Lab 11 Report

Kiran Nadkarni

Elizabeth

11/15/19

Intro to VHDL

Objectives:

The goal of this lab was to accustom ourselves to the process of VHDL coding, a widely used hardware description language (HDL). We want to do this by using VHDL to complete and augment a previous assignment of ours, which we understand fairly well. This will make it easier to implement the VHDL code, since we do not have to worry too much about any extraneous elements in our circuit that are not hard coded. We are also doing this through the medium of LogicWorks so we will be able to see our code implemented as a simulation of hardware. This will strengthen and deepen our understanding of VHDL, as well as show us a practical use for it.

Introduction:

As mentioned in the objectives, we are going to be demonstrating our knowledge of VHDL in this lab. VHDL is a Hardware Description Language. While this is similar to the idea of a programming language, it is still different because unlike programming languages, this language was built specifically to simulate logic hardware and circuitry without needing any physical parts. We are well accustomed to LogicWorks by now, which has many different gates and circuit parts that we can use to simulate circuits we physically build. VHDL can be used within LogicWorks to minimize a set of gates into a single part, which we can freely use within the software. The syntax of VHDL for our purposes is fairly simple. The rightmost word in a line is our output and to represent a set of gates and inputs for that output, we use a <= symbol after the output name. The next parts of the line would be defining the inputs and the gates used, and these can be done with a system of parentheses and input names. Generally speaking, the formula goes, [input-1-name] [gate name] [input-2-name]. to have more than two inputs, we can chain multiple gates together using this same formula, as long as the gate name was between the inputs. In the same way order of operation is ruled by parentheses in math, parentheses in VHDL will mark the gate operations in our circuit that are closer to the beginning. The last important thing about VHDL code, is that all of these lines should end with a semicolon, or the code will not compile.

This is where the circuit we want to build comes into play. We are trying to build a binary adder, which will be capable adding both positive and negative numbers, as well as able to detect overflow in our circuit. The only modifications we are required to make to the binary adder we made previously is that we have to create VHDL part replacements for a single one of our adders, as well as our overflow detector. Since the VHDL binary adder we are making is only going to add 1 bit at a time, I will need 4 of these adders to accomplish the goal of a single 4-bit binary adder that already exists in LogicWorks.

Procedure:

1- The first step in creating a circuit like this is to remember how we created the first one. The main elements of the original adder circuit that we want to keep are the two's complement XOR gates. The reason we have these is because we need the numbers to be in two's complement if we are trying to represent them as negatives, and we do this with the XORs and the switch that will control the sign of the numbers. These XOR gates are connected to the initial adders which only exist to perform the second part of two's complement where we add a digit.

2- The second part is to create our Full Adder circuit, which we will do in VHDL, 1 bit at a time. We can then create the full 4-bit adder as we have done on the diagram below, by connecting the carry output to the c input on the adders. Then our sum outputs function as the 4-bit output, with a and b being the 4-bit inputs. We will make the adder by setting the 3 inputs of the adder to be a, b, and c, with the outputs as sum and carry. These will be the variables we can use in our VHDL code. we then can define our outputs in terms of gates and the inputs. To follow a single bit adder formula, we will set

```
sum <=(a xor b) xor c;

carry<= (a and b) or (c and (a xor b));
```

We choose these values because we know that the sum can only really be a maximum of 3 or 0011 per adder, because we are adding at most three 1's and at least zero 1's, so XOR gates would be perfect for this. We make the carry using a similar thought process, because we know that we can only have a carry out only if we have two or three input 1's. We now just have input this code into LogicWorks and compile it to make sure our adder now works.

3- We can then create the other side of our circuit, where we have another line of XOR gates to take the answer out of two's complement, and we also add a 1 to it in a normal built in adder again to complete to the switch to normal binary. This is our final answer, so we are now done, we can use a hex display to display our output using the 4 input bits.

4- Our last step is to create a VHDL representation for our overflow detector and implement it. As mentioned before, we can use the same input connections that we would for the old implementation of it. The normal gate implementation of it uses three inputs, connected to the positive/negative sign switches for both inputs and the MSB of the main adder we are using. We use these three inputs specifically because this is the relevant information to know if there is overflow, since we need to know if the MSB exceeds the -8 to 7 range and if the inputs are potentially both negative or positive to cause this. We did this before by ANDing the sign switches with the output 2's complement MSB and the NOTs of those inputs with the NOT of the 2's complement output. We do this with two 3-input AND gates, and then OR them together to give us our overflow output. These are to represent both the double positive and double negative number input cases for overflow. So we will have three inputs A, B, and C, and an output O which will lead to a binary

probe telling us 1 for overflow and 0 for no overflow. This is simply represented in VHDL too, as

O<= (A and B and (not C)) or ((not A) and (not B) and C);

Results:

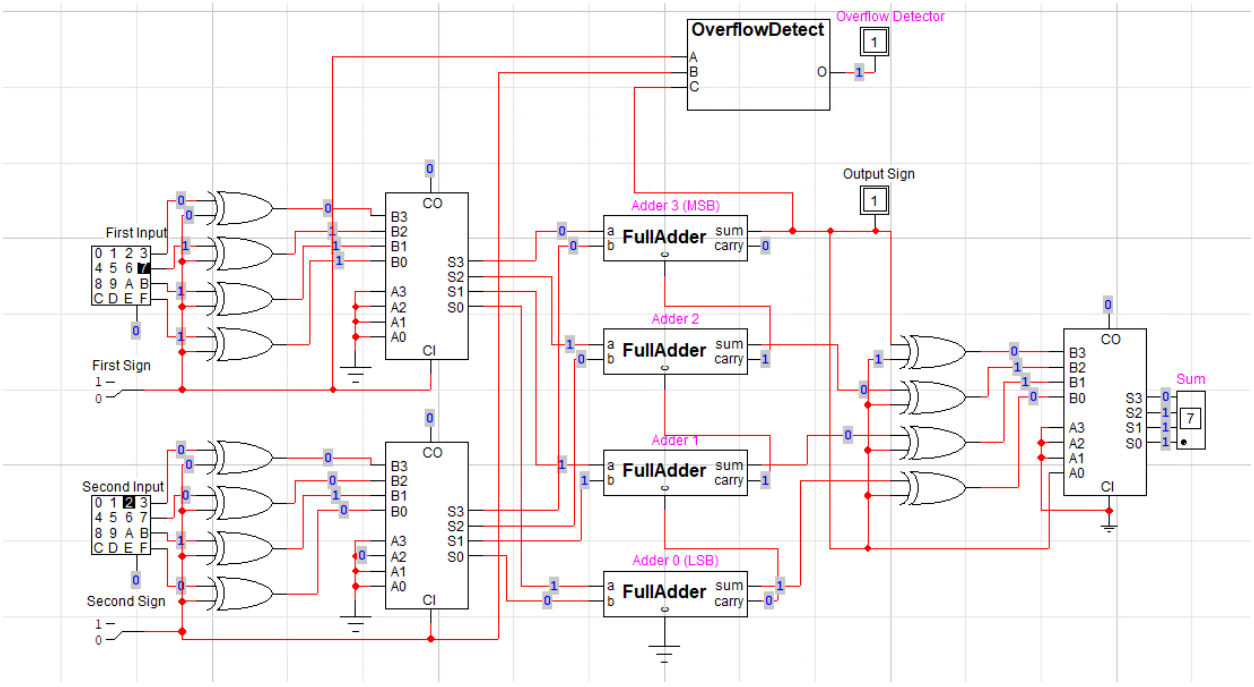

Figure 1: This is the diagram of my completed circuit with the overflow detector and the full adders. Notice the full adders and the overflow detector are marked to show which were coded with VHDL and which circuit components are natively from LogicWorks.

First Num (with sign)	Second Num (with sign)		Result (with sign)	Overflow
2	3		5	0
-5	-2		-7	0
3	-2		1	0
-6	4		-2	0
3	4		7	0
-5	-3		-8	0
6	5		-5	1
-7	-4		5	1
10	-2		-8	0
-11	2		7	0

Figure 2: This is my data from this lab. I have taken values on the low end and high end of both positive and negative values, as well as some in between. You can see that there is appropriate overflow with a sum outside of the range of -8 to 7.

Discussion:

My results were the same as the original adder assignment we had, which shows that I created the circuit correctly and that it works for values at a high, low, and medium range. The overflow detection system worked perfectly, as well as the method of using XOR gates to put the number we are working with into 2's complement form. As expected, the adder only will not encounter overflow when displaying numbers in the range of -8 to 7. By extension, this means that the results for the VHDL code, which was my goal for this lab specifically, were correct, because if the circuit is able to function the same as before, then that means that it is is working fine for all possible output values, as mentioned before.

Conclusion:

I believe that the most difficult part of this lab was the steps surrounding getting our created circuit components into the LogicWorks library. Remaking my circuit was pretty easy because it would always have the same number of inputs and outputs so I could treat remaking the full adder circuit and making the VHDL parts as isolated projects that would work together later. The actual VHDL syntax was fairly straightforward as well, and while having programming experience might help with this lab, I do not even think that it is necessary whatsoever because of how intuitive the syntax was. Going along with this, the creation of the VHDL code was the most entertaining thing to me in this lab, because the possibilities feel freer than if I was strictly building a circuit in LogicWorks or on a protoboard.

Question:

I preferred using VHDL to construct this circuit, because although there might be a steeper learning curve to creating your own circuit components, the rewarding minimization of footprint of the circuit is worth it. I believe that depending on how the circuit is made, it might even be faster to implement the circuit using VHDL rather than dragging parts out of the LogicWorks library.