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".

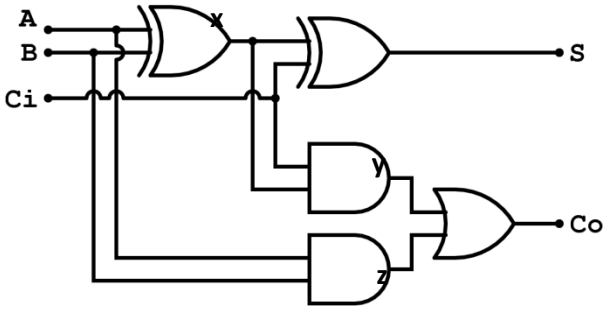
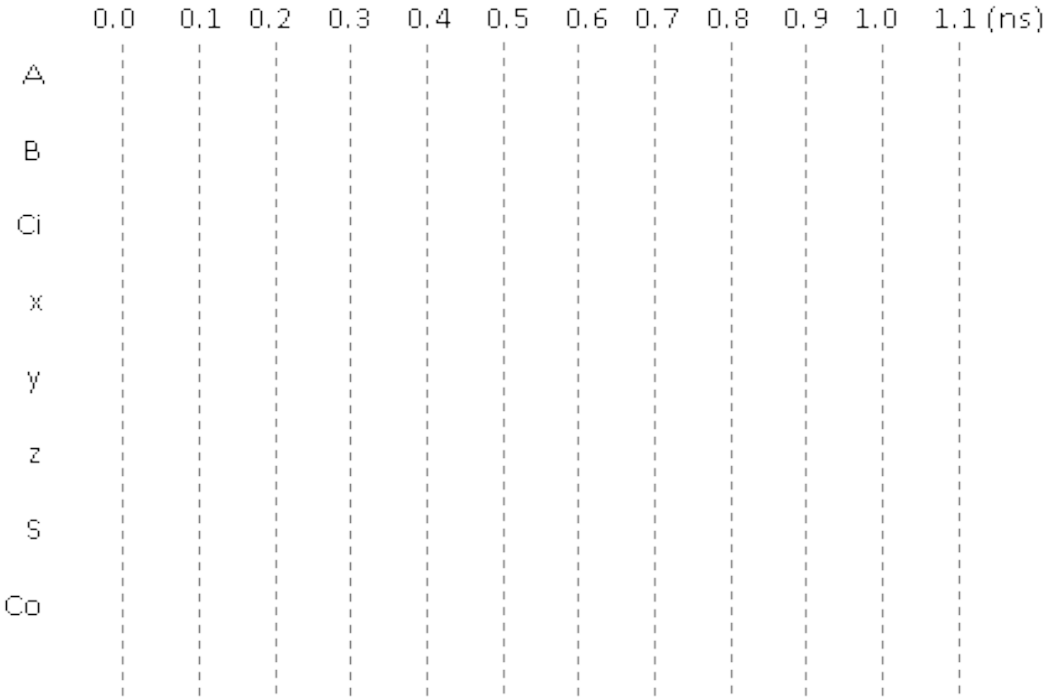
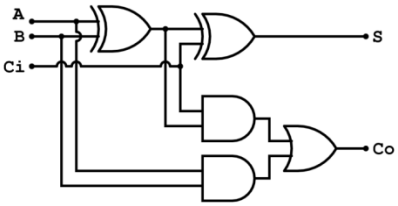
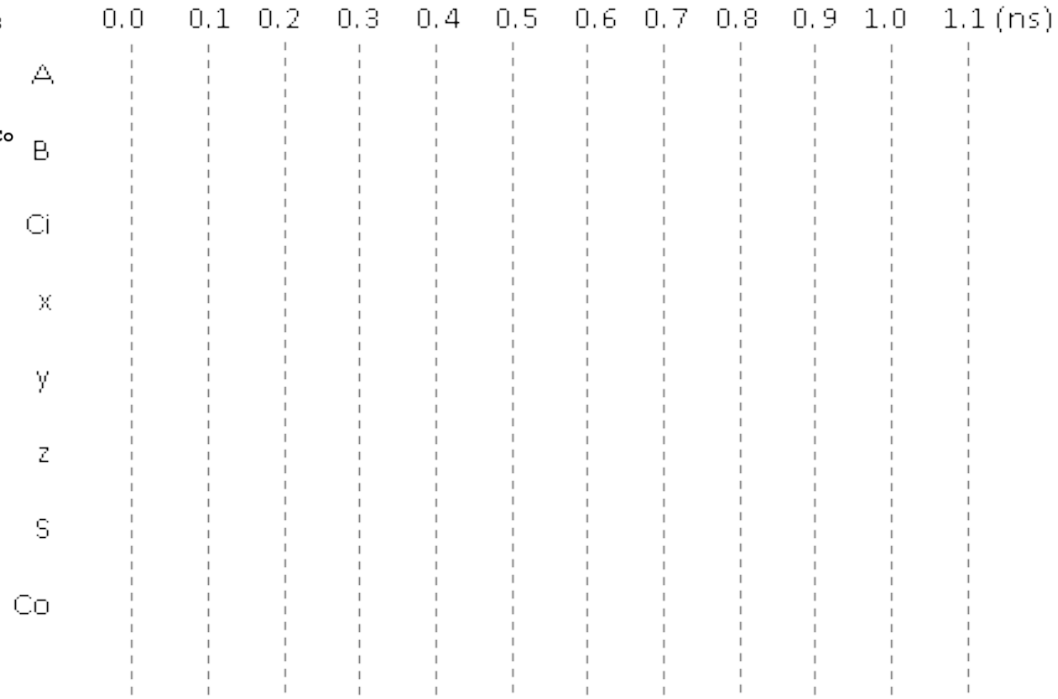


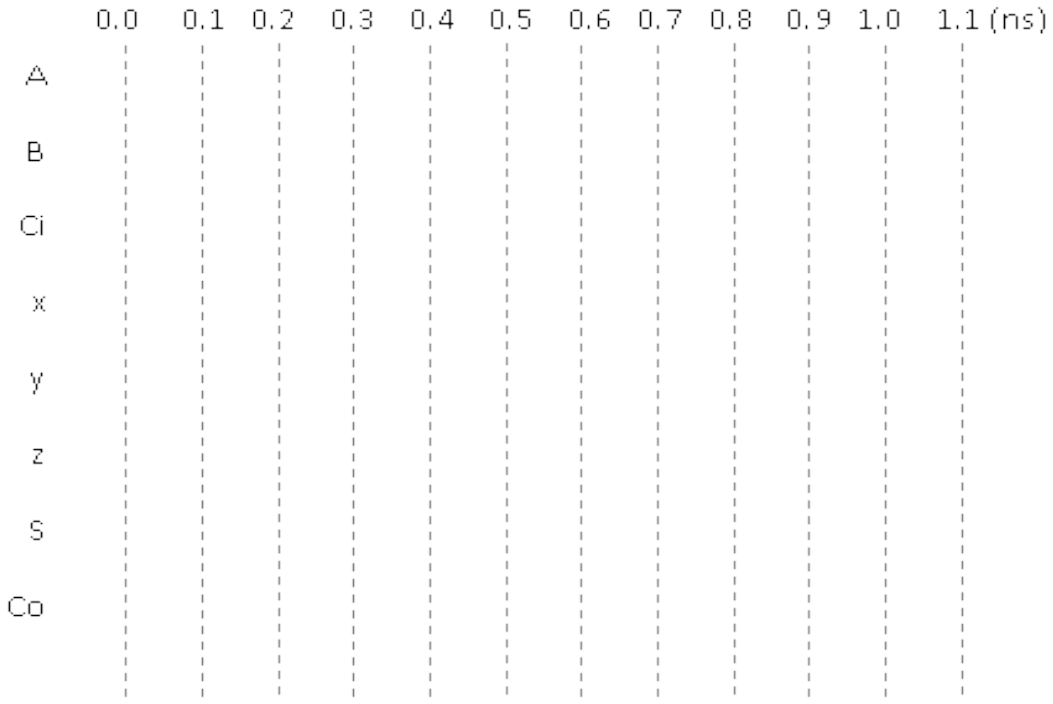
Figure 1: Full adder (from Wikipedia)

- 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 0.1, A and Ci both change to 1. Fill in the timing diagram below…




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.

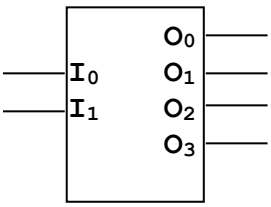
Building blocks that are more complex than gates are sometimes called medium-scale integrated devices, or MSI devices for short. Exactly what makes a device "medium" in scale rather than "large" or "very large" (as in VLSI which you may have heard of) is not always agreed on, but we'll generally stick 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).

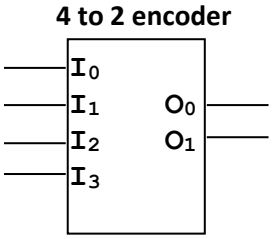
- **Decoder**
  - N inputs, 2^N outputs.
  - It decodes the N-bit binary number and sets exactly one of the decoder's outputs to 1. So if the input of the 2 to 4 decoder shown below is I[1:0]=01, then O_1 will be a "1" and the other three outputs will be a "0".

2 to 4 decoder



  - 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.



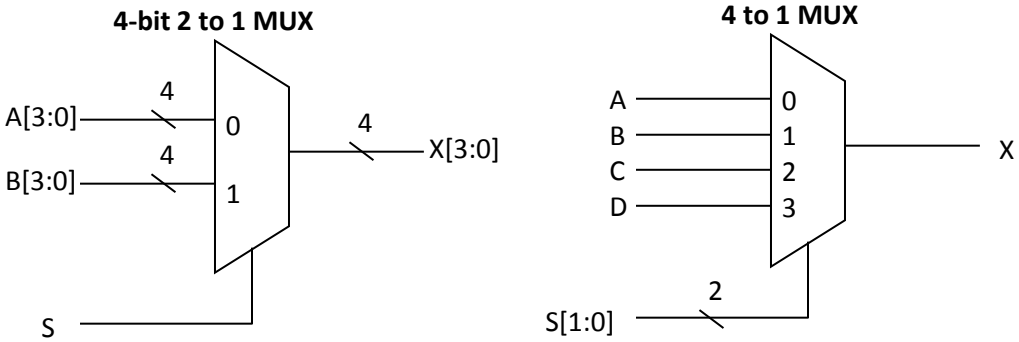
  - Questions/problems
    1. If the I[3:0]=1000, what will the outputs be? _____
    2. If the input I[3:0]=0100, what will the outputs be? _____
    3. If the input I[3:0]=1001, what will the outputs be? _____
    4. Draw logic gates which implement a 4 to 2 encoder. You can assume that exactly one input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

- Say we are using 4-bit numbers. What is the smallest number 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 numbers. What is the smallest number we can represent? _____ The largest?

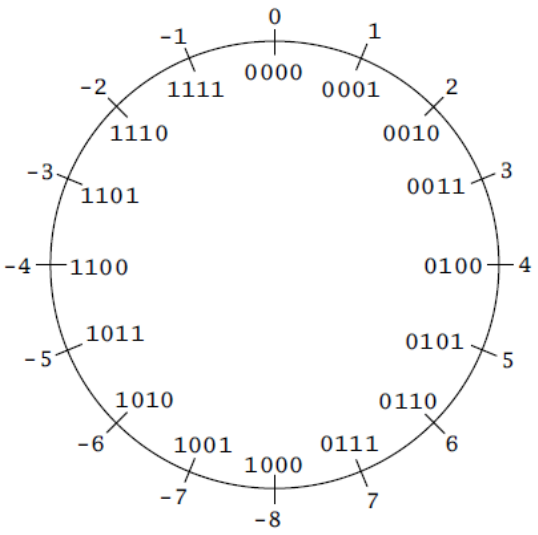
As before we generally have a fixed number of bits per number.

- Say we are using 4-bit 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?