries are much richer, and have a large number of gates.

1. A primitive gate is one where all the inputs directly drive the gate of a transistor in a switch-network that is connected to the output. This means that the gate consists of two switch networks, one connected to Vdd and the other connected to Gnd.

Logic Gates

Many systems provide more complex logic gates than just single inversion structures. These usually include non-inverting gates (AND and OR) as well as more complex functions like XOR and Mux.



The logic gates provided in the EE271 library is pretty limited. They are:

inv, nand2 nand3 nand4, nor2 nor3,

aoi21 aoi22, oai21 oai22,

xor xnor, tri, mux2, latch latch_b

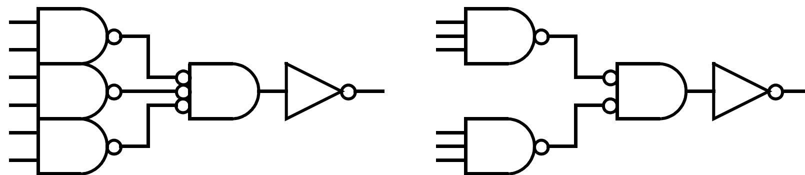
Logic Design Problem

Given a functional specification (or description)

Find an interconnection of gates that generates the same outputs as the specification.

Clearly there are many 'correct' solution to the logic design problem.

For example look at a 6 input NAND gate



And there are many more …

Inv, NOR3,NAND2; Inv, NOR2, NAND3

Bubble Conventions (Aside)

Every gate has two representations depending on its use. When the inputs are active high, the symbol with the bubble on the output should be used. When the inputs are active low (negative true) the symbols with the bubbles on the inputs should be used.

For example a NOR gate can be drawn like[1]:

Goal is to make the bubbles line up so the intent is easier to understand (see previous slide)

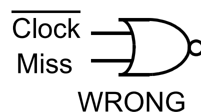
Always used the active low symbol when all the inputs are inverted.

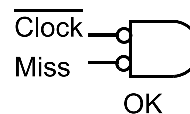
1. This is just Demorgan's Law.

- Bubbles

When some inputs are inverted and others are not, choose the symbol that represents how you think about the function.

For example if you wanted to create a signal that was $\text{Clock} * \overline{\text{Miss}}$, you might use the following circuit





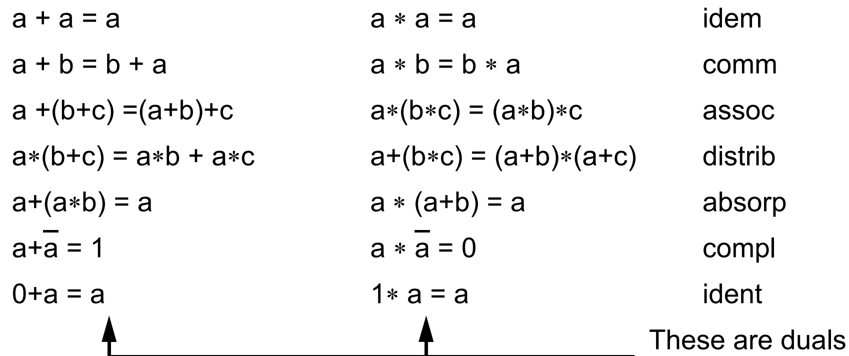
Since the intent is a logical AND function, you should draw the schematic to show an AND gate, rather than a NOR gate.

While this might not seem like a big deal to you, it will make your schematics much easier to read and understand when you need to look at a circuit you designed 6 months ago to fix a newly found bug.

Logic Minimization

There are many ways to implement a functional specification, and some are better than others. Can use the rules of boolean algebra to minimize the expressions:

Some of the boolean equalities:



$a + a = a$	$a * a = a$	idem
$a + b = b + a$	$a * b = b * a$	comm
$a + (b+c) = (a+b)+c$	$a*(b*c) = (a*b)*c$	assoc
$a*(b+c) = a*b + a*c$	$a+(b*c) = (a+b)*(a+c)$	distrib
$a+(a*b) = a$	$a * (a+b) = a$	absorp
$a+\bar{a} = 1$	$a * \bar{a} = 0$	compl
$0+a = a$	$1* a = a$	ident

2-Level Logic Minimization

Called 2-level since it works on minimizing a sum of products representation of the function:

$f = (\ ) + (\ ) + (\ ) + \ldots$

in each () is a AND of input terms, called a product

Example:

$f = a\,b + b\,c + c\,a$

$f = \bar{a}\,b\,c + \bar{c}$

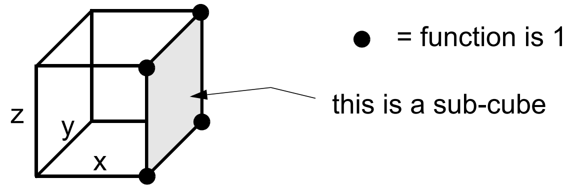
Logic can be simplified by reducing the number of product terms, and the number of inputs in each product by finding the smallest number of boolean subcubes that 'cover' the function

Basic Idea:

$A\,(B + \bar{B}) = A$

Karnaugh Maps

Are a way to view a boolean n-cube:



Create a table where adjacent entries differ by only one variable:

C \ AB	00	01	11	10
0				
1				

	00	01	11	10
00				
01				
11				
10				

Find a small number of large faces (large regions in map) that cover 1s

- Or if the function is mostly 1, invert it, and the function that maps the 0s

- Karnaugh Map Example

For a function, f

CD \ AB	00	01	11	10
00	1	0	0	1
01	0	1	0	0
11	1	1	1	1
10	1	1	1	1

- Karnaugh Map Example

For a function, f

CD \ AB	00	01	11	10
00	1	0	0	1
01	0	1	0	0
11	1	1	1	1
10	1	1	1	1

The min sum of products is

$f = C + \overline{B}\,\overline{D} + \overline{A}\,B\,D$

But sum of products leads to gates with large fanin (both AND and OR)

- Good to see some simplifications
- Not necessarily what you want to implement in CMOS

Logic Minimization

CMOS logic is often minimized with multi-level logic optimization

Logic represented by:

- sum of products of sums of products of sums …
- More levels of logic traded for reduced fanin.

Example:

Sum of Products = adf + aef + bdf + bef + cdf + cef + g

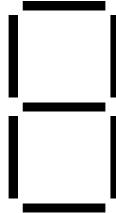
- 6 — 3input AND gates (x6)
- 3 — 3input OR (to implement the 7 input OR)

Multi-level = (a + b + c) (d + e) f + g (factored version)

- 1 — 3 input OR
- 2 — 2 input OR
- 1 — 3 input AND

+ Example of Logic Minimization[1]

- Draw Karnaugh map of functions
- Find *minimal* set of implicants that cover all 1s of all functions
- Example, four segments of 7-segment decoder



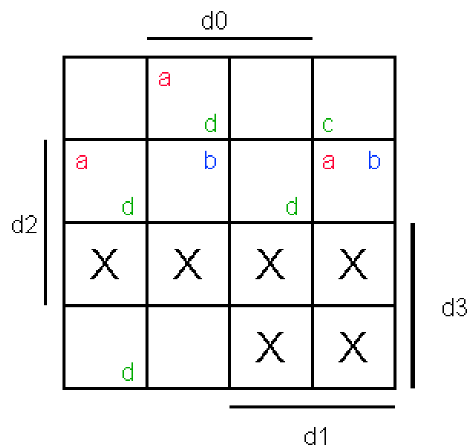
No	d3	d2	d1	d0	sa	sb	sc	sd	se	sf	sg
0	0	0	0	0	1	1	1	1	1	1	0
1	0	0	0	1	0	1	1	0	0	0	0
2	0	0	1	0	1	1	0	1	1	0	1
3	0	0	1	1	1	1	1	1	0	0	1
4	0	1	0	0	0	1	1	0	0	1	1
5	0	1	0	1	1	0	1	1	0	1	1
6	0	1	1	0	0	0	1	1	1	1	1
7	0	1	1	1	1	1	1	0	0	0	0
8	1	0	0	0	1	1	1	1	1	1	1
9	1	0	0	1	1	1	1	0	0	1	1

1. Taken from Prof Dally's lecture notes

+ Direct Synthesis Example

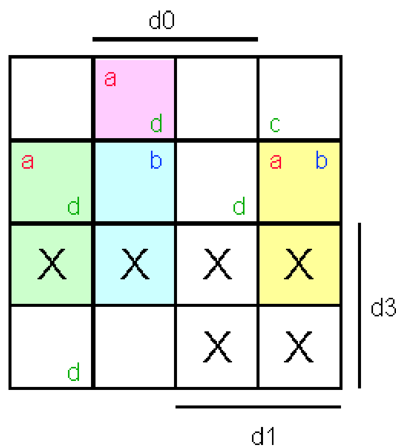
No	d3	d2	d1	d0	sa	sb	sc	sd	se	sf	sg
0	0	0	0	0	1	1	1	1	1	1	0
1	0	0	0	1	0	1	1	0	0	0	0
2	0	0	1	0	1	1	0	1	1	0	1
3	0	0	1	1	1	1	1	1	0	0	1
4	0	1	0	0	0	1	1	0	0	1	1
5	0	1	0	1	1	0	1	1	0	1	1
6	0	1	1	0	0	0	1	1	1	1	1
7	0	1	1	1	1	1	1	0	0	0	0
8	1	0	0	0	1	1	1	1	1	1	1
9	1	0	0	1	1	1	1	0	0	1	1

Build 0 function since more 1s than 0s



The dark lines on the outside of the table indicate where that input is true (i.e. d0 is true in the middle two columns). Place a 'a' 'b' 'c' 'd' in each square where that segment should be 0. Enter an X for each entry that doesn't matter (these are numbers greater than 9, which are not legal input). For the 'X' inputs you don't care what the logic outputs.

+ Implicants



$sa' = (d0' \wedge d1' \wedge d2) \vee (d0 \wedge d1' \wedge d2' \wedge d3') \vee (d0' \wedge d1 \wedge d2)$

$sb' = (d0 \wedge d1' \wedge d2) \vee (d0' \wedge d1 \wedge d2)$

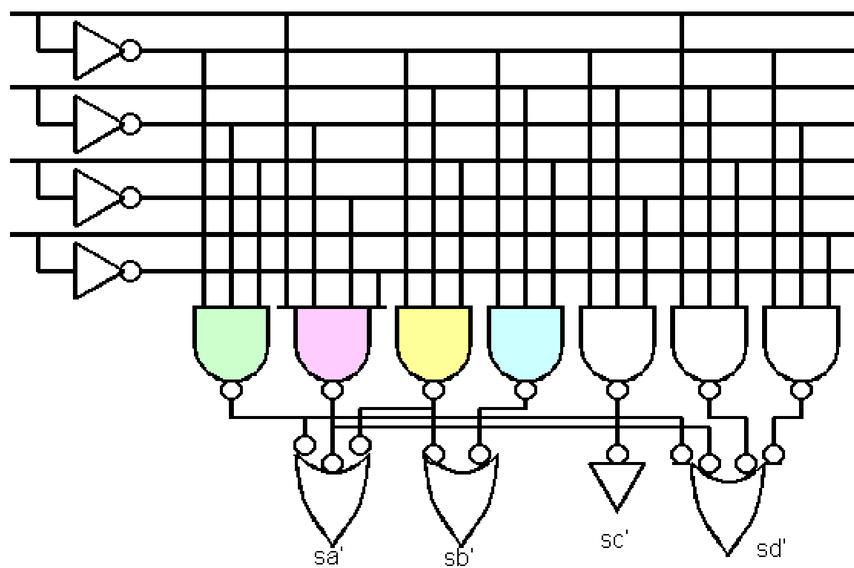
$sc' = (d0' \wedge d1 \wedge d2')$

$sd' = (d0 \wedge d1' \wedge d2' \wedge d3') \vee (d0' \wedge d1' \wedge d2) \vee (d0 \wedge d1 \wedge d2) \vee (d0' \wedge d1' \wedge d3)$

OR all the squares where the output must be 0. Some of the squares (implicants) will be shared

The cost is the sum of all the inputs (called literals) used

+ Resulting Logic

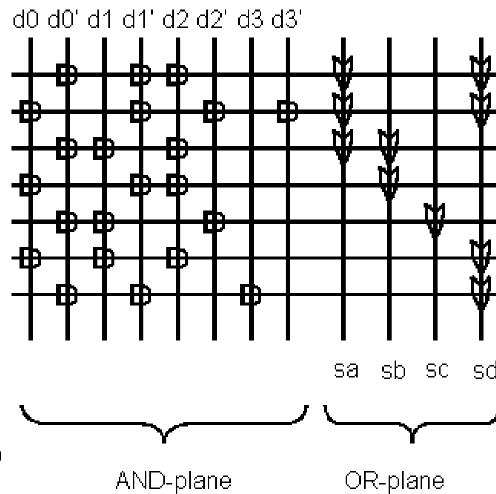


Note that the resulting logic is built from two levels of NAND gates (or could be built from two levels of NOR gates)

+ PLA

Are a way of directly implementing sum of product designs

- These might have large fanin gates, so use nMOS NOR style gates
  - The AND plane is a decoder. The horizontal line rises if all the inputs are low. Each AND symbol represents a pulldown transistor in an nMOS style NOR gate.
  - The OR plane combines the outputs of the AND plane. The horizontal lines become the inputs, and the vertical line falls if any of the inputs are high. Each OR symbol represents a pulldown device
  - Very regular wiring



More about this structure later in the quart

Something is Wrong

Be careful when you talk about optimization

- Make sure you know what you are optimizing
- Make sure you are optimizing something you care about

We have been talking about logic optimization without defining metric!

- There are many logic minimization methods

  Often hard (requires lots of computer time)

  Might not provide you with a better solution for YOUR problem

  Often less logic is not a better solution

What are some possible metrics?

Objective Functions

Here are some issues a logic designer might try to improve:

1. Correctness/Simplicity

   Does it implement the correct function? How hard will it be to verify that the circuit really works? How hard will it be to test the chip?

2. Area

   How much space does it take to build this circuit. Is it small enough? This depends on the implementation technology (board / std cell / custom)

3. Speed

   How fast will the circuit run. Is the maximum clock rate fast enough?

4. Design Time

   How long have you been working on it. Is it time to call it done?

5. Power

Simplicity

Cleverness is often overrated

- Don't be over clever and under smart
- You are responsible for creating a correct implementation
  - Make sure you understand how it works
  - Make sure it works under all cases
  - Generate test vectors that demonstrate that it works

  A simple solution is always the easiest to understand
- Sometimes simple circuits don't meet the other specs
  - Need to innovate on these circuits
  - Try to find the places that give you the most return on your design time.

  Doing function in software is a great solution when it is good enough

Area Objective

Size metric depends heavily on implementation technology

- In a board level design, memory is very cheap, since a 28pin chip space can hold 1-4Mbits of SRAM and 64Mbits of DRAM.

  Very area efficient to use lots of memory for board level designs
- For a gate array solution, memory is quite expensive, since each memory cell uses a few gate positions. In fact the situation is so bad that many gate arrays have the ability to contain embedded memories. Even so, the technology is not the best for memory, and so you get a few memory cells / logic gate
- Custom layout, which can use custom designed memory arrays, and datapaths can save a large amount of area for some regular structures, but requires more effort to create (need to watch design time)
- Chips sometimes have minimum allowable area. Smaller is not necessarily better

Area Estimation

Using Standard Cells

Most libraries will give you the area of each cell you are using (for a gate array / std cell based approach). You should take the cell area and multiply by 3 (for a 2-layer metal design, about 2 for 3-layer metal) to get an area estimate of the design.

Custom Layout

This depends much more on the layout style used. To find the answer usually requires lots of work (you need to do the layout).

On average, a good rule of thumb is that the area will be 25 times the area of the transistor gate area used in the gates.

Arrays tend to be denser (for memories / datapaths) because the wires are more constrained, but at this level you should not worry about it

Bottom Line

Fewer gates, and fewer inputs (less wires) mean smaller areas.

Performance Issues

Also heavily depends on implementation technologies

- Board-level design

    Number of package crossings is the key

    10ns 256kbit SRAM, and 6ns inverter/ buffers

    Logic in a single PAL 10ns, two PALs 20ns

- Chip Design

    Gate speed depends on two factors — resistance and cap

    Resistance is set by size of driving transistors, and # in series.

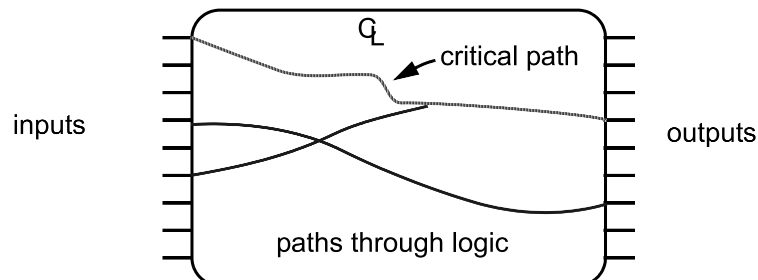
    Capacitance is set by the wiring and the fanout.

    Faster circuits

        Shorter wires, lower fanin, lower fanout, and less gates in series.

Critical Paths

There are many signal paths through a set of combinational logic.



Not all the paths have the same delay

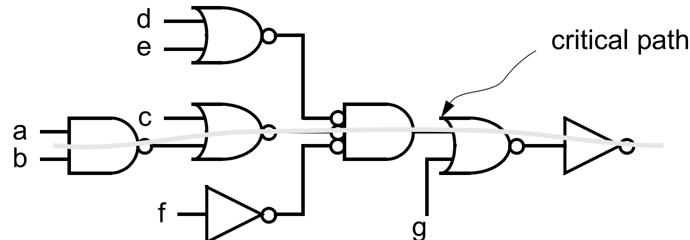
- Path from input to latest output is called the critical path

    It is this path that will slow down machine, since clock has to wait for all the outputs to be valid.

Critical Paths

Look at the function

$(\overline{a\,b} + c)\,(d + e)\,f + g$



If all the inputs change at the same time

Worst-case path will probably be from the inputs {a, b}

Speeding up g won't help much

Critical Paths

There are many paths from the inputs to the outputs, and the machine must run at the speed of the slowest path. In a good design you want to try to balance the delays through all the paths, so no paths are much slower than the rest.

Say the function you were implementing was a decoder

Out = $A_0\ A_1\ A_2\ A_3\ A_4$ RegRead

where RegRead was generated elsewhere, but is

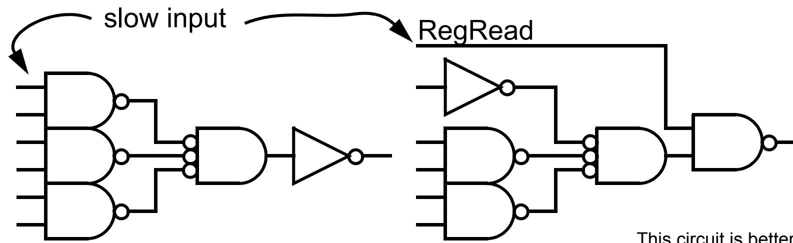
$\overline{\text{MemStall + Istall + Exception}}$

and exception can be generated from a number of other signals

While the function you need is a 6 input AND gate, not all the inputs arrive at the same time. One of the inputs will be later than the others. To minimize the delay for the critical path, will lead to a different implementation than just minimizing the delay of a 6 input AND gate

6 Input NAND with Late Input

The design that minimizes the delay through the gate is shown on the left, while the gate on the right will have a smaller critical path



better

This circuit is better because it decreases the delay from the slow input to the output. To decrease this delay, the circuit actually increases the delay from all the other inputs (since the delay of the NAND gate is slower than the final inverter. As long as these other inputs arrive early enough, it is a win.

Faster circuits

better balanced paths

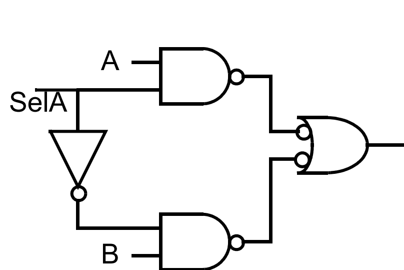
+ Switch Logic

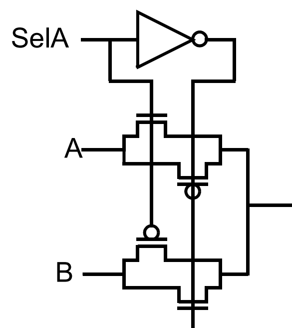
Sometimes you get a great advantage by going back to the switch model

- Switch logic has it place

Compare a 2 input mux

Out = SelA * A + $\overline{\text{SelA}}$ *B





In the design shown, the output resistance of the mux, is equal to the resistance of the transmission gate PLUS the resistance of the gate driving the A (or B) input. Thus you would not want to cascade these gates since the resistance would get too large

Often encapsulated inside of ‘logic gates’

Gate Design Summary

Moved up a level in abstraction. More tools (boolean algebra) are available to improve circuits.

- Need to understand what we want to improve before optimizing
- CAD tools are available to help with this level of optimization

  Espresso for 2-level; mis, synopsys for multilevel

Yet we are still working at a pretty low level

- Average gate has 6ish transistors
- Still need to design stuff with 100K gates
- Writing equations / drawing schematics for all these gates is hard

Want to work at a higher level first

- Work at level of adders, registerfiles, control sections
- Need to check function / algorithm before polish logic gates

Generating Initial Specification

The whole logic design problem is to create a circuit that meets some functional specification.

- How was this spec given to you?
- How do you know what it means?
- How do you know that it works? (does the desired function)

Maybe the functional spec should be executable, and should form the starting point of the logic design.

- Not a new thought
- Languages have been developed to do this, called HDL

  Hardware Description Languages
- Tools can take subsets of HDLs and generate logic

  Talk about these languages in the next lecture