EECS 478
Project 2 Report

Gabrielle Guarino
gguarino

2-input AND gate:

The only time a 2-input AND gate will result in a 1 is when both inputs are also 1's. I only needed to add one thing to the truth table to account for this.

3-input XOR gate:

XOR gates result in a 1 when only one of their inputs is a 1 and the others (in this case, 2) are 0's. For the truth table I added these cases. The cases are as follows: in1 = 0, in2 = 0, in3 = 1; in1 = 0, in2 = 1, in3 = 0; in1 = 1, in2 = 0, in3 = 0.

4-input MUX gate:

MUX gates output the input that corresponds to the value the select bits portray. For a 4-input MUX, we have 2 select bits. When the select bits are 00, I output input1. When the select bits are 01, I output input2. When the select bits are 10, I output input3. Lastly, when the select bits are 11, I output input4. This was handled by entries in the truth table that correspond to each of the select bit value combinations. The rest of the inputs, besides the one that value corresponds to, are don't care bits. The one input we care about for each entry is given a 1 value in the truth table. This works because it checks for when that specific input is 1, not worrying about the other inputs, and outputs it. Otherwise it outputs a 0 (which is what that input would be anyways if it wasn't a 1).

Other gates I created:

I found that in the implementations of my modules, it was very helpful to have this library full of functions that act on nodes. I created more of these logic gates to help with my later modules. I created a 8-input MUX, a 2-input MUX, a XOR2 gate, and an inverter. Some of these I didn't end up using, but they were nice to have in case I needed to.

Adder module:

This module gave me the least amount of trouble because I could handle it mostly all in a loop because of the way adders can be combined and rippled. I started with a single full adder, and after each loop iteration, I fed the outgoing out into the incoming cin for the next full adder. The only time I had to do something unique was on the last loop iteration, instead of feeding that cout into cin, I had to output that cout as it was the final one. The full adder logic that I used is: s = (a xor b xor cin), cout = ((a xor b) & c) | (a & b).

Subtractor module:

The way in which I handled the subtractor module was by adding input2's two's complement inverted value to input1. I flipped all the bits in input2 by using my invertor gate I created. Then, I created an ADD module with input1, the new inverted input2, and a cin of 1 always. I had this cin always be inputting a 1 because when converting to two's complement you must flip all the bits and then add a one to get the desired value.

Multiplier module:

For the multiplier module, I iterated through input2's bits, ANDing each one with every bit from input1 to create partial products. I then shifted this partial product by whatever number bit I was on from input2. I eventually summed these partial products together by looping through and creating a buffer that pushed by previous summed output into the input of the next sum.

A struggle that I had while implementing this module was that it calls the other modules often. In my first implementation of the other modules, I created static names for most of the internal nodes. This doesn't work if I call the module multiple times because these nodes are already created for a different purpose. Therefore, I added the output strings, and sometimes the input strings to my naming conventions. This way when I make nodes I am making unique node names based on the current inputs and outputs.

Other modules I created:

I also created an absolute value module to aid in my datapath module. To do this, I took the first bit of the number, XORed it with all its bits, and added the first bit to it. I then outputted this number. This works because if the number is negative, a two's complement, the first bit will be a 1. If you XOR the 1 with all the bits, it will flip each bit, and then you add the 1 to it at the end to receive the positive-valued two's complement representation of the original negative number. If the input is positive to begin with, the first bit is a 0, you XOR that will all the bits and receive the same number as before, and then add a 0 to it, still getting the same number. This was used in my datapath module which will be explained later.

Verification:

Verifying the adder and subtractor modules was simple because I could create accurate blifs for bits 1-32 on my own. I started with a 1bit blif and then built the next one (2bit) off of that one and so on and so forth. This was time consuming, but was mostly just copy and paste since subtractors and adders can be combined and rippled. I then created a script that would create my program's blifs for each bit 1-32, and then take these blifs and compare them against my known-good blifs that I made using the abc binary. I could view the abc binary output and make sure that all networks said they were equivalent.

I couldn't create multiplier and datapath blifs by hand because the multiplier logic is much more complex than adders and subtractors, and not as simple to just combine them. Therefore, I created a C++ program that creates test cases for me by testing every input from 0 to 2^(numbits) and then placed a truth table in the blif that corresponds to the correct output bits receiving ones. This C++ program worked well, but it took too long to create multiplier blifs for more than 12 bits, or even create the datapath blif at all without taking hours. I did use the program to create accurate mult blifs for 1-11 bits, though, and compared these against the ones I calculated with my program. I ran these through the abc binary and made sure the networks stated they were equivalent.

To further test the datapath and multiplier modules, I created singular test cases. I tried to think of edges cases where both inputs were the same, or the largest values possible, or the smallest values possible. I created a script that ran these test cases through the simulator and gave me the output to view and analyze. I also ran these test cases on my mult32.blif because I figured this one had the most chances of failing. I am currently working on a C++ program that

will test random values, input them into the simulator, calculate the output, receive the simulated output, and compare them. If I can finish that before the due time, I will add it to my submission.

I used many scripts to try to automate my verification process so that I could run a single program and everything could be tested at once. I think this worked well for me, though it was time consuming to create such a test suite.

Datapath module:

For my datapath module, I ran into some trouble. First, we are finding the minimum value first, and then taking the absolute value of it. We are receiving signed inputs. Our multiplier takes in unsigned values. Therefore, I created 3 select bits that you determine from the inputs that tell you which one is the minimum. The first select bit is the first bit of x. This select bit tells you whether your x input is negative. The second select bit is the first bit of x XORed with the first bit of y. I did this because this way, these two select bits will be able to tell you if both inputs are negative, both are positive, or just one is negative. If only one input is negative, you output the value that corresponds to that input because it will always be smallest. If both are positive, you output whichever is smaller, the absolute value of 9*x or the absolute value of 12*y. If both are negative, you output whichever is larger, the absolute value of 9*x or the absolute value of 12*y. The third select bit tells you just that, it is 1 if abs(9x) is larger than abs(12y) and a 0 if smaller. I use my 8-input MUX to differentiate among these different select bit combinations.

The select bit combinations are as follows: 000 -> 9x, 001 -> 12y, 010 -> 12y, 011 -> 12y, 100 -> 12y, 101 -> 9x, 110 -> 9x, 111 -> 9x. These correspond to: 000 = “x and y are positive, abs(12y) is larger”, 001 = “x and y are positive, abs(9x) is larger”, 010 = “x is positive, y is negative, abs(12y) is larger”, 011 = “x is positive, y is negative, abs(9x) is larger”, 100 = “x and y are negative, abs(12y) is larger”, 101 = “x and y are negative, abs(9x) is larger”, 110 = “x is negative, y is positive, abs(12y) is larger”, 111 = “x is negative, y is positive, abs(9x) is larger”.