Use pencil!

Website: https://www.eecs.umich.edu/courses/eecs270/

Policies and class introduction

This class provides an introduction to digital logic. This is the foundation of modern computer design as well as being useful in more specialized contexts such as application-specific integrated circuits (ASICs) and Field-Programmable Gate Arrays (FPGAs). The class has a significant lab component where you will learn to use a "hardware description language" (HDL) called Verilog to design logic on an FPGA. This material is used by people working in chip design, embedded systems, and certain CAD classes (it's a prerequisite for 373, 427, 470, and 478).

Course communication:

You should use the course website as the portal for the class. We are using a large number of online tools.

- **Website**: Has lecture notes (like these), homework assignments, and links to everything else.
- **Canvas**: Has the lab material included your lab grades
- **Gradescope** (entry code: **4VE8Z3**): Turn in homework, also has all non-lab grades
- **Piazza**: Course discussion

Handouts today:

- Course schedule
- Weekly schedule (office hours, lab hours, etc.)
- Course polices
- Lecture notes
- Homework 1 (HW1)
- Group assignment 1 (GA1)

All handouts are on the website. After this only the lecture notes will be provided by the instructor.

How to use these notes

Lectures will be largely given on the whiteboard. The intent of these notes is that you can do your in-class work and take notes on them as we go. No filled-in notes will be provided (and in fact the instructor hasn't created them). It is strongly recommended that you take your notes in pencil as you will be working on in-class problems and may find you want to make some changes as the discussion continues.

Often these notes will have references to parts of our textbook where you can get more information. Our textbook is quite readable and it can be very useful. Readings can also be found in the course schedule.

Lecture start

Say we live in the rather black and white world where things (variables) are either true (**T**) or false (**F**). So, if ***S*** is "Mark is going to the Store" and ***C*** is "Mark likes Computer games" then we'll assume that each phrase is either true or false (as opposed only sort of liking computer games). We can then use connectives to combine the variables.

Mark is going to the store AND Mark likes computer games.

The above statement is only true if both phrases are true. Let that sentence be ***X***. We can now draw the "truth table" for ***X*** (we'll use the other tables in a minute). When is X true?

S	C	X
F	F	
F	T	
T	F	
T	T	

AND

S	C	
F	F	
F	T	
T	F	
T	T	

____

S	C	
F	F	
F	T	
T	F	
T	T	

____

C	
F	
T	

____

What if the statement, ***Y***, were "Mark is going to the store OR Mark likes computer games"? When is that true? How about Mark does NOT like computer games?

(OR vs. XOR)

How about having ***B*** be "Bob's house is brown".

S	C	B	
F	F	F	
F	F	T	
F	T	F	
F	T	T	
T	F	F	
T	F	T	
T	T	F	
T	T	T	

S AND C AND B

S	C	B	
F	F	F	
F	F	T	
F	T	F	
F	T	T	
T	F	F	
T	F	T	
T	T	F	
T	T	T	

S OR C OR B

S	C	B	
F	F	F	
F	F	T	
F	T	F	
F	T	T	
T	F	F	
T	F	T	
T	T	F	
T	T	T	

S OR C OR NOT B

S	C	B	
F	F	F	
F	F	T	
F	T	F	
F	T	T	
T	F	F	
T	F	T	
T	T	F	
T	T	T	

S OR (C AND B)

Representation of Boolean Logic (section 2.6)

Using AND, OR, NOT and XOR gets old. So symbols have been used to represent these notions for quite a while. We'll hit three different representations today:

	Math/Philosophy	Electrical/Computer Engineering	Gate
Y AND Z			
Y OR Z			
NOT Y			
Y XOR Z			

Order of operations?

____

In digital logic we usually think of "1" as being TRUE and "0" as being "FALSE" …

Methods of representation and terminology (p53-54, 64-68))

We can represent logic functions in a number of ways at this point. We can write logic equations, draw gates, and write truth tables. Going between these methods of representation (mostly to/from truth tables) can be interesting. Consider the following truth table. Write a logical formula that is equivalent.

S	C	B	X
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

Terms: (p54-55,69)

- Variable
- Literal
- Product term
- Sum-of-Products
- Minterm
- Canonical Sum-of-Products

Now write a truth table for the following word problem:

> Consider a device with three inputs: A, B and S as well as one output M. M should be equal to A if S=0, else M should be equal to B.

Now, without looking at the Truth table, can you draw gates for this? Hint: it can be done with 4 gates (2 AND, 1 OR, 1 NOT)

A	B	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	

This device is called a multiplexor (MUX), you will see it in lab 1!

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 10010_2 (the subscript indicates the base).

1 0 0 1 0

Now what do you suppose the value of 10.11_2 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 $1F_{16}$ 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[1]. So what is 10011001_2 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 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	R_1	R_0
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.

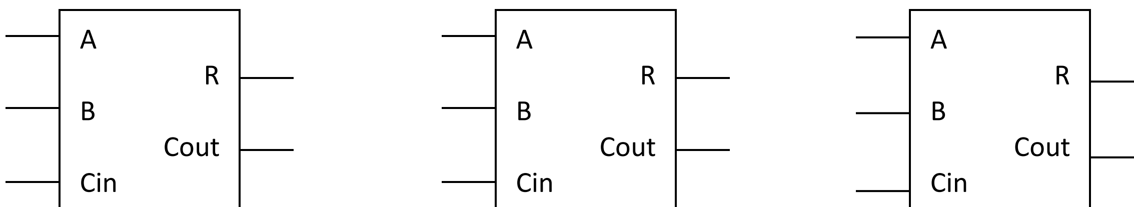
- If we used the truth table scheme, how many rows would there be?
  - What about for a 64-bit addition?
- Can we break the problem down in a way that works better?
  - How?
- Let's work on it…

```
  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.

$$
\begin{array}{rcccc}
 & C_3 & C_2 & C_1 & \\
 & & A_2 & A_1 & A_0 \\
+ & & B_2 & B_1 & B_0 \\
\hline
 & \mathbf{S_3} & \mathbf{S_2} & \mathbf{S_1} & \mathbf{S_0}
\end{array}
$$

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?



4-bit ripple-carry adder.

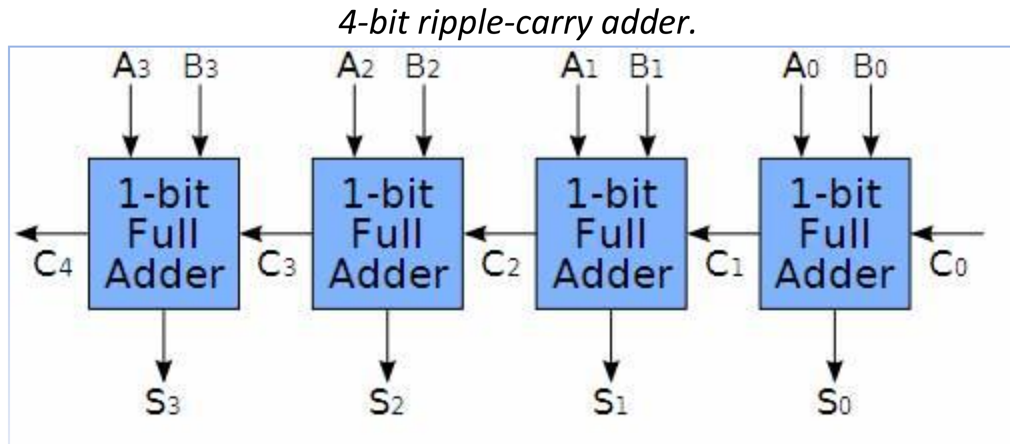


Figure by Dr. Liu, FSU

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 equal 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".

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…

0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0 1.1 (ns)

A

B

Ci

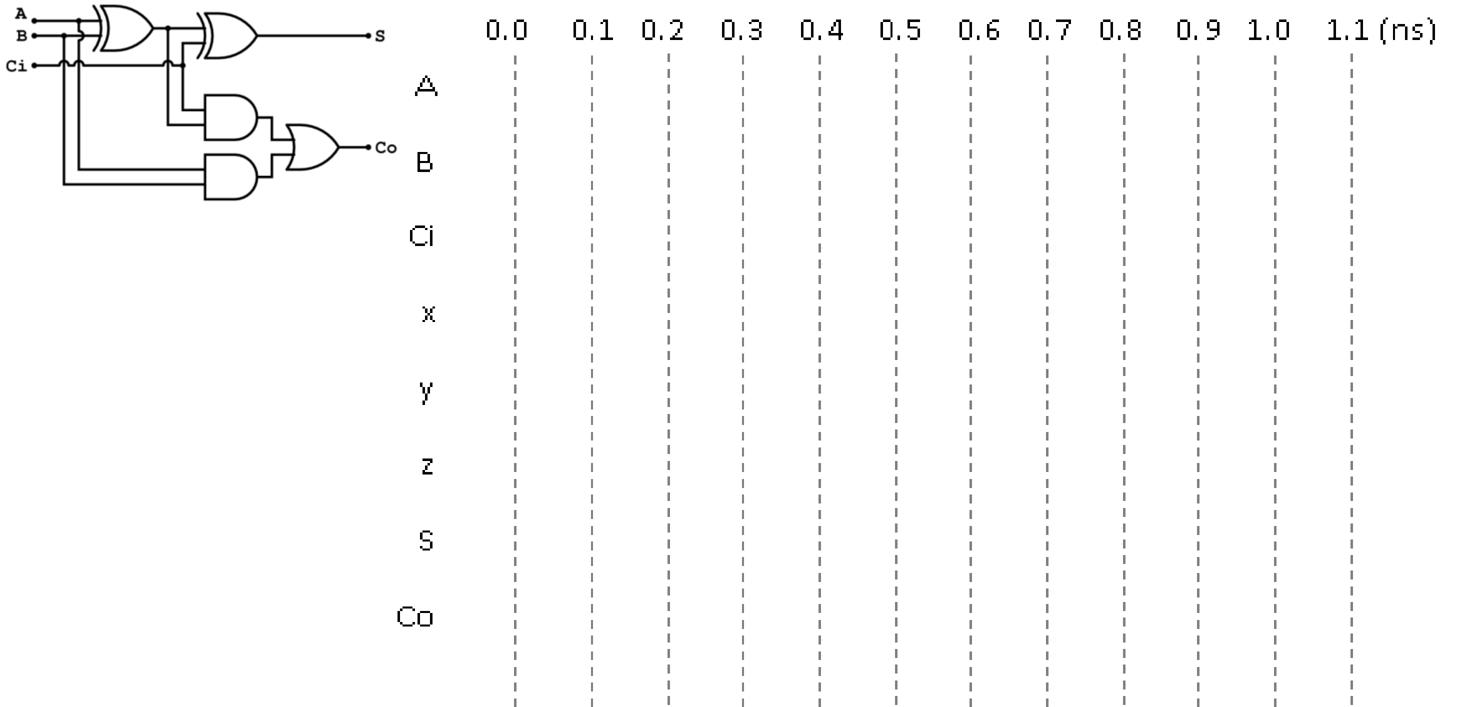
x

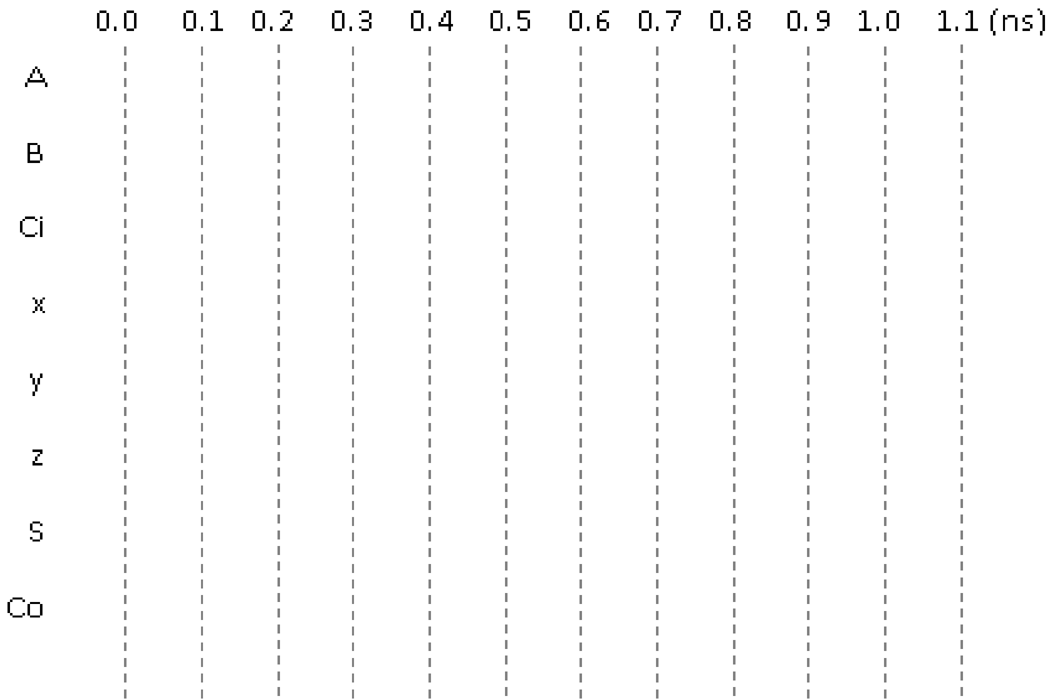
y

z

S

Co


A
B
Ci
S
Co
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1.0
1.1 (ns)
A
B
Ci
x
y
z
S
Co


0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1.0
1.1 (ns)
A
B
Ci
x
y
z
S
Co

Medium Scale Integrated (MSI) devices [Sections 2.9 and 2.10]

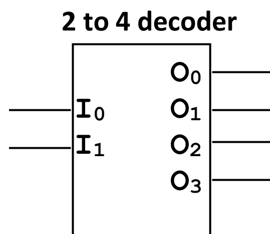
As we've seen, it's sometimes not reasonable to do all the design work at the gate-level—sometimes we just want bigger building blocks. For example, we used full-adders as the building blocks of our ripple-carry adder. You can imagine that as we do designs we might find that there are certain building blocks we use over and over again. As you might guess, being familiar with a commonly-used set of building blocks can make designing digital devices easier.

Building blocks that are more complex than gates are sometimes called medium-scale integrated devices, or MSI devices for short. Exactly what makes a device "medium" in scale rather than "large" or "very large" (as in VLSI which you may have heard of) is not always agreed on, but we'll generally stick with devices that are fairly simple and where small versions of them can be implemented in a couple dozen gates or so.

The devices

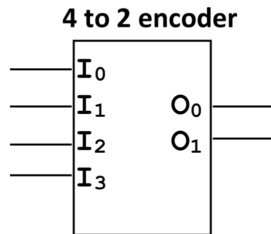
We'll cover four common devices today: decoder, encoder, priority encoder, and MUX. The MUX we've seen before. Other MSI devices you'll need to be familiar with are the comparator, adder, and deMUX (see text).

- **<u>Decoder</u>**
  - N inputs, 2^N outputs.
  - It decodes the N-bit binary number and sets exactly one of the decoder's outputs to 1. So if the input of the 2 to 4 decoder shown below is I[1:0]=01, then O_1 will be a "1" and the other three outputs will be a "0".



  - Questions/problems
    1. If the input I[1:0]=11, what will the outputs be? ____________________
    2. Draw the logic gates needed to build the above 2 to 4 decoder.
    3. Consider the logic equation (ABC+A'BC'+AB'C'). Using only a 3 to 8 decoder and one gate implement that logic function.

- **Encoder**
  - 2^N inputs, N outputs
  - It assumes exactly one input is a "1" and outputs a binary number that corresponds to the input that is a one. Put another way, it undoes the work of the decoder. For example, if I[3:0]=0010, O[1:0]=01.



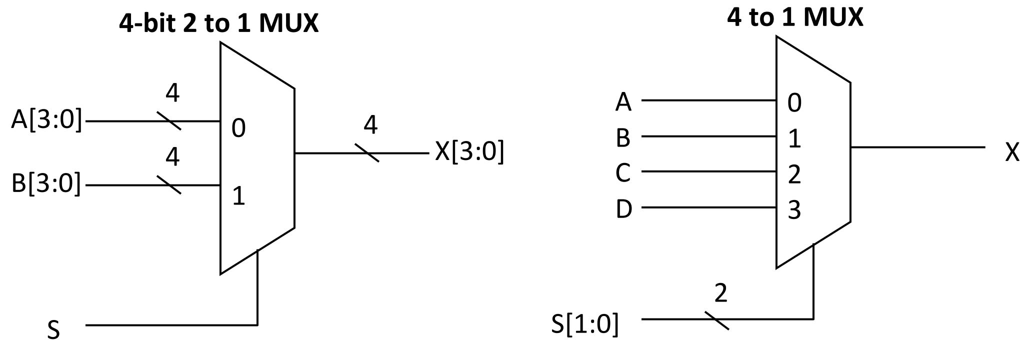
  - Questions/problems
    1. If the I[3:0]=1000, what will the outputs be? ____________________
    2. If the input I[3:0]=0100, what will the outputs be? ____________________
    3. If the input I[3:0]=1001, what will the outputs be? ____________________
    4. Draw logic gates which implement a 4 to 2 encoder. You can assume that exactly one input will be a "1".

- **Priority Encoder**
  - Just like an encoder but if more than one input is a "1" the output is based on the largest input. So if I[3:0]=0101, O[1:0]=10.
  - Questions/problems
    1. If I[3:0]=D_{16}, what will the outputs be? _________
    2. If I[7:0]=01001100, what will the outputs be? ____________
    3. If I[7:0]= $0C_{16}$, what will the outputs be? ____________

Note: it is common to have an "enable output" (EO) signal which is "1" only if at least one input is "1".

MUX

- We've looked at 2 to 1 MUXes previously, but we could have X to 1 where X is any integer greater than 1. We could also have "n-bit" MUXes. For example, below we have a 4-bit 2 to 1 MUX. Here we are just choosing one of two groups of 4 to route to the output.



  Naming conventions for MUXes are not as standard as we might like. That leftmost figure might be called a "4-bit 2:1 MUX", a "4-bit 2-1" MUX, or even an "8 to 4 MUX".

- Questions/problems
  1. For the 4 to 1 MUX, if A=0, B=1, C=0, D=1 and S=01, what is X? ___________

  2. How many select lines would you need for a 2-bit 8 to 1 MUX? _________
  3. Build a 2-bit 2 to 1 MUX out of 2 to 1 MUXes.
  4. Build a 4 to 1 MUX out of 2 to 1 MUXes.
  5. Build a 2 to 1 MUX out of gates.

Solving problems with MSI devices

- In computer architecture it is common to have N devices that could request a given resource. This is often implemented by giving each device a request line (which if it's 1 means they want to use the resource) and a grant line (which if it's one means they have been granted the device.) Using MSI devices design an arbiter takes 4 request lines and asserts only one grant line. If multiple devices are requesting at the same time, you can pick any of them (you pick the priority).

Designing an MSI device

- Design, at the gate level, a 2-bit greater than comparator. That is it takes two 2-digit numbers "A" and "B" and outputs a "1" if and only if A>B.
    - How would you design a 4-bit greater-than comparator?

Representing negative numbers in binary (200-206, p194-200 1st)

When we represent negative numbers in decimal, we use an additional symbol, the minus sign "–", to indicate that the number is negative. When using a computer we only have ones and zeros, so we can't use some extra character to indicate that the number is negative. One trick we could play is to have one of the bits of the number indicate sign.

Signed-magnitude representation

One scheme you could use is just to have one of the bits in the number be the sign bit. Say if the leftmost bit (i.e. the most significant bit) is a one the rest of the number is negative and if it's zero then the number is positive. So 1100 is -4 while 0100 is 4.

- How would we write -6? ___________ . 6? _____________

In a computer or other digital device we generally have a fixed number of bits per number.

- Using a 4-bit signed-magnitude representation , what is the smallest value we can represent?_______ The largest? ________
- For an n-bit number using signed magnitude representation, what is the range of representation? _________________
- How many different values can we represent with n-bits? ____________

This scheme has any number of downsides. The most obvious is that building an equals comparator is a bit tricky. Why?

Another issue is that building an adder which can add two signed-magnitude numbers is a bit tricky…

2's complement representation

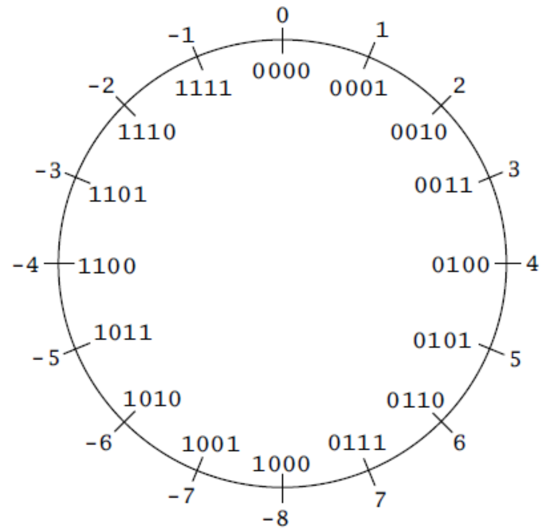
The scheme that is most commonly used to represent negative numbers in a computer is called "2's complement". The text goes into a good degree of background about the name and why it works. But the basic idea is that we treat the most significant bit as being negative but of the same magnitude it would be in a normal (unsigned) binary number. So as an unsigned number 1000_2 is 8. As a 2's complement number it's -8. 1001 as a 2's complement number is -7 (-8+1) while 1111 is -1 (-8+4+2+1).

It turns out that changing the sign of 2's complement numbers isn't as hard as you might think (though harder than signed magnitude). Flip all the bits and add 1. So 1111 = -1. Flip the bits and you get 0000. Add 1 and you get 0001 or 1.

- Say we are using 4-bit 2's complement numbers. What is the smallest number we can represent? _______ The largest? ______
- For an n-bit number using 2's complement representation, what is the range of representation? _________ How many different values can we represent with n-bits? _______________

Adding 2's complement numbers

The reason we tend to use 2's complement numbers is that traditional binary addition works exactly the same as long as the result is in the range of representation. We just lop off any extra bits. For the following problems add the values as if they were unsigned numbers, only keeping the first four digits. Which of these additions are correct for 4-bit unsigned values? For 4-bit 2's complement values?




```
   1 1 1 1         0 1 1 1        0 1 0 0        0 0 1 1
 + 1 1 1 1       + 1 1 1 1      + 0 1 0 0      + 1 1 1 0
   =======        =========      =========      =========
```

Bits is bits

What is the value of the binary value 1000? The right answer is "it depends". As an unsigned number it's 8. As a 2's complement number it's -8. As a signed-magnitude number it is 0.

The key point is that we use 0s and 1s to represent everything. Exactly how we treat those 0s and 1s depends on the context. In a computer the same 0s and 1s could be used to represent different numbers, or letters (ASCII for example), computer instructions, or just about anything else. Bits don't have meaning without context, though we often assume they are unsigned numbers when no context is given.

Questions

- Treated as an 8-bit 2's complement number, what is the value of $4C_{16}$? FF_{16}? $C0_{16}$?
- When doing addition of fixed-bit unsigned numbers, how do you detect overflow? How about with fixed-bit 2's complement numbers?

Verilog

You've been using Verilog in lab already for a while, but at this point it's useful to be a bit more formal.

Basics – the assign statement

The most basic statement is the assign statement. Here we can specify gates and the like with simple symbols. & is AND, | is OR, ^ is XOR and ~ is NOT. For example:

```
assign a=(y|~x)^z;
```

Consider the following gates. Write a logic equation for each output.

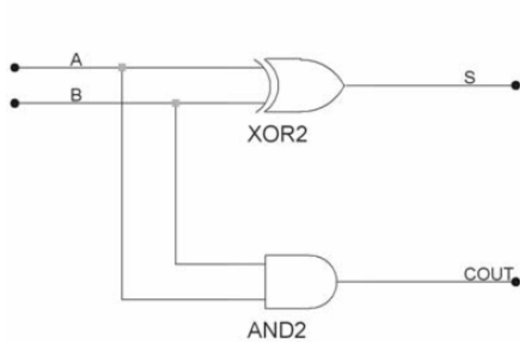


Figure 1a: Half adder

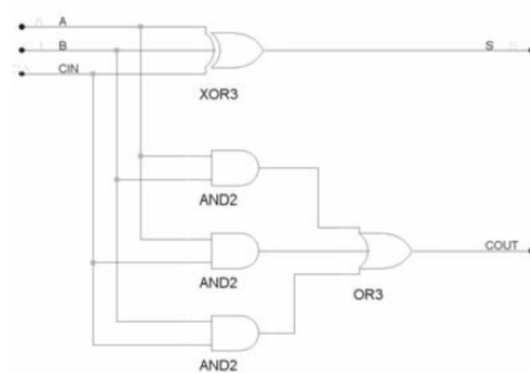


Figure 1b: Full adder

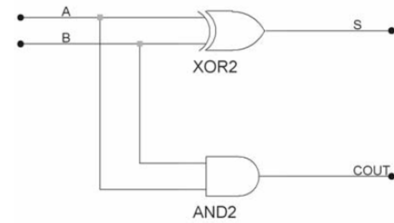
Half adder:

- assign S=
- assign Cout=

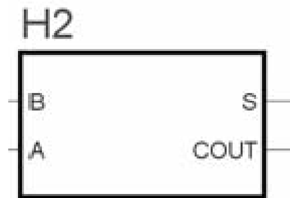
Full adder:

- assign S=
- assign Cout=

Basics – defining a module



While our fundamental building block are gates, we often need something more complex (for example MSI devices). In prelab2 you'll be asked to build "boxes" using schematic capture. In Verilog we often want to do the same thing. So we define a module in a way that is reminiscent of writing a function in C.




```
module add_half( input  a,
                 input  b,
                 output s,
                 output cout );

     assign s = a ^ b;
     assign cout = a & b;
endmodule
```

The half adder can then be used, along with a full adder, to make a 2-bit adder.

Below is an example of some code using slightly different (older) syntax. Don't do things like (port list followed by port descriptions in the body of the module), but we wanted you to see it at least once as you will commonly see code that looks like this!

```
module add_2bit (a, b, s, cout);

   input [1:0] a, b;   // Both a and b are 2 bit inputs
   output [1:0] s;     // s[1] = MSB of s, s[0] = LSB of s
   output cout;
   wire c0;            // intermediate carry between adders

   add_half a1(a[0], b[0], s[0], c0);
   add_full a2(a[1], b[1], c0, s[1], cout);

endmodule
```

The parameter list can take two forms. The first is shown in the example above and is similar to parameter lists for C function calls—the inputs and outputs must be in the exact same order as in the module declaration itself. This is *connecting by order*.

The other way to list parameters for a module instantiation is to explicitly assign signals to the correct ports, in which case the order does not matter. The instantiations from the add_2bit module are rewritten below using this form. This is *connecting by name*.

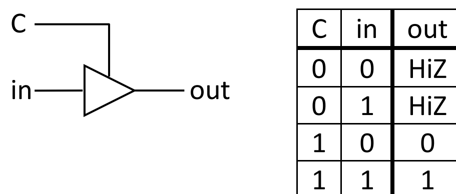
```
add_half a1(.a(a[0]), .b(b[0]), .s(s[0]), .cout(c0));
add_full a2(.a(a[1]), .b(b[1]), .cin(c0), .s(s[1]), .cout(cout));
```

- Write the code for the full adder.
- Modify the above add_2bit to make a 4-bit adder using the connect by name method and use the port list syntax shown in the add_half module.

Finishing up combinational logic

Tri-state devices (page 227, our book isn't so good with this though)

One interesting thing to do is to create a circuit where the output *can be* "disconnected" from the input. Such a device would create an "open circuit" or, equivalently, an (extremely) high resistance connection between the input and the output (an open circuit effectively has an infinite resistance yes?) For historical reasons, such a disconnected output is called "high impedance" (recall impedance is a more generic term for resistance[1]). Because the symbol for impedance is "Z", we use the term "HiZ" to indicate that the output is disconnected. Below is the standard symbol for a tri-state buffer (the "three states are "0", "1", and "HiZ".



C	in	out
0	0	HiZ
0	1	HiZ
1	0	0
1	1	1

- Tri-state devices are commonly used to create MUXes. Design a 2 to 1 MUX using tri-state buffers and an inverter.

We commonly use tri-state devices to connect multiple chips together on a board. We do this because we can put the tri-state buffers on each chip and don't need an external MUX on the board (which could cost serious $$$$).

A common problem students have is "what happens if I drive a gate input with a HiZ output?" The answer is that the input voltage is floating, so you could get a high input, a low input or something in between. ***This is to be avoided.*** You can sometimes get away with it and get reliable outputs (for example, if you put in HiZ as an input to an AND gate and the other input is 0, you'll get a 0 for output), but in general it's a horrible idea (and should be avoided even in cases like the one described above as the device probably wasn't designed to deal with that...)

[1] The total opposition to alternating current by an electric circuit, equal to the square root of the sum of the squares of the resistance and reactance of the circuit and usually expressed in ohms. Symbol: Z (http://dictionary.reference.com/browse/impedance)

Minterms, maxterms and some useful notation

Consider the following truth table:

N2	N1	N0	X
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Recall that we can write the equation for it in canonical sum-of-products form as:
N2'*N1*N0 + N2*N1'*N0 + N2*N1*N0.

Each of those canonical product terms are called "minterms" and our book refers to this as the sum-of-minterms form.

We can also write the above truth table as $\sum_{N2,N1,N0}(3,5,7)$. How did we get that? Why Sigma?

The scheme we use for finding the canonical sum-of-products involves writing a term for each 1. Couldn't I also write a term for each 0 and then AND those terms together?

Maxterms/product-of-sums (brief coverage found on page 70, 64 in 1st)

Just as the canonical sum-of-products form can be useful for converting from a truth table to a logic equation; product-of-sums form can also be useful. Consider the table on the right. The canonical SoP form would have 6 product terms. We could instead write an equation for each zero in "X" then AND those terms together. In this case we'd end up with:

X=(A'+B+C) * (A'+B'+C). Fill in the table with the two terms in parenthesizes. Notice X is just the AND of those two terms.

A	B	C			X
0	0	0			1
0	0	1			1
0	1	0			1
0	1	1			1
1	0	0			0
1	0	1			1
1	1	0			0
1	1	1			1

These "sum terms" that include all variables are called "maxterms" and the overall form is called "Product of Sums" (canonical product of sums in this case). When there are a lot fewer 0s than 1s, this can be quite convenient.

That finishes combinational logic for now…

With tri-state devices, we've covered most of the major topics in combinational logic. There are four significant combinational-logic topics that will be covered later in the semester:

- Optimizing combinational logic (this is a major topic and will take a couple of lectures)
- Implementing gates using transistors
- Switching logic and a more formal/theoretical treatment of combinational logic
- More complex MSI devices (multipliers, barrel shifters, etc.)

All four of these topics aren't needed to be able to be successful with basic logic design. As such, we'll put them off until after we've managed to cover the "basics" of sequential logic. The idea is that once we get basic combinational and basic sequential logic down, we can start doing interesting design problems. And the sooner we do that, the better you will be with them.

Some practice problems…

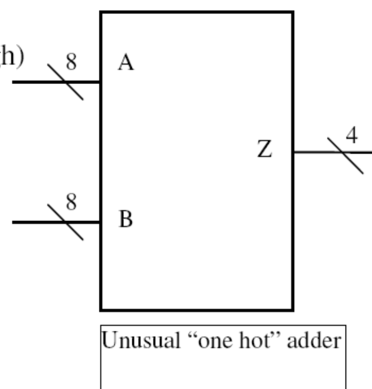
Here are a handful of nice practice questions for to try. Time allowing, me may do a few in class. Answers won't be posted, but feel free to come to office hours if you have questions!

A. Using the rules of logic, convert (A+B)'*(D*C)' into sum-of-products form. Provide the name of the rule used for each step.
B. Using only a decoder and an OR gate (of any number of inputs) create a circuit which implements the following logic: F= (A+B+C)*(A+B')*(B'+C)
C. Design an AND gate using only tri-state buffers and inverters
D. Given a 4-bit unsigned comparator with outputs EQ and GT, design a 4-bit 2's complement comparator with the same outputs. You may use standard MSI devices but try to be efficient.
E.

Consider a very unusual adder. It takes two 8-bit inputs, both representing number using the "one hot" representation scheme (Thus the inputs range in value from 0-7). The output is a 4-bit unsigned number that is a sum of the two inputs. In your design you may use the following devices: (Be sure to ***label*** all of the devices you use other than standard gates!) **[20]**

- 2-input AND gates, 2-input OR gates, 2-input XOR gates, NOT gates
- 3-bit adders (inputs and outputs are both 3-bits) with carry-in and carry-out.
- 8 to 3 encoders (all wires active high)
- 3 to 8 decoders (all wires active high)
- 8 to 4 Multiplexers (all wires active high)

The inputs A[7:0] and B[7:0] are describing input values ranging from 0 to 7 using the "one hot" representation scheme. The output Z[3:0] is to be the sum of the values of the 2 inputs. Z is represented as an unsigned number. For the inputs A7 and B7 are the MSBs. While for the output Z3 is the MSB.



Unusual "one hot" adder