City College of New York

Final Test Project

Zi Xuan Li

CSC 211 Fall 2022

Professor Izidor Gertner

December 11th, 2022

Objective:

The objective of this project is to build and simulate an arithmetic logic unit (ALU), more specifically an adder/subtractor module that can perform addition, subtraction, and accumulate using the 2's complement number representation. The goal is to simulate the addition and subtraction of two 32-bit integers as well as perform the accumulation operation while using addition and subtraction. Along with this, an additional objective is to code a negative flag and zero flag in VHDL that will output 1 when the output of the circuit is negative and zero, respectively. The adder/subtractor module will also be connected to RAM1PORTs that function like SRAMs to read and write data into the adder/subtracter module. The main goal of the lab is to simulate 7 different actions in the VWF file.

Functionality, Specifications, and Simulation:

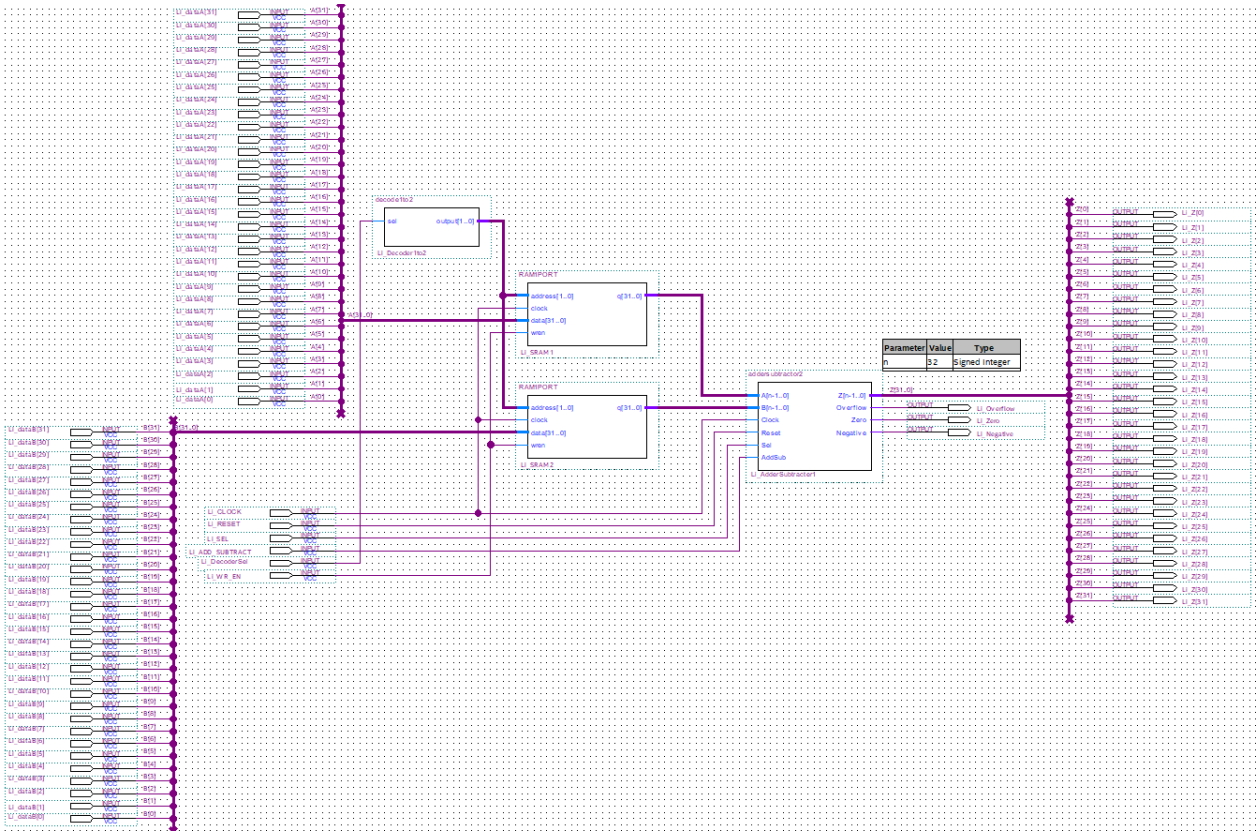

Figure 1: BDF file of Adder/Subtractor with SRAM circuit diagram.

The BDF file contains nothing special, but it does demonstrate all the inputs and outputs that will be seen in the VWF file. There are 2 vector inputs; Li_dataA[31..0] and Li_dataB[31..0] as well as 6 standard inputs; Li_CLOCK, Li_RESET, Li_SEL, Li_ADD_SUBTRACT, Li_DecoderSel, and Li_WR_EN. Additionally there is 1 vector output Li_Z[31..0] and 3 standard outputs Li_Overflow, Li_Zero, Li_Negative.

```vhdl
LIBRARY ieee;
USE ieee.std_logic_1164.all;

LIBRARY altera_mf;
USE altera_mf.altera_mf_components.all;

ENTITY RAM1PORT IS
    PORT
    (
        address     : IN STD_LOGIC_VECTOR (1 DOWNTO 0);
        clock       : IN STD_LOGIC  := '1';
        data        : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
        wren        : IN STD_LOGIC ;
        q           : OUT STD_LOGIC_VECTOR (31 DOWNTO 0)
    );
END RAM1PORT;


ARCHITECTURE SYN OF ram1port IS

    SIGNAL sub_wire0  : STD_LOGIC_VECTOR (31 DOWNTO 0);

BEGIN
    q    <= sub_wire0(31 DOWNTO 0);

    altsyncram_component : altsyncram
    GENERIC MAP (
        clock_enable_input_a => "BYPASS",
        clock_enable_output_a => "BYPASS",
        intended_device_family => "Cyclone V",
        lpm_hint => "ENABLE_RUNTIME_MOD=NO",
        lpm_type => "altsyncram",
        numwords_a => 4,
        operation_mode => "SINGLE_PORT",
        outdata_aclr_a => "NONE",
        outdata_reg_a => "CLOCK0",
        power_up_uninitialized => "FALSE",
        read_during_write_mode_port_a => "NEW_DATA_NO_NBE_READ",
        widthad_a => 2,
        width_a => 32,
        width_byteena_a => 1
    )
    PORT MAP (
        address_a => address,
        clock0 => clock,
        data_a => data,
        wren_a => wren,
        q_a => sub_wire0
    );


END SYN;
```

Figure 2: VHDL code of the component RAM1PORT used in Figure 1.

This VHDL code is generated by Quartus when creating a 2 x 32 RAM1PORT from the Quartus library. This code has the same functionality as a 2 x 32-bit SRAM and will be used to read and write 32-bit integers. No notable elements of this VHDL code were changed before symbol creation to fulfill its role in the project.

```
1   library IEEE;
2   use IEEE.STD_LOGIC_1164.all;
3
4   entity decode1to2 is
5    port(
6     sel : in std_logic;
7     output : out std_logic_vector(1 downto 0));
8   end decode1to2;
9
10  architecture arch of decode1to2 is
11  begin
12   with sel select
13    output <= "01" when '0',
14    "10" when '1',
15    "00" when others;
16  end arch;
```

Figure 3: VHDL code of the component Decoder1to2 used in Figure 1.

I wrote this VHDL code in order to be able to select the address of the address[1..0] vector input of the 2 x 32 RAM1PORT in Figure 2. The VHDL code works as a 1 to 2 decoder, taking a single input and generating 2 lines of output.

```vhdl
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

-- Top-level entity
ENTITY addersubtractor2 IS
    GENERIC ( n : INTEGER := 32 ) ;
    PORT (A, B                  : IN STD_LOGIC_VECTOR(n-1 DOWNTO 0) ;
          Clock, Reset, Sel, AddSub  : IN STD_LOGIC ;
          Z                     : BUFFER STD_LOGIC_VECTOR(n-1 DOWNTO 0) ;
          Overflow              : OUT STD_LOGIC  ;
          Zero                  : OUT STD_LOGIC  ;
          Negative              : OUT STD_LOGIC ) ;
END addersubtractor2 ;

ARCHITECTURE Behavior OF addersubtractor2 IS
    SIGNAL G, M, Areg, Breg, Zreg : STD_LOGIC_VECTOR(n-1 DOWNTO 0) ;
    SIGNAL SelR, AddSubR, over_flow, zero_flag, negative_flag : STD_LOGIC ;

    COMPONENT mux2to1
        GENERIC ( k : INTEGER := 32 ) ;
        PORT ( V, W : IN STD_LOGIC_VECTOR(k-1 DOWNTO 0) ;
               Selm     : IN STD_LOGIC ;
               F        : OUT STD_LOGIC_VECTOR(k-1 DOWNTO 0) ) ;
    END COMPONENT ;

    COMPONENT megaddsub
        PORT (add_sub     : IN STD_LOGIC ;
              dataa, datab   : IN STD_LOGIC_VECTOR(31 DOWNTO 0) ;
              result         : OUT STD_LOGIC_VECTOR(31 DOWNTO 0) ;
              overflow       : OUT STD_LOGIC ) ;
    END COMPONENT ;

BEGIN
-- Define flip-flops and registers
    PROCESS ( Reset, Clock )
    BEGIN
        IF Reset = '1' THEN
            Areg <= (OTHERS => '0'); Breg <= (OTHERS => '0');
            Zreg <= (OTHERS => '0'); SelR <= '0'; AddSubR <= '0'; Overflow <= '0';
        ELSIF Clock'EVENT AND Clock = '1' THEN
            Areg <= A; Breg <= B; Zreg <= M;
            SelR <= Sel; AddSubR <= NOT AddSub; Overflow <= over_flow; Zero <= zero_flag; Negative <= negative_flag;
        END IF ;
        IF (Z(Z'left) = '1') THEN
            Negative <= '1';
        ELSE
            Negative <= '0';
        END IF;

        IF (Z = "00000000000000000000000000000000") THEN
            Zero <= '1';
        ELSE
            Zero <= '0';
        END IF;
    END PROCESS ;
-- Define combinational circuit
    nbit_addsub: megaddsub
        PORT MAP ( AddSubR, G, Breg, M, over_flow ) ;
    multiplexer: mux2to1
        GENERIC MAP ( k => n )
        PORT MAP ( Areg, Z, SelR, G ) ;
    Z <= Zreg ;
END Behavior;

-- k-bit 2-to-1 multiplexer
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY mux2to1 IS
    GENERIC ( k : INTEGER := 32 ) ;
    PORT ( V, W : IN STD_LOGIC_VECTOR(k-1 DOWNTO 0) ;
           Selm : IN STD_LOGIC ;
           F : OUT STD_LOGIC_VECTOR(k-1 DOWNTO 0) ) ;
END mux2to1 ;

ARCHITECTURE Behavior OF mux2to1 IS
BEGIN
    PROCESS ( V, W, Selm )
    BEGIN
        IF Selm = '0' THEN
            F <= V ;
        ELSE
            F <= W ;
        END IF ;
    END PROCESS ;
END Behavior ;

-- 16-bit adder/subtractor LPM created by the MegaWizard
LIBRARY ieee;
```

Figure 4: VHDL code of the Adder/Subtracter component used in Figure 1 (part a).

```
90   USE ieee.std_logic_1164.all;
91   LIBRARY lpm;
92   USE lpm.lpm_components.all;
93
94   ENTITY megaddsub IS
95      PORT ( add_sub : IN STD_LOGIC ;
96         dataa : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
97         datab : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
98         result : OUT STD_LOGIC_VECTOR (31 DOWNTO 0);
99         overflow : OUT STD_LOGIC );
100  END megaddsub;
101
102  ARCHITECTURE SYN OF megaddsub IS
103     SIGNAL sub_wire0 : STD_LOGIC ;
104     SIGNAL sub_wire1 : STD_LOGIC_VECTOR (31 DOWNTO 0);
105     COMPONENT lpm_add_sub
106     GENERIC ( lpm_width : NATURAL;
107        lpm_direction : STRING;
108        lpm_type : STRING;
109        lpm_hint : STRING );
110     PORT ( dataa : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
111        add_sub : IN STD_LOGIC ;
112        datab : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
113        overflow : OUT STD_LOGIC ;
114        result : OUT STD_LOGIC_VECTOR (31 DOWNTO 0) );
115     END COMPONENT;
116  BEGIN
117     overflow <= sub_wire0;
118     result <= sub_wire1(31 DOWNTO 0);
119     lpm_add_sub_component : lpm_add_sub
120     GENERIC MAP ( lpm_width => 32,
121        lpm_direction => "UNUSED",
122        lpm_type => "LPM_ADD_SUB",
123        lpm_hint => "ONE_INPUT_IS_CONSTANT=NO,CIN_USED=NO" )
124     PORT MAP ( dataa => dataa,
125        add_sub => add_sub,
126        datab => datab,
127        overflow => sub_wire0,
128        result => sub_wire1 );
129  END SYN;
130
```

Figure 5: VHDL code of the Adder/Subtracter component used in Figure 1 (part b).

The VHDL code above is built following the instructions from the Intel FPGA, “Using Library Modules in VHDL Designs” pdf. It includes the mandatory adder and subtracter unit as required by project guidelines. In this code, I changed the size of the vector inputs k to 32 or 31..0 in multiple of the areas in the code to follow project guidelines. I also added the negative & zero flag for when the results are negative and zero respectively into the VHDL code. The code can be found on lines 44 to lines 54.

```
44    IF (Z(Z'left) = '1') THEN
45       Negative <= '1';
46    ELSE
47       Negative <= '0';
48    END IF;
49
50    IF (Z = "00000000000000000000000000000000") THEN
51       Zero <= '1';
52    ELSE
53       Zero <= '0';
54    END IF;
```

Figure 6: VHDL code of the negative and zero flag.

The VHDL code I wrote is relatively simple, it checks whether the msb (most significant bit) of the result Z is 1, and if it is 1 sets the negative flag to 1 and if not, sets the negative flag to 0. This works because the adder/subtracter module uses the 2’s complement number representation, and when the msb is 1, it means the integer is negative. The zero-flag works because when the result Z is 32 0s, the output is literally 0, so the zero-flag is set to 1; otherwise, the zero-flag is set to 0.

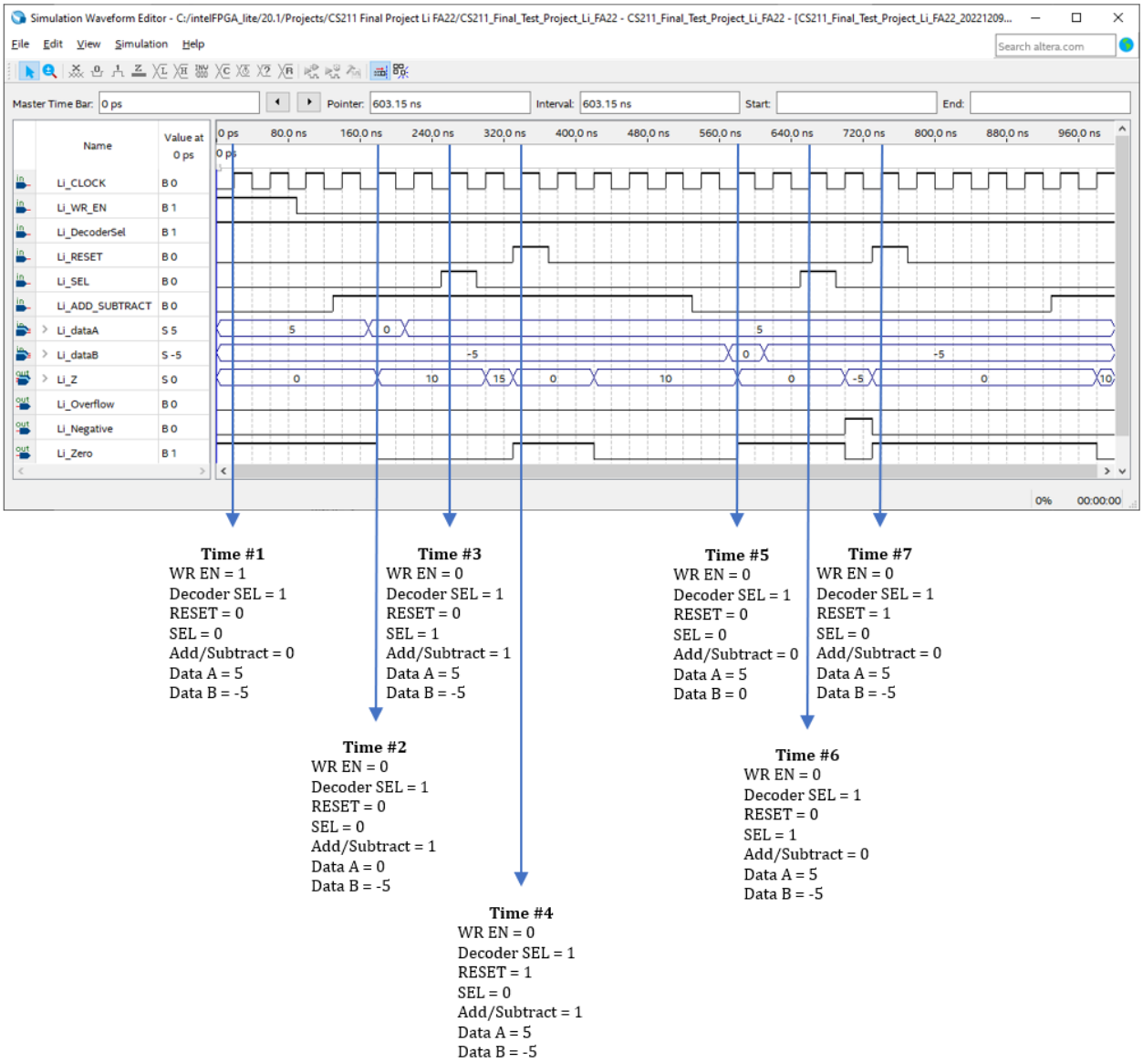

Figure 7: VWF file of Adder/Subtractor with SRAM in Figure 1.

The project specifically asks that 1) we preload two 32-bit integers into the SRAM, 2) Load the first integer from SRAM into register A, 3) Load the second integer from SRAM into register B, and 4a) Add registers A plus B and store the result in Z, 4b) Subtract registers B from A and store the result in register Z, 4c) Perform accumulation operation using ADD, and 4d) Perform accumulation operation using SUB.

To understand if we're doing these correctly, we need to first understand that the entire circuit is connected to the input clock. This means that by nature, the entire circuit will function accordingly to the clock. This is part of the reason why in Figure 7, Time #1..7 does not include the value of the input Li_CLOCK. The other reason is because we are measuring the values of each input during the HIGH state of the clock signal. Therefore, it is important to note, that output Z only changes during the next cycle of the clock. With that being said, we need to understand the functionality of the input and output signals in

the waveform. The outputs in the circuit are Li_Z[31..0], Li_Overflow, Li_Negative, and Li_Zero. Li_Z[31..0] is a vector output that reads the sum or difference of the operation performed by the adder/subtractor module. Li_Overflow is a flag that outputs 1 when overflow has occurred in the operation else it outputs 0. Li_Negative is another flag that outputs 1 when Z is a negative value else it outputs 0. Li_Zero is yet another flag that outputs 1 when Z is equal to 0 else it outputs 0. To explain the inputs, it is easier to divide the inputs into their corresponding components.

For the component “Decoder1to2”, the input that corresponds to this component is Li_DecoderSel. This input allows the “Decoder1to2” component to create 2 output signals that will be connected to the address input on the “Ram 1 Port” component, which is used to determine which row of the SRAM will be turned on to read or write data.

For the component “Ram1Port”, the inputs that correspond to this component are Li_dataA[31..0], Li_dataB[31..0] and Li_WR_EN. The two vector inputs are used to load data in the SRAM so that it can be read out later. The write enable input is used to allow the SRAM to write the data into memory and store said data.

For the component “AdderSubtractor”, the inputs that correspond to this component are Li_RESET, Li_SEL, and Li_ADD_SUBTRACT. The reset input makes the adder/subtractor module set the output Z to 0. The Li_SEL input allows the module to perform accumulation or regular arithmetic depending on the state of the Li_SEL signal. When Li_SEL = 1, the operation that is performed is called accumulation, which is Z = previous Z + B, or Z = previous Z – B. When Li_SEL = 0, the operation that is performed is Z = A + B or Z = A – B. Whether the module performs the accumulation addition or accumulation subtraction, or regular addition or regular subtraction is entirely dependent on the input signal Li_ADD_SUBTRACT. This signal allows the module to perform addition when Li_ADD_SUBTRACT = 0 and subtraction when Li_ADD_SUBTRACT = 1.

TO SIMULATE

The first 3 requirements (1,2,3) are performed in Time #1. During the high state of the clock, write enable and decoder select are both high. This means that the data read from data A and data B are being loaded into the SRAM. Afterwards, because write enable is high, the data in the SRAM is being loaded into the register A and register B on the adder/subtracter component. The integer that is loaded into register A is 5 and register B is -5.

The requirement (4.a) is performed at Time #5. During the high state of this time stamp, write enable = 0, decoder select = 1, reset = 0, select = 0, and add/subtract = 0. This allows the circuit to perform addition, specifically regular addition where the output Z = A + B because add/subtract = 0 and select = 0. Although data A and B are reset in Time#4, data A reads 5, and data B reads 0, because decoder select is 1, 5 is being loaded in register A and -5 is being loaded into register B from RAM1PORT every clock cycle since RAM1PORT never stored the change of data A to 0 or B to 0 because write enable was 0 after the first 90ns. Therefore, Z = 0 at the next clock cycle, because 5 + (-5) = 0.

The requirement (4.b) is performed at Time #2. During the high state of this time stamp, write enable = 0, decoder select = 1, reset = 0, select = 0, and add/subtract = 1. This allows the circuit to perform subtraction, specifically regular subtraction where the output Z = A – B because add/subtract = 1 and select = 0. Similarly, to the explanation given in (4.a), even though data A reads 0 and data B reads -5, the integer stored in register A is 5 and register B is -5. Therefore, Z = 10 at the next clock cycle, because 5 – (-5) = 10.

The requirement (4.c) is performed at Time #6. During the high state of this time stamp, write enable = 0, decoder select = 1, reset = 0, select = 1, and add/subtract = 0. This allows the circuit to perform addition, specifically accumulation addition where output Z = previous Z + B because add/subtract = 0 and select = 1. Here, the previous Z has a value of 0, and B has a value of -5. Therefore, Z = -5 at the next clock cycle, because 0 + (-5) = -5.

The requirement (4.d) is performed at Time #3. During the high state of this time stamp, write enable = 0, decoder select = 1, reset = 0, select = 1, and add/subtract = 1. This allows the circuit to perform subtraction, specifically accumulation subtraction where output Z = previous Z - B because add/subtract = 1 and select = 1. Here, the previous Z has a value of 10, and B has a value of -5. Therefore Z = 15 at the next clock cycle, because 10 – (-5) = 15.

You can also see that the negative and zero flags are working as intended because the output Li_Zero is 1 when Z is 0 and Li_Negative is 1 when Z is -5. Additionally, I included Time#4 and Time#7 to demonstrate the output Z is 0 when the reset input is 1.

Conclusions:

In this project, I learned about the functionality of the Intel FPGA Adder/Subtracter circuit. This includes the purpose of its inputs and outputs. The circuit includes 6 inputs; A[31..0], B[31..0], Clock, Reset, Sel, Add Sub and 4 outputs; Z[31..0], Overflow, Zero, and Negative. The two vector inputs A[31..0] and B[31..0] are the 32-bit integers that are used to perform the different operations in this project. The input “clock” is to enable all the functionalities of the circuit. The input “reset” sets both dataA and dataB to 0. The input “sel” controls the operation A + B/A – B when sel = 0 and Z = Z – B when sel = 1. The input “addsub” controls whether the circuit adds or subtracts the two integers. The vector output Z[31..0] is the result of the arithmetic operation. Outputs “overflow”, “zero”, and “negative” are flags for the circuit to display whether the result is overflow, zero or negative. In addition to this, I was able to deepen my understanding and knowledge of the topic of SRAM and its functionalities. I also learned some VHDL syntax as well as some new functions.