CS 211, Spring 2017

Laboratory Project: VHDL Adder / Subtractor with overflow detection, carry look ahead

Instructor: Professor Izidor Gertner
March 22, 2017 by 11:59 PM

Lab Project: VHDL Adder / Subtractor

Objective:

In this lab you will implement and compare performance of 4-bit adder/subtractors in Quartus using the following approaches:

- Ripple-carry (RCA)
- Carry look-ahead (CLA)
- LPM (Library of Parameterized Modules)

You will perform your design using hardware description language VHDL. I will explain the language in class. You could use the following link as a reference to VHDL.
http://www.seas.upenn.edu/~ese171/vhdl/vhdl_primer.html

First, we will practice by implementing the 4-bit versions of the adder. Then it is left for you to expand the code to implement the 8-bit versions of both adder and subtractor circuits.

1. RIPPLE CARRY ADDER

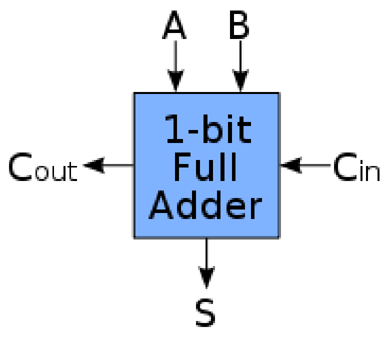
Background

In this section we will implement a 1-bit full bit adder in VHDL and then use it to make a 4-bit ripple carry adder. Below is the block diagram schematic of a full adder with input ports a, b and carry input C_{in}, an output port sum and carry output C_{out}.

CS 211, Spring 2017

Laboratory Project: VHDL Adder / Subtractor with overflow detection, carry look ahead

Instructor: Professor Izidor Gertner
March 22, 2017 by 11:59 PM



This system takes its inputs, performs the logical arithmetic "addition" operation and returns their sum and a carry out as the output. The truth table for this system can be seen below:

Input			Output	
A	B	Cin	S	Cout
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1

where A, B and Cin(Carry in) are the inputs and S (Sum) and Cout are the outputs. The full adder is described by the following equations:

$$S = (A \oplus B) \oplus C_{in}$$

$$C_{out} = (A \times B) + ((A \oplus B) \times C_{in})$$

Implementation

CS 211, Spring 2017

Laboratory Project: VHDL Adder / Subtractor with overflow detection, carry look ahead

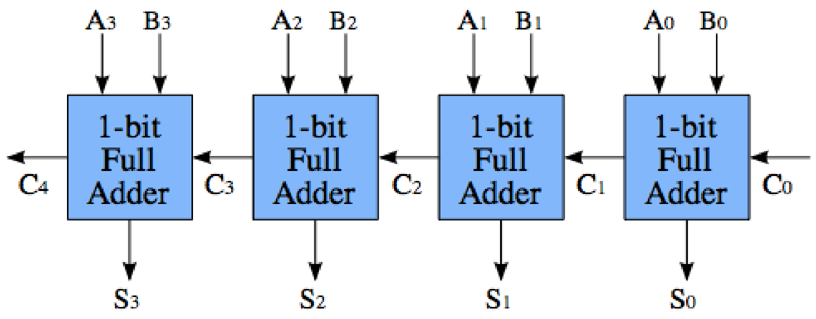
Instructor: Professor Izidor Gertner
March 22, 2017 by 11:59 PM

Here is the partial VHDL code for the full adder. The VHDL statement for calculating “sum” is pretty straight forward; it‟s just addition of the three input operands. The Boolean equation for “sum” can be written as: **sum = a** xor **b** xor **cin.** Please complete line 12 for *cout* assignment based on the equation given above for Cout.

FULL ADDER FILE: fa.vhd (partial code)

```
 1 Library ieee;
 2 use ieee.std_logic_1164.all;
 3
 4 entity FA is
 5     port(a, b : in std_logic;
 6          cin : in std_logic;
 7          cout : out std_logic;
 8          sum : out std_logic);
 9 end FA;
10
11 architecture arch of FA is
12 begin
13     cout <= ??? ; --complete this
14     sum <= a xor b xor cin;
15 end arch;
```

To create the 4-bit RCA, the full adder component will be instantiated four times and structurally connected as shown in the following schematic diagram:



CS 211, Spring 2017

Laboratory Project: VHDL Adder / Subtractor with overflow detection, carry look ahead

Instructor: Professor Izidor Gertner
March 22, 2017 by 11:59 PM

Since our implementation of 4-bit Full adder relies on the 1-bit Full Adder component, make sure you complete and compile your 1-bit Full Adder before implementing the 4-bit Full Adder. Next is the (partial) code for the 4-bit adder, complete where necessary:

RIPPLE CARRY ADDER FILE: rca.vhd (partial code)

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

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

architecture arch of RCA is

    signal c: std_logic_vector(4 downto 0); --"internal" carry tracker
    component FA is
        port(a, b, cin: ??? ; --complete at every “???”
             cout : ??? ;
             sum : ??? );
        end component;
    begin
        c(0) <= '0';
        FA0 : FA port map(a(0), b(0), c(0), c(1), sum(0));
        FA1 : FA port map( ??? );
        FA2 : FA port map( ??? );
        FA3 : FA port map( ??? );
        cout <= ???;
end arch;
```

This structural model of a 4-bit RCA instantiates four FA components by “*port mapping*” them. The first adder’s carry in is set to 0 as shown above. For the rest of the full adders, the carry input is the carry output of the previous full adder. Hence, the carries ripple up in this circuit, which gives it the name, ripple-carry adder. This 4-bit RCA has two input ports **a** and **b** each of 4-bit width. You will need to “*port map*” the remaining three full adder components to complete the design.

In short, the sum for the 4-bit RCA is contributed by four full adders and the carry is propagated from the first through fourth full adder to the output carry port. It is to be noted that, the addition is described as ***a xor b xor cin*** which means the carry generation in between the additions is taken into consideration at every step of addition operation.

2. CARRY LOOK-AHEAD ADDER

Background

A system of ripple-carry adders is a sequence of standard full adders that makes it possible to add numbers that contain more bits than that of a single full adder. Each full adder has a carryin (Cin) and a carryout (Cout) bit, and the adders are connected by connecting Cout on step k to Cin on step k+1.

The challenge with ripple-carry adders is the propagation delay of the carry bits. Assume that, in an instant the values of A and B change, such that

$A_1 = 0 \quad B_1 = 1$	$A_0 = 1 \quad B_0 = 1$

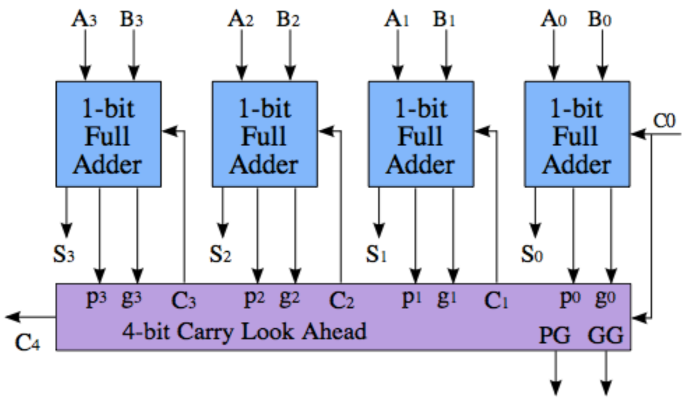
Since A_0 and B_0 are high, the first full adder will produce a carry, i. e. $C_1 = 1$. However, it takes some time for the logic to settle down, so C_1 doesn't change until a little after A_1 and B_1 changed. Thus, before C_1 shows up, the second full adder does not produce a carry, but as C_1 shows up, the second adder will recompute and produce a carry, i. e. $C_2 = 1$. In the worst case, C_4 is not correctly computed until 4*propagation delay, and C_n is not computed until n*propagation delay.

A carry look-ahead adder system solves this problem, by computing whether a carry will be generated before it actually computes the sum. There are multiple schemes of doing this, so there is no one exclusive implementation that constitutes a look-ahead adder. One schematic diagram of a carry look-ahead adder may look something like this:

CS 211,   Spring 2017

Laboratory Project: VHDL Adder / Subtractor with overflow detection, carry look ahead

Instructor: Professor   Izidor Gertner
March 22,   2017 by 11:59 PM



The important to note part of the picture, is that the purple block is producing three values: C_4, PG (Propagate) and GG (Generate). PG goes high if this block will propagate Cin to Cout, and GG goes high if the block will generate an overflow regardless of Cin. (Also, the block may neither propagate nor generate a carry, in which case both PG and GG are low, and Cout is 0.) PG and GG can be calculated in the purple block regardless of the value of C0 - thus, when C0 finally arrives, the purple block can simply consult its previously calculated result, and if the result is a "propagate," then C0 is propagated directly to C4; this is four times faster than propagating through all the four full adders.

The reason why the block has the outputs PG and GG is so that, in a hierarchal fashion, we can acquire even greater propagation speedups.

Implementation

From the full-adder implementation, we add two reference signal conditions that will allow us to determine the carry bit value of our circuit: *generate G* and *propagate P*. With these, we obtain a set of equations to implement our carry look-ahead circuit:

$$\begin{matrix} P_i = A_i \oplus B_i \\ G_i = A_i B_i \end{matrix} \quad \Longrightarrow \quad \begin{matrix} S_i = P_i \oplus C_i \\ C_{i+1} = G_i + P_i C_i \end{matrix}$$

Note: You will need these equations for P and G to complete your VHDL code.

CS 211, Spring 2017

Laboratory Project: VHDL Adder / Subtractor with overflow detection, carry look ahead

Instructor: Professor Izidor Gertner
March 22, 2017 by 11:59 PM

For the 4-bit carry look-ahead diagram provided, the logic for the generate (g) and propagate (p) values are given below. Note that the numeric value determines the signal from the circuit above, starting from 0 on the far left to 3 on the far right:

$$\begin{aligned} C_1 &= G_0 + P_0 \cdot C_0 \\ C_2 &= G_1 + P_1 \cdot C_1 \\ C_3 &= G_2 + P_2 \cdot C_2 \\ C_4 &= G_3 + P_3 \cdot C_3 \end{aligned}$$

Substituting C1 into C2, then C2 into C3, then 3 into C4 yields the expanded equations:

$$\begin{aligned} C_1 &= G_0 + P_0 \cdot C_0 \\ C_2 &= G_1 + G_0 \cdot P_1 + C_0 \cdot P_0 \cdot P_1 \\ C_3 &= G_2 + G_1 \cdot P_2 + G_0 \cdot P_1 \cdot P_2 + C_0 \cdot P_0 \cdot P_1 \cdot P_2 \\ C_4 &= G_3 + G_2 \cdot P_3 + G_1 \cdot P_2 \cdot P_3 + G_0 \cdot P_1 \cdot P_2 \cdot P_3 + C_0 \cdot P_0 \cdot P_1 \cdot P_2 \cdot P_3 \end{aligned}$$

To determine whether a bit pair will generate a carry, the following logic works:

$$G_i = A_i \cdot B_i$$

To determine whether a bit pair will propagate a carry, either of the following logic statements work:

$$\begin{aligned} P_i &= A_i \oplus B_i \\ P_i &= A_i + B_i \end{aligned}$$

Modifying the initial code of our full adder to include propagate P and generate G signals, the (partial) code will looks as follows:

FULL ADDER (WITH GENERATE AND PROPAGATE) FILE: fagp.vhd (partial code)

Laboratory Project: VHDL Adder / Subtractor with overflow detection, carry look ahead

Instructor: Professor   Izidor Gertner
March 22,   2017 by 11:59 PM

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

entity full_adder_g_p is
    port( a, b, cin: in std_logic;
    sum, g, p: out std_logic);
end full_adder_g_p;

architecture arch of full_adder_g_p is
    begin
        sum <= a xor b xor cin;
        p <= ???; --complete this
        g <= ???; --complete this
end arch;
```

Please complete lines 12 and 13 for p and g from the equations given at the beginning of this section.

Once we have a 1-bit full adder compiled and working with P and G signals, it is time to create a 4 bit carry look-ahead. We will implement it in two steps: first we will create code that holds the generate/propagate logic, and then we create a circuit that performs the carry.

CARRY LOOK-AHEAD LOGIC FILE: cla_logic.vhd (partial code)

CS 211, Spring 2017

Laboratory Project: VHDL Adder / Subtractor with overflow detection, carry look ahead

Instructor: Professor Izidor Gertner
March 22, 2017 by 11:59 PM

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

entity cla_logic is
    port(
        G, P: in std_logic_vector(3 downto 0);
        CIN: in std_logic;
        C: out std_logic_vector(2 downto 0); -- “internal” carry
        CGOUT, CPOUT: out std_logic);
end cla_logic;

architecture arch of cla_logic is

  begin
    C(0) <= G(0) or (P(0) and CIN);

    C(1) <= ???;

    C(2) <= ???;

    CGOUT<= ???; --can think of it as a C(3), it's the last carry

    CPOUT<= (P(3) and P(2) and P(1) and P(0));

end arch;
```

And here is the partial code that performs the carry, complete where necessary:

CARRY LOOK-AHEAD IMPLEMENTATION FILE: cla4.vhd (partial code)

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

entity cla4 is -- 4-bit CLA structural model: top entity
        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 cla4;

architecture arch of cla4 is

        component full_adder_g_p -- component declaration full adder
                port( a, b, cin: in std_logic;
```

```vhdl
                        sum, cg, cp: out std_logic);
          end component;

          component cla_logic -- component declaration CLA-generator
                  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); --local signals
          signal cout: std_logic;

          begin
          carry(0) <= carryin;
          ADD0:
            full_adder_g_p
                  port map (a(0), b(0), carry(0), sum(0), cg(0), cp(0));
          ADD1:
            full_adder_g_p
                  port map (???);
          ADD2:
            full_adder_g_p
                  port map (???);
          ADD3:
            full_adder_g_p
                  port map (???);

          --generate carries from
          --propagate and generate values
          --from full_adder_g_p_g_p
          CLA:
            cla_logic
                  port map(cg, cp, carryin, carry(3 downto 1), cout, cpout);

          cgout <= cout;
          overflow <= carry(3) xor cout;

end arch;
```

3. SUBTRACTOR

Once we have a working design that performs addition, we can also add some extra specification to allow it to perform subtraction. The way we will do it is by the 2’s complement technique.

Let’s do an example. Calculate the following: 0111 – 0001:

i.	First, we need to convert 0001 to its negative equivalent in 2's complement.	$\begin{array}{rr} 0111 & (7) \\ \underline{-\ 0001} & \underline{-\ (1)} \end{array}$
ii.	To do this we change all the 1's to 0's and 0's to 1's and add one to the number. Notice that the most-significant digit is now 1 since the number is negative.	$\begin{array}{rr} 0001\ \text{->} & 1110 \\ & \underline{1} \\ & 1111 \end{array}$
iii.	Next, we add the negative value we computed to 0111. This gives us a result of 10110.	$\begin{array}{rr} 0111 & (7) \\ \underline{+\ 1111} & \underline{+(-1)} \\ 10110 & (?) \end{array}$
iv.	Notice that our addition caused an overflow bit. Whenever we have an overflow bit in 2's complement, we discard the extra bit. This gives us a final answer of 01102 (or 6_{10}).	$\begin{array}{rr} 0111 & (7) \\ \underline{-\ 0001} & \underline{-\ (1)} \\ 0110 & (6) \end{array}$

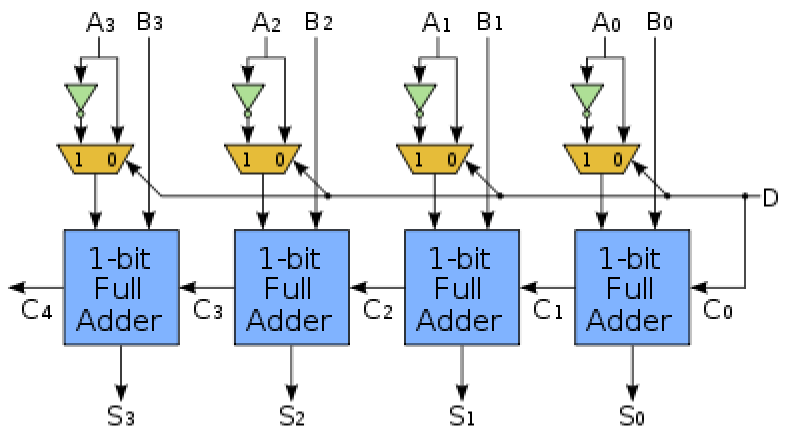
We now have to add some logic to let our circuit know whether we want to perform addition or subtraction.

In this section we will implement the subtraction feature to the carry look-ahead method. Ripple carry subtraction is left for you as an exercise; you can use the next figure for reference.

CS 211, Spring 2017

Laboratory Project: VHDL Adder / Subtractor with overflow detection, carry look ahead

Instructor: Professor Izidor Gertner
March 22, 2017 by 11:59 PM



Basically, one control signal makes a subtractor from the adder by inverting the second operand and adding one to the least significant 1-bit adder. Usually this is done by feeding 1 as a carry in.

Note that from the figure instead of feeding input A straight into the addition, we have an input signal D that determines the operation to perform. Depending on the value of D, either input A or A' will be fed into the adder. It is up to you whether you choose input A or input B as the subtrahend. When you choose to do subtraction you should also feed C_{in}=1.

Let's now focus on the carry look-ahead code. We need to do two things with our code:

1. We need to add a multiplexer that feeds either the value of one of the inputs if we do addition or its compliment if we choose to do subtraction.
2. We need to expand on the implementation of the cla4.vhd file to include an extra input for operation select and add/subtract feature.

Here is the complete code for the multiplexer that takes an input vector and a select.

MULTIPLEXER CHOOSES B OR B' DEPENDING ON: mux2to1.vhd

Laboratory Project: VHDL Adder / Subtractor with overflow detection, carry look ahead

Instructor: Professor Izidor Gertner

March 22, 2017 by 11:59 PM

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

entity 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 mux2to1;

architecture behavior of mux2to1 is
begin
    process (sel, a, b)
        begin
            if (sel = '1') then
                y <= b;
            else
                y <= a;
            end if;
    end process;
end behavior;
```

Here is the complete code for the carry look-ahead adder/subtractor:

CARRY LOOK-AHEAD ADD/SUBTRACT FILE: cla4_add_subtract.vhd

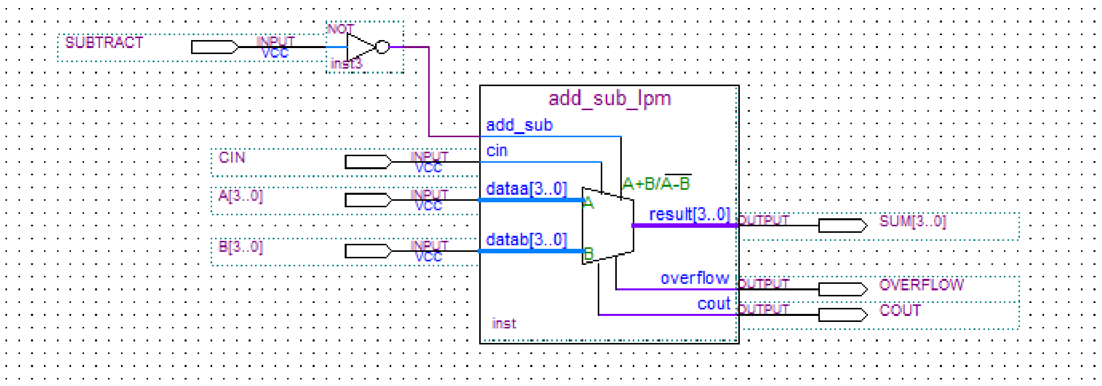
```vhdl
library IEEE;
use IEEE.std_logic_1164.all;

entity cla4_add_subtract is
        port(a, b: in std_logic_vector(3 downto 0);
                cin: in std_logic;
                subtract: in std_logic;
                sum: out std_logic_vector(3 downto 0);
                cout: out std_logic;
                overflow: out std_logic);
end cla4_add_subtract;

architecture cla4_add_subtract_arch of cla4_add_subtract is
        component mux2to1
        port(
                a, b: in std_logic_vector(3 downto 0);
                sel: in std_logic;
                y: out std_logic_vector(3 downto 0));
        end component;

        component cla4
        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 component;

        signal carry: std_logic;
        signal b_not: std_logic_vector(3 downto 0);
        signal b_actual: std_logic_vector(3 downto 0);

        begin
          b_not <= not b;
          carry <= cin;
          MUX_SUB:
            mux2to1
              port map (b, b_not, subtract, b_actual);
          ADD0:
            cla4
              port map (a, b_actual, carry, sum, cout, open, overflow);
end cla4_add_subtract_arch;
```

4. LPM ADDER / SUBTRACTOR

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

1. Open the MegaWizard Plug-In Manager (Tools > MegaWizard Plug-In Manager).
2. Make sure "Create new…" is selected and hit next.
3. Under "Arithmetic" find and select the "LPM_ADD_SUB" megafunction.
4. Give your module a name.
5. Make sure VHDL is the selected type of output file. Hit next.
6. You should see a window where you can choose the input bus size. In this case, the bus size is 4.
7. In the same window, we can choose to add the "add_sub" input. Hit next.

8. Then, select "both values vary" and "unsigned". Hit next.
9. Check all of the boxes to get all of the available carry/overflow inputs/outputs.
10. Now hit next or finish until you are done.
11. Once the module has been created, it can be turned into a symbol, as shown:



5. YOUR TASK

1. Circuits to implement:

We just showed you how to create 4-bit adders and explained how to expand in order to perform subtraction as well. For this lab you have to create the following circuits:

- 8-bit Ripple Carry Adder/Subtractor
- 8-bit Carry Look-Ahead Adder/Subtractor
- 8-bit LPM Adder/Subtractor

1A DISPLAY

Display results of all circuits in BINARY WITH LEDS and in addition using 7 SEGMENT DISPLAY. You should be able to display negative numbers on display. A special led should indicate if there is an overflow.

CS 211, Spring 2017

Laboratory Project: VHDL Adder / Subtractor with overflow detection, carry look ahead

Instructor: Professor Izidor Gertner
March 22, 2017 by 11:59 PM

1.1 Challenge problems REQUIRED!

- Design overflow detection circuit.
- Integrate the BITWISE OPERATIONS block with Add/SUB unit.

2. Functional and Timing Simulations

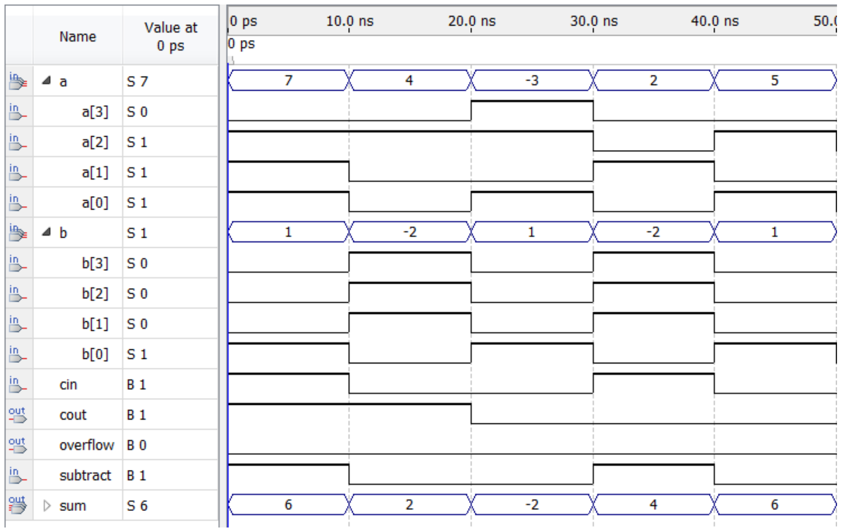
- To ensure that our look-ahead and ripple carry adder/subtractors work correctly, we compare their behavior against that of the adder imported from the Library of Parameterized Modules.
- For this lab you are allowed to use the Waveform Simulation tool within Quartus. The reason for this is that you will need to perform functional AND timing simulations. Generally ModelSim will require a special SDF file for VHDL to do timing simulations. The generation of this file falls outside of the requirements for this lab.
- You should run **functional waveform simulations** separately on each of the implementations and check that you obtain identical results for your sum, carry out, etc. Take screenshots with several test inputs.
- Remember to feed a 1 to carry in when performing subtraction. Subtract=0 performs addition, Subtract=1 does subtraction.
- Perform **timing waveform simulations** to compare which implementation of the adder performs faster. Take screenshots of your results.
- Do waveform simulations for 4 bit and 8 bit circuits.

Here is an example of a functional waveform simulation with the look-ahead adder/subtractor and some test input values:

CS 211, Spring 2017

Laboratory Project: VHDL Adder / Subtractor with overflow detection, carry look ahead

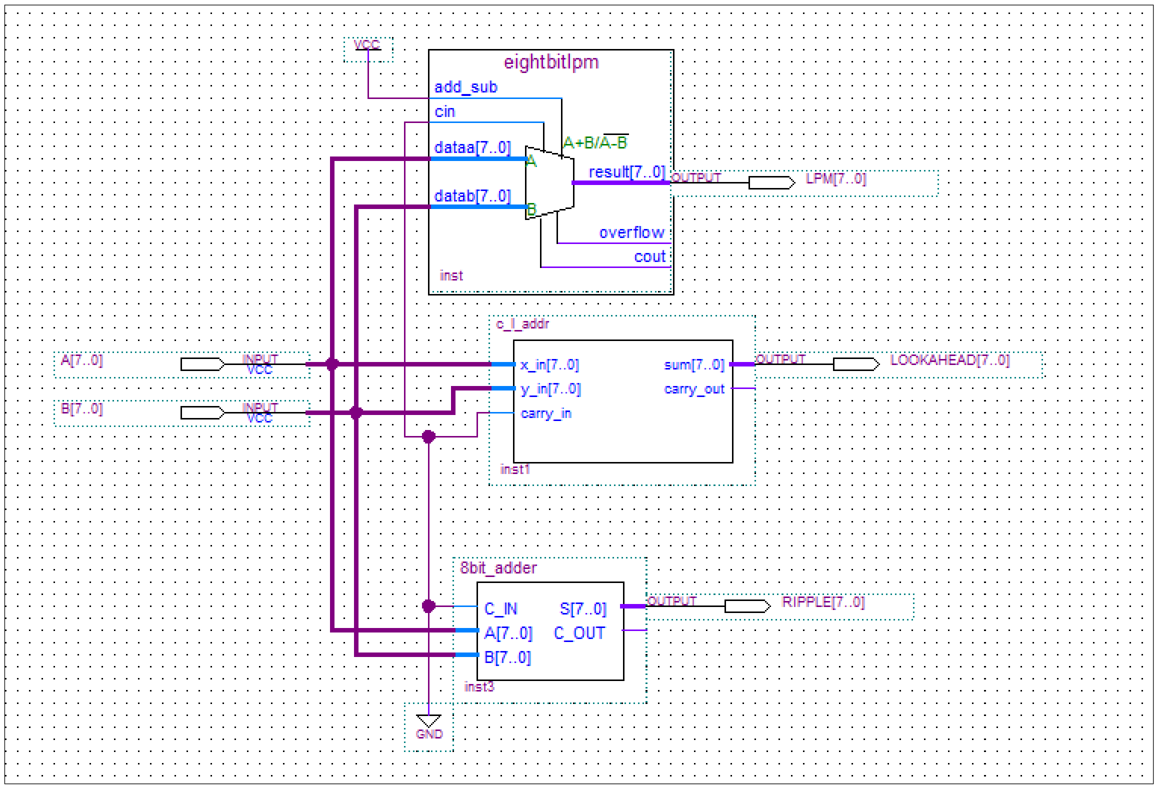
Instructor: Professor Izidor Gertner
March 22, 2017 by 11:59 PM



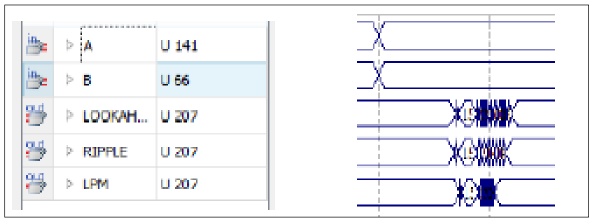
To perform functional comparison, you may put all your circuits in one block diagram file and then run a timing waveform simulation sharing the inputs. Use this image for reference:

Laboratory Project: VHDL Adder / Subtractor with overflow detection, carry look ahead

Instructor: Professor Izidor Gertner
March 22, 2017 by 11:59 PM



The following is a screenshot of a timing waveform simulation that compares the three implementations; you should run several instances with different values to evaluate for speed consistency:



A	U 141
B	U 66
LOOKAH...	U 207
RIPPLE	U 207
LPM	U 207

The following steps show how to create a Waveform file in Quartus. If you are familiar with this, you can safely ignore these steps and skip to the pin-assignments section next.

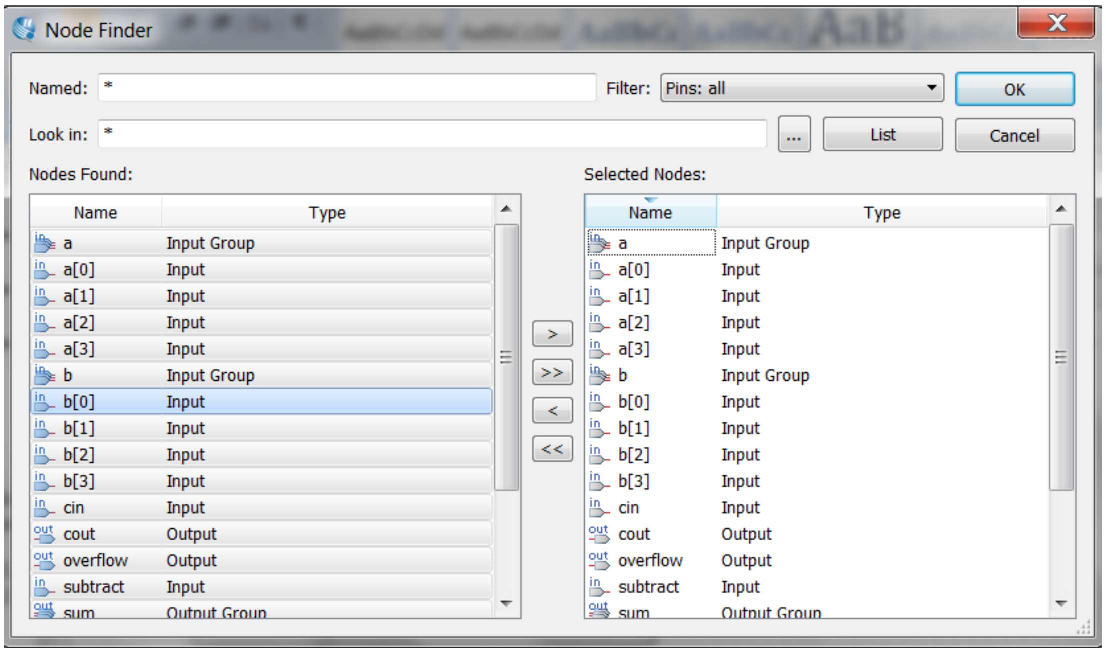
CS 211, Spring 2017

Laboratory Project: VHDL Adder / Subtractor with overflow detection, carry look ahead

Instructor: Professor Izidor Gertner
March 22, 2017 by 11:59 PM

In Quartus you can perform **Timing** and **Functional** Simulations. To create a waveform, perform the following steps:

1. Compile the file you want to simulate.
2. Now go to File > New > University Program VWF.
3. Go to Edit > Insert > Insert Node or Bus…
4. Click the Node Finder button
5. Click the List Button, this will include all the inputs and outputs available in your design.
6. Click the “>>” button, this will copy the available pins to include them in your waveform, then hit OK, then OK again.



Named: * Filter: Pins: all

Look in: *

Nodes Found:

Name	Type
a	Input Group
a[0]	Input
a[1]	Input
a[2]	Input
a[3]	Input
b	Input Group
b[0]	Input
b[1]	Input
b[2]	Input
b[3]	Input
cin	Input
cout	Output
overflow	Output
subtract	Input
sum	Output Group

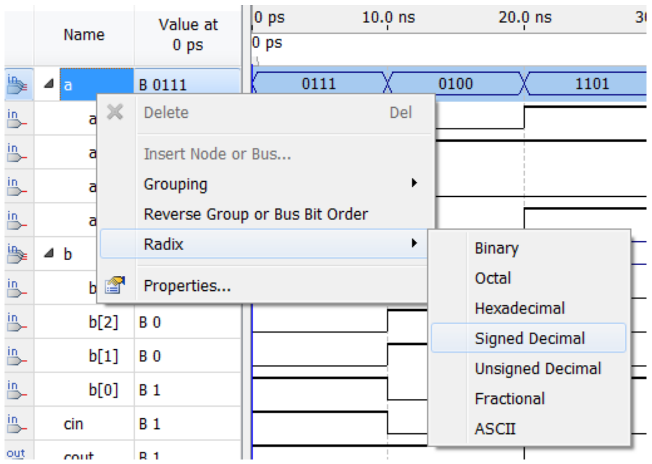
Selected Nodes:

Name	Type
a	Input Group
a[0]	Input
a[1]	Input
a[2]	Input
a[3]	Input
b	Input Group
b[0]	Input
b[1]	Input
b[2]	Input
b[3]	Input
cin	Input
cout	Output
overflow	Output
subtract	Input
sum	Output Group

7. Now you can draw the waveform by selecting a region with the mouse and clicking the 0 or 1 buttons in the menu bar at the top to give the desired value.
8. Once you finished drawing your desired waveform, go to Simulation > Run Functional Simulation OR Run Timing Simulation.
9. To toggle the view of your inputs and outputs between binary or decimal, right click on the input, select Radix and then choose the radix you want to display, either binary or signed decimal.

Laboratory Project: VHDL Adder / Subtractor with overflow detection, carry look ahead

Instructor: Professor   Izidor Gertner
March 22,   2017 by 11:59 PM



---

3. Pin-Assignments and DE2 board testing

Please pay attention you are usinga different board with smaller number of pins.

After testing your circuit's behavior using waveforms, it is now time to run your design in the DE2 board. Have your VHDL code in Quartus and compile. Then create a new text file to add your pin assignments:

For your inputs:
a[48 bits]: Assign to switches 0 to 3
b[48 bits]: Assign to switches 4 to 8
carry-in: Assign to switch 16
subtract: Assign to switch 17

For your outputs:
sum[4 bits]: Assign to LEDR 0 to 7 and seven segment display
carry-out: Assign to LEDR16
overflow: Assign to LEDR17

Click here for pin assignment manual

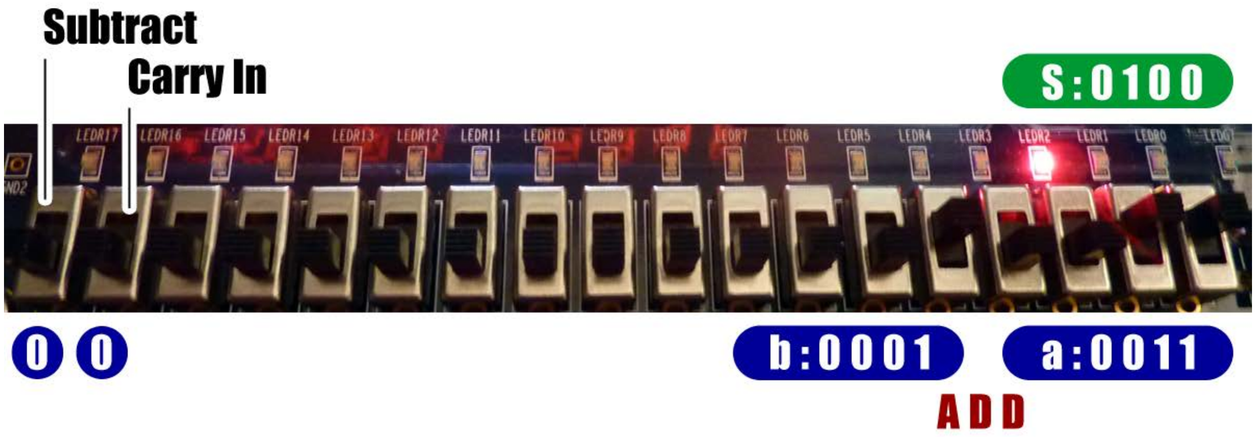
CS 211, Spring 2017

Laboratory Project: VHDL Adder / Subtractor with overflow detection, carry look ahead

Instructor: Professor Izidor Gertner
March 22, 2017 by 11:59 PM

Below is an example of the behavior in the DE2 board for a 4-bit adder/subtractor. When you test your 8-bit adder/subtractors, make sure you take screenshots to include in your report.

In the first image we are performing addition, noted by 0 value in subtract, where a=0011 and b=0001, this results in binary addition s=0100 and no overflow, no carryout.

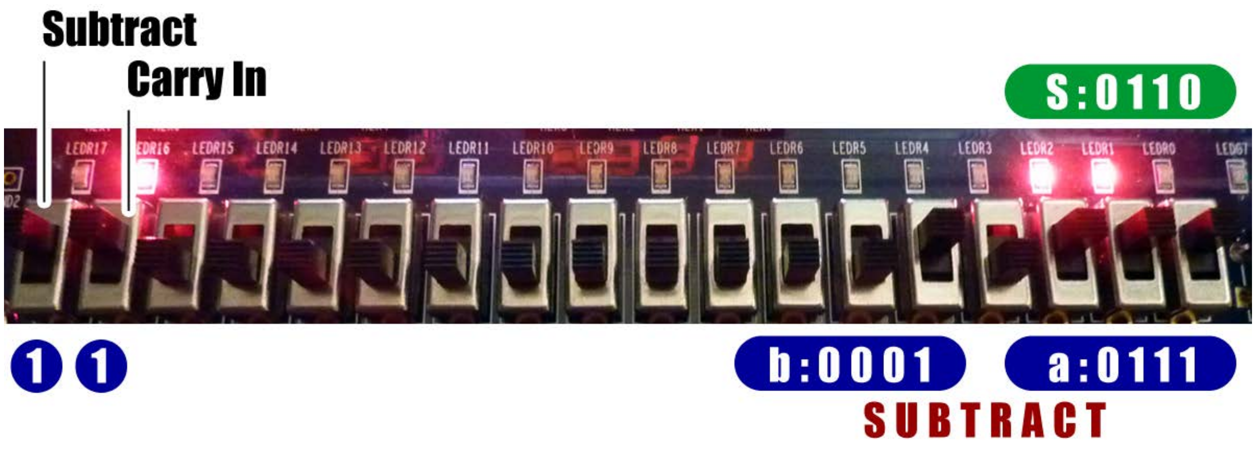


In the second image we are performing subtraction instead, noted by 1 value passed to subtract and carryin of 1, where a=0111 and b=0001, this results in s=0110 and carryout.

CS 211,   Spring 2017

Laboratory Project: VHDL Adder / Subtractor with overflow detection, carry look ahead

Instructor: Professor Izidor Gertner
March 22, 2017 by 11:59 PM



4. Report

Please write a lab report that addresses the following bullet points. Do not limit yourself to these; feel free to address more ideas/experiences with this lab in your summary.

Objective

- What is the goal of this assignment?

Design

- Include the corresponding VHDL code for the 8-bit ripple-carry and carry-look-ahead adder/subtractor circuits.
- Describe the VHDL code, inputs, outputs, etc.
- Include a screenshot of your 8-bit LPM adder/subtractor.

Functionality

CS 211, Spring 2017

Laboratory Project: VHDL Adder / Subtractor with overflow detection, carry look ahead

Instructor: Professor Izidor Gertner
March 22, 2017 by 11:59 PM

- Provide a walkthrough of the circuit behavior.
- List all the operations your circuit performs. Explain what each is supposed to do and how you accomplish them.

Simulation

- Include screenshots of your functional waveform results from each implementation.
- Run several timing simulations with similar values for each of the adder/subtractor implementations.
- Include images from your test in the DE2 board for several addition and subtraction cases, make sure you test for carry out, overflow, etc.

Analysis

- In which cases do you get an Overflow?
- When do you get Carry out = 1?
- What did you do to implement a Ripple-carry adder/subtractor?
- Do you see a difference in speed performance between a 4-bit ripple-carry vs. 8-bit ripple-carry adder? If so, can you explain why? Is this noticeable?
- In general, which implementation performs the fastest out of the three?
- Do you see any significant difference in speed performance among the 4-bit circuits designed? How about among the 8-bit circuits?
- Can you predict what would happen in a 64-bit ripple-carry adder in terms of speed? How about the carry look-ahead adder?

Conclusion

- Give a brief explanation on what you learned about this assignment.
- Are the speed test results conclusive enough for you to decide which implementation performs better?