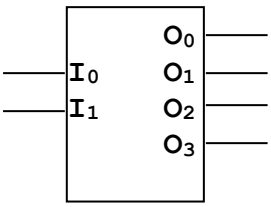
Medium Scale Integrated (MSI) devices [Sections 2.9 and 2.10]

Last time we discussed the fact that certain combinations of gates appear fairly often. So much like some standard library functions in C or C++, we have standard parts in digital logic. Last time we looked at encoders, decoders, and priority encoders.

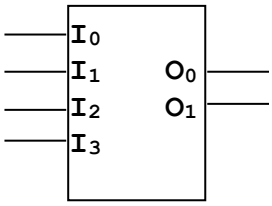
Review

2 to 4 decoder



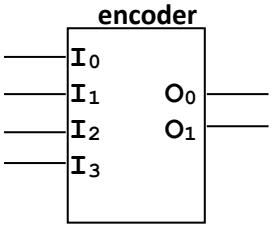
1. If the input I[1:0]=10, what will the outputs be? O[3:0]=_____
2. If you have a 3-input decoder, how many outputs do you have? _____

4 to 2 encoder



1. If the input I[3:0]=0010, what will the outputs be? O[1:0]=_____
2. If the input I[3:0]=1010, what will the outputs be? O[1:0]=_____
3. If you have a 16-input encoder how many outputs do you have? _____

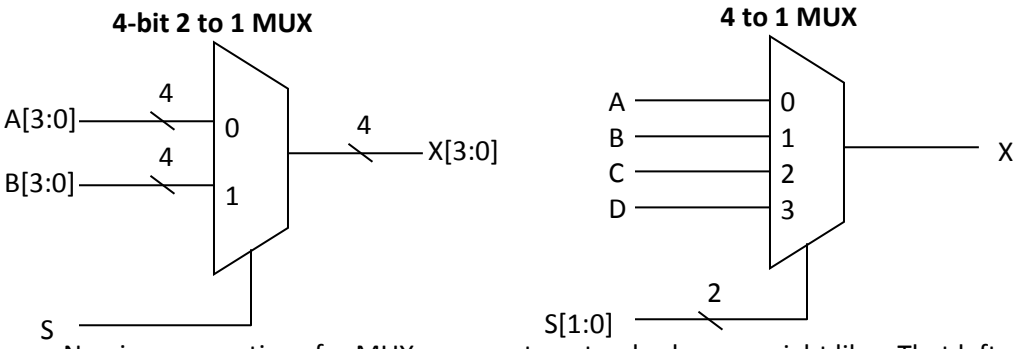
4 to 2 priority encoder



1. If the input I[3:0]=0010, what will the outputs be? O[1:0]=_____
2. If the input I[3:0]=1010, what will the outputs be? O[1:0]=_____

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.

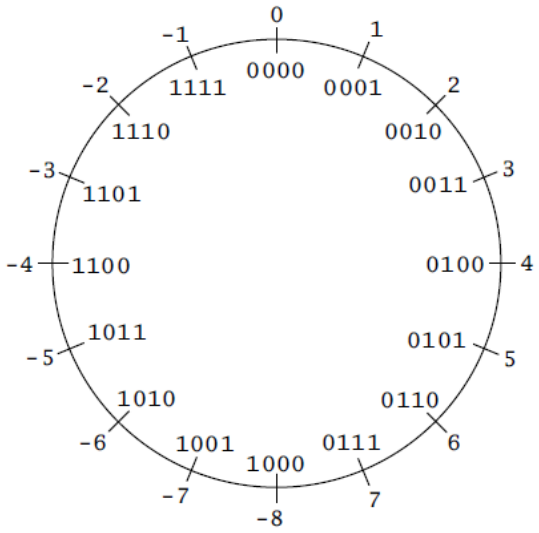
As before we generally have a fixed number of bits per number.

- 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?

Sign change and Subtraction (4.6)

One important "trick" with 2's complement numbers is that you can change the sign of a number fairly easily. Just flip all the bits (1's to 0's and 0's to 1's) then add one. Seems a bit weird, but it works[1] (unless we are starting with the smallest (closest to negative infinity) number our representation handles). Try it. We'll use 4-bit numbers to illustrate.

0001 is 1. Flip the bits → **1110**. Now add one → **1111** which is -1.

0110 is 6. Flip the bits → _____. Now add one → _____ which is _____

1111 is -1. Flip the bits → _____. Now add one → _____ which is _____

0000 is 0. Flip the bits → _____. Now add one → _____ which is _____

1000 is -8. Flip the bits → _____. Now add one → _____ which is _____

Why doesn't that last one work?

Subtraction

From this, we can see that one way to compute A-B is just to find the bitwise complement of B, add it to A and then add one more. With a carry-in on a 4-bit adder, that +1 is pretty easy to do as shown on the right (+5V being a "1")[2]. B_{out} is "borrow out".

Our text discusses "full subtractors" that are an analogy to a full adder. Take a look at it.

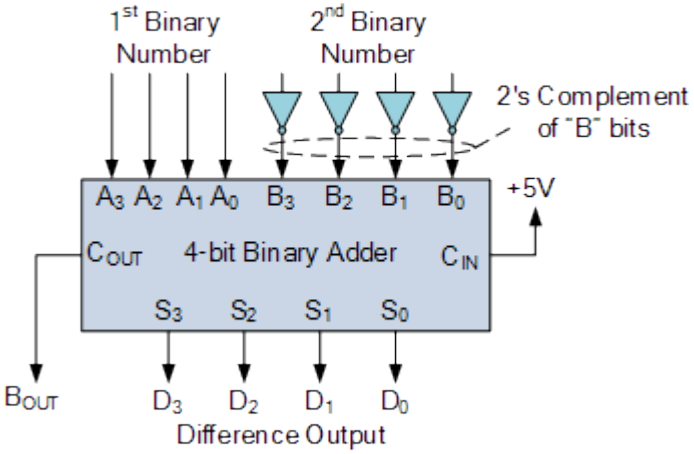


Figure 1: A subtractor built from an adder.

Multipliers, dividers, and ALUs

While addition and subtraction of both signed and unsigned numbers is pretty straightforward, multiplication and division are not. In multiplication is considerably more complex, and thus generally bigger and slower than addition. Division is even worse. Most reasonable multiplication algorithms require multiple steps and we'll want to use sequential logic rather than combinational logic.

One term you should know is "ALU" or arithmetic logic unit. It's a device that you provide the inputs and an operation and it does the operation. Operations might include addition, bit-wise and, subtraction, and maybe even multiplication and division.

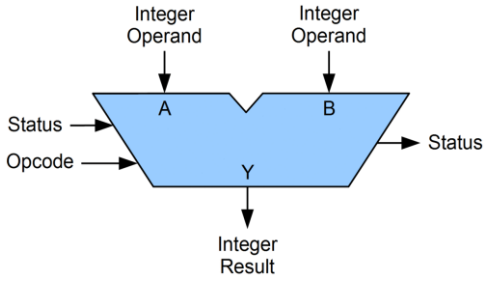


Figure 2: An ALU

[1] http://www.tfinley.net/notes/cps104/twoscomp.html has a discussion, I think there are probably betters ways to think about it…
[2] Figure 1 from http://www.electronics-tutorials.ws/combination/comb35.gif, fair use claimed. Figure 2 from Wikipedia.

Verilog

In lab we are currently drawing the devices and gates when implementing logic on the FPGA. We call is drawing "schematic capture". As you've likely noticed, it is generally a lot easier to write the logic equations than draw the gates. As such people don't generally use schematic capture. Rather they use a text-based scheme called a "hardware description language" (HDL). Not only does it make it easier to enter logic equations, but later we'll see that we can specify some pretty high-level notions and ask the tools to design the hardware.

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.

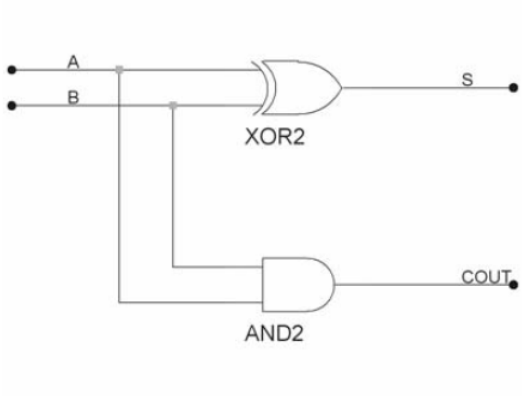


Figure 1a: Half adder

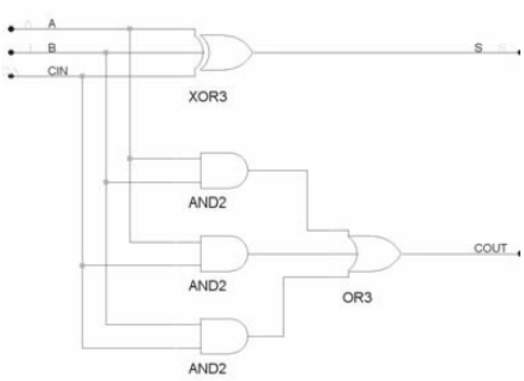


Figure 1b: Full adder

Half adder:

- S=
- Cout=

Full adder:

- S=
- Cout=

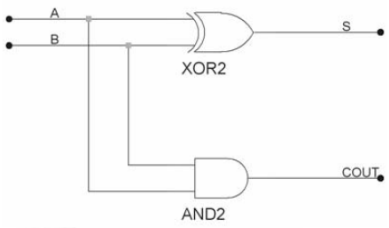
Basics – defining a module

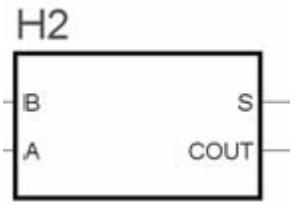
While our fundamental building block are gates, we often need something more complex (for example MSI devices). In prelab2 you'll be asked to build "boxes" using schematic capture. In Verilog we often want to do the same thing. So we define a module in a way that is reminiscent of writing a function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

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

```
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 [1:0] s;
    wire 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
```

- Write the code for the full adder.
- Modify the above add_2bit to make a 4-bit adder.