Lab 5 VHDL Adder Subtractor

July 12, 2017

Cs211 Summer 2017

Xu Mingzhi

Contents

[1. Objective 3](#_Toc487665088)

[2. 4-bit Ripple Carry Adder Subtractor 4](#_Toc487665089)

[2.1 Functionality and Specification 4](#_Toc487665090)

[2.2 Simulation 17](#_Toc487665091)

[2.3 Demonstration 18](#_Toc487665092)

[3. 4-bit Carry Lookahead Adder Subtractor 24](#_Toc487665093)

[3.1 Functionality and Specification 24](#_Toc487665094)

[3.2 Simulation 27](#_Toc487665095)

[4. 4-bit LPM Adder Subtractor 29](#_Toc487665096)

[4.1 Functionality and Specification 29](#_Toc487665097)

[4.2 Simulation 37](#_Toc487665098)

[5. Conclusion 39](#_Toc487665099)

# **Objective**

In this lab, we will use what we have learned in the previous labs about adders, multiplexers, decoders, and seven-segment display and implement it by using VHDL files. VHDL stands for Very High Speed Integrated Circuits Hardware Description Language. The way how VHDL works is slightly different from other coding languages such as C++ or Java. Unlike C++ or Java where when the codes are compiled, it executes line by line which means ordering is important but VHDL compiles parallelly which mean order doesn’t matter if each part is implemented correctly into the body. We will be using VHDL to create a package library to store each component we will be creating to build up the final Ripple Adder Subtractor and Carry Look Ahead Adder Subtractor.

The Adder Subtractor we will be designing in this lab will be

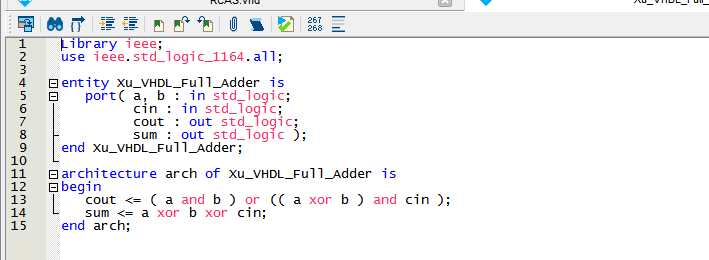
1. 4-bit Ripple Carry Adder Subtractor
2. 4-bit Carry Look Ahead Adder Subtractor
3. 4-bit LPM Adder Subtractor

# **4-bit Ripple Carry Adder Subtractor**

## ***Functionality and Specification***

The 4-bit Ripple Carry Adder Subtractor is something we have done in lab 1 where we use Full Adders as components. Since we are building a 4-bit Ripple Carry Adder Subtractor, we will use 4 Full Adders as components which is adding two 4-bit binary number together and produce a 5-bit output including a carry-out cout as the fifth bit result. In this lab we will be implementing this 4-bit Ripple Carry Adder Subtractor using VHDL files using VHDL languages which is more efficient and organize if something with more bits are being implemented instead of using block diagrams.

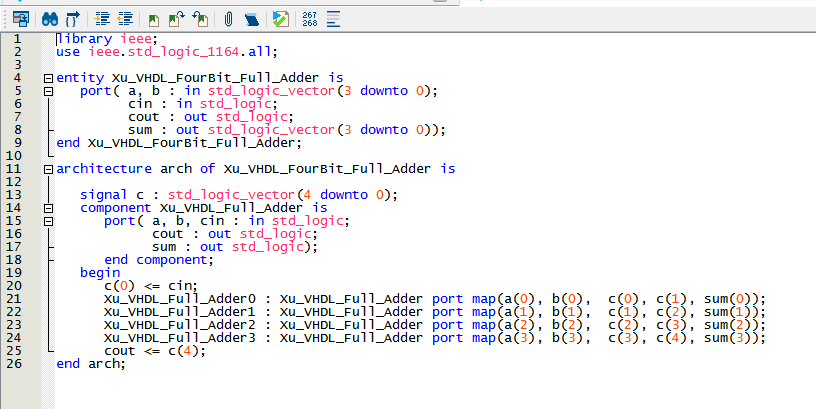
Since the 4-bit Ripple Carry Adder Subtractor require 4 Full Adders as components we will first implement the Full Adder in VHDL which later we will create a package that functions as a library and store this component so we can call and use it in another VHDL files of the same directory.

Figure 1: VHDL code for Full Adder component.

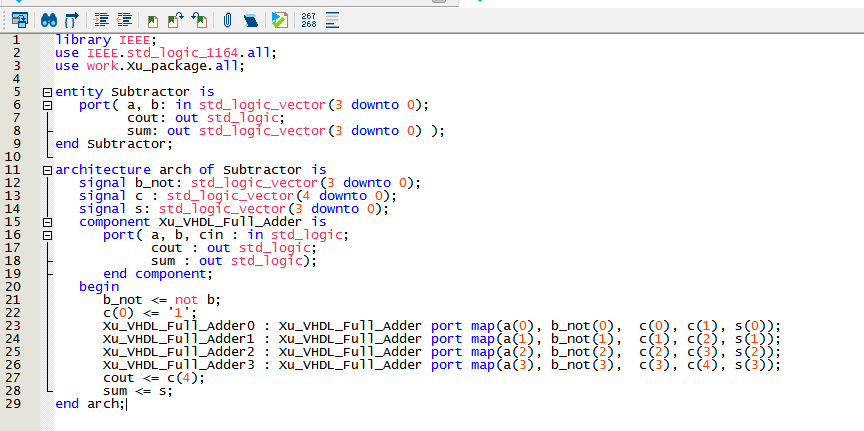
This is the VHDL code for the Full Adder, we will need to declare input for a, b and cin, output for cout and the sum using port under the entity. Then we will implement the Boolean functions under the architecture which the operators representing the gates in a block diagram. The Boolean function for the output cout and sum will be

cout = (a and b) or ((a xor b) and cin)

sum = a xor b xor cin

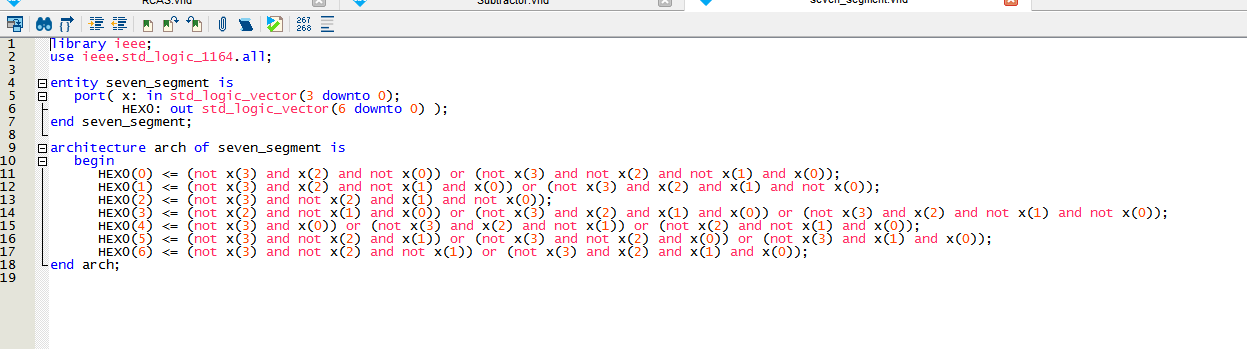
Next, we will implement the 4-bit Full Adder using the Full Adders as components and this would be the base of constructing the final 4-bit Ripple Carry Adder Subtractor.Figure 2: VHDL code for 4-bit Full Adder.

This is the VHDL code for 4-bit Full Adder, we will need to declare input a and b as 4-bit by using std\_logic\_vector(3 downto 0) which means that the input a will have the grouping of a(0), a(1), a(2), and a(3). It goes the same for other inputs and outputs. We will also declare a carry in cin input and have outputs cout and sum in 4-bit which includes sum(0), sum(1), sum(2), and sum(3) since we will be implementing 4 Full Adders so there will be 4 sum outputs. Next, we will need to create a signal c under the architecture for 4-bit Full Adder which functions as a local storage to store the carry-ins and carry-outs that will go into and out from each Full Adder component. We will then call the component Full Adder we had created, so we can put in inputs and receive outputs from it. c(0) will be the cin that goes into the first Full Adder and generate cout into c(1) and a sum into sum(0), c(1) will be the cin for the second Full Adder which generate cout into c(2) and a sum into sum(1), c(2) will be the cin for the third Full Adder which generate cout into c(3) and a sum into sum(2), c(3) will be the cin for the last Full Adder which generate cout into c(4) and a sum into sum(3). Then we will declare c(4) as the cout output for this 4-bit Full Adder and the sums generated from the Full Adders will automatically assigned into the sum output since that is the sum output.

Now we will need to add a subtractor into the 4-bit Full Adder to make it into a 4-bit Full Adder Subtractor. First, we need to create a subtractor to see the idea of how it works then we will implement it into the 4-bit Full Adder later.Figure 3: VHDL code for subtractor.

This is the code for subtractor, it has the same inputs and outputs from the 4-bit Full Adder. Although it said it’s a subtractor and you would expect it to do subtraction, but that’s wrong. The subtractor will be using Full Adders as components. According to the algorithm described in the lab manual about subtractor, we will need to invert the b input and add one to it and by doing the implementations as 4-bit Full Adder, the result you’ll get will be the same as subtracting but you’re still doing adder. For example, 0111 – 0001 = 0110, but what the subtractor is really doing is 0111 + 0001, invert the 0001 into 1110 add one to it and becomes 1111 then 0111 + 1111 = 10110 ignore the fifth-bit which becomes 0110. We also used the package we have created to store the components we have created so that is why there is no need to call the component for Full Adder in the Subtractor architecture unlike the 4=bit Full Adder which you need to call the component for Full Adder.

We will also need to display the inputs and outputs of the 4-bit Ripple Carry Adder Subtractor on the seven-segment display which we have done in lab 3 where we converted binary to decimal and have the seven-segment display to display decimals. Therefore, we will need to implement the seven-segment decoder using VHDL.

Figure 4: VHDL code for seven-segment decoder.

This is the VHDL code for seven-segment decoder which includes a 4-bit x input and a 7-bit output which correspond to the a, b, c, d, e, f, g element on the seven-segment display. The VHDL code for seven-segment decoder uses the same Boolean function as the one that we have implemented in lab 3. The Boolean function will be

a = HEX0(0) = (not x(3) and x(2) and not x(0)) or (not x(3) and not x(2) and not x(1) and x(0));

b= HEX0(1) = (not x(3) and x(2) and not x(1) and x(0)) or (not x(3) and x(2) and x(1) and not x(0));

c = HEX0(2) = (not x(3) and not x(2) and x(1) and not x(0));

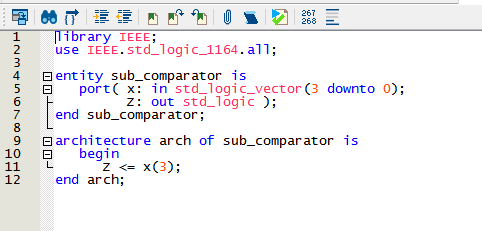
d = HEX0(3) = (not x(2) and not x(1) and x(0)) or (not x(3) and x(2) and x(1) and x(0)) or (not x(3) and x(2) and not x(1) and not x(0));

e = HEX0(4) = (not x(3) and x(0)) or (not x(3) and x(2) and not x(1)) or (not x(2) and not x(1) and x(0));

f = HEX0(5) = (not x(3) and not x(2) and x(1)) or (not x(3) and not x(2) and x(0)) or (not x(3) and x(1) and x(0));

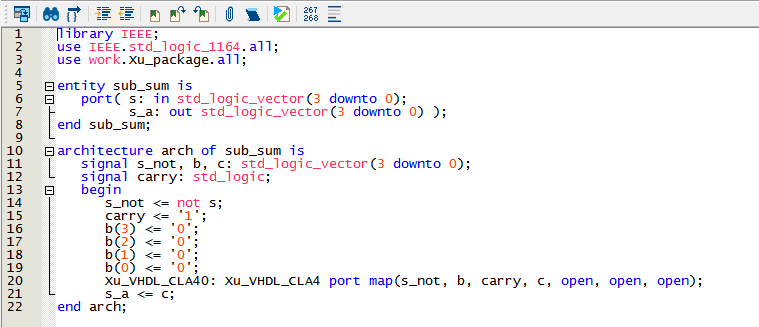
g = HEX0(6) = (not x(3) and not x(2) and not x(1)) or (not x(3) and x(2) and x(1) and x(0));

In this lab, we will be using signed decimals which is different from lab 3 where we only worked with unsigned decimals which goes from 0 to 15. Signed decimal for 4-bit will be ranged from -8 to 7. Therefore, we will need to create a comparator to check if the input of a and b and the output sum will be over 7 or not. In lab 3 the comparator we created was to check if the binary input and output are greater than 10 because we were using unsigned decimal. We need to check if the binary input and output is greater than 7 in this lab because we are using signed decimal and since the range for 4-bit binary only ranges from -8 to 7. If the number is greater than 7 it will be consider as an overflow and it will go back to the negative. For example, 0111 + 0001 = 1000, which is 7 + 1 = 8 but since it is signed decimal it will be 7 + 1 = -8.

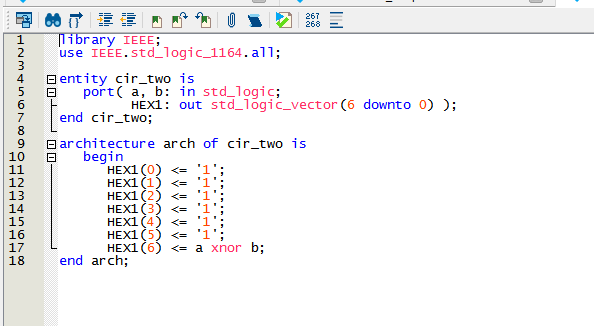
Figure 5: VHDL code for comparator.

This is the VHDL code for the comparator we will be using to check if the binary inputs and output will be greater than 7 and to check that all we need to check if fourth bit is 1 since 1000 will be 8 and 1000 to 1111 will be considered as -8 to -1.

Since we are working with signed decimal we will need a circuit one similarly to the one we have done in lab 3 where a new set of input will be used if the binary input is greater than 10. In this case the circuit one we will be creating is going to be an inverter which when binary input or output is greater than 7 it will go to 8 then back to 7 until it reaches 1. For example, 1001 will be 7 instead of 9. 1001 in signed decimal will be -7, so we will have a circuit two that displays the negative sign using the output of the comparator to determine if there is a negative sign or not.

Figure 6: VHDL code for Circuit One.

This is the VHDL code for Circuit One that will invert the 4-bit input s to the 4-bit output s\_a. We will need to create a 4-bit s\_not to store the inverted input s by doing not s, then we will create a 4-bit input b to be 0000 and have a carry to be 1. The process involve using the Carry LookAhead Adder which will be described in another section, but the function of a carry lookahead adder will be the same as 4-bit full adder, then we will have c to store the sum output from the adder and output it with s\_a.

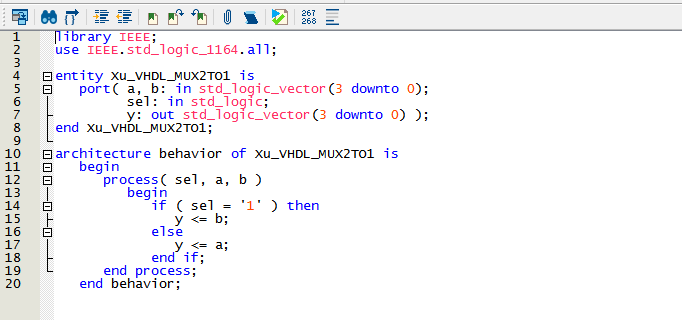
Figure 7: VHDL code for Circuit Two.

This is the VHDL code for Circuit Two that will display a negative sign – on the seven-segment display, unlike in lab 3, the Circuit Two displays 0 or 1. The inputs will be a and b and outputs 7-bit output onto the seven-segment display. Since we will be only using the HEX(6) which is g to display negative sign – all the other light on the display will be off which will be set to one. The Boolean function for g to light up will be

HEX1(6) = a xnor b

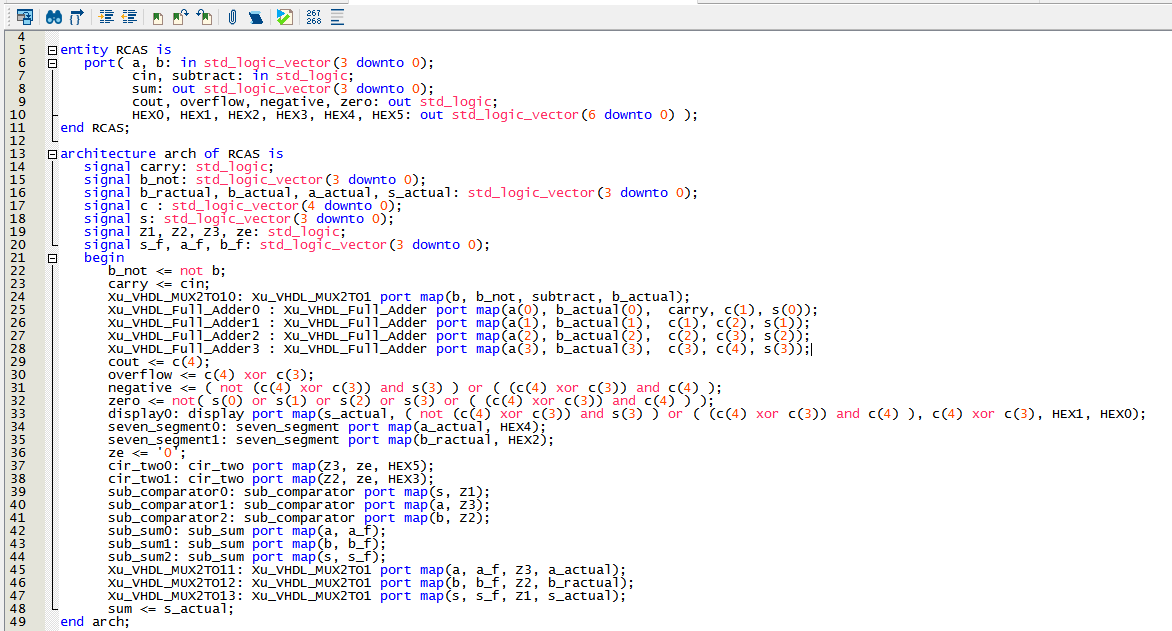
It involves using the negative and overflow detection to determine if it will display the negative sign – onto the display.

We will also need a 2:1 multiplexer to alternate inputs and outputs on the 4-bit Ripple Carry Adder Subtrator.

Figure 8: VHDL code for 2:1 Multiplexer.

This is the VHDL code for a 2 to 1 Multiplexer which is the same as the one we have done in lab 2 about multiplexers, but this time in VHDL. There will be 4-bit input a and b and a selector input and a 4-bit output y which is determined by the selector. The process will be if selector is 1 then output y will be b and if selector is 0 then the output y will be a.

Now we will create the 4-bit Ripple Carry Adder Subtractor by using all the components that is described in this section. First, we will use the 4-bit Full Adder as the base and we will implement the components into it.

Figure 9: VHDL code for 4-bit Ripple Carry Adder Subtractor.

This is the VHDL code for 4-bit Ripple Carry Adder Subtractor. In the port we will have 4-bit binary input for a and b, a carry in cin input, a subtract input indicating that it will do subtraction, the 4-bit sum output, then detection output for cout, overflow, negative, zero, lastly we will have outputs for HEX0, HEX1, HEX2, HEX3, HEX4, HEX5 to the seven-segment display.

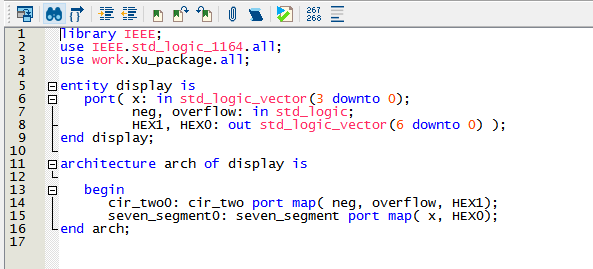
In the architecture, we will have a signal carry that stores the carry-in cin. There will be a signal b\_not that stores not b. Then there will be a 2:1 MUX that determines if it will be doing adder or subtractor, it uses the input subtract as selector and store b or b\_not into b\_actual. Then we will have 4 Full Adder components that takes in a, b\_actual, and carry as inputs and have outputs carry-out stored in signal c and the output sums stored in signal s. output cout will be determined by c(4) since that will be the last carry-out outputted from the last Full Adder component. Then we will have detections for overflow which means if the output is not in the range of -8 to 7, detection for negative shows that the output is a negative number, detection for zero shows that the output sum is zero. The Boolean functions are

overflow <= c(4) xor c(3);

negative <= ( not (c(4) xor c(3)) and s(3) ) or ( (c(4) xor c(3)) and c(4) );

zero <= not( s(0) or s(1) or s(2) or s(3) or ( (c(4) xor c(3)) and c(4) ) );

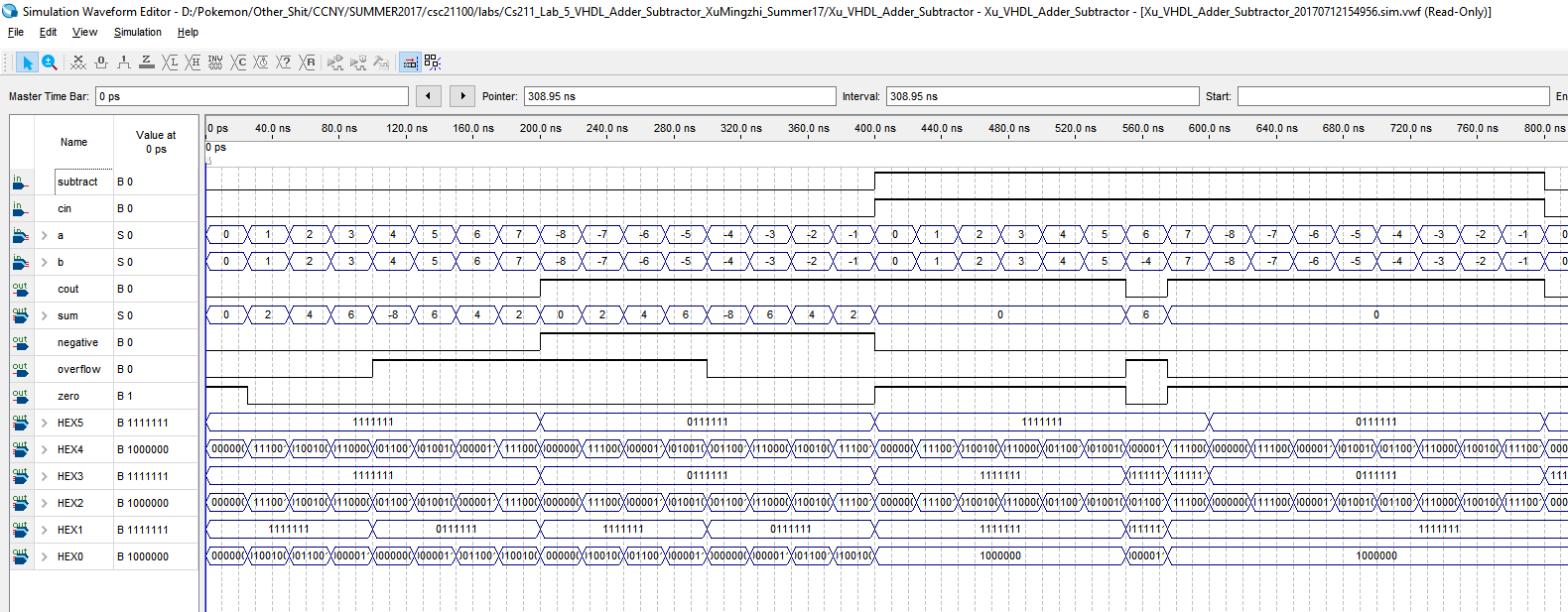
Now we have all the outputs and detections displayed on the LEDR of the board, we will need to display the inputs and output on the seven-segment display. Since we will only display decimal numbers from -8 to 7, we will need to have inputs a and b and signal s to go through Circuit One which produces an inverted binary output to be stored in signals a\_f, b\_f, s\_f. Then will have inputs a and b and signal s to go through the Comparator to check if the binary is greater seven or not, the output from the Comparator will be stored in the signals Z1, Z2, and Z3. Then we will use 2:1 MUXs to determine which number we will be using for the display, we will have Z1 as a selector to determine s\_actual will store s or s\_f. Z2 will be the selector to determine b\_ractual will store b or b\_f. Z3 will be the selector to determine a\_actual will store a or a\_f. Now we have binary inputs and outputs in the range of -8 to 7, we will use a\_actual and b\_ractual as inputs to the seven-segment decoder components and a\_actual will have a HEX4 output and b\_ractual will have a HEX2 output. The Z2 and Z3 will be used as a inputs for Circuit Two to determine if there will be negative sign displayed on the seven-segment display or not, since the g in Circuit Two has a Boolean function of a XNOR b, we will need another input 0 which is stored in ze as the b input. It is slightly different for the output sum display, it uses a display component which includes Circuit Two and Seven-segment decoder.

Figure 10: VHDL code for display component.

This is the VHDL code for the display component, it will a 4-bit input x, input neg and input overflow and outputs 7-bit HEX1 and HEX0. The architecture includes Circuit Two and a Seven-segment decoder. Ultimately it is the same as just calling each one in the Ripple Carry Adder Subtractor but doing it this way shows that there are ways to lessen the code that can be written. This time the Circuit Two will not take Z1 which is the output for the comparator for s, it will use negative and overflow detection to determine if the output sum on the display will have a negative sign or not and the seven-segment decoder will display s\_actual which is also the final output sum.

## ***Simulation***

In the simulation, we will give values of 0 and 1 to the inputs at varying intervals. Input subtract and cin will have values of 0 and 1 at each 800-ns interval. Input a(3) and b(3) will have values of 0 and 1 at each 400-ns interval. Input a(2) and b(2) will have values of 0 and 1 at each 200-ns interval. Input a(1) and b(1) will have values of 0 and 1 at each 100-ns interval. Input a(0) and b(0) will have values of 0 and 1 at each 50-ns interval.

 Figure 11: Vector waveform simulation for 4-bit Ripple Carry Adder Subtractor.

Since we have inputs a and b and the output sum in a group, we can have it to display signed decimal. We can observe from the simulation that when subtract is 0, it will be doing Adder between input a and b, and when subtract is 1, it will be doing Subtractor between input a and b, and when subtract is 1, cin should always be 1 because the algorithm for subtractor to work requires a cin of 1. We can also observe that the outputs is functioning according to the Boolean functions that are implemented in the VHDL code.

## ***Demonstration***

The PIN assignment of the inputs and outputs on the DE1-SoC Board.

a[0] is assigned to SW[0] which is PIN\_AB12

a[1] is assigned to SW[1] which is PIN\_AC12

a[2] is assigned to SW[2] which is PIN\_AF9

a[3] is assigned to SW[3] which is PIN\_AF10

b[0] is assigned to SW[4] which is PIN\_AD11

b[1] is assigned to SW[5] which is PIN\_AD12

b[2] is assigned to SW[6] which is PIN\_AE11

b[3] is assigned to SW[7] which is PIN\_AC9

cin is assigned to SW[8] which is PIN\_AD10

subtract is assigned to SW[9] which is PIN\_AE12

sum[0] is assigned to LEDR[0] which is PIN\_V16

sum[1] is assigned to LEDR[1] which is PIN\_W16

sum[2] is assigned to LEDR[2] which is PIN\_V17

sum[3] is assigned to LEDR[3] which is PIN\_V18

cout is assigned to LEDR[4] which is PIN\_W17

negative is assigned to LEDR[5] which is PIN\_W19

zero is assigned to LEDR[6] which is PIN\_Y19

overflow is assigned to LEDR[7] which is PIN\_W20

HEX0[0] is assigned to HEX0[0] which is PIN\_AE26

HEX0[1] is assigned to HEX0[1] which is PIN\_AE27

HEX0[2] is assigned to HEX0[2] which is PIN\_AE28

HEX0[3] is assigned to HEX0[3] which is PIN\_AG27

HEX0[4] is assigned to HEX0[4] which is PIN\_AF28

HEX0[5] is assigned to HEX0[5] which is PIN\_AG28

HEX0[6] is assigned to HEX0[6] which is PIN\_AH28

HEX1[0] is assigned to HEX1[1] which is PIN\_AJ29

HEX1[1] is assigned to HEX1[1] which is PIN\_AH29

HEX1[2] is assigned to HEX1[2] which is PIN\_AH30

HEX1[3] is assigned to HEX1[3] which is PIN\_AG30

HEX1[4] is assigned to HEX1[4] which is PIN\_AF29

HEX1[5] is assigned to HEX1[5] which is PIN\_AF30

HEX1[6] is assigned to HEX1[6] which is PIN\_AD27

HEX2[0] is assigned to HEX2[0] which is PIN\_AB23

HEX2[1] is assigned to HEX2[1] which is PIN\_AE29

HEX2[2] is assigned to HEX2[2] which is PIN\_AD29

HEX2[3] is assigned to HEX2[3] which is PIN\_AC28

HEX2[4] is assigned to HEX2[4] which is PIN\_AD30

HEX2[5] is assigned to HEX2[5] which is PIN\_AC29

HEX2[6] is assigned to HEX2[6] which is PIN\_AC30

HEX3[0] is assigned to HEX3[0] which is PIN\_AD26

HEX3[1] is assigned to HEX3[1] which is PIN\_AC27

HEX3[2] is assigned to HEX3[2] which is PIN\_AD25

HEX3[3] is assigned to HEX3[3] which is PIN\_AC25

HEX3[4] is assigned to HEX3[4] which is PIN\_AB28

HEX3[5] is assigned to HEX3[5] which is PIN\_AB25

HEX3[6] is assigned to HEX3[6] which is PIN\_AB22

HEX4[0] is assigned to HEX4[0] which is PIN\_AA24

HEX4[1] is assigned to HEX4[1] which is PIN\_Y23

HEX4[2] is assigned to HEX4[2] which is PIN\_Y24

HEX4[3] is assigned to HEX4[3] which is PIN\_W22

HEX4[4] is assigned to HEX4[4] which is PIN\_W24

HEX4[5] is assigned to HEX4[5] which is PIN\_V23

HEX4[6] is assigned to HEX4[6] which is PIN\_W25

HEX5[0] is assigned to HEX5[0] which is PIN\_V25

HEX5[1] is assigned to HEX5[1] which is PIN\_AA28

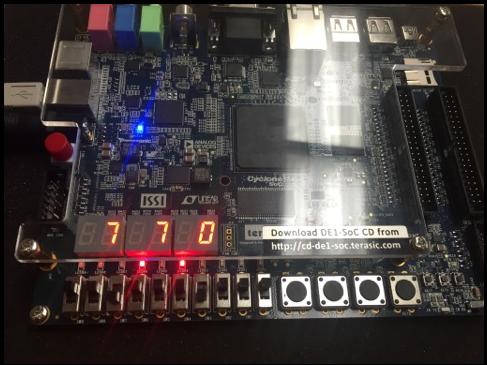
HEX5[2] is assigned to HEX5[2] which is PIN\_Y27

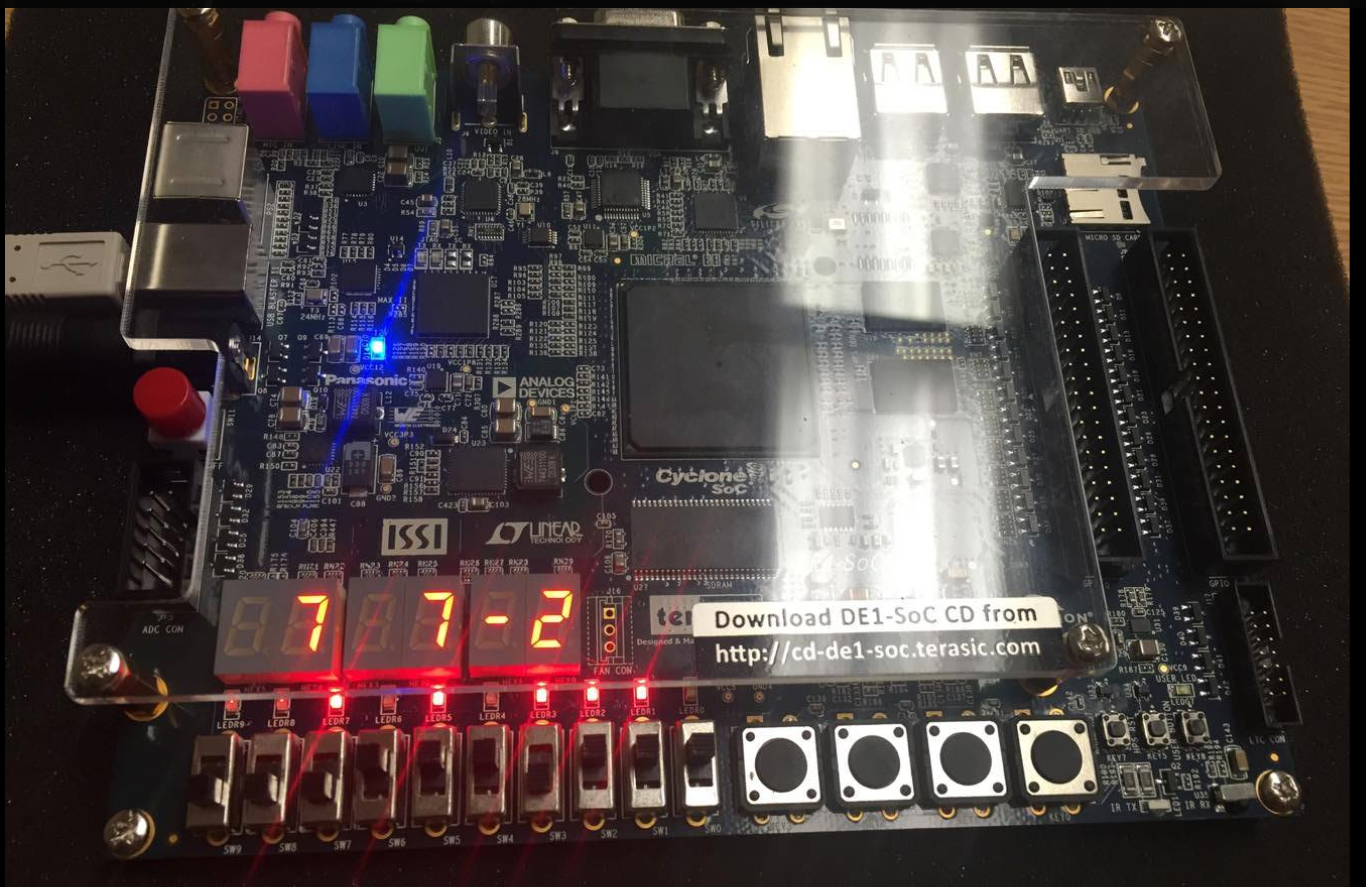
HEX5[3] is assigned to HEX5[3] which is PIN\_AB27

HEX5[4] is assigned to HEX5[4] which is PIN5\_AB26

HEX5[5] is assigned to HEX5[5] which is PIN\_AA26

HEX5[6] is assigned to HEX5[6] which is PIN\_AA25

Figure 12: 7 – 7 = 0.

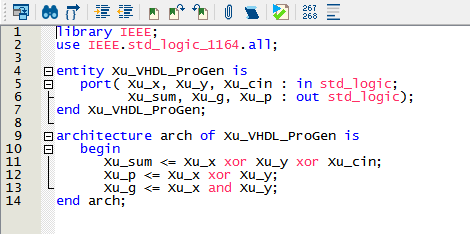
Figure 13: 7 + 7 = 14 so it is an overflow so it will become -2.

# **4-bit Carry Lookahead Adder Subtractor**

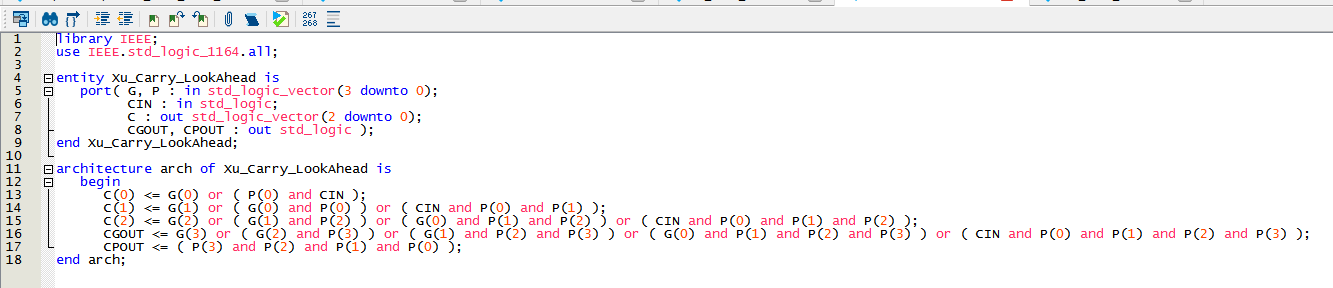
## ***Functionality and Specification***

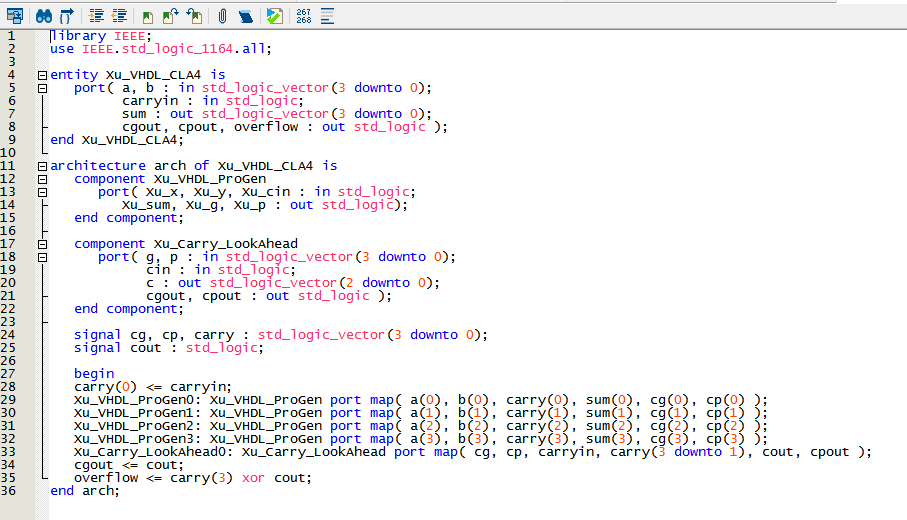
The 4-bit Carry Lookahead Adder Subtractor is something new that is introduced in this lab. The functionality of this Carry Lookahead Adder Subtractor is the same as a Ripple Carry Adder Subtractor which takes in two binary inputs and produce a binary output sum. The difference between the Carry Lookahead Adder Subtractor and Ripple Carry Adder Subtractor is that the Carry Lookahead Adder Subtractor will be more efficient and faster than the Ripple Carry Adder Subtractor due to the carry-ins and carry-outs. For example, if it takes 1-ns to go through a Full Adder component it will take n-ns to go through a n-bit Ripple Carry Adder Subtractor due to the delay needed to take the cout from each Full Adder component and set it as the new cin for the next Full Adder Component. Why does the Carry Lookahead Adder Subtractor function more efficient than the Ripple Carry Adder Subtractor? It is because there is a component in the Carry Lookahead Adder Subtractor call Carry Lookahead which calculates the final cout without having the carry-ins and carry-outs going from one Full Adder component to the other.

First we will need to create a Propagate Generate component which takes input a, b, and cin and outputs sum, Propagate p, and Generate g.

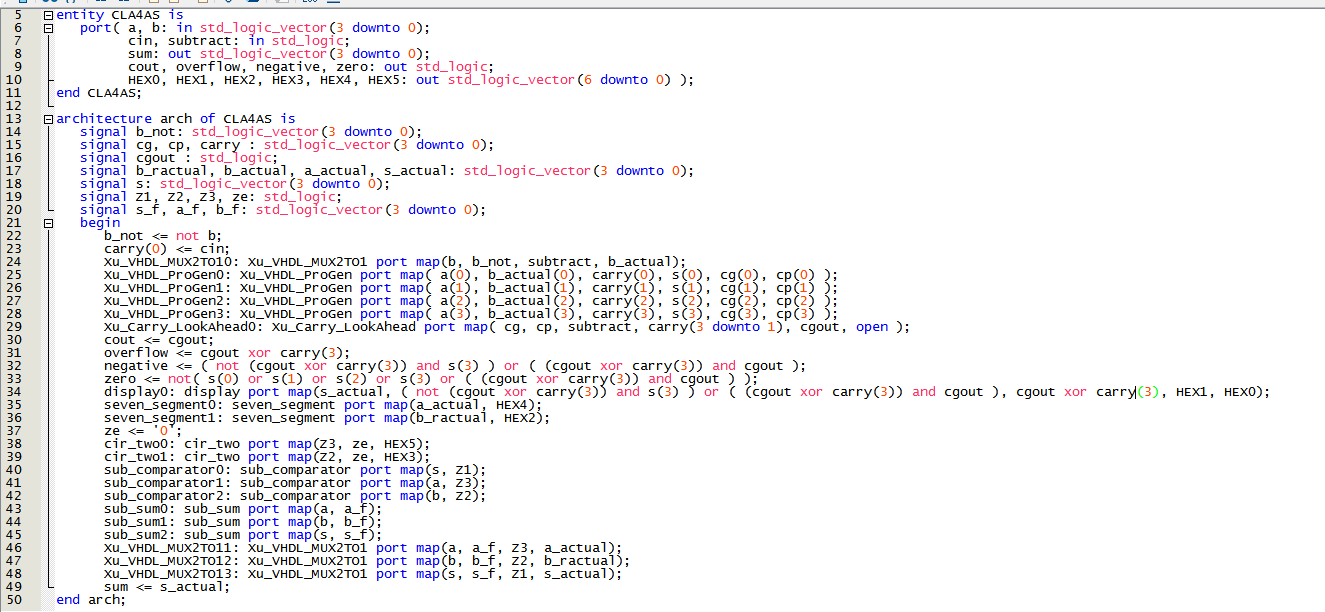
Figure 14: VHDL code for Propagate Generate component.

This is the VHDL code for the Propagate Generate component, similarly to a Full Adder component which produces a output sum but instead of outputting a cout it will have two outputs which are Propagate and Generate to be used in a Carry Lookahead component to compute a final cout.

Figure 15: VHDL code for Carry Lookahead component.

This is the VHDL code for the Carry Lookahead component. It will take the outputs p and g produced by the Propagate Generate component and cin as inputs and outputs the carry-ins that will go into the next Propagate Generate component and the final carry-out at the same time so there will be no delay like the Ripple Carry Adder Subtractor.Figure 16: VHDL code for Carry Lookahead Adder.

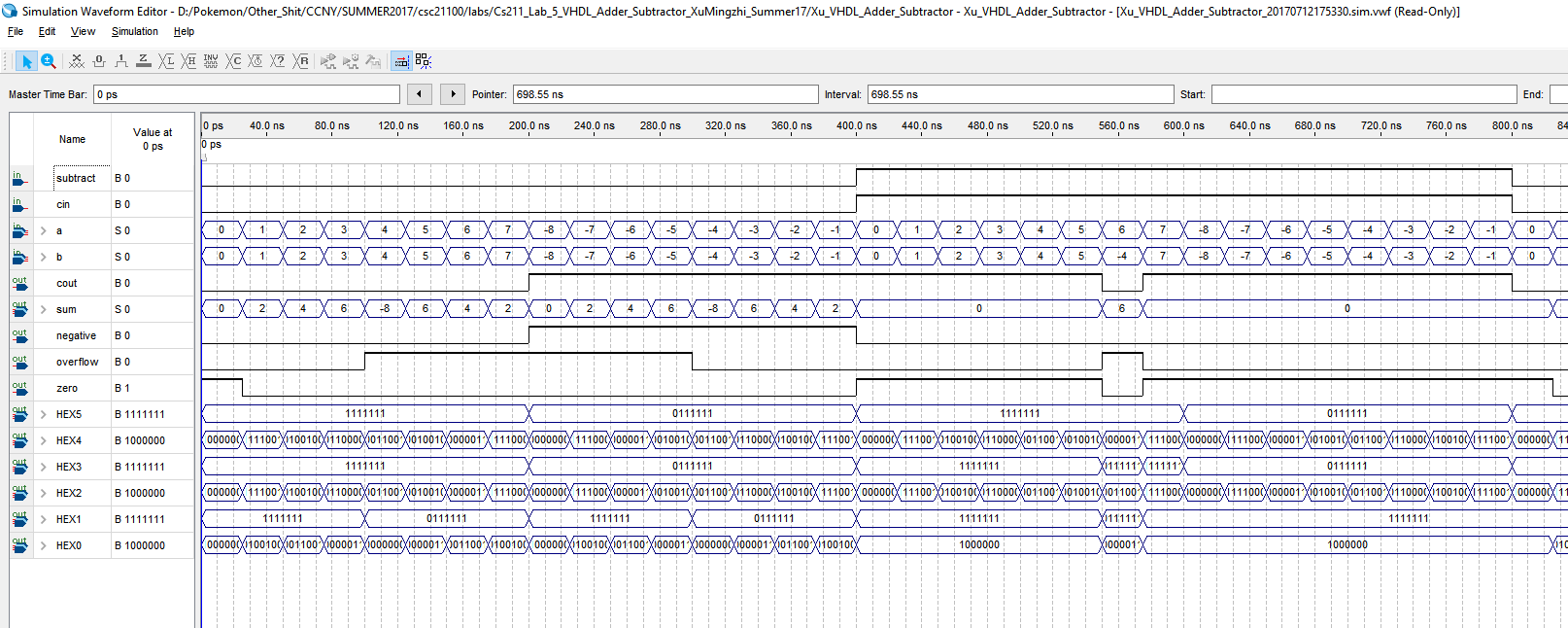
This is the VHDL code for the Carry Lookahead Adder which is implemented using the two components Propagate Generate and Carry Lookahead. It looks similarly like the 4-bit Full Adder, the Propagate Generate component is like the Full Adder component but the Carry Lookahead Adder will have no delay for the carry-ins and carry-out since it will all be generated by the Carry Lookahead component which the Ripple Carry Adder don’t have. This is possible due to the way how VHDL language are being executed unlike other compute languages.

Figure 17: VHDL code for 4-bit Carry Lookahead Adder Subtractor.

This is the VHDL code for the 4-bit Carry Lookahead Adder Subtractor, it will have same inputs and outputs as the 4-bit Ripple Carry Adder Subtractor and uses the same components to manipulate the binary inputs and outputs which are described in the last section so there is no need to restate everything again. The only difference is the Carry Lookahead component which the Ripple Carry Adder Subtractor don’t and this component will produce the carry-ins and the final carry-outs with no delay.

## ***Simulation***

In the simulation, we will give values of 0 and 1 to the inputs at varying intervals. Input subtract and cin will have values of 0 and 1 at each 800-ns interval. Input a(3) and b(3) will have values of 0 and 1 at each 400-ns interval. Input a(2) and b(2) will have values of 0 and 1 at each 200-ns interval. Input a(1) and b(1) will have values of 0 and 1 at each 100-ns interval. Input a(0) and b(0) will have values of 0 and 1 at each 50-ns interval.

Figure 18: Vector waveform simulation for 4-bit Carry Lookahead Adder Subtractor.

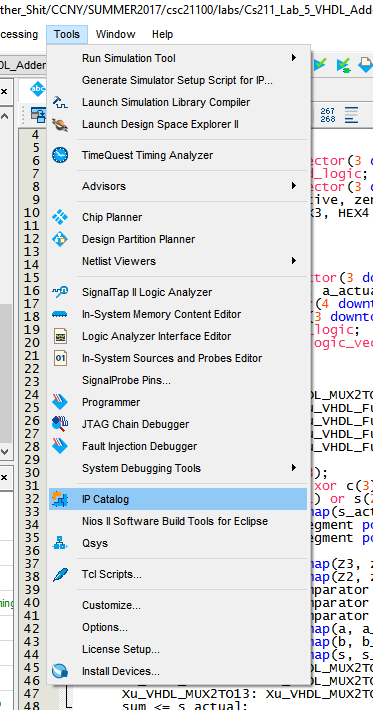
Since we have inputs a and b and the output sum in a group, we can have it to display signed decimal. We can observe from the simulation that when subtract is 0, it will be doing Adder between input a and b, and when subtract is 1, it will be doing Subtractor between input a and b, and when subtract is 1, cin should always be 1 because the algorithm for subtractor to work requires a cin of 1. As we can see, the simulation produced by the 4-bit Carry Lookahead Adder Subtractor is the same as 4-bit Ripple Carry Adder Subtractor so there is no need to load it onto the board since it will do the same thing as the 4-bit Ripple Carry Adder Subtractor.

# **4-bit LPM Adder Subtractor**

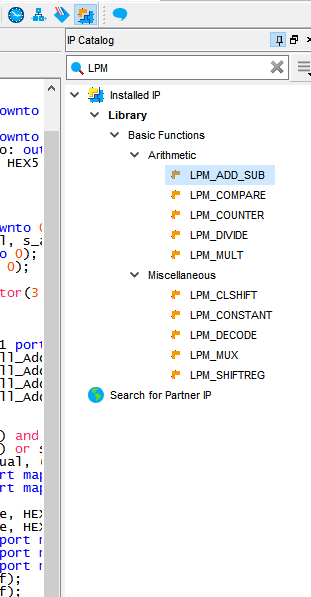
## ***Functionality and Specification***

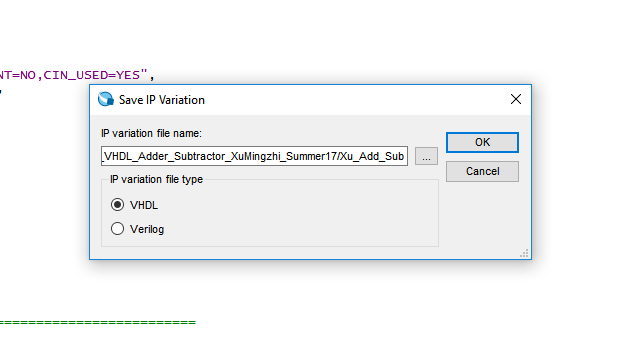
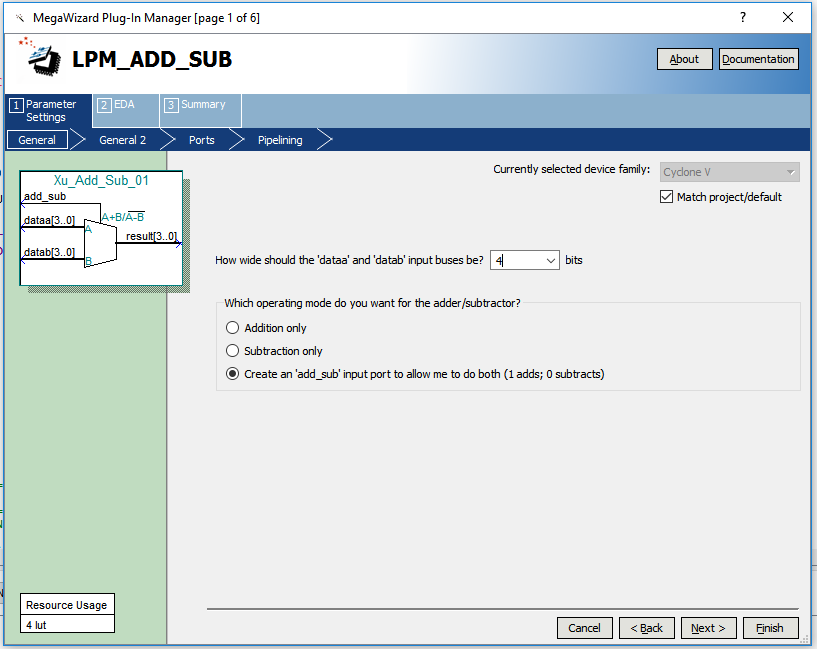
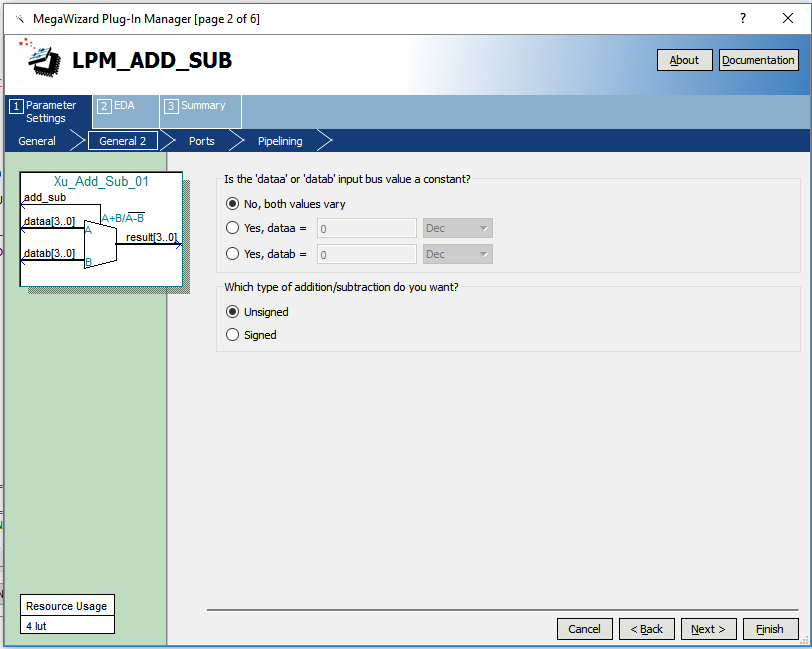
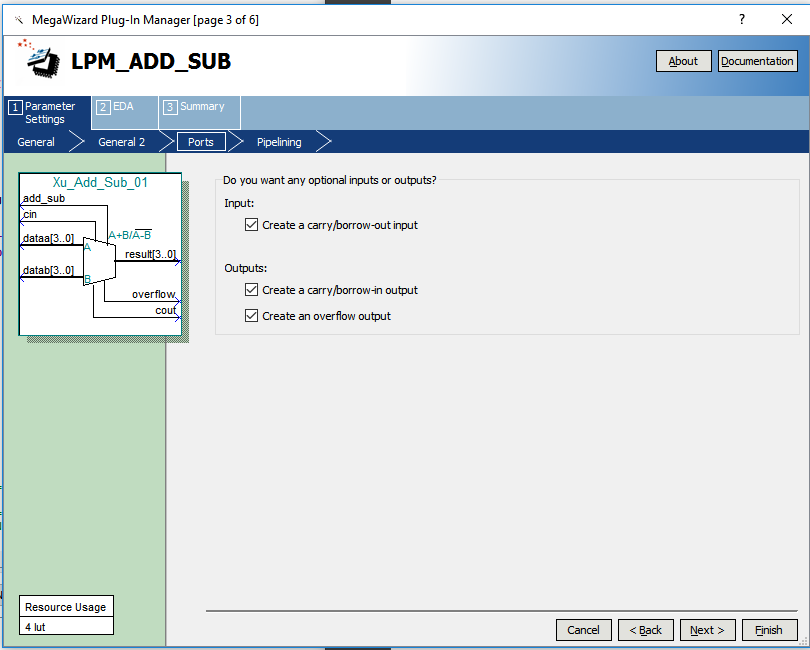
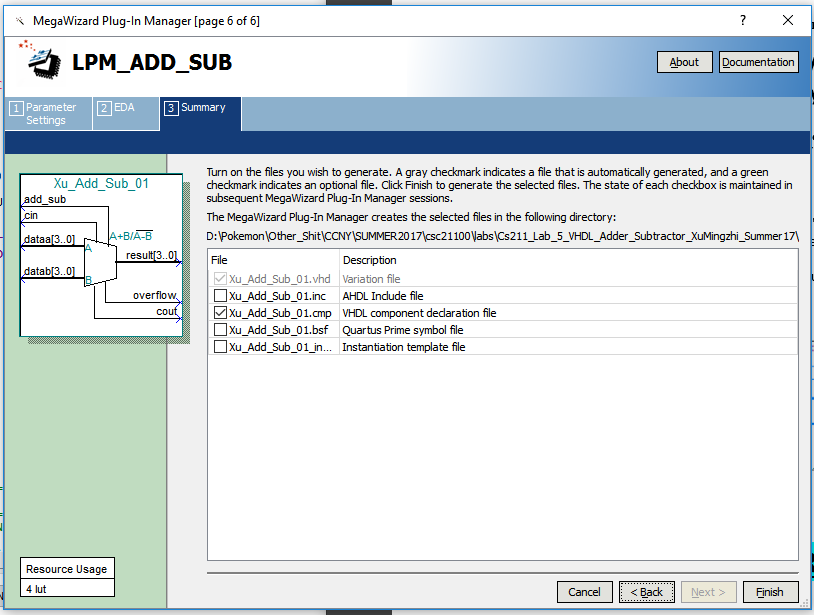
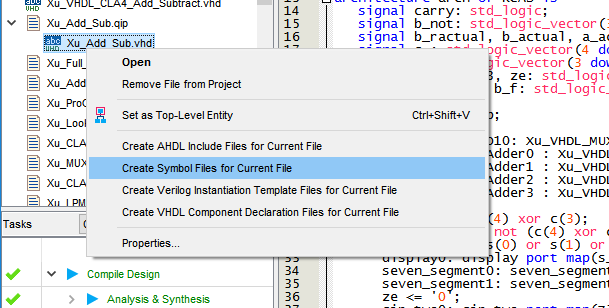
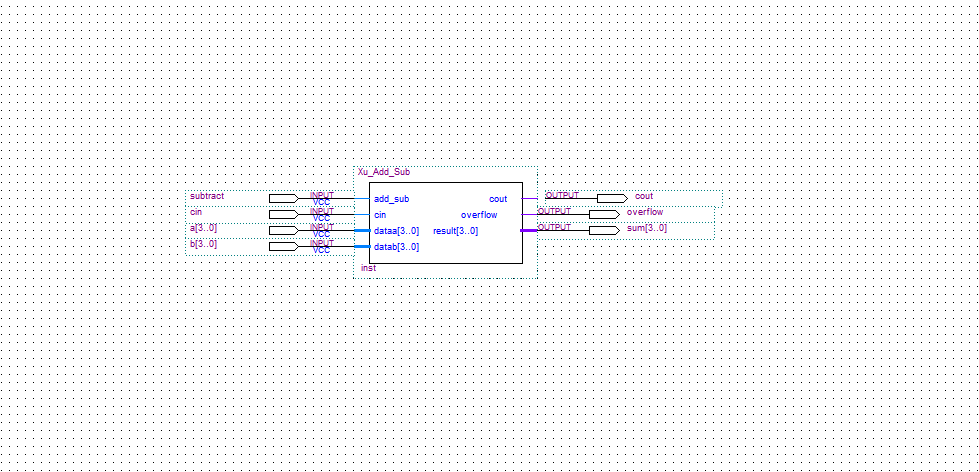
We want to also experiment with the performance of the Adder Subtractor from the Library of Parameterized Modules (LPM) available in Quartus Prime. To create an Adder Subtractor from the LPM:

1. We will need to go to tools section in Quartus Prime and select IP Catalog



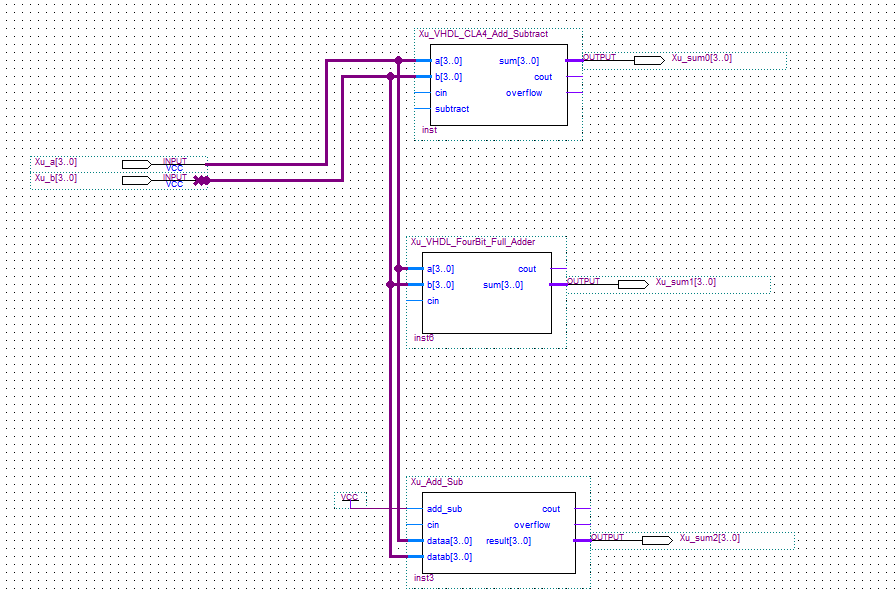
1. It will most likely appear on the right side of your Quartus Prime window and type in LPM and select LPM\_ADD\_SUB



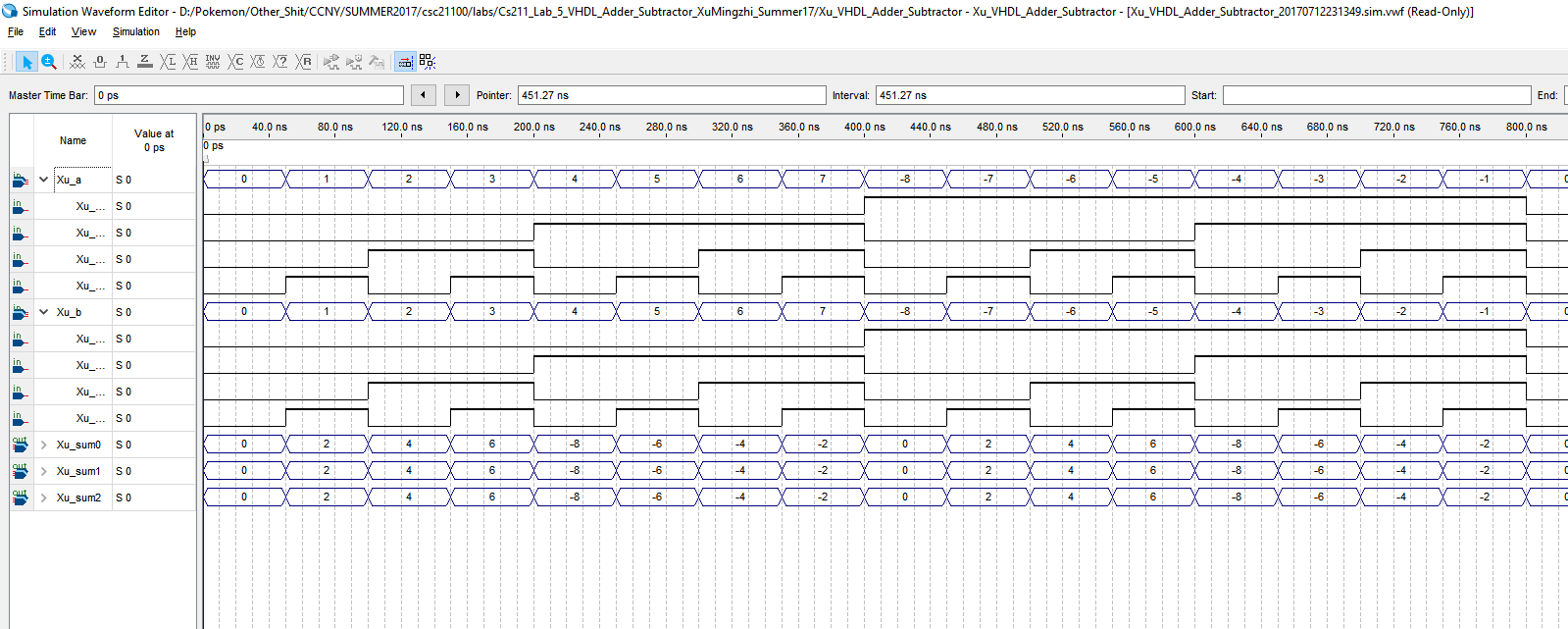
1. Save IP variation as VHDL file
2. Now the MegaWizard Plug-in Manage will appear and we want to select the one that does both adder and subtractor
3. We want it to set no for constant input and have it to do unsigned addition and subtraction
4. Then we will create a carry-in input, a carry-out output, and an overflow output
5. The file should be VHDL component declaration file
6. Then we will create a symbol for the LPM Adder Subtractor
7. Next, we will create a block diagram and use the symbol we have created

## ***4.2 Simulation***

In the simulation, we will compare output sum of the LPM Adder Subtractor with the Carry Lookahead Adder and the Ripple Adder to check if the Adders are functioning in the same and outputs the same results. Therefore, we will create a block diagram files that includes the symbols for the LPM Adder, Ripple Adder, and the Carry Lookahead Adder.



Here is a block diagram of two 4-bit inputs connecting to the three adders and each outputting a sum. Carry Lookahead Adder will have sum0, Ripple Carry Adder will have sum1, and LPM Adder will have sum3.



As we can observe from the simulation, all 3 adders will have the same output in signed decimal form.

# **Conclusion**

In this lab, we learned how to implement everything we have done in the previous labs in VHDL format. We also learned how to create a package to store our components so we don’t have to call it again in another VHDL file. We also created a 4-bit Ripple Carry Adder Subtractor, a 4-bit Lookahead Adder Subtractor and compare with the LPM Adder Subtractor that’s in the Quartus Prime to check if the result will be the same. Apparently the 4-bit Carry Lookahead Adder Subtractor will compute faster than the 4-bit Ripple Carry Adder Subtractor due to the Carry Lookahead component which computes the carry-in and the final carry-out with Boolean functions instead of in the Ripple Carry Adder Subtractor it is delay be the carry-outs from each Full Adder component going into the carry-in of the next Full Adder component.