*Use pencil!*

# Manipulation of logic (2.5, The Axioms of Logic handout on webpage)

There are lots of logic properties, and most have a dual. You can prove these in a number of ways, but the easiest is to show that the truth tables of both are equivalent. This is informally called “proof by truth table” and formally called “perfect induction”

|  |  |  |
| --- | --- | --- |
| **Property** |  |  |
| Commutative | a+b = b+a | a\*b=b\*a |
| Associative | (a+b)+c = a+(b+c) |  |
| Combining\* | a\*b+a\*b’=a |  |
| DeMorgan’s | a’\*b’=(a+b)’ |  |
| Distributive | a\*b+a\*c=a\*(b+c) |  |
|  |  |  |

\*This one isn’t in our book AFAIK.

Can we use Combining to go from Canonical SoP to our other solution for the MUX?

There is a lot more in this section, you’ll need to read it.

# And now for something completely different… (maybe)Binary (and Hex) numbers (1.2)

Consider the number 123

1 2 3

Each *place* has a value. We normally work in base 10, so each place is 10 times bigger than the last.

In binary we work in base 2. Consider the number 100102 (the subscript indicates the base).

1 0 0 1 0

Now what do you suppose the value of 10.112 is?

1 0 . 1 1

Let’s convert 21 into base 2.

*(Time allowing we’ll cover this in class. You need to know it in any case.)*

Now consider base 16. We need symbols for 0-15 but only have them for 0-9 in decimal. So we’ll use A=10, B=11, C=12, D=13, E=14, F=15.

What is 1F16 in decimal? What is 44 in hexadecimal (base 16)?

Converting between base 2 and base 16 is easy. Just group the binary digits into groups of 4 starting at the decimal point. So what is 100110012 in hex? We commonly use hex to represent large numbers which we’d prefer to use binary for just to make it more readable…

# And the payoff

Why is binary

# And the payoff…

Why did we do binary numbers on the first day? How are they relevant to digital logic?

The point is that we can represent binary numbers using basic logic. Consider a device that adds two one-digit binary numbers and outputs a 2 digit binary number. Let the inputs be A and B and the output be R[1:0].

(The R[1:0] is a way two write that there are two outputs, R1 and R0. We might also write R[7:0] to indicate that there are 8 outputs: R7, R6, R5, R4, R3, R2, R1, and R0.)

 A

+ B

====

 R1R0

Write the truth table for this adder. R1 is to be the most significant digit (farthest to the left in the 2’s place in this example) while R0 is to be the least significant digit (farthest to the right, in the 1’s place). Then draw the logic gates.

|  |  |  |  |
| --- | --- | --- | --- |
| A | B | R1 | R0 |
| 0 | 0 |  |  |
| 0 | 1 |  |  |
| 1 | 0 |  |  |
| 1 | 1 |  |  |

The point is that we can do arithmetic using basic logic. You may say “great, I can add two one-bit numbers”. But it turns out we can use this basic idea of using logic states to represent numbers to do all kinds of math. A modern computer can easily do 5-10 billion additions of 64-bit numbers in a second! All based on this basic idea.

Consider adding two 3-digit binary numbers.

* If we used the truth table scheme, how many rows would there be?
	+ What about for a 64-bit addition?
* Can we break the problem down in a way that works better?
	+ How?
* Let’s work on it…

# Solving problems with logic—Computer Arithmetic (2.7, 4.3) *Continued*

 0 1 0 1 0 1 1 1 1 1 1 1

 + 1 0 1 + 0 1 1 + 0 0 1 + 1 1 1

 ======= ======= ======= =======

Now, given that practice, we can see that when *we* do the work we have the two 3-digit numbers as inputs, up to 4 bits of output and 3 “internal” bits—the carries.

 *C3 C2 C1*

 A2 A1 A0

+ B2 B1 B0

==========

 **S3 S2 S1 S0**

Solving this problem all at once is pretty tricky (and big, that’s 64 truth table entries!), and probably unreasonable. So instead let’s take advantage of the symmetry of the problem. What I mean is that for a given *column* above is one “A” one “B” and one “C” as input and one “S” and one “C” as output. Let’s say we magically had a box that took A, B and Cin as inputs and generated S and Cout as outputs. We’d need 3 of them (one for each column). How would we connect them?

A

R

B

Cout

Cin

A

R

B

Cout

Cin

A

R

B

Cout

Cin

*4-bit ripple-carry adder.*



Figure by Dr. Liu, FSU

Now let’s figure out the logic for each of these boxes (full adders)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| A | B | Cin | Cout | S |
| 0 | 0 | 0 |  |  |
| 0 | 0 | 1 |  |  |
| 0 | 1 | 0 |  |  |
| 0 | 1 | 1 |  |  |
| 1 | 0 | 0 |  |  |
| 1 | 0 | 1 |  |  |
| 1 | 1 | 0 |  |  |
| 1 | 1 | 1 |  |  |

## What did we do?

We’ve done a bunch of things.

* First we identified a fairly complex problem (6 inputs, 4 outputs)
* We then broke it down (by noticing the symmetry) into smaller parts (3 inputs, 2 outputs) that we could solve “by truth table”.
* We also came up with a scalable solution (it will work for 8-bit numbers too, right?)
* And we built one of the basic building blocks of digital logic, the full adder.

The kind of adder we got when we combined those full adders is a “ripple carry adder”. It is fairly slow because we need to wait for the carries to “ripple” down from one full adder to the next. That might not be obvious, but we’ll tackle the notion of delay later.

# More Gates (section 2.8)

|  |  |  |
| --- | --- | --- |
| **Name** | **Logic equation** | **Symbol** |
| NOR | !(X + Y) |  |
| NAND | !(X\*Y) |  |
| XNOR | !(X⊕Y) |  |

## Multiple input gates…

Keep in mind that we can have more than two inputs on a gate. So A\*B\*C can be shown as a single gate.

## Bubble Bubble, Toil and Trouble…

Also, you can use a bubble as a short way of indicating a “not”. So really the NAND and NOR gate symbols are just that—an AND (or OR) followed by a not.

|  |  |  |
| --- | --- | --- |
| S | C | NOR |
| 0 | 0 |  |
| 0 | 1 |  |
| 1 | 0 |  |
| 1 | 1 |  |

|  |  |  |
| --- | --- | --- |
| S | C | NAND |
| 0 | 0 |  |
| 0 | 1 |  |
| 1 | 0 |  |
| 1 | 1 |  |

|  |  |  |
| --- | --- | --- |
| S | C | XNOR |
| 0 | 0 |  |
| 0 | 1 |  |
| 1 | 0 |  |
| 1 | 1 |  |

**Problem:**

Create a 3-bit equals comparator. That is, a device that checks to see if two 3-bit numbers are equal. Clearly solving this with a truth table isn’t a great idea (6 inputs). Hint: consider XNOR…

# Logical completeness (section 2.8 still, page 82)

Given that we can (in theory) create a circuit for any truth table by finding the canonical sum-of-products, that means we can generate any Boolean function. Could you do it with just AND gates? Just OR? How about just OR and NOT gates?

# Delay (2.10)

As we’ve mentioned in class before, in the real world gates take time to change. This has all sorts of very important effects.

* For one thing, we have to wait a short time after the inputs change for the outputs to change. On a modern computer that will often be on the order of 0.5ns or less for most logic. On our FPGA it might be 5ns or so.
* Perhaps more importantly, when we change the inputs we might see the output go “bad” for a short period of time as those changes propagate through the circuit. These bad outputs are called “glitches”.

**x**

**y**

**z**



Figure 1: Full adder (from Wikipedia)

* Consider the above circuit diagram. If XOR gates have a delay of 0.2ns while AND and OR gates have a delay of 0.1ns, how soon after we change Ci (while holding everything else constant) might it be before the outputs are steady? What about for B?
* Now consider what happens if we start with A=B=Ci=0 and then at time 0.0 A and Ci both change to 1. Fill in the timing diagram below…



