ENGIN 341 – Advanced Digital Design

Engineering Department

University of Massachusetts Boston

Semester – Spring 2021

Instructor – Dr. Michael Rahaim

Lab Report for

Lab #2: ALU Design

By

Name: Tyler McKean

Student ID: 01098154

Date: 3/14/2021

I pledge to uphold the governing principles of the Code of Student Conduct of the University of Massachusetts Boston. I will refrain from any form of academic dishonesty or deception, cheating, and plagiarism. I pledge that all the work submitted here is my own, and that I have clearly acknowledged and referenced other people's work. I am aware that it is my responsibility to turn in other students who have committed an act of academic dishonesty; and if I do not, then I am in violation of the Code. I will report to formal proceedings if summoned.

Signed:

CONTENTS

NOTE: Right click and select “Update Field” above to automatically update page numbers.

NOTE: Remove all highlighted text prior to submissio

OVERVIEW

For Lab 2, I was tasked with designing and implementing an Arithmetic Logic Unit (ALU) that performs up to 11 functions consisting of math, binary and unary operations on a set of switches and pushbuttons on the Zybo board hardware. In order to choose which operation the ALU would be performing, additional switches attached to the J Connector peripheral ports served as the Function Select inputs to the ALU. The results would then be projected as a hexadecimal value on a 7-Segment Display PMOD Device. A 2-bit full adder design was used from a previous lab to create both an N-bit generic ripple carry adder and subtractor. With a provided Twos Complement vhd file, the remainder of the lab involved designing a 2-to-1 Overflow multiplexer to handle the Carry Out and Borrow Out from the adder and subtractor operations, and an 11-to-1 Output multiplexer that drew from a particular operation based on the Function Select switches. With these components all nested together, the ALU design was further implemented into a Structural architecture that also featured the 7-Segment Driver to display the results of the operation to the peripheral device. With all the components contained into one structural vhd file, I synthesized the design and assigned the correct pins on the Zybo board using the I/O Planning Tool for the necessary pushbuttons and switches. The following sections describe the design process and synthesized results I obtained.

DESIGN DESCRIPTION

The first step in my design process was to alter a previously designed 2-bit Full Adder from the previous lab into an N-bit wide Ripple Carry Adder. The design used generic statements to create inputs and outputs that were N-bits wide vectors. The architecture for the Ripple Carry Adder included the Full Adder component previously made in Lab. The Full Adder consists of two half adders made from XOR and AND gates with a sum and carry output each. To make the inputs and outputs N-bit wide, I could use a generic statement that defined any integer value of bits. This Ripple Carry Adder could then be generated used a for loop to create N number of Full Adders for my design. This module would output a N-bit sum value to the ALU Output Multiplexer and its Carry Out value would go to a 2-to-1 Overflow Multiplexer to signify if an overflow carry occurred.

The next steps I took were to design an N-bit Rippler Carry Subtractor, which involved a similar design process as the 2-bit Full Adder from Lab 1. A Full Subtractor uses the same number of XOR and AND gates as a Full Adder but includes a NOT gate tied to the input of A inside the first half subtractor module and a NOT gate tied to the half subtractor's sum output inside the second half subtractor. A schematic diagram of the Full Subtractor can be seen in Figure 1 below. This full subtractor is defined by this relationship of both half subtractors resulting in a differential output and borrow out created from an OR gate between the borrow outs from both half subtractors. With the full subtractor module design, the N-bit Ripple Carry Subtractor could then be created using generic and generate statements similarly to the N-bit Ripple Carry Adder. The design of a 2-bit Ripple Carry Subtractor can be seen in Figure 2 below.

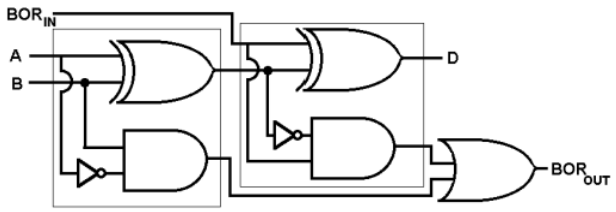

Figure 1 - Design Schematic of a Full Subtractor

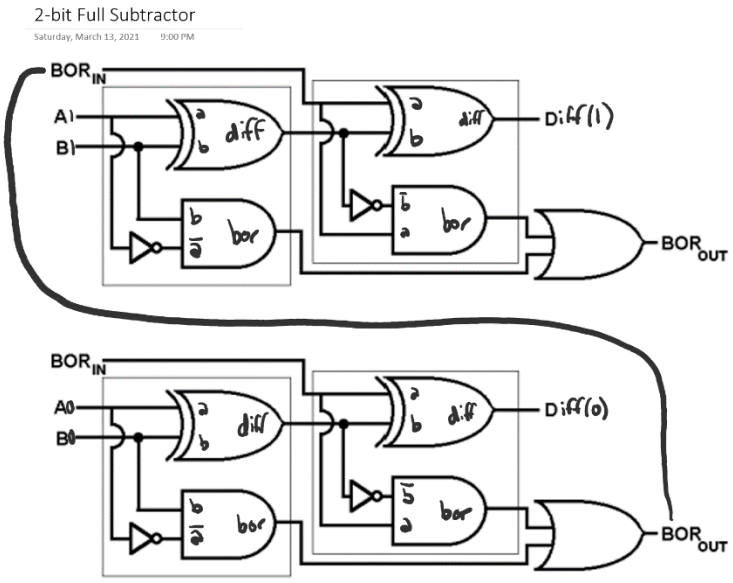

Figure 2 - Design Schematic of a 2-bit Full Subtractor

The Two's Complement module was provided to us, but its inputs were dependent on the differential output from the N-bit Ripple Carry Subtractor. The design for the Two's Complement consisted of an XOR gate tied to the least significant bit, followed by an OR gate tied to the least significant bit and the latter bit. This sequence of XOR and OR gates repeat until reaching the MSB of the differential input. The circuit starts from the LSB and copies every 0 until reaching the first 1, which is copied before inverting the remainder of the bits. A diagram showing the schematic for the Two's Complement is shown in Figure 3 below.

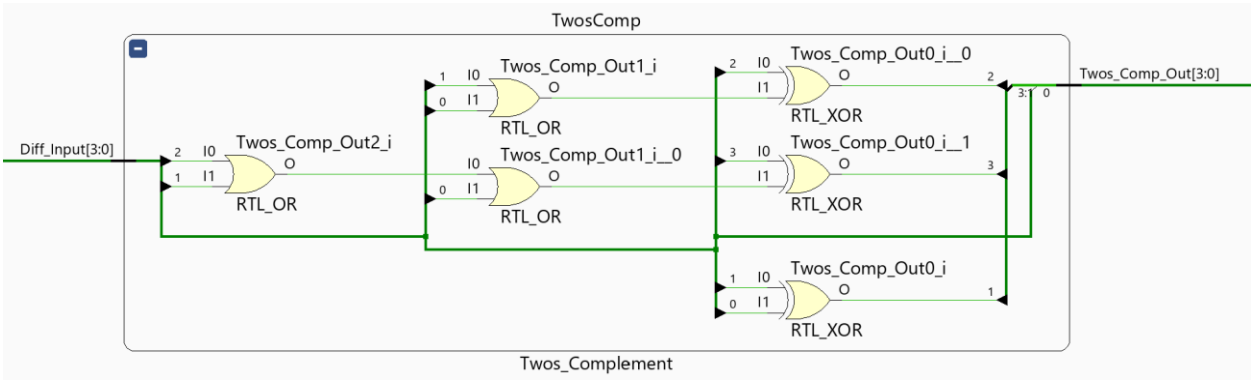

Figure 3 - Circuit Schematic for the Two's Complement of 4 input bits

The Logic operations inside the ALU consisted of an AND, OR, and XOR gates which would be performed on the inputs A and B. These operations would be synthesized by simple modules within the ALU vhd file. The AND module output will consist of 1s when both A and B have 1s at the same designated bits, the OR module will output 1s whenever either A or B's bits have a 1, otherwise the output will be 0, and the XOR will output 1s whenever A and B have different values than one another. The last four operations in the ALU would only be performed on input A, exclusively. They consisted of a NOT, Shift Left, Shift Right, Rotate Right, and Rotate Left operations. The NOT operation would flip the values of A with 1s turning to

0s and 0s turning to 1s. The Shift Left operation would offset the 4-bit sequence one bit to the left and replace the LSB with a 0. Alternatively, the Shift Right operation offsets the 4-bit sequency one bit to the right and replaces the MSB with a 0. The Rotate Left would perform a similar operation except it saves the MSB and places it in the LSB position while offsetting the 4-bit sequence to the left by one bit. The Rotate Right operation would do the opposite and save the LSB value and place it in the MSB position while offsetting the 4-bit sequence by one bit to the right.

The outputs of all 11 functions would then go into an Output Multiplexer that the Function Select switches dictate which operation to perform on the two inputs A and B. To signify that there was an overflow from the operation, or if the result was a negative number, an Overflow Multiplexer was also designed. This Multiplexer would take inputs from the Carry out and Borrow out results of the Ripple Adder and Subtractor with the Function Select again distinguishing which result to display. The overflow result would then be assigned to an LED on the Zybo board to display the overflow or negative result. The 4-bit output of the Output Multiplexer would be sent to a 7-Segment Driver that converts the results of the ALU operation into a hexadecimal value that is displayed on the PMOD attachment. These 7-bit values were assigned to pins on the Zybo board and the connection between the Zybo board and PMOD attachment can be seen in Figure 4 below.

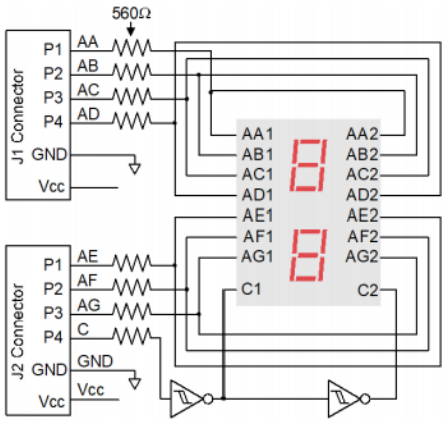

Figure 4 - 7-Segment Display pin connections to J1 and J2 connectors of Zybo board.

DESIGN ENTRY

The N-bit Ripple Carry Adder was the first module I created when starting this Lab. I essentially altered the vhd file from Lab 1, but this time included a generic statement within the entity definition to create N-bit wide inputs and outputs. The code for the Ripple Carry Adder can be seen in [1] in the Appendix, where I declared inputs *a*, *b*, and output *sum* as std_logic_vectors (N-1 downto 0). I assigned my generic value to be 4, so this created vectors that were 4 bits in length. Both the *carry_in* and *carry_out* bits remained as single bit definitions. The architecture contained the full adder defined in the previous lab, but I utilized a generate statement along with a for loop to create N number of full adders. An internal signal named *Cc* served as a carry chain that interconnected *CC(i)* and *CC(i +1)* within the for loop generate statement. Since my generic variable was set at 4, this generated architecture created 4 full adders as design sources in Vivado. More full adders could be created by simply changing the generic variable to a higher integer

variable, which confirms the design of the N-bit Ripple Carry Adder being able to expand to N number of full adders.

With the Adder module designed, I created two new source files for the half subtractors that make up the full subtractor module. The first half subtractor's architecture contained a differential output with an XOR gate between inputs *a* and *b* with a borrow output being a result of NOT *a* AND *b*. The architecture of the second half subtractor had the same inputs and outputs as the first half subtractor with its differential output being an XOR gate between a and b, but contained an AND operation between *a* AND NOT *b*. With both half subtractors being named half_sub1 and half_sub2, I created a full subtractor module with a mixed architecture that contained these half subtractors as components. The full subtractor code is [2] in the Appendix and it contains inputs *a, b, bor_in* with outputs *diff* and *bor_out*. I created internal signals called *d1, b1,* and *b2* that interconnected the two half subtractors. Half_sub1 took in inputs *a* and *b* and its outputs for *diff* and *borrow* were assigned to *d1* and *b1*. For half_sub2, it's input *a* was assigned to *bor_in*, input *b* was assigned to internal signal *d1, diff* assigned to the differential output, and *borrow* assigned to internal signal *b2*. The output *bor_out* was then instantiated to be the result of *b1* OR *b2*. This concluded the mixed architecture of the full subtractor to which I then created another vhd file that would use this as a component to create the N-bit Ripple Carry Subtractor. The code for the N-bit Ripple Carry Subtractor [3] shows the utilization of a generic variable to create N-bit long input vectors: *a, b,* and *diff* with single bit outputs *bor_in* and *bor_out*. Its generated architecture was created by including the full subtractor module component, internal signal *Bc* to serve as a borrow chain, and an instantiated for loop to generate N number of full subtractors. Again, like the Ripple Adder, the internal signal *Bc* interconnects borrows from the current number in the for loop to the next. This generated structure generates N number of full subtractors and concludes the design for the N-bit Ripple Carry Subtractor.

After finishing the Ripple Carry Subtractor module, I created a vhd file for the Overflow MUX [4]. This multiplexer contained single bit inputs *Cout, Bout*, and 4-bit input from the Function Select switches with single bit output *Overflow_out*. The dataflow architecture contained a with statement that specified if the *Function_Select* input was the value 0000, then the *Overflow_out* would be equal to the *Cout* input. This *Function_Select* value was specified for the Ripple Carry Adder operation, so incase of a carry overflow, this multiplexer illuminate an LED. If the *Function_Select* input was the value 0001, then *Overflow_out* would be equal to the *Bout* input, which was associated with the borrow overflow of the Ripple Carry Subtractor indicating the result was negative. All the other possible values for the *Function_Select* input were disregarded in this multiplexer's architecture.

The Output Multiplexer was created with 12 inputs corresponding to the results of the ALU operations and Function select switches with a 4-bit Output [5]. The dataflow architecture used a with statement to cycle through 11 values of the Function Select and specified which input of the Output MUX that the Output would be selected from. A table of values for the Function Select were given to us and can be seen in Figure 5 below.

Function Select	Function
0000	Add
0001	Subtract
0010	AND
0011	Two's Complement
0100	OR
0101	XOR
0110	NOT
0111	Shift Left
1000	Shift Right
1001	Rotate Left
1010	Rotate Right
1011	--
1100	--
1101	--
1110	--
1111	--

Figure 5 - Output MUX Functions with Control Signals

The values that were not assigned to a particular function were instantiated as undefined. With the two Multiplexer modules built, I created a vhd file for the ALU that contained all the functions and multiplexers into one behavioral architecture.

The code for the ALU [6] contains 4-bit input vectors *A, B,* and *Function_Select* with a 4-bit output vector *R* and *Overflow* output bit. Within the architecture of the ALU, I defined the Overflow_MUX, Output_MUX, both N-bit Ripple Carry Adder and Subtractor, and the Twos_Complement module as components. The internal signals I defined within the architecture served as the connections to tie every component to the correct input and output. For the Overflow_MUX, I defined signals *Carry_out* and *Borrow_out* that would tie the Carry out of the Ripple Adder and the Borrow out of the Subtractor to the inputs of the multiplexer. Within both the Ripple Carry Adder and Subtractor, I had instantiated that the *carry_in* and *bor_in* to logic low since the only inputs considered for this design were the *A* and *B* input vectors. The Twos Complement module took an internal input signal called diff from the output of the Subtractor and outputted an internal signal called twos. For the remaining binary and unary operations, I defined 4-bit std_logic_vectors within the architecture for each operation. For the AND operation, the signal *and_out* performed an AND expression on inputs *A* and *B*. The internal signal *or_gate* instantiated an OR expression on both *A* and *B* inputs. Likewise, the signal *xor_out* instantiated an XOR expression on inputs *A* and *B*. The signal *not_out* instantiated a NOT operation on just the input *A*. For the Shift Left operation, the signal *sl_out* took the highest 3 bits of input *A* and added a 0 to the LSB position. The Shift Right operation was instantiated with internal signal *sr_out* taking the lowest 3 bits of *A* and replacing the MSB with a 0. The Rotate Left signal, *rl_out*, was instantiated by taking the lowest 3-bits of A and using an AND operation to place the LSB in the MSB position. Similarly, the Rotate Right operation was instantiated to signal *rr_out* by taking the LSB of input *A* and performing an AND operation with its highest 3-bits. The port map of the Output_MUX was then defined such that the inputs to each operation were one of the created internal signals within its architecture.

With the internal components of the ALU tied together, the next steps were to create one last vhd file with both the ALU and 7-Segment driver. The code for the Lab2_Design structural vhd file can be seen as [7] in the Appendix. The entity is defined with 4-bit inputs *A*, *B*, and *Function_Select* with a 7-bit output vector *Seg_Output* and single bit output *Overflow_Out*. The structural architecture includes the

Arithmetic_Logic_Unit and Seven_Segment_Driver modules as components with a 4-bit internal signal called *Seven_Seg_In*. This internal signal ties the output of the ALU to the input of the 7-Segment driver and concludes all the necessary design procedures for the lab.

RESULTS AND OBSERVATIONS

Simulation Results

After completing the designs of all the modules. I started by created some testbench files for the Ripple Carry Adder, Ripple Carry Subtractor, Twos Complement, and the ALU module. The first testbench I created was for a 2-bit instantiation of the Ripple Carry Adder and its results can be seen in Figure 6 below.

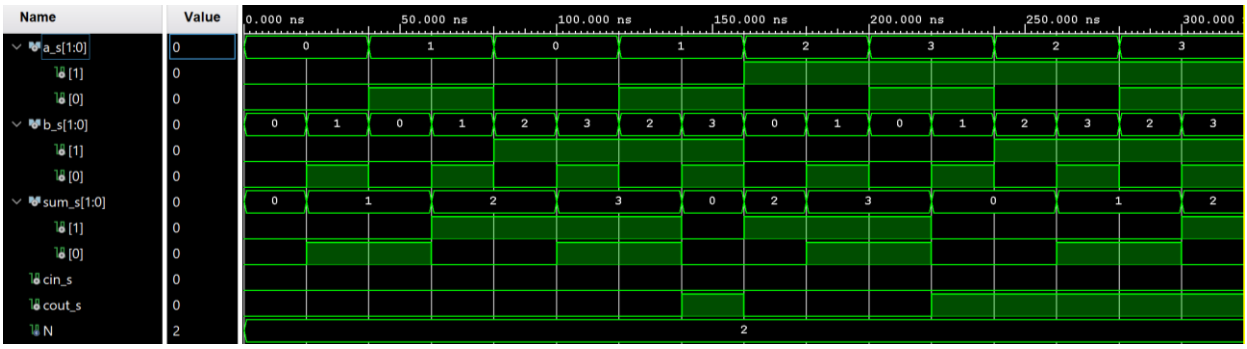

Figure 6 - Results of 2-bit Ripple Carry Adder Testbench

Looking over the results, the Ripple Carry Adder appears to be functioning correctly. The code for the Adder testbench [8] was initialized with the *cin_s* signal to zero, since the ALU did not depend on the carry input port of Ripple Adder. When signals *b0_s* or *a0_s* are high, *sum0_s* was high as well. If both *a0_s* and *b0_s* were high, then the carry chain results in *sum1_s* being high. When *a1_s* or *b1_s* was high, this resulted in *sum1_s* having a logical high and when both *a1_s* and *b1_s* were logical high, this resulted in *cout_s* being logical high too. The remainder of the timetable simulation appeared to be working correctly so the next testbench I created was for the Ripple Carry Subtractor.

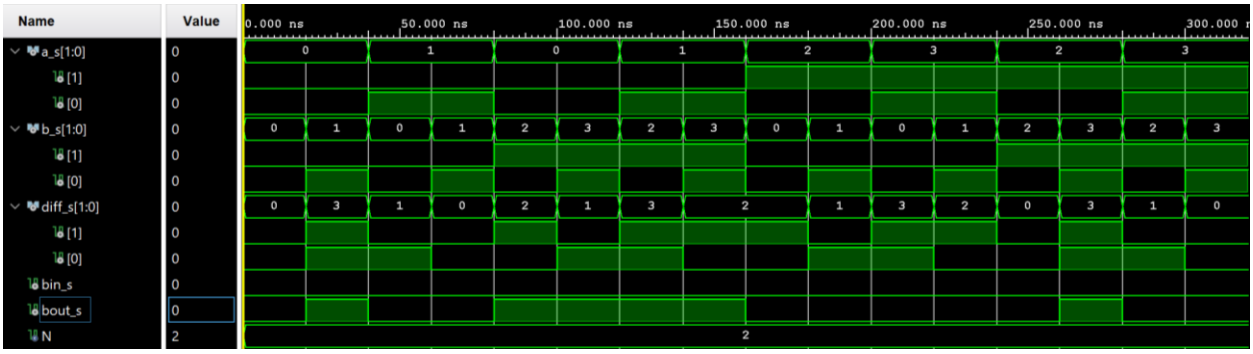

Figure 7 - Results of 2-bit Ripple Carry Subtractor Testbench

Analyzing the results of this testbench, I was able to confirm that my Ripple Carry Subtractor was designed correctly. Here, whenever just *b1_ s* or *b0_s* were high, the bout_s was high signifying a negative output. When only *b0_s* was high this resulted in both *diff1_s* and *diff0_s* as logical high. When either *a1_s* and

b1_s or a*0_s* and *b0_s* were high, the results were all logical high, showing a zero from subtracting 1-1 or 2-2. Again, I initialized the *bin_s* to be zero since the ALU design did not rely on a borrow input for the Subtractor. The code for the testbench can be found in the Appendix as [9]. With the Ripple Carry Subtractor operating appropriately, the next testbench I created was for the Twos Complement module.

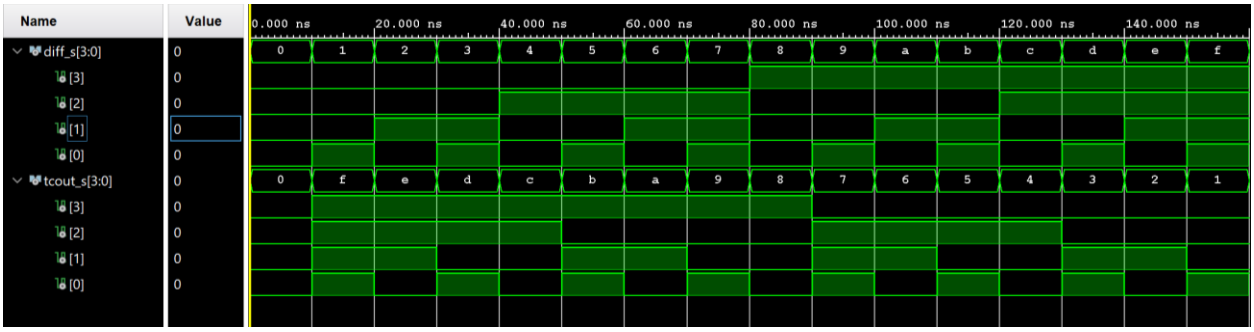

Figure 7 - Results of Twos Complement Testbench

The results of the Twos Complement testbench can be observed in Figure 7 above and the code for the testbench is [10] in the Appendix. As the differential input counts in hexadecimal values from 0 to F, the tcout_s signals count down from hex values of F to 0, but both start with 0. Since this function is taking the differential output from the Subtractor, if both input values of *A* and *B* are equal the result with be 0 because the inverted bits would result in 1111, but a one would add to this value clearing the four bits to 0000. When the *diff_s* inputs were 0001, the output was equal to 1111 or F. The function is taking the input, flipping the bits, and then adding one to the inverted bits to get the *tcout_s* results. So, in the case of 0001, when the signal is inverted, it would be 1110 before an addition one is added to it, resulting in a value of F for the output. The remaining combination of values were correct as well and proved the design of the Twos Complement module to be a success.

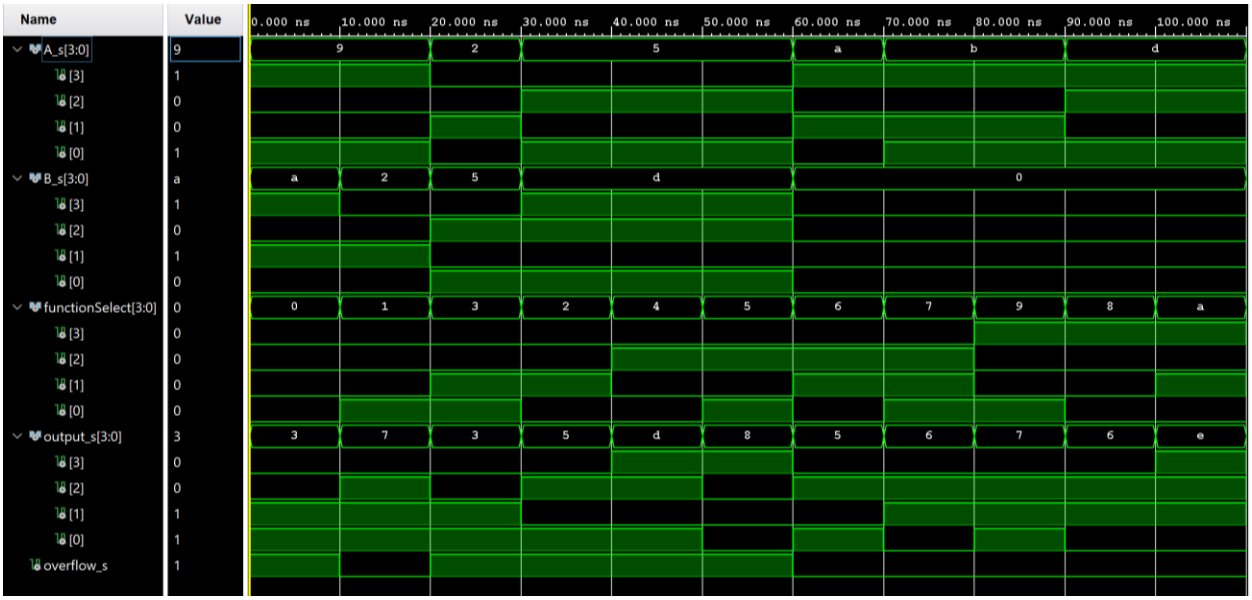

Figure 8 - Results of Arithmetic Logic Unit Testbench

The last testbench I created before synthesizing the design was the ALU testbench. For this testbench, we were asked to test all 11 functions of the ALU, so I setup my simulation to run through each function every 10 ns [11]. In addition to creating the ALU simulation, we were asked to provide handwritten calculations for each operation, which can be seen in Figure 9 below.

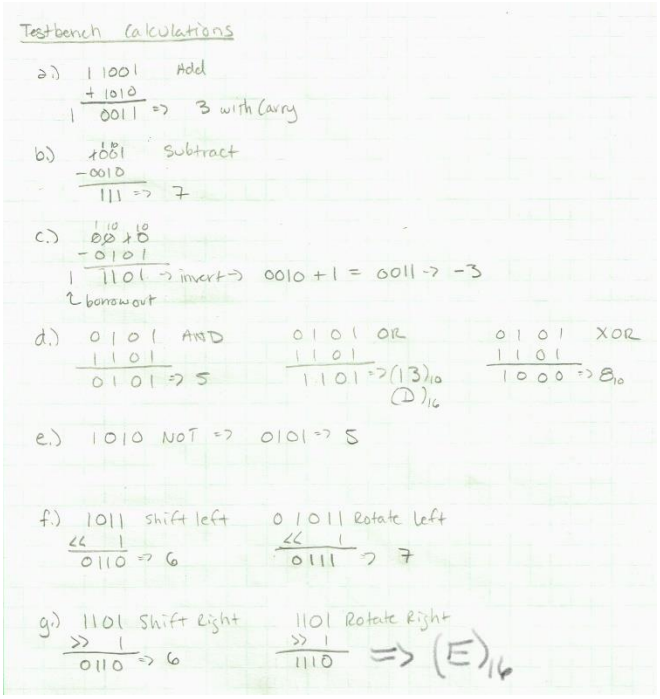

Figure 9 - Hand calculations from each of the 11 functions tested in the ALU testbench.

The first operation was performed from 0-10ns with an ADD operation between *A* = 1001 and *B* = 1010. This operation resulted with the *output_s* signals equaling 3 (0011) with the *overflow_s* signal being high. Referring to my hand calculations, my result was also 3 with a carry out from the overflow of the most significant bits being added. From 10-20ns, the next operation instantiated was the Subtraction of *A* = 1001 and *B* = 0010. This resulted in the *output_s* signal being 7 (0111), which I also got from my hand calculations. The next function tested at 20-30ns was the Twos Complement operation where *A* = 0010 was subtracted from *B* = 0101. This resulted in a value of 3 in the output with a borrow out making the *overflow_s* signal a logical high, which indicates the result was negative. Checking my notes, the subtraction result was 1101, which was inverted to 0010 and then 1 was added to it resulting in 0011, which was correct. From 30-60ns, the following three operations were the AND, OR, and XOR functions with *A* = 0101 and *B* = 1101. The AND operation yielded 0101 as a result, the OR operation produced 1101 as a result and the XOR operation resulted in 1000 as a result. Checking my handwritten notes, I also got the same results for these logical operations. The last five operations were only performed on the input *A*. From 60-70ns, the NOT operation was performed with *A* = 1010 and resulted in 0101, which was correct. The Shift Left operation was executed between 70-80ns with *A* = 1011 and resulted in 0110, which was correct because the MSB value was lost during the operation and a '0' was added to the LSB position. The Rotate Left operation was performed next between 80-90ns with *A* = 1011 and the *output_s* signals resulted in 0111, which is what I also got from my handwritten notes. At time 90-100ns, the Shift Right operation was performed on *A* = 1101 and resulted in 0110. Then the last operation at 100-110ns was the Rotate Right

operation giving a result of 1110. My handwritten notes matched every result of the ALU testbench, which indicates my design was successful. The next steps were to synthesize the design and use the I/O planning tool to map the inputs and outputs of the Lab 2 design to the switches and pushbuttons of the Zybo board.

Synthesized Results

The Figure below shows the synthesized design of the Lab 2 design schematic synthesized in Vivado.

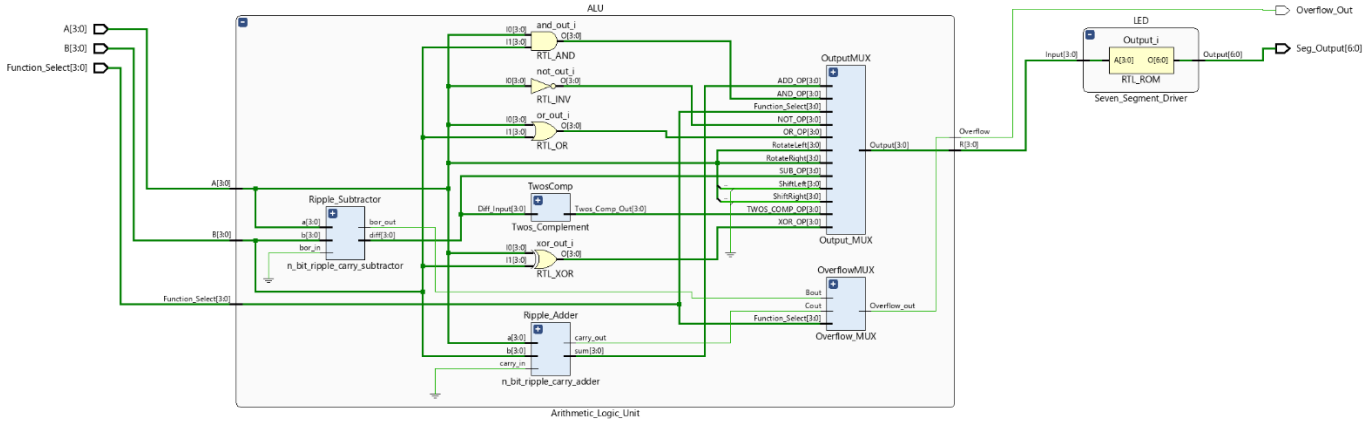

Figure 10 - RTL Design of Lab 2 schematic

Using the I/O planning tool and referencing the Zybo board manual I assigned the necessary pins all the switches, pushbuttons, and LED according to my handwritten notes in Figure 11.

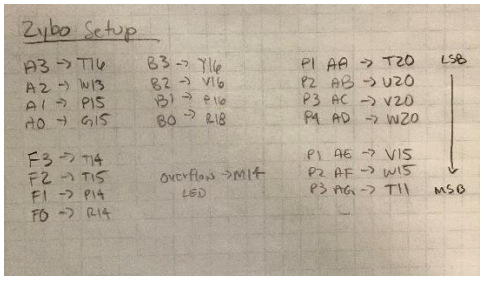

Figure 11 - Handwritten Pin Constraints for Lab 2

These pin assignments set input bits *A* to the onboard switches, input *B* to the onboard pushbuttons, the Function Select PMOD switches to the JD Connector and the 7-Segment Display to the JB/JC Connectors of the Zybo board. With the design synthesized, I programmed the board to test the ALU functions I simulated in the testbench. The following sections shows physical demonstrations of each of the 11 functions simulated in the ALU testbench.

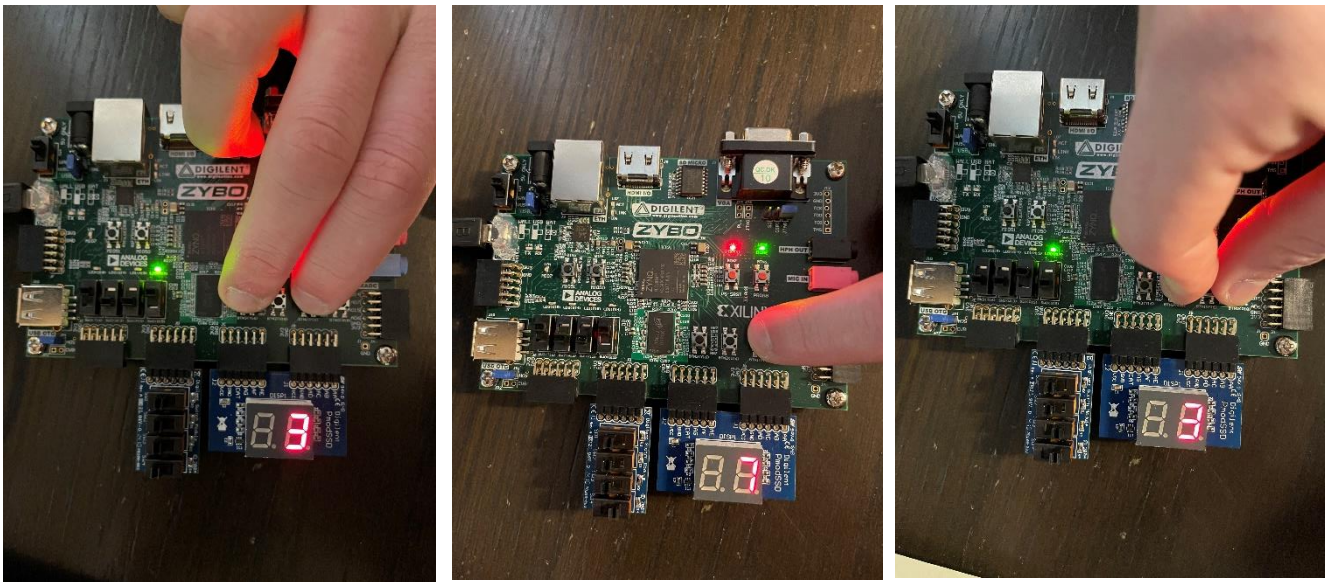

Figure 12 - Results from Add (left), Subtract (center), and Twos Complement (right) on the Zybo Board

Figure 12 above shows the operation results for the Add, Subtract, and Twos Complement function with the 7-Segment Display illuminating the hexadecimal result. My results from the testbench for the Add operation between 1001 and 1010 was 0011, thus the number 3 is displayed as a result. The Subtraction operation between 1001 and 0010 resulted in 0111, which displayed a 7 in the 7-Segement Display. The Twos Complement of 0010-0101 was equaled to 3 being displayed on the 7-Segment and LED M14 illuminating, which indicates the negative result.

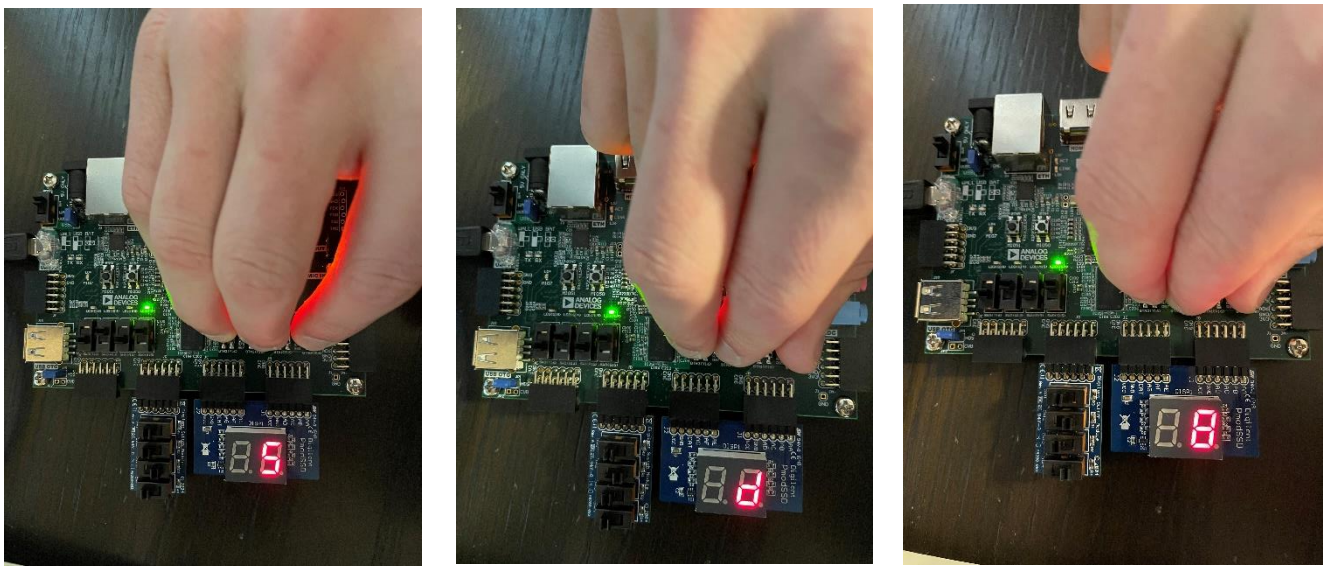

Figure 13 - Results from AND (left), OR (center), and XOR (right) operations with 7-Segment displaying results.

Figure 13 above shows the synthesized results of the logical operations: AND, OR, and XOR. The AND operation of 0101 with 1101 resulted in a value of 0101, which is seen as 5 on the display. The OR operation of 0101 with 1101 resulted in 1101, which is a hexadecimal value of D and is shown as its lowercase form

as a result on the 7-Segment display. The XOR operation of 0101 with 1101 resulted in 1000, so the 7-Segment displayed the numerical value 8 on the display.

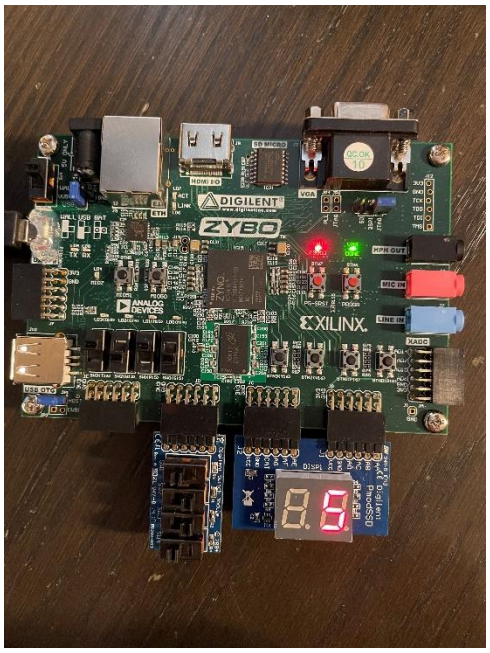

Figure 14 - NOT operation of input switches A on Zybo board.

The NOT operation of 1010 resulted in 0101 with the number 5 being displayed and can be seen in Figure 14 above.

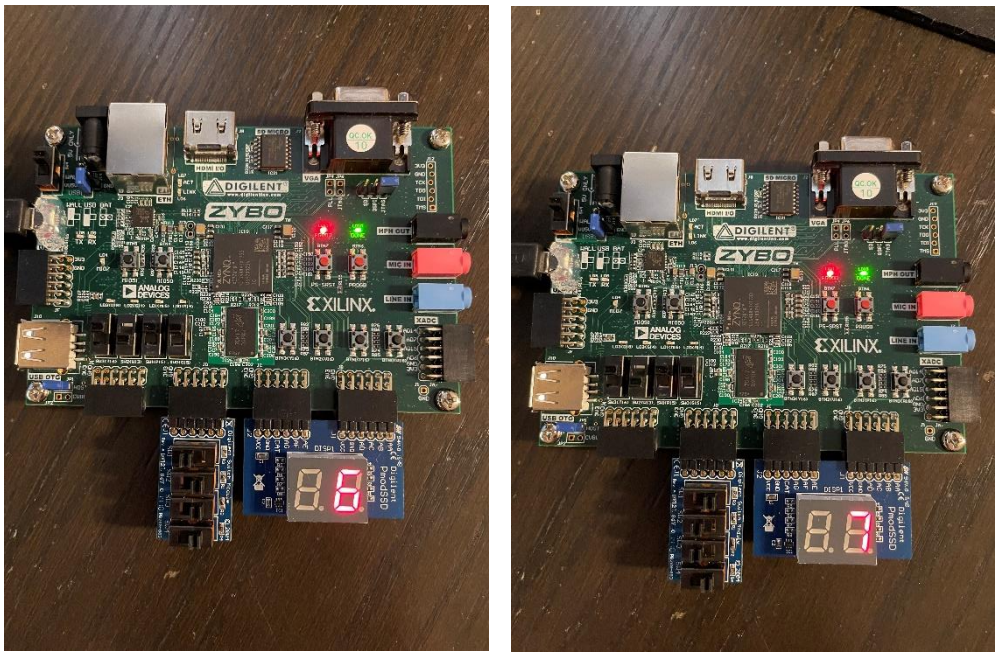

Figure 15 - Results of Shift Left (left) and Rotate Left (right) of input switches A on Zybo Board.

Figure 15 shows the results of the Shift Left and Rotate Left operation. For the Shift Left, the input value of 1011 resulted in outcome of 0110 with a 6 displayed. The Rotate Left operation on value 1011 resulted in 0111 as an outcome with the number 7 displayed.

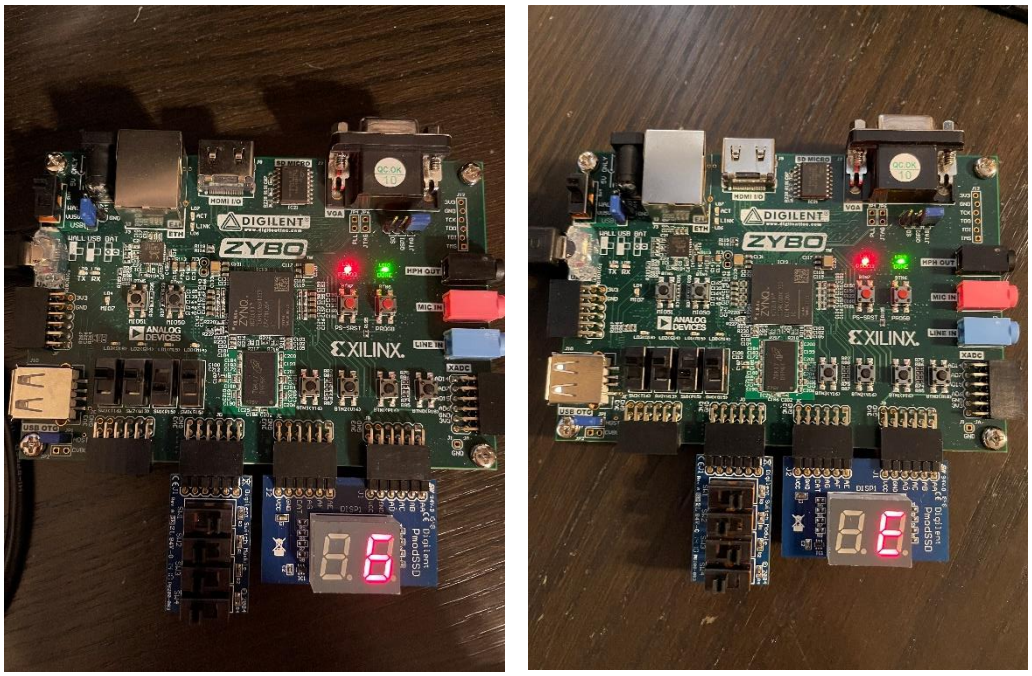

Figure 16 - Results of Shift Right (left) and Rotate Right (right) operation on input switches A of Zybo Board.

Figure 16 shows the outcomes of the Shift Right and Rorate Right operations on the input switches. For the Shift Right function, it was performed on inputs 1101 and its outcome was 0110, which displayed 6 on the 7-Segment display. The last operation performed was the Rotate Right operation on inputs 1101 and its outcome was 1110, which resulted with a hexadecimal value of E displayed. This concludes all the synthesized tests I performed on the physical switches and pushbuttons on the Zybo board.

SUMMARY

The overall purpose of this lab was to design an Arithmetic Logic Unit that could perform 11 functions on two sets of 4-bit inputs and display the results to a 7-Segment Display. My design process started by altering a previous 2-bit full adder module into an N-bit Ripple Carry Adder. Following this redesign, I created an N-bit Ripple Carry Subtractor in a similar design flow. An Overflow Multiplexer was designed to take the Carry and Borrow outs of the Adder and Subtract and output a single bit that would illuminate an LED when there was a carry overflow and when the outcome of the Subtractor was a negative number. With the provided Twos Complement Module, I instantiated the remaining functions within the ALU architecture with interconnecting internal signals. An Output Multiplexer was designed to take the inputs from all 11 of the functions and output a 4-bit value to the 7-Segment Driver. With the ALU designed, I created testbenches for the Ripple Carry Adder, the Ripple Carry Subtractor, the Twos Complement module, and the entire ALU design. My testbench simulations proved to be successful, which I then used with the provided 7-Segment Drive module and Lab 2 design vhd file to interconnect the entire top-level design all within one structural architecture. I then synthesized the design and programmed my Zybo board where I tested each of the 11 functions performed in the ALU testbench. My synthesized results matched my simulation results and indicated that my design for the ALU was successful.

APPENDIX

```
[1] library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity n_bit_ripple_carry_adder is
    generic(N : positive:= 4);
    Port ( a, b : in STD_LOGIC_VECTOR (N-1 downto 0);
           carry_in : in STD_LOGIC;
           sum : out STD_LOGIC_VECTOR (N-1 downto 0);
           carry_out : out STD_LOGIC);
end n_bit_ripple_carry_adder;
architecture Generated of n_bit_ripple_carry_adder is
component full_adder is
    port(
        a : in STD_LOGIC;
        b : in STD_LOGIC;
        carry_in : in STD_LOGIC;
        sum : out STD_LOGIC;
        carry_out : out STD_LOGIC
        );
end component;
-- Internal signals
```

```
signal Cc : std_logic_vector (N downto 0); -- Carry chain
begin
   Cc(0) <= carry_in;
   full_adders: for i in 0 to N-1 generate
            FAx: full_adder port map (A(i), B(i), CC(i), sum(i), Cc(i+1));
            end generate;
   carry_out <= Cc(N);
end Generated;
```

[2]

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity full_sub is
   Port ( a : in STD_LOGIC;
          b : in STD_LOGIC;
          bor_in : in STD_LOGIC;
          diff : out STD_LOGIC;
          bor_out : out STD_LOGIC);
end full_sub;
architecture mixed of full_sub is
component half_sub1 is
   port(
      a : in STD_LOGIC;
      b : in STD_LOGIC;
      diff : out STD_LOGIC;
      borrow : out STD_LOGIC
      );
end component;
component half_sub2 is
   port(
      a : in STD_LOGIC;
      b : in STD_LOGIC;
      diff : out STD_LOGIC;
```

```
        borrow : out STD_LOGIC
        );
end component;
signal d1, b1, b2 : STD_LOGIC;
begin
    hs1: half_sub1 port map (
        a => a,
        b => b,
        diff => d1,
        borrow => b1 );
    hs2: half_sub2 port map (
        a => bor_in,
        b => d1,
        diff => diff,
        borrow => b2 );
    or_gate: bor_out <= b1 OR b2;
end mixed;
```

[3]

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity n_bit_ripple_carry_subtractor is
    generic(N : positive := 4);
    Port ( a, b : in STD_LOGIC_VECTOR (N-1 downto 0);
           bor_in : in STD_LOGIC;
           diff : out STD_LOGIC_VECTOR (N-1 downto 0);
           bor_out : out STD_LOGIC);
end n_bit_ripple_carry_subtractor;
architecture Generated of n_bit_ripple_carry_subtractor is
component full_sub is
    port(
        a : in STD_LOGIC;
        b : in STD_LOGIC;
```

```
        bor_in : in STD_LOGIC;
        diff : out STD_LOGIC;
        bor_out : out STD_LOGIC
        );
end component;
-- Internal signals
signal Bc : std_logic_vector (N downto 0); -- Borrow chain
begin
    Bc(0) <= bor_in;
    full_subs: for i in 0 to N-1 generate
                FSx: full_sub port map (a(i), b(i), BC(i), diff(i), Bc(i+1));
                end generate;
    bor_out <= Bc(N);
end Generated;
```

[4]

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity Overflow_MUX is
        generic (N: positive:= 4);
        Port(
        Cout, Bout: in std_logic;
        Function_Select: in std_logic_vector (N-1 downto 0);
        Overflow_out: out std_logic );
end Overflow_MUX;
architecture Dataflow of Overflow_MUX is
begin
    with Function_Select select Overflow_out <=
    Cout when "0000",
    Bout when "0001",
    '-' when others;
end Dataflow;
```

[5] library IEEE;

use IEEE.STD_LOGIC_1164.ALL;

entity Output_MUX is

generic (N: integer:= 4);

Port(

ADD_OP: in std_logic_vector (N-1 downto 0);

SUB_OP: in std_logic_vector (N-1 downto 0);

TWOS_COMP_OP: in std_logic_vector (N-1 downto 0);

AND_OP: in std_logic_vector (N-1 downto 0);

OR_OP: in std_logic_vector (N-1 downto 0);

XOR_OP: in std_logic_vector (N-1 downto 0);

NOT_OP: in std_logic_vector (N-1 downto 0);

ShiftLeft: in std_logic_vector (N-1 downto 0);

ShiftRight: in std_logic_vector (N-1 downto 0);

RotateLeft: in std_logic_vector (N-1 downto 0);

RotateRight: in std_logic_vector (N-1 downto 0);

Function_Select: in std_logic_vector (N-1 downto 0);

Output: out std_logic_vector (N-1 downto 0));

end Output_MUX;

architecture Dataflow of Output_MUX is

begin

with Function_Select select

Output <=

ADD_OP when "0000",

SUB_OP when "0001",

AND_OP when "0010",

TWOS_COMP_OP when "0011",

OR_OP when "0100",

XOR_OP when "0101",

NOT_OP when "0110",

```
    ShiftLeft when "0111",
    ShiftRight when "1000",
    RotateLeft when "1001",
    RotateRight when "1010",
    (others => 'U') when others;
end Dataflow;
```

[6]

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity Arithmetic_Logic_Unit is
    generic (N: positive:= 4);
    Port (
    A,B: in std_logic_vector(N-1 downto 0);
    Function_Select: in std_logic_vector (N-1 downto 0);
    Overflow: out std_logic;
    R: out std_logic_vector(N-1 downto 0));
end Arithmetic_Logic_Unit;
architecture Behavioral of Arithmetic_Logic_Unit is
component Overflow_MUX is
    Port(
        Cout, Bout: in std_logic;
        Function_Select: in std_logic_vector (N-1 downto 0);
        Overflow_out: out std_logic );
end component;
component Output_MUX is
    Port(
        ADD_OP: in std_logic_vector (N-1 downto 0);
        SUB_OP: in std_logic_vector (N-1 downto 0);
        TWOS_COMP_OP: in std_logic_vector (N-1 downto 0);
```

```
        AND_OP: in std_logic_vector (N-1 downto 0);
        OR_OP: in std_logic_vector (N-1 downto 0);
        XOR_OP: in std_logic_vector (N-1 downto 0);
        NOT_OP: in std_logic_vector (N-1 downto 0);
        ShiftLeft: in std_logic_vector (N-1 downto 0);
        ShiftRight: in std_logic_vector (N-1 downto 0);
        RotateLeft: in std_logic_vector (N-1 downto 0);
        RotateRight: in std_logic_vector (N-1 downto 0);
        Function_Select: in std_logic_vector (N-1 downto 0);
        Output: out std_logic_vector (N-1 downto 0));
end component;
component n_bit_ripple_carry_adder is
    Port ( a, b : in STD_LOGIC_VECTOR (N-1 downto 0);
           carry_in : in STD_LOGIC;
           sum : out STD_LOGIC_VECTOR (N-1 downto 0);
           carry_out : out STD_LOGIC);
end component;
component n_bit_ripple_carry_subtractor is
    Port ( a, b : in STD_LOGIC_VECTOR (N-1 downto 0);
           bor_in : in STD_LOGIC;
           diff : out STD_LOGIC_VECTOR (N-1 downto 0);
           bor_out : out STD_LOGIC);
end component;
component Twos_Complement is
    Port ( Diff_Input : in STD_LOGIC_VECTOR (3 downto 0);
           Twos_Comp_Out : out STD_LOGIC_VECTOR (3 downto 0));
end component;
signal Cin, Bin, Carry_out, Borrow_out : std_logic;
signal sum, diff, twos, add_in, sub_in, and_in, or_in, xor_in, not_in, shiftleft_in, shiftright_in, rotateleft_in, rotateright_in : std_logic_vector(N-1
downto 0);
signal add, sub, and_out, or_out, xor_out, not_out, sl_out, sr_out, rl_out, rr_out: std_logic_vector (N-1 downto 0);
begin
```

```
OverflowMUX: Overflow_MUX port map(
   Cout => Carry_out,
   Bout => Borrow_out,
   Function_Select => Function_select,
   Overflow_out => Overflow );
Ripple_Adder: n_bit_ripple_carry_adder port map(
   a => A,
   b => B,
   carry_in => '0',
   sum => sum,
   carry_out => Carry_out );
Ripple_Subtractor: n_bit_ripple_carry_subtractor port map(
   a => A,
   b => B,
   bor_in => '0',
   diff => diff,
   bor_out => Borrow_out );
TwosComp: Twos_Complement port map(
   Diff_Input => diff,
   Twos_Comp_Out => twos );
AND_GATE: and_out <= A and B;
OR_GATE: or_out <= A or B;
XOR_GATE: xor_out <= A xor B;
NOT_GATE: not_out <= not A;
SHIFTLEFT: sl_out <= A(N-2 downto 0) & '0';
SHIFTRIGHT: sr_out <= '0' & A(N-1 downto 1);
ROTATELEFT: rl_out <= A(N-2 downto 0) & A(N-1);
ROTATERIGHT: rr_out <= A(0) & A(N-1 downto 1);
OutputMUX: Output_MUX port map(
   ADD_OP => sum,
   SUB_OP => diff,
   TWOS_COMP_OP => twos,
```

```
        AND_OP => and_out,
        OR_OP => or_out,
        XOR_OP => xor_out,
        NOT_OP => not_out,
        ShiftLeft => sl_out,
        ShiftRight => sr_out,
        RotateLeft => rl_out,
        RotateRight => rr_out,
        Function_Select => Function_Select,
        Output => R );
end Behavioral;
```

[7]

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Lab2_Design is
    Port ( A : in STD_LOGIC_VECTOR (3 downto 0);
           B : in STD_LOGIC_VECTOR (3 downto 0);
           Function_Select : in STD_LOGIC_VECTOR (3 downto 0);
           Seg_Output : out STD_LOGIC_VECTOR (6 downto 0);
           Overflow_Out : out STD_LOGIC);
end Lab2_Design;

architecture Structural of Lab2_Design is

component Arithmetic_Logic_Unit is
    Port ( A : in STD_LOGIC_VECTOR (3 downto 0);
           B : in STD_LOGIC_VECTOR (3 downto 0);
           Function_Select : in STD_LOGIC_VECTOR (3 downto 0);
           R : out STD_LOGIC_VECTOR (3 downto 0);
           Overflow : out STD_LOGIC);
```

```
end component;
component Seven_Segment_Driver is
    Port ( Input : in STD_LOGIC_VECTOR (3 downto 0);
           Output : out STD_LOGIC_VECTOR (6 downto 0));
end component;
signal Seven_Seg_In : STD_LOGIC_VECTOR (3 downto 0);
begin
ALU: Arithmetic_Logic_Unit port map (
    A => A,
    B => B,
    Function_Select => Function_Select,
    R => Seven_Seg_In,
    Overflow => Overflow_Out);
LED: Seven_Segment_Driver port map (
    Input => Seven_Seg_In,
    Output => Seg_Output);
end Structural;
```

[8]

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity n_bit_ripple_carry_tb is
end n_bit_ripple_carry_tb;
architecture n_bit_ripple_carry_tb_stimulus of n_bit_ripple_carry_tb is
component n_bit_ripple_carry_adder is
    generic(N : integer := 2);
    Port ( a, b : in STD_LOGIC_VECTOR (N-1 downto 0);
           carry_in : in STD_LOGIC;
           sum : out STD_LOGIC_VECTOR (N-1 downto 0);
           carry_out : out STD_LOGIC);
end component;
-- Stimulus signals initialized to '0'
```

```
constant N: integer := 2;
signal a_s, b_s, sum_s : std_logic_vector (N-1 downto 0) := "00";
signal cin_s, cout_s : STD_LOGIC := '0';
begin
-- Instantiate a Unit Under Test (UUT) and connect to internal stimuli
    UUT: n_bit_ripple_carry_adder
    generic map (N => 2)
    port map (
        a(0) => a_s(0),
        a(1) => a_s(1),
        b(0) => b_s(0),
        b(1) => b_s(1),
        carry_in => cin_s,
        sum(0) => sum_s(0),
        sum(1) => sum_s(1),
        carry_out => cout_s );
    a_s(1) <= not a_s(1) after 160ns;
    a_s(0) <= not a_s(0) after 40ns;
    b_s(1) <= not b_s(1) after 80ns;
    b_s(0) <= not b_s(0) after 20ns;
    cin_s <= '0';
end n_bit_ripple_carry_tb_stimulus;
```

[9]

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity n_bit_ripple_carry_sub_tb is
end n_bit_ripple_carry_sub_tb;
architecture n_bit_ripple_carry_sub_tb_stimulus of n_bit_ripple_carry_sub_tb is
component n_bit_ripple_carry_subtractor is
    generic(N : integer := 2);
    Port ( a, b : in STD_LOGIC_VECTOR (N-1 downto 0);
```

```
        bor_in : in STD_LOGIC;
        diff : out STD_LOGIC_VECTOR (N-1 downto 0);
        bor_out : out STD_LOGIC);
end component;
-- Stimulus signals initialized to '0'
constant N: integer := 2;
signal a_s, b_s, diff_s: std_logic_vector (N-1 downto 0) := "00";
signal bin_s, bout_s : STD_LOGIC := '0';
begin
-- Instantiate a Unit Under Test (UUT) and connect to internal stimuli
   UUT: n_bit_ripple_carry_subtractor
   generic map (N => 2)
   port map (
      a(0) => a_s(0),
      a(1) => a_s(1),
      b(0) => b_s(0),
      b(1) => b_s(1),
      bor_in => bin_s,
      diff(0) => diff_s(0),
      diff(1) => diff_s(1),
      bor_out => bout_s );
   a_s(1) <= not a_s(1) after 160ns;
   a_s(0) <= not a_s(0) after 40ns;
   b_s(1) <= not b_s(1) after 80ns;
   b_s(0) <= not b_s(0) after 20ns;
   bin_s <= '0';
end n_bit_ripple_carry_sub_tb_stimulus;
```

[10]

```
 library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
```

```
entity tc_tb is
end tc_tb;
architecture tc_tv_stimulus of tc_tb is
component Twos_Complement is
    Port ( Diff_Input : in STD_LOGIC_VECTOR (3 downto 0);
           Twos_Comp_Out : out STD_LOGIC_VECTOR (3 downto 0));
end component;
-- Stimulus signals initialized to '0'
signal diff_s : std_logic_vector (3 downto 0) := "0000";
signal tcout_s : std_logic_vector (3 downto 0) := "0000";
begin
-- Instantiate a Unit Under Test (UUT) and connect to internal stimuli
    UUT: Twos_Complement
    port map (
    Diff_Input(0) => diff_s(0),
    Diff_Input(1) => diff_s(1),
    Diff_Input(2) => diff_s(2),
    Diff_Input(3) => diff_s(3),
    Twos_Comp_Out(0) => tcout_s(0),
    Twos_Comp_Out(1) => tcout_s(1),
    Twos_Comp_Out(2) => tcout_s(2),
    Twos_Comp_Out(3) => tcout_s(3) );
    diff_s(0) <= not diff_s(0) after 10ns;
    diff_s(1) <= not diff_s(1) after 20ns;
    diff_s(2) <= not diff_s(2) after 40ns;
    diff_s(3) <= not diff_s(3) after 80ns;
end tc_tv_stimulus;
```

[11]

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
```

```
entity alu_tb is

end alu_tb;

architecture alu_tb_stimulus of alu_tb is

component Arithmetic_Logic_Unit is

    generic (N: positive := 4);

    Port (

    A,B: in std_logic_vector(N-1 downto 0);

    Function_Select: in std_logic_vector (N-1 downto 0);

    Overflow: out std_logic;

    R: out std_logic_vector(N-1 downto 0));

end component;

-- Stimulus signals initialized to '0'

signal A_s, B_s,functionSelect, output_s: std_logic_vector(3 downto 0) := "0000";

signal overflow_s: std_logic := '0';

begin

-- Instantiate a Unit Under Test (UUT) and connect to internal stimuli

    UUT: Arithmetic_Logic_Unit

    port map(

       A => A_s,

       B => B_s,

       Function_Select => functionSelect,

       Overflow => overflow_s,

       R => output_s );

       functionSelect <= "0000" after 0ns, "0001" after 10ns, "0011" after 20ns, "0010" after 30ns, "0100" after 40ns, "0101" after 50ns,

       "0110" after 60ns, "0111" after 70ns, "1001" after 80ns, "1000" after 90ns, "1010" after 100ns, "0000" after 110ns;


       A_s <= "0000", "1001" after 0ns, "1001" after 10ns, "0010" after 20ns, "0101" after 30ns, "0101" after 40ns, "0101" after 50ns,

       "1010" after 60ns, "1011" after 70ns, "1011" after 80ns, "1101" after 90ns, "1101" after 100ns, "0000" after 110ns;

       B_s <= "0000", "1010" after 0ns, "0010" after 10ns, "0101" after 20ns, "1101" after 30ns, "1101" after 40ns, "1101" after 50ns, "0000" after
60ns;

end alu_tb_stimulus;
```