Lab 5 VHDL Adder Subtractor

July 12, 2017

Cs211 Summer 2017

Xu Mingzhi

Contents

1. 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

a. 4-bit Ripple Carry Adder Subtractor
b. 4-bit Carry Look Ahead Adder Subtractor
c. 4-bit LPM Adder Subtractor

2. 4-bit Ripple Carry Adder Subtractor

2.1 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.

```
Library ieee;
use ieee.std_logic_1164.all;

entity Xu_VHDL_Full_Adder is
   port( a, b : in std_logic;
         cin : in std_logic;
         cout : out std_logic;
         sum : out std_logic );
end Xu_VHDL_Full_Adder;

architecture arch of Xu_VHDL_Full_Adder is
begin
   cout <= ( a and b ) or (( a xor b ) and cin );
   sum <= a xor b xor cin;
end arch;
```

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.

```
library ieee;
use ieee.std_logic_1164.all;

entity Xu_VHDL_FourBit_Full_Adder is
   port( a, b : in std_logic_vector(3 downto 0);
         cin : in std_logic;
         cout : out std_logic;
         sum : out std_logic_vector(3 downto 0));
end Xu_VHDL_FourBit_Full_Adder;

architecture arch of Xu_VHDL_FourBit_Full_Adder is

   signal c : std_logic_vector(4 downto 0);
   component Xu_VHDL_Full_Adder is
      port( a, b, cin : in std_logic;
            cout : out std_logic;
            sum : out std_logic);
      end component;
   begin
      c(0) <= cin;
      Xu_VHDL_Full_Adder0 : Xu_VHDL_Full_Adder port map(a(0), b(0),  c(0), c(1), sum(0));
      Xu_VHDL_Full_Adder1 : Xu_VHDL_Full_Adder port map(a(1), b(1),  c(1), c(2), sum(1));
      Xu_VHDL_Full_Adder2 : Xu_VHDL_Full_Adder port map(a(2), b(2),  c(2), c(3), sum(2));
      Xu_VHDL_Full_Adder3 : Xu_VHDL_Full_Adder port map(a(3), b(3),  c(3), c(4), sum(3));
      cout <= c(4);
end arch;
```

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.

```
library IEEE;
use IEEE.std_logic_1164.all;
use work.Xu_package.all;

entity Subtractor is
   port( a, b: in std_logic_vector(3 downto 0);
         cout: out std_logic;
         sum: out std_logic_vector(3 downto 0) );
end Subtractor;

architecture arch of Subtractor is
   signal b_not: std_logic_vector(3 downto 0);
   signal c : std_logic_vector(4 downto 0);
   signal s: std_logic_vector(3 downto 0);
   component Xu_VHDL_Full_Adder is
      port( a, b, cin : in std_logic;
            cout : out std_logic;
            sum : out std_logic);
      end component;
   begin
      b_not <= not b;
      c(0) <= '1';
      Xu_VHDL_Full_Adder0 : Xu_VHDL_Full_Adder port map(a(0), b_not(0),  c(0), c(1), s(0));
      Xu_VHDL_Full_Adder1 : Xu_VHDL_Full_Adder port map(a(1), b_not(1),  c(1), c(2), s(1));
      Xu_VHDL_Full_Adder2 : Xu_VHDL_Full_Adder port map(a(2), b_not(2),  c(2), c(3), s(2));
      Xu_VHDL_Full_Adder3 : Xu_VHDL_Full_Adder port map(a(3), b_not(3),  c(3), c(4), s(3));
      cout <= c(4);
      sum <= s;
end arch;
```

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.

```
library ieee;
use ieee.std_logic_1164.all;

entity seven_segment is
   port( x: in std_logic_vector(3 downto 0);
         HEX0: out std_logic_vector(6 downto 0) );
end seven_segment;

architecture arch of seven_segment is
   begin
      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));
      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));
      HEX0(2) <= (not x(3) and not x(2) and x(1) and not x(0));
      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));
      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));
      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));
      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));
end arch;
```

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.

```
library IEEE;
use IEEE.std_logic_1164.all;

entity sub_comparator is
   port( x: in std_logic_vector(3 downto 0);
         Z: out std_logic );
end sub_comparator;

architecture arch of sub_comparator is
   begin
      Z <= x(3);
end arch;
```

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.

```
library IEEE;
use IEEE.std_logic_1164.all;
use work.Xu_package.all;

entity sub_sum is
   port( s: in std_logic_vector(3 downto 0);
         s_a: out std_logic_vector(3 downto 0) );
end sub_sum;

architecture arch of sub_sum is
   signal s_not, b, c: std_logic_vector(3 downto 0);
   signal carry: std_logic;
   begin
      s_not <= not s;
      carry <= '1';
      b(3) <= '0';
      b(2) <= '0';
      b(1) <= '0';
      b(0) <= '0';
      Xu_VHDL_CLA40: Xu_VHDL_CLA4 port map(s_not, b, carry, c, open, open, open);
      s_a <= c;
end arch;
```

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.

```
library IEEE;
use IEEE.std_logic_1164.all;

entity cir_two is
   port( a, b: in std_logic;
         HEX1: out std_logic_vector(6 downto 0) );
end cir_two;

architecture arch of cir_two is
   begin
      HEX1(0) <= '1';
      HEX1(1) <= '1';
      HEX1(2) <= '1';
      HEX1(3) <= '1';
      HEX1(4) <= '1';
      HEX1(5) <= '1';
      HEX1(6) <= a xnor b;
end arch;
```

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.

```
library IEEE;
use IEEE.std_logic_1164.all;

entity Xu_VHDL_MUX2TO1 is
   port( a, b: in std_logic_vector(3 downto 0);
         sel: in std_logic;
         y: out std_logic_vector(3 downto 0) );
end Xu_VHDL_MUX2TO1;

architecture behavior of Xu_VHDL_MUX2TO1 is
   begin
      process( sel, a, b )
         begin
            if ( sel = '1' ) then
               y <= b;
            else
               y <= a;
            end if;
      end process;
   end behavior;
```

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.

```
entity RCAS is
    port( a, b: in std_logic_vector(3 downto 0);
          cin, subtract: in std_logic;
          sum: out std_logic_vector(3 downto 0);
          cout, overflow, negative, zero: out std_logic;
          HEX0, HEX1, HEX2, HEX3, HEX4, HEX5: out std_logic_vector(6 downto 0) );
end RCAS;

architecture arch of RCAS is
    signal carry: std_logic;
    signal b_not: std_logic_vector(3 downto 0);
    signal b_ractual, b_actual, a_actual, s_actual: std_logic_vector(3 downto 0);
    signal c : std_logic_vector(4 downto 0);
    signal s: std_logic_vector(3 downto 0);
    signal z1, z2, z3, ze: std_logic;
    signal s_f, a_f, b_f: std_logic_vector(3 downto 0);
    begin
        b_not <= not b;
        carry <= cin;
        Xu_VHDL_MUX2TO10: Xu_VHDL_MUX2TO1 port map(b, b_not, subtract, b_actual);
        Xu_VHDL_Full_Adder0 : Xu_VHDL_Full_Adder port map(a(0), b_actual(0),  carry, c(1), s(0));
        Xu_VHDL_Full_Adder1 : Xu_VHDL_Full_Adder port map(a(1), b_actual(1),  c(1), c(2), s(1));
        Xu_VHDL_Full_Adder2 : Xu_VHDL_Full_Adder port map(a(2), b_actual(2),  c(2), c(3), s(2));
        Xu_VHDL_Full_Adder3 : Xu_VHDL_Full_Adder port map(a(3), b_actual(3),  c(3), c(4), s(3));
        cout <= c(4);
        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) ) );
        display0: display port map(s_actual, ( not (c(4) xor c(3)) and s(3) ) or ( (c(4) xor c(3)) and c(4) ), c(4) xor c(3), HEX1, HEX0);
        seven_segment0: seven_segment port map(a_actual, HEX4);
        seven_segment1: seven_segment port map(b_ractual, HEX2);
        ze <= '0';
        cir_two0: cir_two port map(z3, ze, HEX5);
        cir_two1: cir_two port map(z2, ze, HEX3);
        sub_comparator0: sub_comparator port map(s, z1);
        sub_comparator1: sub_comparator port map(a, z3);
        sub_comparator2: sub_comparator port map(b, z2);
        sub_sum0: sub_sum port map(a, a_f);
        sub_sum1: sub_sum port map(b, b_f);
        sub_sum2: sub_sum port map(s, s_f);
        Xu_VHDL_MUX2TO11: Xu_VHDL_MUX2TO1 port map(a, a_f, z3, a_actual);
        Xu_VHDL_MUX2TO12: Xu_VHDL_MUX2TO1 port map(b, b_f, z2, b_ractual);
        Xu_VHDL_MUX2TO13: Xu_VHDL_MUX2TO1 port map(s, s_f, z1, s_actual);
        sum <= s_actual;
end arch;
```

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.

```
library IEEE;
use IEEE.std_logic_1164.all;
use work.xu_package.all;

entity display is
   port( x: in std_logic_vector(3 downto 0);
         neg, overflow: in std_logic;
         HEX1, HEX0: out std_logic_vector(6 downto 0) );
end display;

architecture arch of display is

   begin
      cir_two0: cir_two port map( neg, overflow, HEX1);
      seven_segment0: seven_segment port map( x, HEX0);
end arch;
```

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.

2.2 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.

2.3 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

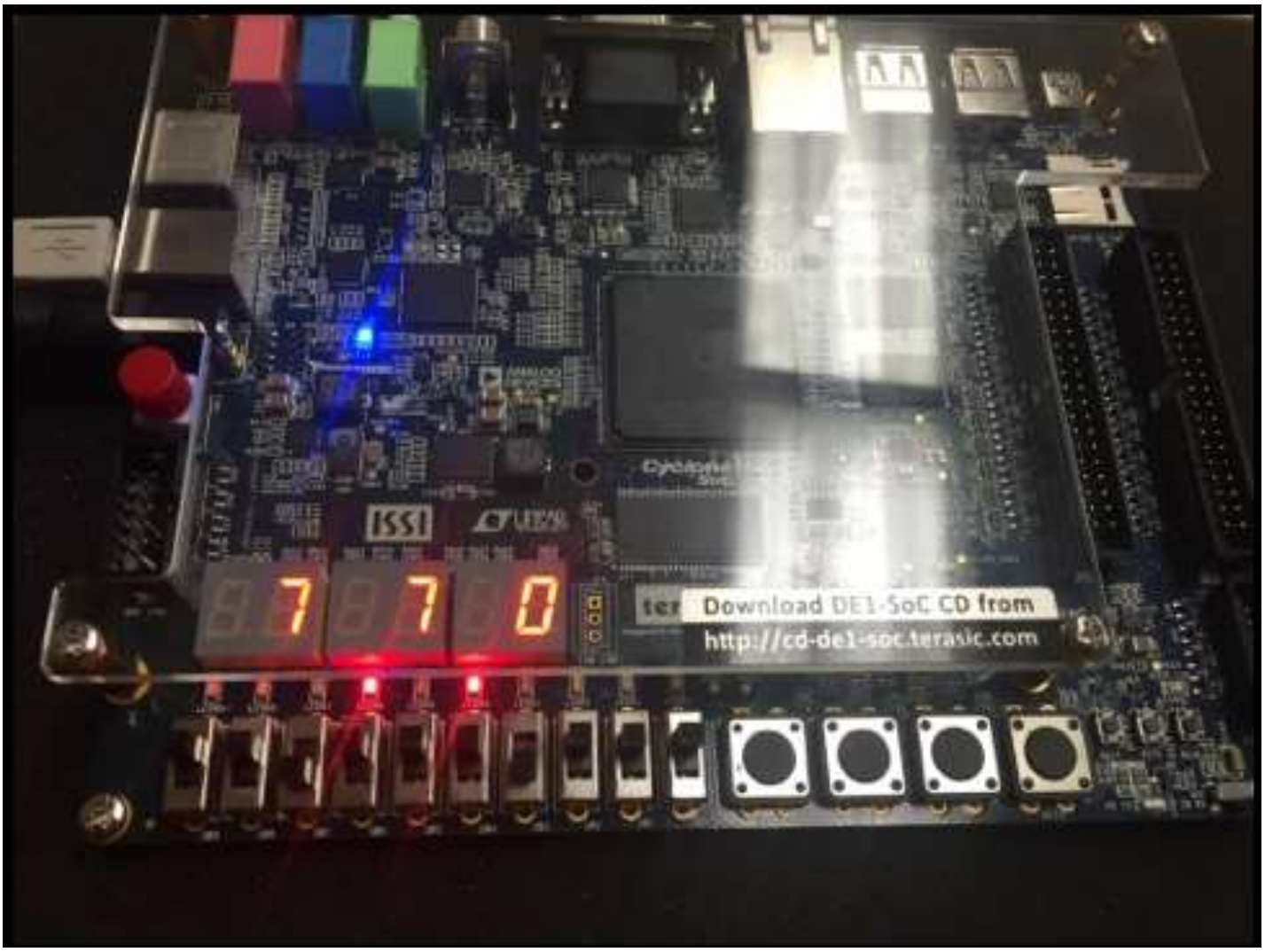


Figure 12: 7 – 7 = 0.

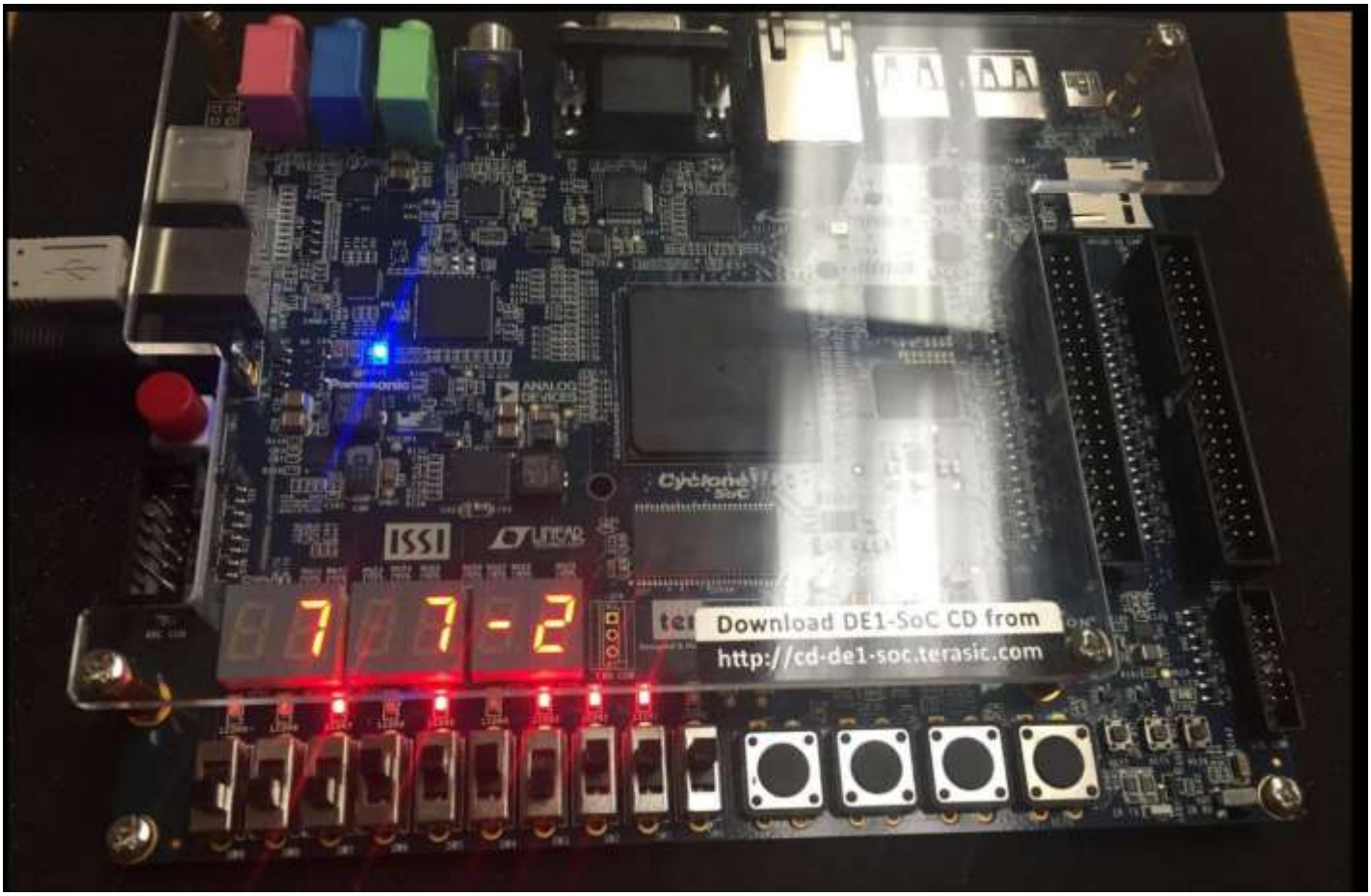


Figure 13: 7 + 7 = 14 so it is an overflow so it will become -2.

3. 4-bit Carry Lookahead Adder Subtractor

3.1 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.

```
library IEEE;
use IEEE.std_logic_1164.all;

entity Xu_VHDL_ProGen is
   port( Xu_x, Xu_y, Xu_cin : in std_logic;
         Xu_sum, Xu_g, Xu_p : out std_logic);
end Xu_VHDL_ProGen;

architecture arch of Xu_VHDL_ProGen is
   begin
      Xu_sum <= Xu_x xor Xu_y xor Xu_cin;
      Xu_p <= Xu_x xor Xu_y;
      Xu_g <= Xu_x and Xu_y;
end arch;
```

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.

```
library IEEE;
use IEEE.std_logic_1164.all;

entity Xu_Carry_LookAhead is
   port( G, P : in std_logic_vector(3 downto 0);
         CIN : in std_logic;
         C : out std_logic_vector(2 downto 0);
         CGOUT, CPOUT : out std_logic );
end Xu_Carry_LookAhead;

architecture arch of Xu_Carry_LookAhead is
   begin
      C(0) <= G(0) or ( P(0) and CIN );
      C(1) <= G(1) or ( G(0) and P(0) ) or ( CIN and P(0) and P(1) );
      C(2) <= G(2) or ( G(1) and P(2) ) or ( G(0) and P(1) and P(2) ) or ( CIN and P(0) and P(1) and P(2) );
      CGOUT <= G(3) or ( G(2) and P(3) ) or ( G(1) and P(2) and P(3) ) or ( G(0) and P(1) and P(2) and P(3) ) or ( CIN and P(0) and P(1) and P(2) and P(3) );
      CPOUT <= ( P(3) and P(2) and P(1) and P(0) );
end arch;
```

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.

```
library IEEE;
use IEEE.std_logic_1164.all;

entity Xu_VHDL_CLA4 is
   port( a, b : in std_logic_vector(3 downto 0);
         carryin : in std_logic;
         sum : out std_logic_vector(3 downto 0);
         cgout, cpout, overflow : out std_logic );
end Xu_VHDL_CLA4;

architecture arch of Xu_VHDL_CLA4 is
   component Xu_VHDL_ProGen
      port( Xu_x, Xu_y, Xu_cin : in std_logic;
            Xu_sum, Xu_g, Xu_p : out std_logic);
   end component;

   component Xu_Carry_LookAhead
      port( g, p : in std_logic_vector(3 downto 0);
            cin : in std_logic;
            c : out std_logic_vector(2 downto 0);
            cgout, cpout : out std_logic );
   end component;

   signal cg, cp, carry : std_logic_vector(3 downto 0);
   signal cout : std_logic;

   begin
   carry(0) <= carryin;
   Xu_VHDL_ProGen0: Xu_VHDL_ProGen port map( a(0), b(0), carry(0), sum(0), cg(0), cp(0) );
   Xu_VHDL_ProGen1: Xu_VHDL_ProGen port map( a(1), b(1), carry(1), sum(1), cg(1), cp(1) );
   Xu_VHDL_ProGen2: Xu_VHDL_ProGen port map( a(2), b(2), carry(2), sum(2), cg(2), cp(2) );
   Xu_VHDL_ProGen3: Xu_VHDL_ProGen port map( a(3), b(3), carry(3), sum(3), cg(3), cp(3) );
   Xu_Carry_LookAhead0: Xu_Carry_LookAhead port map( cg, cp, carryin, carry(3 downto 1), cout, cpout );
   cgout <= cout;
   overflow <= carry(3) xor cout;
end arch;
```

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.

```
entity CLA4AS is
    port( a, b: in std_logic_vector(3 downto 0);
          cin, subtract: in std_logic;
          sum: out std_logic_vector(3 downto 0);
          cout, overflow, negative, zero: out std_logic;
          HEX0, HEX1, HEX2, HEX3, HEX4, HEX5: out std_logic_vector(6 downto 0) );
end CLA4AS;

architecture arch of CLA4AS is
    signal b_not: std_logic_vector(3 downto 0);
    signal cg, cp, carry : std_logic_vector(3 downto 0);
    signal cgout : std_logic;
    signal b_ractual, b_actual, a_actual, s_actual: std_logic_vector(3 downto 0);
    signal s: std_logic_vector(3 downto 0);
    signal Z1, Z2, Z3, ze: std_logic;
    signal s_f, a_f, b_f: std_logic_vector(3 downto 0);
    begin
        b_not <= not b;
        carry(0) <= cin;
        Xu_VHDL_MUX2TO10: Xu_VHDL_MUX2TO1 port map(b, b_not, subtract, b_actual);
        Xu_VHDL_ProGen0: Xu_VHDL_ProGen port map( a(0), b_actual(0), carry(0), s(0), cg(0), cp(0) );
        Xu_VHDL_ProGen1: Xu_VHDL_ProGen port map( a(1), b_actual(1), carry(1), s(1), cg(1), cp(1) );
        Xu_VHDL_ProGen2: Xu_VHDL_ProGen port map( a(2), b_actual(2), carry(2), s(2), cg(2), cp(2) );
        Xu_VHDL_ProGen3: Xu_VHDL_ProGen port map( a(3), b_actual(3), carry(3), s(3), cg(3), cp(3) );
        Xu_Carry_LookAhead0: Xu_Carry_LookAhead port map( cg, cp, subtract, carry(3 downto 1), cgout, open );
        cout <= cgout;
        overflow <= cgout xor carry(3);
        negative <= ( not (cgout xor carry(3)) and s(3) ) or ( (cgout xor carry(3)) and cgout );
        zero <= not( s(0) or s(1) or s(2) or s(3) or ( (cgout xor carry(3)) and cgout ) );
        display0: display port map(s_actual, ( not (cgout xor carry(3)) and s(3) ) or ( (cgout xor carry(3)) and cgout ), cgout xor carry(3), HEX1, HEX0);
        seven_segment0: seven_segment port map(a_actual, HEX4);
        seven_segment1: seven_segment port map(b_ractual, HEX2);
        ze <= '0';
        cir_two0: cir_two port map(Z3, ze, HEX5);
        cir_two1: cir_two port map(Z2, ze, HEX3);
        sub_comparator0: sub_comparator port map(s, Z1);
        sub_comparator1: sub_comparator port map(a, Z3);
        sub_comparator2: sub_comparator port map(b, Z2);
        sub_sum0: sub_sum port map(a, a_f);
        sub_sum1: sub_sum port map(b, b_f);
        sub_sum2: sub_sum port map(s, s_f);
        Xu_VHDL_MUX2TO11: Xu_VHDL_MUX2TO1 port map(a, a_f, Z3, a_actual);
        Xu_VHDL_MUX2TO12: Xu_VHDL_MUX2TO1 port map(b, b_f, Z2, b_ractual);
        Xu_VHDL_MUX2TO13: Xu_VHDL_MUX2TO1 port map(s, s_f, Z1, s_actual);
        sum <= s_actual;
end arch;
```

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.

3.2 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.

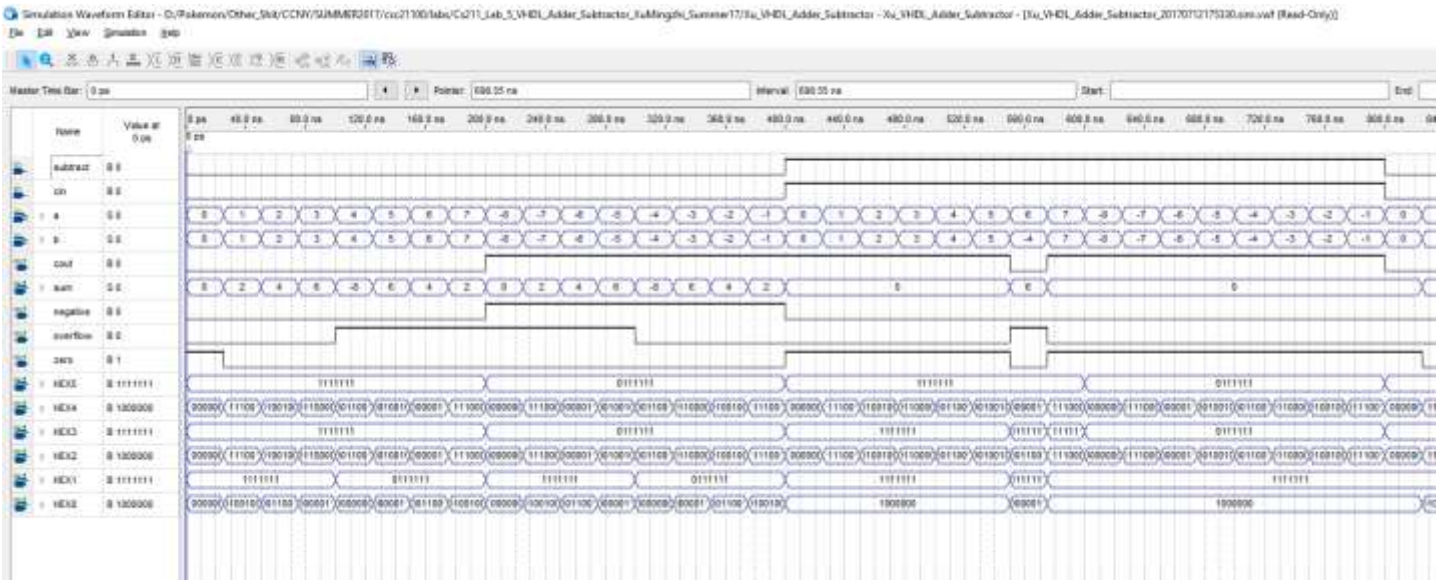


Figure 18: Vector waveform simulation for 4-bit Carry Lookahead Adder Subtractor.

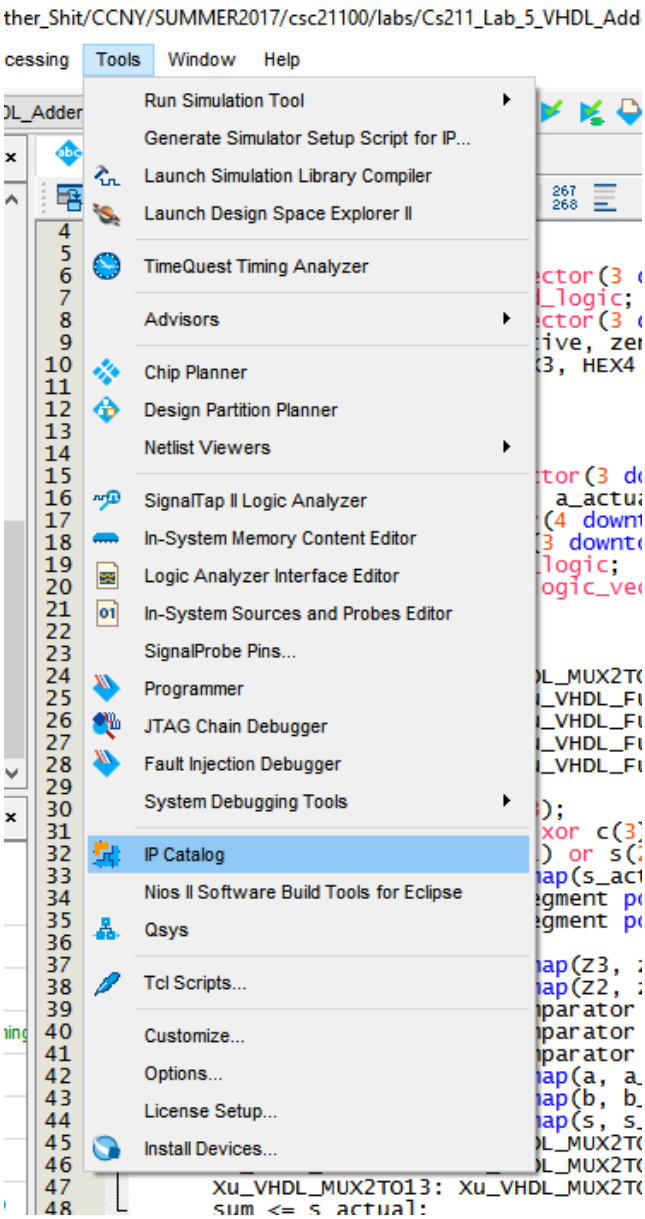
Since we have inputs a and b and the output sum in a group, we can have it to display signed decimal. We can observe from the simulation that when subtract is 0, it will be doing Adder between input a and b, and when subtract is 1, it will be doing Subtractor between input a and b, and when subtract 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. 4-bit LPM Adder Subtractor

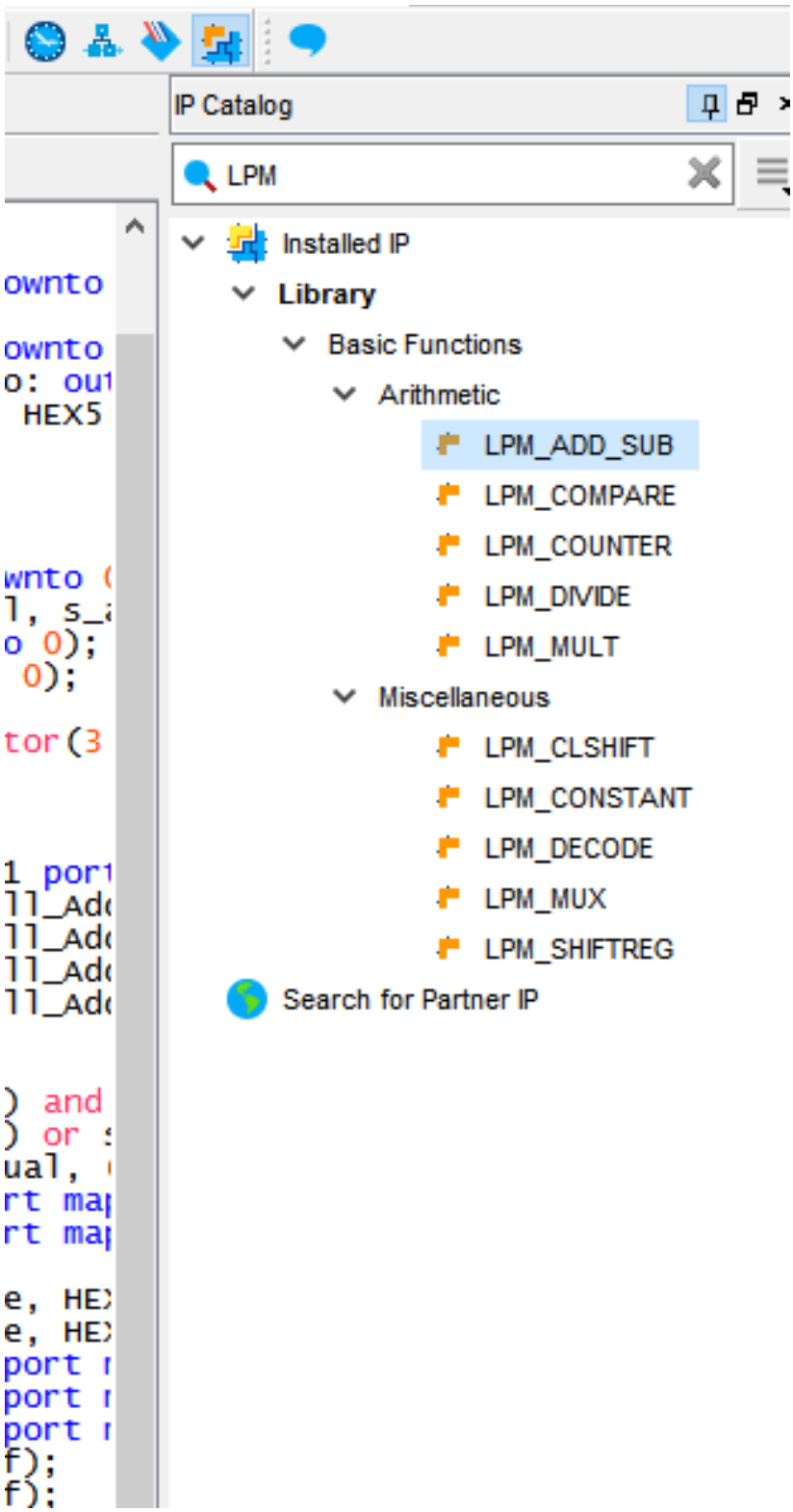
4.1 Functionality and Specification

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:

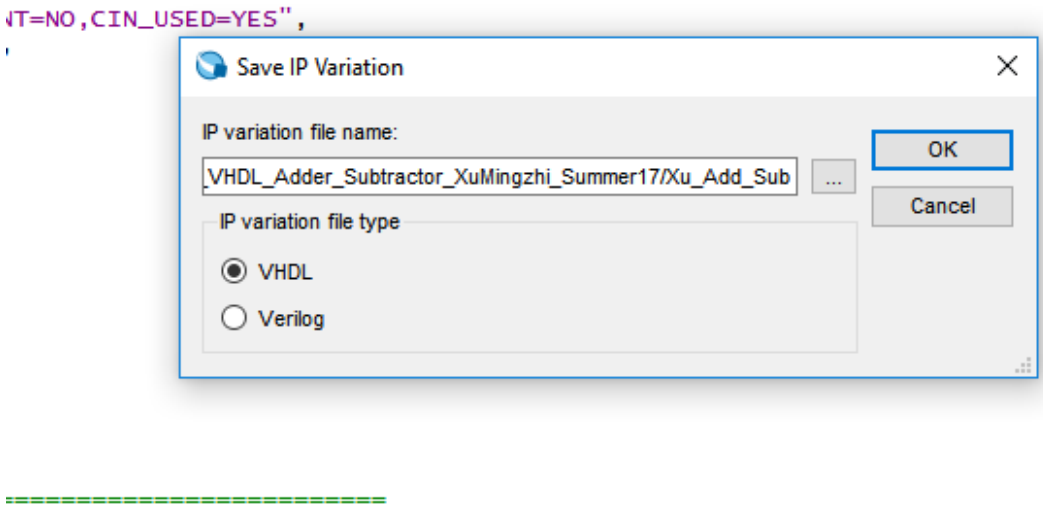
1. We will need to go to tools section in Quartus Prime and select IP Catalog



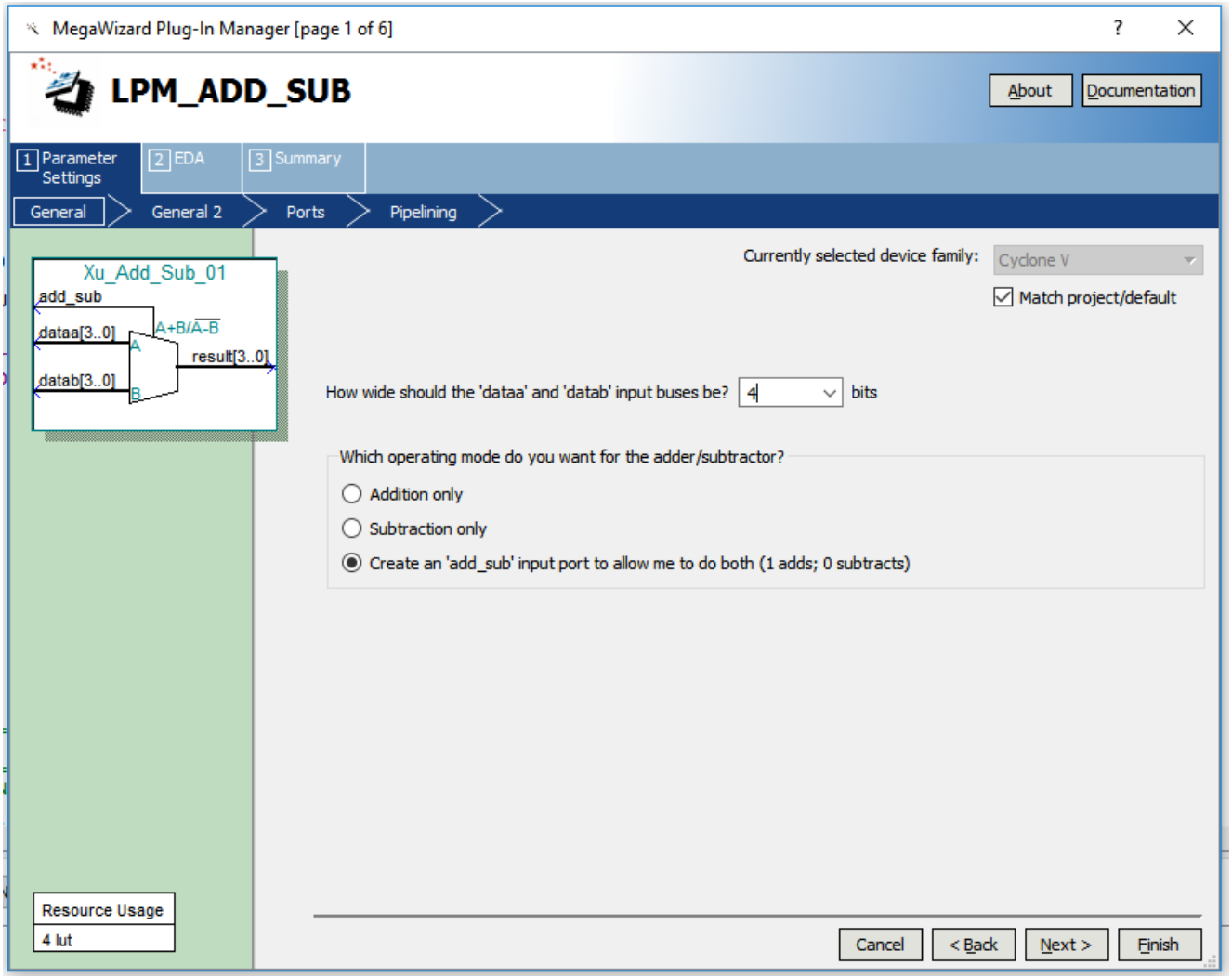
2. It will most likely appear on the right side of your Quartus Prime window and type in LPM and select LPM_ADD_SUB



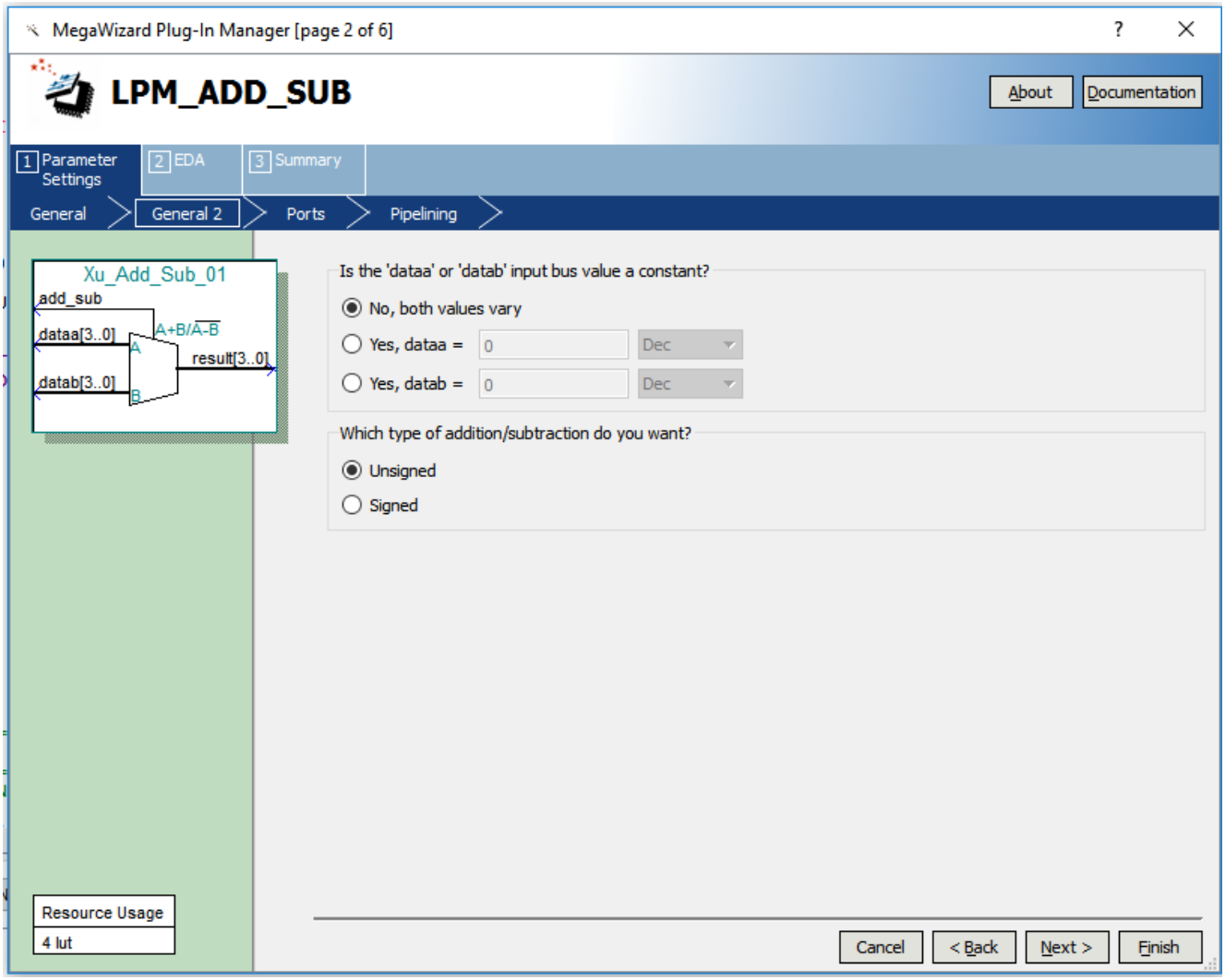
3. Save IP variation as VHDL file



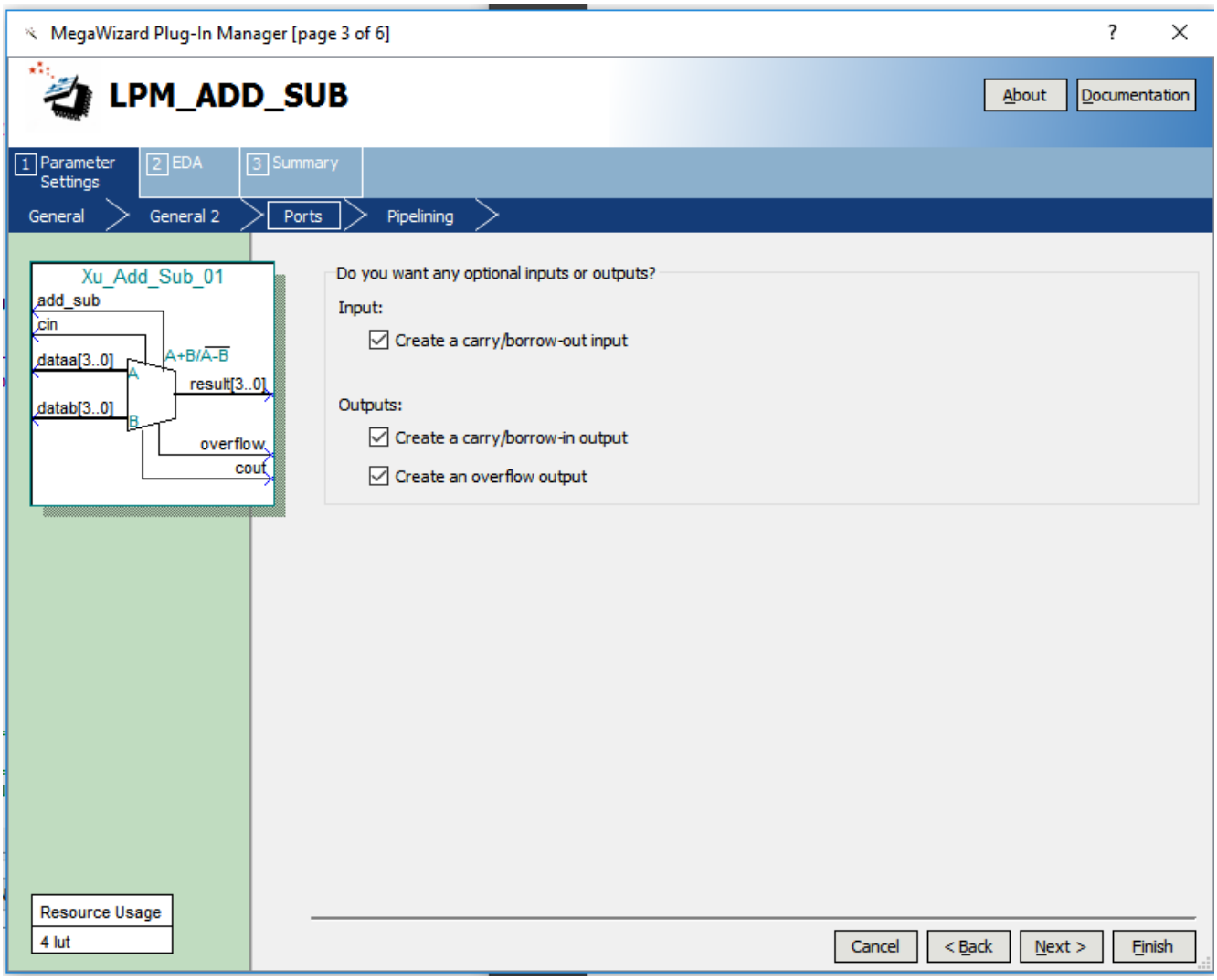
4. Now the MegaWizard Plug-in Manage will appear and we want to select the one that does both adder and subtractor



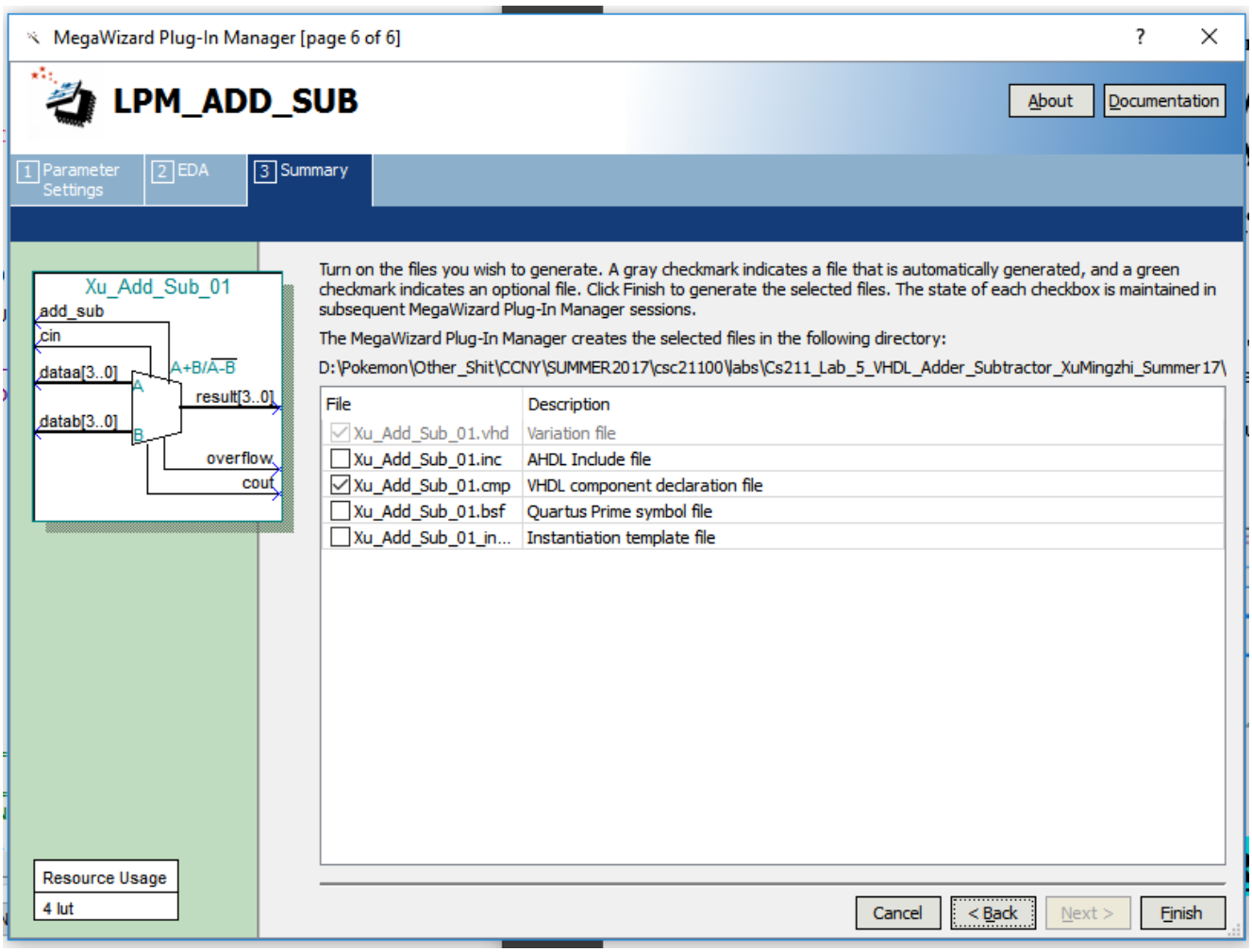
5. We want it to set no for constant input and have it to do unsigned addition and subtraction



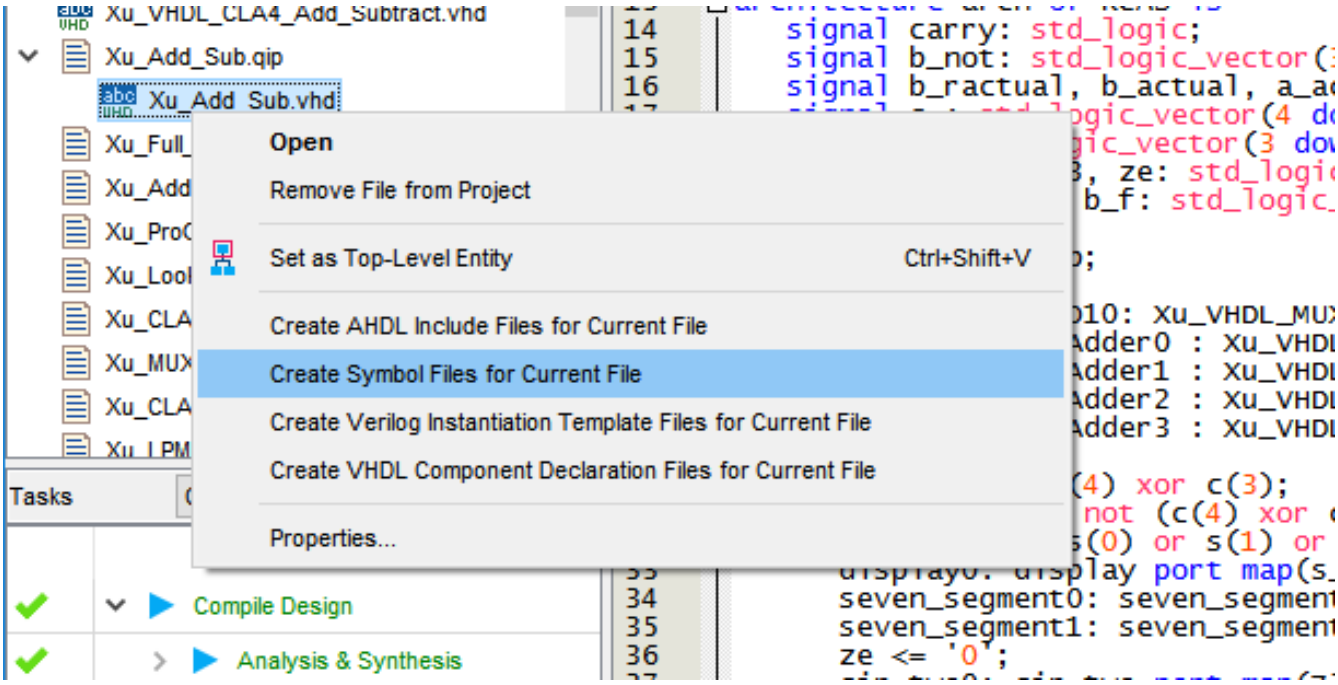
6. Then we will create a carry-in input, a carry-out output, and an overflow output



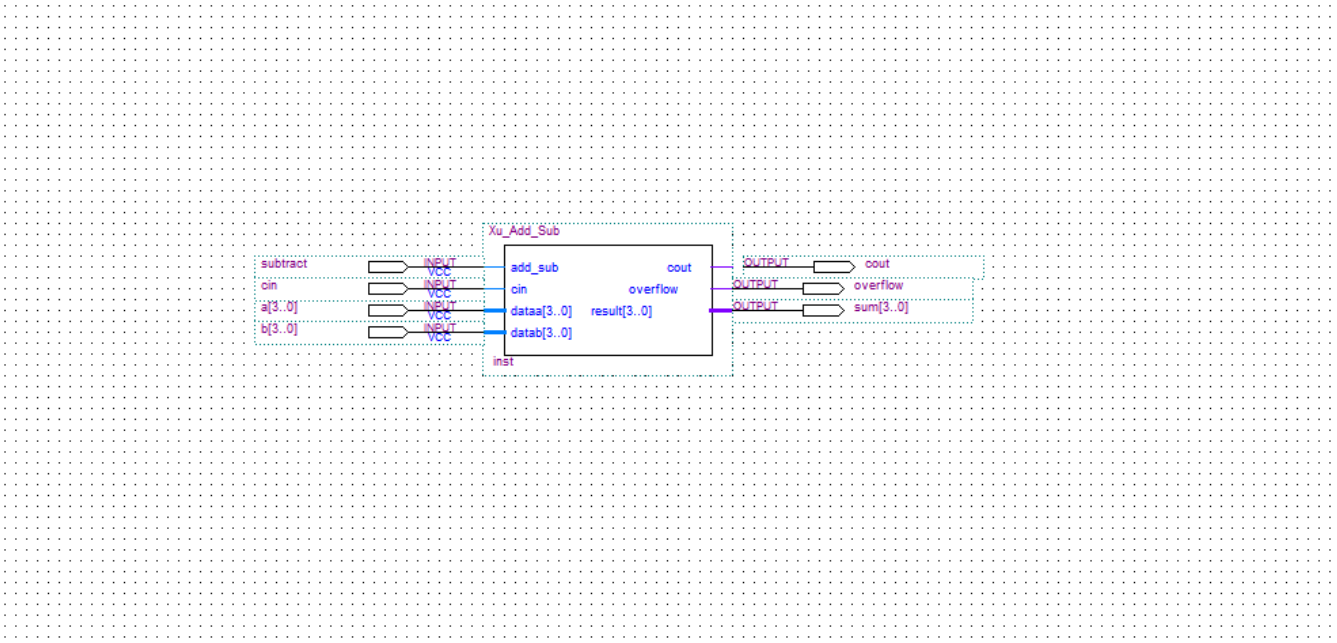
7. The file should be VHDL component declaration file



8. Then we will create a symbol for the LPM Adder Subtractor

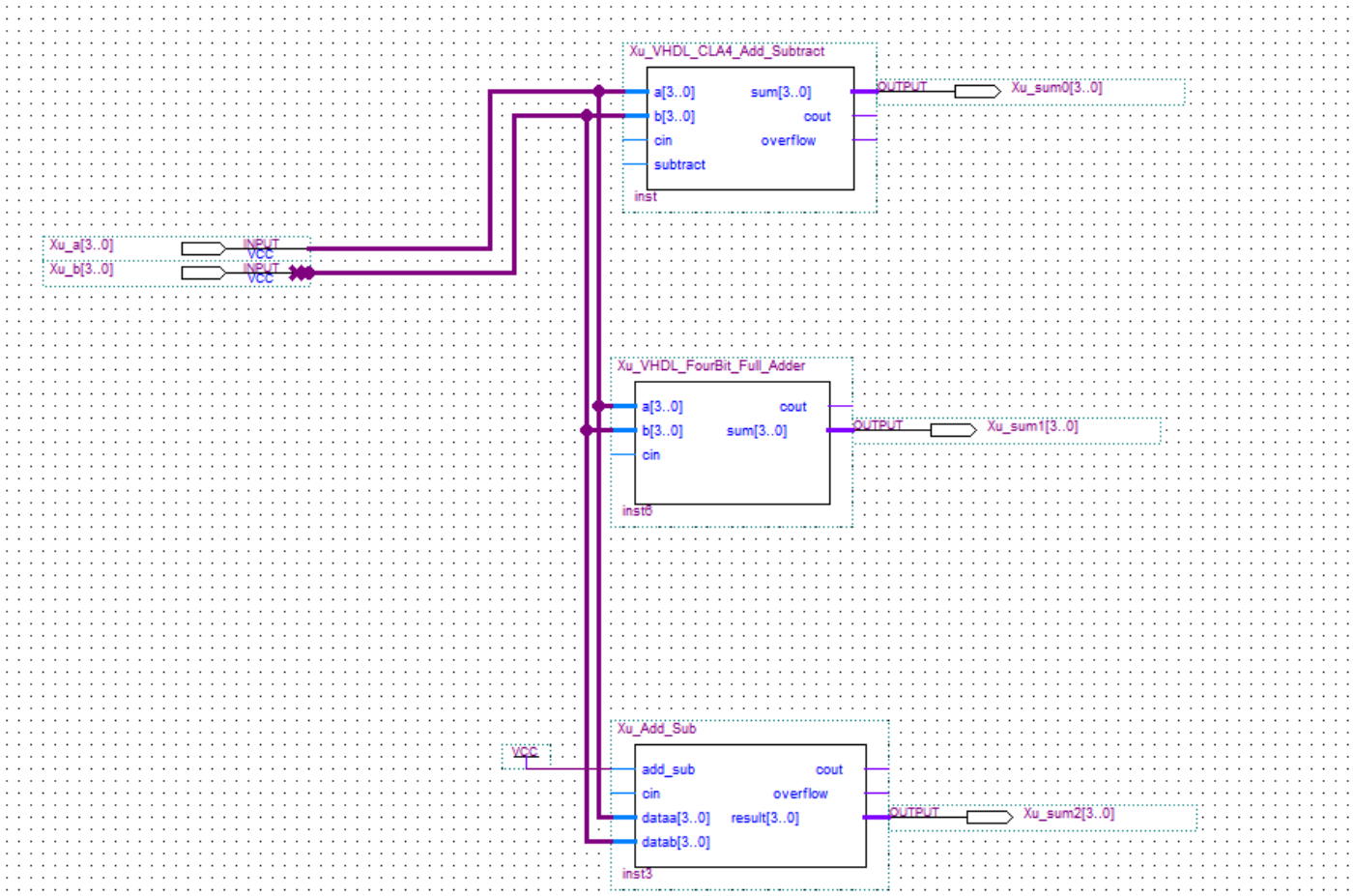


9. Next, we will create a block diagram and use the symbol we have created



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.

5. 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.