Finishing up combinational logic

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.

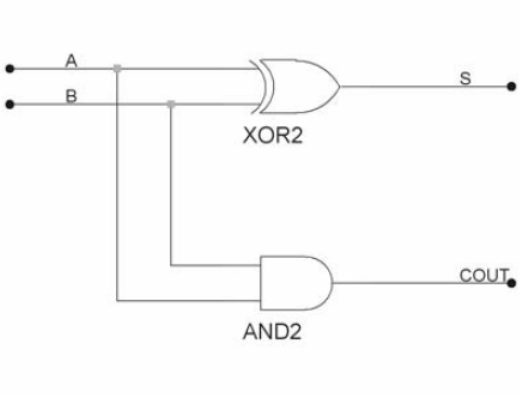


Figure 1a: Half adder

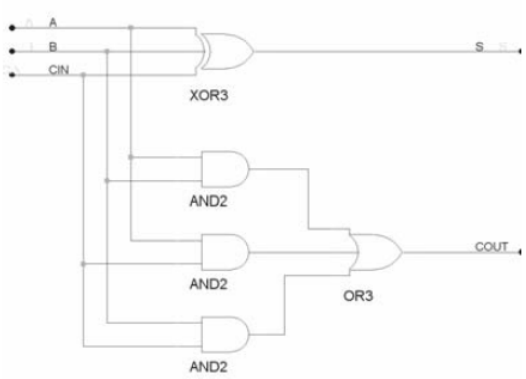


Figure 1b: Full adder

Half adder:

- S=
- Cout=

Full adder:

- S=
- Cout=

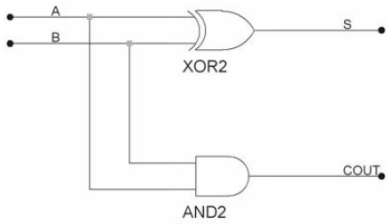
Basics – defining a module

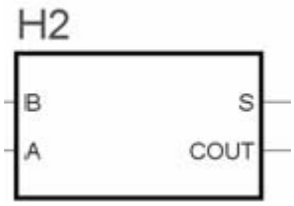
While our fundamental building block are gates, we often need something more complex (for example MSI devices). In prelab2 you'll be asked to build "boxes" using schematic capture. In Verilog we often want to do the same thing. So we define a module in a way that is reminiscent of writing a function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

Other syntax

- You can also declare the type of the ports in the port list.

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

When doing that, it is generally written as using multiple lines:

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

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

- We can do port connections (similar to passing arguments in C) in two ways. Above we saw connections by "ordered list" or connections by order.
    - **`add_half a1(a[0], b[0], s[0], c0);`**

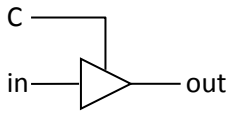
In that case, a[0] is connected to a in the module, b[0] to b, etc. because they are listed in the same order. Because Verilog modules often have a very long list of ports (20+ isn't unheard of when designing microprocessors) it can be pretty easy to make a mistake somewhere. So another option is to do ports connections "by name".

- **`add_half a1(.a(a[0]), .b(b[0]), .s(s[0]), .cout(c0));`**

Note: you could then list the ports in any order.

Tri-state devices (page 227, our book isn't so good with this either though, p206 in 1st)

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…

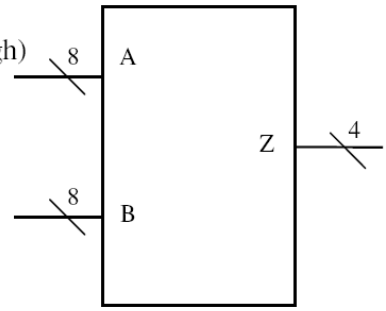
Here are a handful of nice practice questions for to try. Time allowing, me may do a few in class today. Answers won't be posted, but feel free to come to office hours (mine or the GSIs') if you have questions!

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)

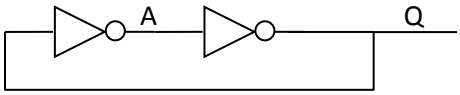
The inputs A[7:0] and B[7:0] are describing input values ranging from 0 to 7 using the "one hot" representation scheme. The output Z[3:0] is to be the sum of the values of the 2 inputs. Z is represented as an unsigned number. For the inputs A7 and B7 are the MSBs. While for the output Z3 is the MSB.



Unusual "one hot" adder

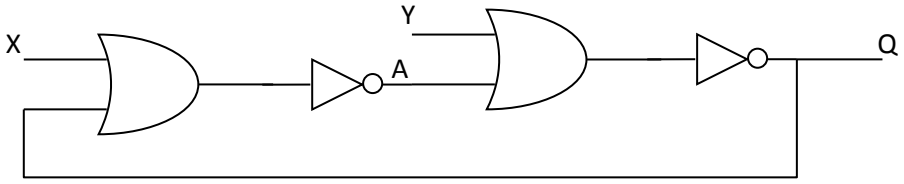
Sequential Logic (3.1 and 3.2. 3.2 is a long difficult section you really should read!)

One thing we have carefully avoided so far is feedback—all of our signals have gone from left-to-right. What if we don't do that?



Consider the figure on the left. It is a "bi-stable" device. That means there are two ways it could be stable. If A=0 then Q=1 and you'd expect that to hold its value (because A is then driven to 0 by Q. In the same way A=1, Q=0 is stable. So we've got two different sets of values for which this could be stable.

So is such a device useful? It would then *seem* like a good candidate for a memory device—after all it can keep one of two values. The problem is that we can't write to that device as both A and Q are already being driven by something else. So let's consider the following device instead



Now consider the above diagram. Recall that 0+X=X. So if X = Y = 0, those OR gates act like wires. So *in that case* the device functions exactly the same as the pair of inverters.

Questions:

1. What is Q if X=0 and Y=1?
2. What is Q if X=1 and Y=0?
3. What is Q if X=0 and Y=1 and then X changes to a 0?

Definition

From our text:

> A ***sequential circuit*** is a circuit whose output depends not only on the circuit's present inputs, but also on the circuit's present state, which is all the bits stored in the circuit. The circuit's state in turn depends on the past sequence of the circuit's input values.

The net effect is that we are using feedback (as shown above) to create ***memory***. That memory allows us to have state (the information stored in the circuit).