CEG3156 | Spring 2024



**GROUP 10**

Laboratory #1: Floating Point Multiplication

Course Professor: **Dr. Rami Abielmona**

|  |  |
| --- | --- |
| Name | Student ID |
| Josue Dazogbo | 300258449 |
| Christopher King | 300226522 |

# Introduction

The purpose of this lab is to design, implement and test a floating-point adder and multiplier unit in VHDL. These components will be designed using the ASM method involving separate data paths and control paths using one flip fop per state. The ASM design will be based off the algorithms provided for addition and multiplication. Testing of the adder and multiplier unit will be done using predetermined hand calculations and comparing the result of the hand calculations to the adder and multiplier output. This lab will help strengthen understanding of floating-point representation and floating-point arithmetic.

# Objectives

### Main Objective

The primary objective of this laboratory is to design and build a floating-point multiplier in VHDL. This involves understanding the behavior of floating-point arithmetic, implementing the necessary hardware components, and verifying their functionality through simulation and on a physical FPGA board.

### Learned Objectives

Throughout the course of this lab, the following objectives were achieved:

* **Design and Implementation of a Floating-Point Adder**: Designed, realized, and tested a floating-point adder unit, gaining hands-on experience with floating-point addition operations.
* **Design and Implementation of a Floating-Point Multiplier**: Designed, realized, and tested a floating-point multiplier unit, learning the principles of floating-point multiplication.
* **Understanding of Floating-Point Arithmetic**: Developed a comprehensive understanding of floating-point arithmetic, including the IEEE standard for floating-point representation, normalization, and the handling of special cases such as overflow and underflow.
* **VHDL Coding and Simulation**: Students practiced writing VHDL code at the structural level, ensuring a deeper understanding of Register Transfer Logic (RTL) design. They also simulated their designs to verify correctness against theoretical calculations.
* **FPGA Implementation and Testing**: Students implemented their designs on a Cyclone FPGA board, using push-buttons to input floating-point numbers and LEDs to display the output. This provided practical experience with hardware design and debugging.
* **Problem-Solving and Troubleshooting**: Through the process of design, simulation, and physical testing, students enhanced their problem-solving and troubleshooting skills, learning how to overcome various design obstacles.

By completing these tasks, we not only met the main objective of the lab but also gained valuable skills and knowledge that will be applicable in future digital logic problems.

# Problem and Solution Discussion

In this laboratory, the task is to design and implement a floating-point multiplier using VHDL. This involves understanding the floating-point representation as defined by the IEEE standard and implementing the arithmetic operations required for multiplication.

Floating-point numbers are used to represent real numbers that can have a wide range of values, including very small fractions and very large numbers. The challenge lies in accurately performing arithmetic operations while adhering to the specific format of floating-point representation.

To implement the floating point addition circuit, we first based our design on the algorithm provided in the lab manual

A diagram of a number of numbers

Description automatically generated

Figure 1: Floating Point Addition Flowchart

We then derived the appropriate ASM Charts (ASM Chart, Datapath, Detailed Chart and Control Path) for a possible adder. Since our design was for an 8-bit mantissa, a 7 bit exponent, we used 9 bit registers to hold both mantissa (with the implicit 1) and the exponent.

A screenshot of a computer

Description automatically generated

Figure 2: Floating point addition ASM chart

A diagram of a computer system

Description automatically generated with medium confidence

Figure 3: Floating point Adder Data Path

A screenshot of a computer flowchart

Description automatically generated

Figure 4: Floating point addition detailed ASM chart

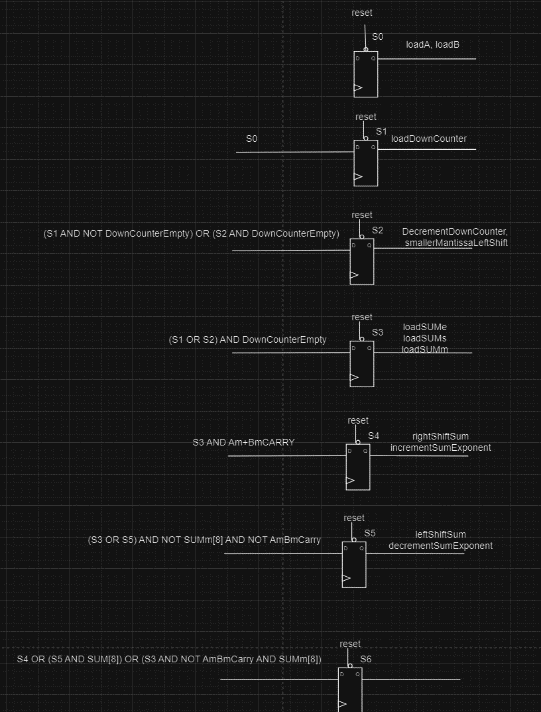


Figure 5: Control path for floating point adder

The same process was followed for the creation of the Floating Point Multiplier. The flow chart for multiplication was derived from the lecture slides and from online research

A diagram of a product

Description automatically generated

Figure 6: Floating Point Multiplier Flow Chart

A diagram of a product

Description automatically generated

Figure 7: Floating Point Multiplier ASM Chart

A diagram of a machine

Description automatically generated

Figure 8: Floating Point Multiplier Data Path

A diagram of a flowchart

Description automatically generated

Figure 9: Floating Point Multiplier Detailed Chart

A diagram of a computer system

Description automatically generated with medium confidence

Figure 10: Floating Point Multiplier Control Path A close-up of a computer circuit

Description automatically generated

Figure 11: Multiplier Final Circuit

# Verification

### General Purpose Register

The Nine-Bit General-Purpose Register (NineBitGPRegister) described in the VHDL code is designed to support multiple operations including loading values, shifting left and right, incrementing, decrementing, and serial data input. It includes several ports: i\_resetBar for active-low reset, i\_load to load a new value, i\_shiftLeft and i\_shiftRight for shifting, i\_decrement and i\_increment for decrementing and incrementing, i\_serial\_in\_left and i\_serial\_in\_right for serial inputs, i\_clock for synchronization, i\_Value for input values, and o\_Value for output. Internal signals like d\_in, q\_out, and enable\_reg manage data input and state, while operation\_selectors and operation\_decoder manage operation selection. Key components include enARdFF\_2 D flip-flops, EightToOneMux multiplexers, EigthToThreeEncoder, and NineBitAdderSubtractor for incrementing and decrementing. The enable\_reg activates the flip-flops for any operation, while operation\_decoder determines the operation. Multiplexers select the input for each bit, and D flip-flops update the register state. The output o\_Value represents the register's current value. This design allows the register to efficiently perform various operations based on control signals.

A screenshot of a computer

Description automatically generated

Figure : General Purpose in Load, Shift Left, Right Shift and Increment Modes

### Unsigned Multiplier Logic

The MultiplierFinal VHDL component performs multiplication of two 9-bit inputs (IN1, IN2), producing an 18-bit product (Product). The component is driven by a clock signal (CLK) and an asynchronous reset signal (async\_reset\_bar). The architecture includes two main components: MultiplierCP (Control Path) and MultiplierDP (Data Path). The MultiplierCP manages control signals, such as left and right shifts (LSHFTA, RSHFTB), load signals (LDA, LDB), increment and clear instructions (INCI, CLRI), and product-related signals (LDP, CLRP). It receives inputs from the control path (B0IS1, ILT7) and synchronizes operations using the clock and reset signals. The MultiplierDP performs the actual data manipulation, using the control signals to shift, load, increment, and clear data as needed. It outputs the status signals (ILT7, B0IS1) back to the control path and produces the final product (Product). The internal signals (LSHFTA, RSHFTB, LDA, LDB, INCI, CLRI, LDP, CLRP, B0IS1, ILT7) facilitate communication between the data path and control unit, ensuring accurate and efficient multiplication.

A screenshot of a computer

Description automatically generated

Figure : Multiplier Test Bench Simulation Result

Our value here is incorrect as the shift register connected to the output shifts the value by 2 over the normal expected behavior. Its final value should be 1101 instead of 110100.

### Floating Point Adder Logic

The FloatingPointAdder VHDL component facilitates the addition of two floating-point numbers. It comprises input ports for clock (i\_clock) and reset (i\_reset), sign bits (i\_signA, i\_signB), mantissas (i\_mantissaA, i\_mantissaB), and exponents (i\_exponentA, i\_exponentB). Output ports include the result sign bit (o\_sign), overflow indicator (o\_overflow), resulting mantissa (o\_mantissa), and resulting exponent (o\_exponent). Internally, the component features an AdderDataPath for performing the arithmetic operations and an AdderControlUnit for managing control signals. The AdderDataPath component receives the input signs, mantissas, and exponents, along with control signals, and outputs the result. The AdderControlUnit orchestrates the addition process by generating control signals based on the current state and inputs. Signals such as int\_SHFTM, int\_LDDC, int\_DECDC, int\_LDAM, int\_LDBM, int\_LDSM, int\_LSHFTM, int\_RSHFTM, int\_LDSE, int\_INCSE, int\_DECSE, int\_CLRS, int\_LDAS, int\_DCEMT, int\_MantissaCarry, and int\_MantissaSumMSB are used for internal communication between the control unit and data path. This setup ensures efficient management of floating-point addition, handling normalization, alignment, and rounding of the mantissas while keeping track of the sign and exponent adjustments necessary for the final output.

A screenshot of a computer

Description automatically generated

Figure Floating point adder data path simulation

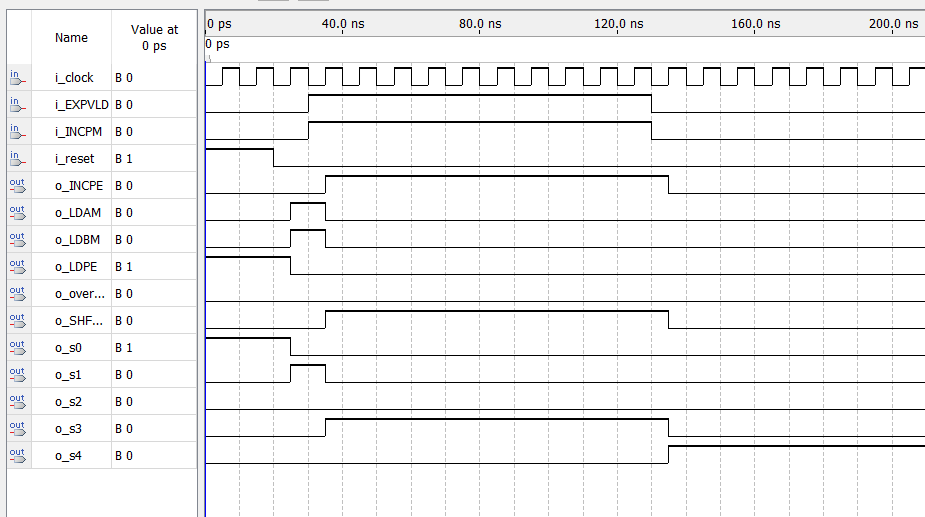


Figure Floating point adder control path simulation

Without Overflow

A screenshot of a computer screen

Description automatically generated

Figure : Floating point adder simulation using hand calculations

The values output by the Floating Point Adder **matched our expected results**.

With Overflow

A screenshot of a computer

Description automatically generated

Figure : Floating point adder simulation using hand calculations with overflow

Result is as expected (and shown in Appendix – Example Calculations) however, the overflow flag is not working correctly.

### Floating Point Multiplier

The FloatingMultiplier VHDL component performs the multiplication of two floating-point numbers. It accepts inputs for clock (i\_clock), reset (i\_reset), sign bits (i\_signA, i\_signB), mantissas (i\_mantissaA, i\_mantissaB), and exponents (i\_exponentA, i\_exponentB), and outputs the resulting sign bit (o\_sign), overflow indicator (o\_overflow), mantissa (o\_mantissa), and exponent (o\_exponent). The architecture includes two main components: MultiplierDataPath and MultiplierControlUnit. The MultiplierDataPath handles arithmetic operations, receiving inputs and producing the resulting sign, mantissa, exponent, and overflow signal, while also generating control signals (o\_INCPM and o\_EXPVLD) indicating the multiplication process status. The MultiplierControlUnit manages control logic, receiving clock, reset, and status signals from the data path and producing control signals (o\_LDPE, o\_LDAM, o\_LDBM, o\_overflow, o\_SHFTPM, o\_INCPE, and o\_LDPM) to coordinate multiplication steps, such as loading data, shifting mantissas, and adjusting exponents. Internal signals (INCPM, EXPVLD, LDPE, LDAM, LDBM, overflow, SHFTPM, INCPE, and LDPM) facilitate communication between the data path and control unit, ensuring synchronized operations. This component effectively aligns exponents, multiplies mantissas, adjusts the resulting exponent, and handles overflows, providing efficient and accurate floating-point multiplication suitable for various computational applications.

A screen shot of a computer

Description automatically generated

Figure : Floating Point Multiplier Control Path

A screenshot of a computer

Description automatically generated

Figure : Floating Point Multiplier Output

As we can see from this screenshot, the multiplier mantissa output does not work. Since the multiplier used for the circuit was combinational, we should have created a delay reset signal that would start the multiplier circuit a bit later than the rest of the design to let the register values update first. Other than that, the exponent part of the multiplier works other than the renormalization step (See Appendix, Multiplier – Case Without Overflow).

# Demo

## Top Level Entity

Due to limitations of the DE2-115 board, it was not possible to input both 16-bit floating point numbers at the same time. As such, a top-level entity was created containing 2 registers that could be loaded with only 16 of the switches and 2 push buttons to select the register to load to. This top-level entity was then simulated to verify its functionality.

A screenshot of a computer

Description automatically generated

Figure Simulation of top-level adder entity

## Pin Assignments

Pine assignments were completed using a file provided with previous labs. 16 switches were used to enter the 16-bit operand values, 2 push buttons were used to set the values of A and B, and 2 more push buttons were used to reset the A and B registers and to reset the adder ASM.

A screenshot of a computer

Description automatically generated

Figure Pin assignments on DE2 board

# Discussion

The final floating point adder design did not run successfully on the Altera board. Despite numerous attempts, no output was correctly displayed on the board with multiple different inputs. All simulations of the floating-point adder including the top level entity were working correctly and therefore it is likely that there was an error in the pin mappings. Multiple sources listed different pin mappings for the board. The table provided with the lab manual could not be used due to errors within Quartus.

### Discussion of Challenging Problems

The floating point multiplier was not successfully simulated and was not tested on the board. These issues were due to the integer multiplier not working correctly as seen in figure 12. This prevents the mantissa portion from being multiplied correctly and provides an incorrect answer. Despite this we were able get the exponent portion of the result working correctly. With lower time constraints, the integer multiplier could have been fixed and the full multiplier could have been functional.

We also had issues with the pin assignment in Quartus. We assigning pins to the board but the Floating Point adder was only successfully loaded on the board once where the LEDs displayed a result. Any subsequent attemps were fruitless. We tried using multiple files for the pin assignment (CEG 3155 File, File found on the internet, etc.) but none of those yielded a consistent result.

# Appendix

## Example calculations

### Adder – Case Without Overflow

A = 6.5 B = 10.25

A = 110.1

= 1.101 \* 2^2

= 1+.10100000 \* 2^65-63

A = 0 1000001 10100000

B =1010.01

= 1.01001 \* 2^3

= 1+.01001 \* 2^66-63

B = 0 1000010 01001000

A + B:

6.5 + 10.25 = 16.75

A = 0 1000001 10100000

B = 0 1000010 01001000

1. Shift A mantissa to the right by 1
2. Add mantissas

A 0 .110100000

B 1. 01001000

-------------------

10.00011000

1. Since carry is set, shift right by one and increment exponent

1.00001100 exp = 66 + 1 = 67

Final solution = 0 1000011 00001100

### Adder – Case With Overflow

A = 1 1111111 11111111

B = 1 1111110 11111111

1. Shift B mantissa to the right by 1
2. Add mantissas

1.11111111

0. 1111111

--------------

10.01111110

3. Normalize sum by shifting to the right and increasing exponent by 1

Mantissa: 1.00111111 Exponent: 0000000!!!

Exponent overflow!

Final solution: 1 0000000 00111111

### Multiplier – Case Without Overflow

A = 6.5 B = 10.25

A = 110.1

= 1.101 \* 2^2

= 1+.10100000 \* 2^65-63

A = 0 1000001 10100000

B =1010.01

= 1.01001 \* 2^3

= 1+.01001 \* 2^66-63

B = 0 1000010 01001000

A\*B = 6.5\*10.25 = 66.625

1. Add exponents and subtract bias

1000001 + 1000010 – 0111111 = 1000100

1. Multiply mantissa

1.10100000 \* 1.01001000 = 01000010101.00000000

3. Normalize by shifting to the right 9 times and incrementing the exponent by 9

Mantissa: 1.00001010 Exponent 1001011

4. Since signs area both pos, result is pos

Final solution 0 1001011 00001010

### Multiplier – Case With Overflow

A = 1 1111111 11111111

B = 1 1111110 11111111

1. Add exponents and subtract bias

1111111 + 1111110 – 0111111 = 1 0111111 (Overflow already detected here b/c bit 8 is set)

1. Multiply mantissa

11111111 \* 11111111 = 011111110.00000001

1. Normalize by shifting right 7 times and incrementing exponent by 7

1.11111100 Exp = 1 0000110 Oh wow still overflow!

1. Signs are both neg, therefore output is positive

Final solution: 0 0000110 11111100

## VHDL Code

Top level FP adder for altera board:

library IEEE;

use IEEE.std\_logic\_1164.all;

entity AdderTop is

port(

i\_mantissa : IN STD\_LOGIC\_VECTOR(7 DOWNTO 0);

i\_exponent : IN STD\_LOGIC\_VECTOR(7 DOWNTO 0);

i\_resetAdder, i\_resetReg, i\_clock, i\_loadA, i\_loadB : IN STD\_LOGIC;

o\_sign, o\_overflow : OUT STD\_LOGIC;

o\_mantissa : OUT STD\_LOGIC\_VECTOR(7 DOWNTO 0);

o\_exponent : OUT STD\_LOGIC\_VECTOR(6 DOWNTO 0)

);

end AdderTop;

architecture rtl of AdderTop is

COMPONENT EightBitGPRegister

PORT

(

-- Register Operations

i\_resetBar : IN STD\_LOGIC;

i\_load, i\_shiftLeft, i\_shiftRight : IN STD\_LOGIC;

i\_decrement, i\_increment : IN STD\_LOGIC;

-- Register Signals

i\_serial\_in\_left, i\_serial\_in\_right : IN STD\_LOGIC;

i\_clock : IN STD\_LOGIC;

i\_Value : IN STD\_LOGIC\_VECTOR(7 DOWNTO 0);

o\_Value : OUT STD\_LOGIC\_VECTOR(7 DOWNTO 0)

);

END COMPONENT;

component FloatingPointAdder

PORT

(

i\_clock, i\_reset : IN STD\_LOGIC;

i\_signA, i\_signB : IN STD\_LOGIC;

i\_mantissaA, i\_mantissaB : IN STD\_LOGIC\_VECTOR(7 DOWNTO 0);

i\_exponentA, i\_exponentB : IN STD\_LOGIC\_VECTOR(6 DOWNTO 0);

o\_sign, o\_overflow : OUT STD\_LOGIC;

o\_mantissa : OUT STD\_LOGIC\_VECTOR(7 DOWNTO 0);

o\_exponent : OUT STD\_LOGIC\_VECTOR(6 DOWNTO 0)

);

end component;

signal int\_resetBAR : STD\_LOGIC;

signal int\_mantissaA, int\_mantissaB : STD\_LOGIC\_VECTOR(7 DOWNTO 0);

signal int\_exponentA, int\_exponentB : STD\_LOGIC\_VECTOR(7 DOWNTO 0);

begin

int\_resetBAR <= not i\_resetReg;

mantissaA : EightBitGPRegister

PORT MAP

(

i\_resetBar => int\_resetBAR,

i\_load => i\_loadA,

i\_shiftLeft => '0',

i\_shiftRight => '0',

i\_decrement => '0',

i\_increment => '0',

i\_serial\_in\_left => '0',

i\_serial\_in\_right => '0',

i\_clock => i\_clock,

i\_Value => i\_mantissa,

o\_Value => int\_mantissaA

);

exponentA : EightBitGPRegister

PORT MAP

(

i\_resetBar => int\_resetBAR,

i\_load => i\_loadA,

i\_shiftLeft => '0',

i\_shiftRight => '0',

i\_decrement => '0',

i\_increment => '0',

i\_serial\_in\_left => '0',

i\_serial\_in\_right => '0',

i\_clock => i\_clock,

i\_Value => i\_exponent,

o\_Value => int\_exponentA

);

mantissaB : EightBitGPRegister

PORT MAP

(

i\_resetBar => int\_resetBAR,

i\_load => i\_loadB,

i\_shiftLeft => '0',

i\_shiftRight => '0',

i\_decrement => '0',

i\_increment => '0',

i\_serial\_in\_left => '0',

i\_serial\_in\_right => '0',

i\_clock => i\_clock,

i\_Value => i\_mantissa,

o\_Value => int\_mantissaB

);

exponentB : EightBitGPRegister

PORT MAP

(

i\_resetBar => int\_resetBAR,

i\_load => i\_loadB,

i\_shiftLeft => '0',

i\_shiftRight => '0',

i\_decrement => '0',

i\_increment => '0',

i\_serial\_in\_left => '0',

i\_serial\_in\_right => '0',

i\_clock => i\_clock,

i\_Value => i\_exponent,

o\_Value => int\_exponentB

);

FPA:FloatingPointAdder

PORT MAP

(

i\_clock => i\_clock,

i\_reset => i\_resetAdder,

i\_signA => int\_exponentA(7),

i\_signB => int\_exponentB(7),

i\_mantissaA => int\_mantissaA,

i\_mantissaB => int\_mantissaB,

i\_exponentA => int\_exponentA(6 DOWNTO 0),

i\_exponentB => int\_exponentB(6 DOWNTO 0),

o\_sign => o\_sign,

o\_overflow => o\_overflow,

o\_mantissa => o\_mantissa,

o\_exponent => o\_exponent

);

end rtl;

Floating point adder (connection with datapath and control path)

LIBRARY ieee;

USE ieee.std\_logic\_1164.all;

ENTITY FloatingPointAdder is

PORT

(

i\_clock, i\_reset : IN STD\_LOGIC;

i\_signA, i\_signB : IN STD\_LOGIC;

i\_mantissaA, i\_mantissaB : IN STD\_LOGIC\_VECTOR(7 DOWNTO 0);

i\_exponentA, i\_exponentB : IN STD\_LOGIC\_VECTOR(6 DOWNTO 0);

o\_sign, o\_overflow : OUT STD\_LOGIC;

o\_mantissa : OUT STD\_LOGIC\_VECTOR(7 DOWNTO 0);

o\_exponent : OUT STD\_LOGIC\_VECTOR(6 DOWNTO 0);

o\_s0, o\_s1, o\_s2, o\_s3, o\_s4, o\_s5, o\_s6 : OUT STD\_LOGIC

);

END FloatingPointAdder;

ARCHITECTURE rtl of FloatingPointAdder is

COMPONENT AdderDataPath IS

PORT

(

-- Adder Input Signals

SignA, SignB : IN STD\_LOGIC;

MantissaA, MantissaB : IN STD\_LOGIC\_VECTOR(7 downto 0);

ExponentA, ExponentB : IN STD\_LOGIC\_VECTOR(6 downto 0);

GClock, GResetBAR : IN STD\_LOGIC;

-- Adder Output Signals

SignOut : OUT STD\_LOGIC;

MantissaOut : OUT STD\_LOGIC\_VECTOR(7 downto 0);

ExponentOut : OUT STD\_LOGIC\_VECTOR(6 downto 0);

Overflow : OUT STD\_LOGIC;

--Control Path Links

SHFTM : IN STD\_LOGIC;

LDDC, DECDC : IN STD\_LOGIC;

LDAM, LDBM : IN STD\_LOGIC;

LDSM, LSHFTM, RSHFTM : IN STD\_LOGIC;

LDSE, INCSE, DECSE : IN STD\_LOGIC;

CLRS, LDAS : IN STD\_LOGIC;

DCEMT, MantissaCarry, MantissaSumMSB : OUT STD\_LOGIC;

o\_register\_Am\_result, o\_register\_Bm\_result : OUT STD\_LOGIC\_VECTOR(8 DOWNTO 0)

);

END COMPONENT;

COMPONENT AdderControlUnit IS

PORT

(

i\_clock, i\_reset : IN STD\_LOGIC;

i\_downCounterEmpty, i\_mantissaCarry, i\_mantissaSumMSB : IN STD\_LOGIC;

o\_loadA, o\_loadB : OUT STD\_LOGIC;

o\_loadDownCounter, o\_decrementDownCounter : OUT STD\_LOGIC;

o\_smallerMantissaLeftShift : OUT STD\_LOGIC;

o\_loadSumE, o\_loadSumM, o\_loadSumS, o\_rightShiftSum, o\_incrementSumExponent, o\_leftShiftSum, o\_decrementSumExponent : OUT STD\_LOGIC;

o\_s0, o\_s1, o\_s2, o\_s3, o\_s4, o\_s5, o\_s6 : OUT STD\_LOGIC

);

END COMPONENT;

SIGNAL int\_SHFTM, int\_LDDC, int\_DECDC, int\_LDAM, int\_LDBM, int\_LDSM, int\_LSHFTM, int\_RSHFTM, int\_LDSE, int\_INCSE, int\_DECSE, int\_CLRS, int\_LDAS, i\_reset\_BAR : STD\_LOGIC;

SIGNAL int\_DCEMT, int\_MantissaCarry, int\_MantissaSumMSB : STD\_LOGIC;

SIGNAL int\_s0, int\_s1, int\_s2, int\_s3, int\_s4, int\_s5, int\_s6 : STD\_LOGIC;

SIGNAL int\_register\_Am\_result, int\_register\_Bm\_result : STD\_LOGIC\_VECTOR(8 DOWNTO 0);

BEGIN

i\_reset\_BAR <= not i\_reset;

dp : AdderDataPath

PORT MAP(

SignA => i\_signA,

SignB => i\_signB,

MantissaA => i\_mantissaA,

MantissaB => i\_mantissaB,

ExponentA => i\_exponentA,

ExponentB => i\_exponentB,

GClock => i\_clock,

GResetBAR => i\_reset\_BAR,

SignOut => o\_sign,

MantissaOut => o\_mantissa,

ExponentOut => o\_exponent,

Overflow => o\_overflow,

SHFTM => int\_SHFTM,

LDDC => int\_LDDC,

DECDC => int\_DECDC,

LDAM => '1',

LDBM => '1',

LDSM => int\_LDSM,

LSHFTM => int\_LSHFTM,

RSHFTM => int\_RSHFTM,

LDSE => int\_LDSE,

INCSE => int\_INCSE,

DECSE => int\_DECSE,

CLRS => int\_CLRS,

LDAS => int\_LDAS,

DCEMT => int\_DCEMT,

MantissaCarry => int\_MantissaCarry,

MantissaSumMSB => int\_MantissaSumMSB,

o\_register\_Am\_result => int\_register\_Am\_result,

o\_register\_Bm\_result => int\_register\_Bm\_result

);

cp : AdderControlUnit

PORT MAP

(

i\_clock => i\_clock,

i\_reset => i\_reset,

i\_downCounterEmpty => int\_DCEMT,

i\_mantissaCarry => int\_MantissaCarry,

i\_mantissaSumMSB => int\_MantissaSumMSB,

o\_loadA => int\_LDAM,

o\_loadB => int\_LDBM,

o\_loadDownCounter => int\_LDDC,

o\_decrementDownCounter => int\_DECDC,

o\_smallerMantissaLeftShift => int\_SHFTM,

o\_loadSumE => int\_LDSE,

o\_loadSumM => int\_LDSM,

o\_loadSumS => open,

o\_rightShiftSum => int\_RSHFTM,

o\_incrementSumExponent => int\_INCSE,

o\_leftShiftSum => int\_LSHFTM,

o\_decrementSumExponent => int\_DECSE,

o\_s0 => int\_s0,

o\_s1 => int\_s1,

o\_s2 => int\_s2,

o\_s3 => int\_s3,

o\_s4 => int\_s4,

o\_s5 => int\_s5,

o\_s6 => int\_s6

);

o\_s0 <= int\_s0;

o\_s1 <= int\_s1;

o\_s2 <= int\_s2;

o\_s3 <= int\_s3;

o\_s4 <= int\_s4;

o\_s5 <= int\_s5;

o\_s6 <= int\_s6;

END ARCHITECTURE;

Top level multiplier:

LIBRARY ieee;

USE ieee.std\_logic\_1164.all;

ENTITY FloatingMultiplier is

PORT

(

i\_clock, i\_reset : IN STD\_LOGIC;

i\_signA, i\_signB : IN STD\_LOGIC;

i\_mantissaA, i\_mantissaB : IN STD\_LOGIC\_VECTOR(7 DOWNTO 0);

i\_exponentA, i\_exponentB : IN STD\_LOGIC\_VECTOR(6 DOWNTO 0);

o\_sign, o\_overflow : OUT STD\_LOGIC;

o\_mantissa : OUT STD\_LOGIC\_VECTOR(7 DOWNTO 0);

o\_exponent : OUT STD\_LOGIC\_VECTOR(6 DOWNTO 0)

);

END FloatingMultiplier;

ARCHITECTURE rtl of FloatingMultiplier is

COMPONENT MultiplierControlUnit is

PORT

(

i\_clock, i\_reset : IN STD\_LOGIC;

i\_INCPM, i\_EXPVLD : IN STD\_LOGIC;

o\_LDPE, o\_LDAM, o\_LDBM, o\_overflow, o\_SHFTPM, o\_INCPE, o\_LDPM : OUT STD\_LOGIC

);

END COMPONENT;

COMPONENT MultiplierDataPath is

PORT

(

-- Multiplier Input Signals

SignA, SignB : IN STD\_LOGIC;

MantissaA, MantissaB : IN STD\_LOGIC\_VECTOR(7 downto 0);

ExponentA, ExponentB : IN STD\_LOGIC\_VECTOR(6 downto 0);

GClock, GReset : IN STD\_LOGIC;

-- Multiplier Output Signals

SignOut : OUT STD\_LOGIC;

MantissaOut : OUT STD\_LOGIC\_VECTOR(7 downto 0);

ExponentOut : OUT STD\_LOGIC\_VECTOR(6 downto 0);

Overflow : OUT STD\_LOGIC;

--Control Path Links

o\_INCPM, o\_EXPVLD : OUT STD\_LOGIC;

i\_LDPE, i\_LDAM, i\_LDBM, i\_overflow, i\_SHFTPM, i\_INCPE : IN STD\_LOGIC

);

END COMPONENT;

SIGNAL INCPM, EXPVLD, LDPE, LDAM, LDBM, overflow, SHFTPM, INCPE, LDPM : STD\_LOGIC;

BEGIN

dp : MultiplierDataPath

PORT MAP

(

-- Multiplier Input Signals

SignA => i\_signA,

SignB => i\_signB,

MantissaA => i\_mantissaA,

MantissaB => i\_mantissaB,

ExponentA => i\_exponentA,

ExponentB => i\_exponentB,

GClock => i\_clock,

GReset => i\_reset,

-- Multiplier Output Signals

SignOut => o\_sign,

MantissaOut => o\_mantissa,

ExponentOut => o\_exponent,

Overflow => o\_overflow,

--Control Path Links

o\_INCPM => INCPM,

o\_EXPVLD => EXPVLD,

i\_LDPE => LDPE,

i\_LDAM => LDAM,

i\_LDBM => LDBM,

i\_overflow => overflow,

i\_SHFTPM => SHFTPM,

i\_INCPE => INCPE

);

cp : MultiplierControlUnit

PORT MAP

(

i\_clock => i\_clock,

i\_reset => i\_reset,

i\_INCPM => INCPM,

i\_EXPVLD => EXPVLD,

o\_LDPE => LDPE,

o\_LDAM => LDAM,

o\_LDBM => LDBM,

o\_overflow => overflow,

o\_SHFTPM => SHFTPM,

o\_INCPE => INCPE,

o\_LDPM => LDPM

);

END ARCHITECTURE;

## All code:

All code from the Quartus project can be found here: <https://github.com/chriswkingg/ceg3156-lab/tree/main/Lab%201>