

University of Texas at Arlington

MavMatrix

Computer Science and Engineering Theses

Computer Science and Engineering Department

2023

DESIGN OF SINGLE PRECISION FLOATING POINT UNIT (32-BIT NUMBERS) ACCORDING TO IEEE 754 STANDARD USING VERILOG, AND CREATION OF AN EDUCATION MODEL FOR ADVANCED DIGITAL LOGIC AND DESIGN COURSES

Kartikey Sharan

Follow this and additional works at: https://mavmatrix.uta.edu/cse_theses



Part of the [Computer Sciences Commons](#)

Recommended Citation

Sharan, Kartikey, "DESIGN OF SINGLE PRECISION FLOATING POINT UNIT (32-BIT NUMBERS) ACCORDING TO IEEE 754 STANDARD USING VERILOG, AND CREATION OF AN EDUCATION MODEL FOR ADVANCED DIGITAL LOGIC AND DESIGN COURSES" (2023). *Computer Science and Engineering Theses*. 517. https://mavmatrix.uta.edu/cse_theses/517

This Thesis is brought to you for free and open access by the Computer Science and Engineering Department at MavMatrix. It has been accepted for inclusion in Computer Science and Engineering Theses by an authorized administrator of MavMatrix. For more information, please contact leah.mccurdy@uta.edu, erica.rousseau@uta.edu, vanessa.garrett@uta.edu.

© 2023 Kartikey Sharan

DESIGN OF SINGLE PRECISION FLOATING POINT UNIT (32-BIT NUMBERS)
ACCORDING TO IEEE 754 STANDARD USING VERILOG, AND CREATION OF AN EDUCATION
MODEL FOR ADVANCED DIGITAL LOGIC AND DESIGN COURSES

BY

KARTIKEY SHARAN



MASTER THESIS

Submitted in partial fulfilment of the requirements for the degree of
Master of Science in Computer Engineering in the Graduate College of the
University of Texas at Arlington, May 2024

Arlington, Texas

Advisor:

DR. BILL D CARROLL

The University of Texas at Arlington

Professor Computer Science and Engineering

Table of Contents

Table of Figures	VII
Acknowledgments	X
Abstract	XI
Chapter 1: Introduction	1
1.1 Floating Point Numbers	2
1.2 The IEEE 754 Standard	3
1.2.1 Overview.....	4
1.2.2 Binary Format Encodings.....	5
Exponent.....	5
Significand.....	6
1.2.3 Examples of Floating Point Representation.....	7
1.2.4 Floating Point Parameters.....	8
1.2.5 Range of Floating-Point Number.....	8
1.2.6 Rounding of Floating-Point Number.....	9
1.2.7 Data Types of 32 bit Floating-Point Number.....	10
1.3 Floating-Point Arithmetic	11
1.3.1 Exponent Overflow.....	11
1.3.2 Exponent Underflow.....	11
1.3.3 Significand Overflow.....	11
1.3.4 Significand Underflow.....	11
Chapter 2: 32-bits Floating Point Adder	12
2.1 Floating Point Addition Algorithm	13
2.1.1 Floating-Point Addition Example.....	14
2.2 Floating Point Adder Flowchart	16
2.3 Floating Point Adder Hardware	17
2.3.1 Floating Point Adder Hardware Architecture.....	18
2.3.2 Floating Point Adder Hardware Implementation.....	19
2.3.2.1 Sign Bit Calculation.....	19
2.3.2.2 Modular Exponent Subtractor.....	20
2.3.2.3 Mantissa Shifter Multiplexer.....	30
2.3.2.4 Mantissa Right Shifter.....	33
2.3.2.5 Mantissa Adder Multiplexer.....	35
2.3.2.6 Mantissa Carry Look Ahead Adder.....	36
2.3.2.7 Exponent Increment Multiplexer.....	41
2.3.2.8 Controlled Incrementor.....	42
2.3.2.9 Mantissa Normalizer.....	46
2.4 Floating Point Adder Results	47
2.4.1 Floating Point Adder Compilation Report.....	47
2.4.2 Floating Point Adder Testbench.....	48
2.4.3 Floating Point Adder Simulation Results.....	49
Case A:.....	49
Case B:.....	50
Case C:.....	51
Case D:.....	52
2.5 Conclusion	53

Chapter 3: Floating Point Subtractor	54
3.1 Floating Point Subtraction Algorithm	55
3.1.1 Floating-Point Subtraction Example	56
3.2 Floating Point Subtractor Flowchart	58
3.3 Floating Point Subtractor Hardware	59
3.3.1 Floating Point Subtractor Hardware Architecture	60
3.3.2 Floating Point Subtractor Hardware Implementation	61
3.3.2.1 Sign Bit Calculation	61
3.3.2.2 Modular Exponent Subtractor	62
3.3.2.3 Mantissa Shifter Multiplexer	62
3.3.2.4 Mantissa Right Shifter	64
3.3.2.5 Mantissa Subtractor Multiplexer	65
3.3.2.6 Mantissa Ripple Carry Subtractor	66
3.3.2.7 Exponent Decrement Multiplexer	69
3.3.2.8 Controlled Decrement	70
3.3.2.9 Mantissa Normalizer	75
3.4 Floating Point Subtractor Results	78
3.4.1 Floating Point Subtractor Compilation Report	78
3.4.2 Floating Point Subtractor Testbench	79
3.4.3 Floating Point Subtractor Simulation Results	80
Case A:	80
Case B:	81
Case C:	82
Case D:	83
3.5 Conclusion	84
Chapter 4: 32-bits Floating Point Multiplier	85
4.1 Floating Point Multiplication Algorithm	86
4.1.1 Floating-Point Multiplication Example	87
4.2 Floating Point Multiplier Flowchart	90
4.3 Floating Point Multiplier Hardware	91
4.3.1 Floating Point Multiplier Hardware Architecture	92
4.3.2 Floating Point Multiplier Hardware Implementation	93
4.3.2.1 Sign Bit Calculation	93
4.3.2.2 Data Classification Module	94
4.3.2.3 Exponent Carry Lookahead Adder	100
4.3.2.4 Modular Bias Subtractor	101
4.3.2.5 Mantissa Append Module	102
4.3.2.6 Mantissa 32-bit Wallace Multiplier	105
4.3.2.7 Mantissa Right Shifter	114
4.3.2.8 Mantissa Product Rounding	115
4.3.2.9 Exponent Incrementor	117
4.3.2.10 Compute Flags	118
4.3.2.11 Compute Output	122
4.4 Floating Point Multiplier Results	124
4.4.1 Floating Point Multiplier Compilation Report	124
4.4.2 Floating Point Multiplier Testbench	125
4.4.3 Floating Point Multiplier Simulation Results	126
Case A:	126
Case B:	127
Case C:	128

Case D:.....	129
Case E:	130
4.5 Conclusion	131
Chapter 5: 32-bits Floating Point Divider.....	132
5.1 Floating Point Division Algorithm.....	133
5.1.1 Floating-Point Division Example.....	134
5.2 Floating Point Divider Flowchart	137
5.3 Floating Point Divider Hardware	138
5.3.1 Floating Point Divider Hardware Architecture	139
5.3.2 Floating Point Divider Hardware Implementation	140
5.3.2.1 Sign Bit Calculation	140
5.3.2.2 Data Classification Module.....	141
5.3.2.3 Modular Exponent Subtractor.....	143
5.3.2.4 Carry Lookahead Bias Adder.....	144
5.3.2.5 Mantissa Append Module	145
5.3.2.6 Mantissa 24-bit Divider	147
5.3.2.7 Mantissa Left Shifter	148
5.3.2.8 Mantissa Division Rounding	149
5.3.2.9 Exponent Decrement.....	150
5.3.2.10 Compute Flags	151
5.3.2.11 Compute Output	152
5.4 Floating Point Divider Results	153
5.4.1 Floating Point Divider Compilation Report	153
5.4.2 Floating Point Divider Testbench	154
5.4.3 Floating Point Divider Simulation Results	155
Case A:.....	155
Case B:	156
Case C:	157
Case D:.....	158
Case E:	159
5.5 Conclusion	160
Chapter 6: Floating Point Unit.....	161
6.1 Floating Point Unit Block Diagram.....	162
6.2 Floating Point Unit Verilog Code	163
6.3 Floating Point Unit RTL Diagram.....	164
Chapter 7: Education Module.....	165
7.1 Lab 1	166
7.2 Lab 2.....	174
7.3 Lab 3	181
7.4 Lab 4	187
7.5 Lab 5	194
7.6 Lab 6.....	200
7.7 Lab 7	206
7.8 Lab 8	213

7.9 Lab 9	219
7.10 Lab 10	225
Conclusion	229
Chapter 8: Conclusion	230
8.1 Future Scope of Work	231
Bibliography	232
About the Author	236

Table of Figures

Figure 1 Base 10 Notation	Figure 2 Base 2 Notation	2
Figure 3 PDP 10		3
Figure 4 32-bit Floating Point Format		5
Figure 5 Binary Significands		6
Figure 6 Significand Representation		6
Figure 7 Subnormal Representation		7
Figure 8 Examples of Binary Floating Format		7
Table 1 Floating Point Parameters		8
Figure 9 Range of Binary 32-bit Format		9
Table 2 Data Types		10
Table 3 Arithmetic Operations		11
Figure 10 Binary Representation for Add		14
Figure 11 Adder Exponent Subtraction		14
Figure 12 Adder Mantissa Addition		15
Figure 13 FPA Example Result		15
Figure 14 Floating Point Adder Flowchart		16
Figure 15 Floating Point Adder Architecture		18
Table 4 Sign Operations		19
Figure 16 Modular Subtractor RTL		20
Figure 17 Exponent Subtractor Code		21
Figure 18 Modular Exponent Sub Block		22
Figure 19 Ripple Carry Subtractor Block		23
Figure 20 Ripple Carry RTL Diagram		24
Figure 21 Ripple Carry Subtractor Code		25
Figure 22 Twos Complementor Block Diagram		26
Figure 23 Twos Complementor RTL Diagram		27
Figure 24 Twos Complementor Code		28
Figure 25 Mux Block Diagram		30
Figure 26 Mantissa MUX RTL		31
Figure 27 Mantissa MUX Code		32
Table 5 MUX Truth Table		32
Figure 28 Right Shifter Block Diagram		33
Figure 29 Right Shifter Code		34
Table 6 Multiplexer 2 Table		35
Figure 30 Mantissa CLA Block Diagram		37
Figure 31 Generate AND		38
Figure 32 Propagate OR		38
Figure 33 Carry Out Logic		39
Figure 34 Carry Lookahead Adder Code		40
Table 7 Multiplexer 3 Table		41
Figure 35 Controlled Incrementor Block		42
Figure 36 Controlled Incrementor RTL		44
Figure 37 Controlled Incrementor Code		45
Figure 38 FPA Compilation Report		47
Figure 39 FPA Testbench		48
Figure 40 Case A Result		49
Figure 41 Case B Result		50
Figure 42 Case C Result		51
Figure 43 Case D Result		52
Figure 44 Binary Representation Sub Example		56
Figure 46 Sub Exponent Subtraction		56
Figure 47 Subtract Mantissa Subtraction		57
Figure 48 FPS Example Result		57
Figure 49 Floating Point Subtractor Flowchart		58
Figure 50 Floating Point Subtractor Architecture		60

Table 8 Sign Operations	61
Figure 51 Mux Block Diagram 2.....	62
Table 9 MUX Truth Table	63
Table 10 Multiplexer 2 Table	65
Figure 52 24 bit Ripple Carry Sub RTL.....	67
Figure 53 24-bit Ripple Cary Sub Code	68
Table 11 Multiplexer 3 Table	69
Figure 54 Controlled Decrement Block.....	71
Figure 55 Controlled Decrement RTL.....	73
Figure 56 Controlled Decrement Code	74
Figure 57 Left Shifter Block Diagram	75
Figure 58 Left Shifter Code.....	76
Figure 59 FPS Compilation Report.....	78
Figure 60 FPS Testbench	79
Figure 61 FPS Case A Result	80
Figure 62 FPS Case B Result.....	81
Figure 63 Case C Result	82
Figure 64 Case D Result.....	83
Figure 65 Binary Presentation Mul Example	87
Figure 66 XOR Sign Subtraction.....	87
Figure 67 Mul Exponent Addition	88
Figure 68 Mul Bias Subtraction	88
Figure 69 Mantissa Multiplication.....	89
Figure 70 Incrementing Exponent	89
Figure 71 FPM Example Result	89
Figure 72 Floating Point Multiplier.....	90
Figure 73 Floating Point Multiplication Architecture.....	92
Table 13 Sign Operations Mul	93
Figure 74 sNaN Format	94
Figure 75 qNaN Format	94
Figure 76 Plus Infinity Format	95
Figure 77 Negative Infinity Format.....	95
Figure 78 Positive Zero Format	96
Figure 79 Negative Zero Format.....	96
Figure 80 Subnormal Format.....	97
Figure 81 Normal Format	97
Figure 82 Data Classification Verilog Code	98
Figure 83 Data Classification RTL Diagram	99
Figure 84 Exponent Adder Instantiation	100
Figure 85 Bias Subtraction Instantiation	101
Figure 86 Mantissa Append Block Diagram	102
Figure 87 Mantissa Append Verilog Code.....	103
Figure 88 Mantissa MUX RTL	104
Table 14 Append Mantissa Truth Table.....	104
Figure 89 Wallace Multiplication Stages	106
Figure 90 Wallace Multiplication Stage 0	107
Figure 91 Wallace Multiplication Stage 1	107
Figure 92 Wallace Multiplication Stage 2	108
Figure 93 Wallace Multiplication Stage 3	108
Figure 94 Wallace Multiplication Step 3.....	108
Figure 95 Wallace Multiplier Flow Diagram	109
Figure 96 Wallace Tree Mul Block Diagram	110
Figure 97 Wallace Multiplier Code	111
Figure 98 Wallace Tree 32-bit RTL.....	112
Figure 99 Wallace Tree 16-bit RTL.....	112
Figure 100 Wallace Tree 8-bit RTL.....	113
Figure 101 Right Shifter Instantiation	114

Figure 102 Product Rounding Verilog Code	115
Figure 103 Product Rounding RTL Diagram	116
Figure 104 Exponent Incrementor Instantiation.....	117
Figure 105 Compute Flags RTL Diagram	119
Figure 106 Compute Flags Verilog Code	121
Figure 107 Compute Output RTL Diagram	123
Figure 108 Compute Output Verilog Code.....	123
Figure 109 FPM Compilation Report	124
Figure 110 FPM Testbench	125
Figure 111 FPM Case A Result	126
Figure 112 FPM Case B.....	127
Figure 113 FPM Case C.....	128
Figure 114 FPM Case D.....	129
Figure 115 FPM Case E	130
Figure 116 Binary Presentation Divide Example	134
Figure 117 XOR Sign Division	134
Figure 118 Divide Exponent Subtraction	135
Figure 119 Divide Bias Subtraction.....	135
Figure 120 Mantissa Division	136
Figure 121 FPD Example Result	136
Figure 122 Floating Point Divider	137
Figure 123 Floating Point Division.....	139
Table 15 Sign Operations Divide	140
Figure 124 Data Classification Instantiation B	142
Figure 125 Data Classification Instantiation	142
Figure 126 Mode Subtractor Instantiation	143
Figure 127 Bias Addition Instantiation	144
Table 16 Append Mantissa Truth Table.....	145
Figure 128 Append Mantissa A Instantiation	146
Figure 129 Append Mantissa A Instantiation	146
Figure 130 24-bit Divider Code.....	147
Figure 131 Left Shifter Instantiation.....	148
Figure 132 Division Rounding Instantiation.....	149
Figure 133 Exponent Decrement Instantiation.....	150
Figure 134 Compute Flags Instantiation	151
Figure 135 Compute Out Instantiation.....	152
Figure 136 FPD Compilation Report	153
Figure 137 FPD Testbench.....	154
Figure 138 FPD Case A Result	155
Figure 139 FPD Case B.....	156
Figure 140 FPD Case C.....	157
Figure 141 Case D	158
Figure 142 FPD Case E	159
Figure 143 Floating Point Unit Block Diagram	162
Figure 144 Floating Point Unit Verilog Code.....	163
Figure 145 Floating Point Unit RTL Diagram	164

Acknowledgments

This project would not have been possible without the support of many people. Many thanks to my adviser, Dr. Bill D Carroll, who read my numerous revisions and helped make some sense of the confusion. Also, thanks to my committee members, Prof. David Levine, and Dr. Jason Losh, who offered guidance and support. Thanks to the University of Texas at Arlington Graduate College for providing me with support and infrastructure to complete this project. And finally, thanks to lord *Hanuman*, my parents, my girlfriend, and numerous friends who endured this long process with me, always offering support and love.

Abstract

In today's day and age of arithmetic, Floating Point Arithmetic is by far the most industry sanctioned way of approximating real number arithmetic for making numerical calculations on all computers used by industries on an everyday basis.

In the year 1985, IEEE 754 standard was established that defined a single universal standard for all different arithmetic formats [1]. Before this, for a long period each computer had a different arithmetic format and size for bases, significand, and exponents. This format allowed industries all around the world to compute floating point arithmetic in a universal way and facilitated open communication between all worlds.

The first objective of this project is implementing a single precision binary floating point processing unit in accordance with the IEEE 754 standard using Verilog hardware description language and writing test benches to run ModelSim simulations for testing.

The second objective of this project is to convert the implementation of the single precision floating point unit into an education model. The purpose of this education model will be to educate future Digital Logic & Design students in the field of floating-point processing and provide them a roadmap to build their own floating-point processor using a series of lab assignment.

Chapter 1: Introduction

Manipulating real numbers efficiently has been the basis of computing in various fields ranging from engineering & science, to finance. There have been many ways to approximate and compute real numbers in an efficient way ever since computers have been introduced.

The most efficient one of these computing methods and representing real numbers in computers has been identified to be floating point arithmetic. The representation of such an infinite, continuous set with a finite set is a difficult task. Such a task requires a lot of compromises to be made in terms of speed, accuracy, implementation, and memory cost [2].

Considering the challenges of speed and accuracy, floating point arithmetic is the perfect compromise that can be made for most numerical applications that needs to be performed in today's day and age of computation.

1.1 Floating Point Numbers

The floating-point numbers representation is based on scientific notation for representing floating point numbers [3]. This notation consists of four major elements: sign, mantissa, base, and exponent. In the scientific notation, the decimal point is not set in a fixed position in the bit sequence. The position of decimal is indicated as a base power.

Example of floating-point representation using different bases:

$$\begin{array}{c}
 \text{Sign} \\
 \underbrace{+} \\
 \text{Mantissa} \quad \cdot \quad \text{Base} \\
 \underbrace{5.05} \quad \cdot \quad \underbrace{10} \\
 \text{Exponent} \\
 \underbrace{-25}
 \end{array}$$

Figure 1 Base 10 Notation

$$\begin{array}{c}
 \text{Sign} \\
 \underbrace{+} \\
 \text{Mantissa} \quad \cdot \quad \text{Base} \\
 \underbrace{1.01110} \quad \cdot \quad \underbrace{2} \\
 \text{Exponent} \\
 \underbrace{-1101}
 \end{array}$$

Figure 2 Base 2 Notation

Floating point numbers expressed in scientific notation consist of four primary components:

- Sign: This element conveys information about the sign of the number (0 is used for positive numbers, 1 for negative numbers).
- Mantissa: This element consists of the value of the number.
- Exponent: This element consists of the value of the base power in a biased form.
- Base: The base of a number is implied and is universally known for all number system (2 for binary numbers, 10 for decimal numbers).

This kind of free hand format allowed for each individual architect or programmer to design their own floating point number system for use. This allowed for different elements to be interpreted using different bit numbers with no uniformity overall.

The very first modern-day implementation of a floating-point arithmetic in a computer was built using a radix-2 number system. This number system consisted of 14-bit significand,

7-bit exponents, and 1-bit sign. On the other hand, another computer PDP-10, Figure 3, used a radix-8 number system and IBM 360 used a radix-16 floating point arithmetic [4].



Figure 3 PDP 10

This non uniformity of number system in different architectures led to the need for a universal standard of floating-point number which developed a clear and concise format to be used by all developers worldwide. This was called the IEEE 754 standard.

1.2 The IEEE 754 Standard

IEEE is acronym for the Institute of Electrical and Electronics Engineers (IEEE). IEEE is the world's largest technical professional organization dedicated to advancing technological innovations and excellence for the benefit of humanity.

The American Institute of Electrical Engineers was the organization that gave birth to the IEEE in 1884. The competing Institute of Radio Engineers was established in 1912. Although the IRE drew more students and grew larger by the middle of the 1950s, the AIEE was initially larger. In 1963, the IRE and the AIEE amalgamated to form IEEE.

The IEEE Operations Center in Piscataway, New Jersey, initially established in 1975, is where most of the business is conducted. The IEEE headquarters are in New York City [5].

One of the many responsibilities of IEEE is serving as a major standards development organization for the creation of industrial standards in a variety of fields like nanotechnology, consumer electronics, and telecommunications. The IEEE 754-2008 is one of such standards.

1.2.1 Overview

The IEEE 754 standard, revised in 2008, specified important aspects of floating-point arithmetic such as formats and methods to operate with floating point numbers. The IEEE 754 standard specifies four different formats for representing floating point numbers are:

- Half Precision Floating Point (16 bits)
- Single Precision Floating Point (32 bits)
- Double Precision Floating Point (64 bits)
- Quadruple Precision Floating Point (128 bits)

32 bits and 64 bits are the most found representation of floating-point numbers. In this thesis we will be working with the single precision floating point number system i.e., 32 bits of operands [6].

Furthermore, this IEEE 754 standard dictates that all the computational work done with floating point numbers will output the same result irrespective of whether the processing was done in software or hardware, and irrespective of the method of implementation.

The IEEE 754 standard specifies the following:

- Formats for binary and decimal floating-point numbers for arithmetic and data transaction between modules.
- Instructions for various operations such as addition, subtraction, multiplication, and other similar operations.
- Variety of parameters to be considered when rounding binary numbers after arithmetic operations and conversions.
- Setting parameters for conversions between integer to floating point format.
- Guidelines for floating point data classification, exceptions, and their handlings such as NaN, Zero, Normal, Subnormal, and Infinity data types.

1.2.2 Binary Format Encodings

Floating point binary number is represented in accordance with IEEE 754 standard in three separate fields and 32 bits for single precision number as shown in Figure 4 [7]:

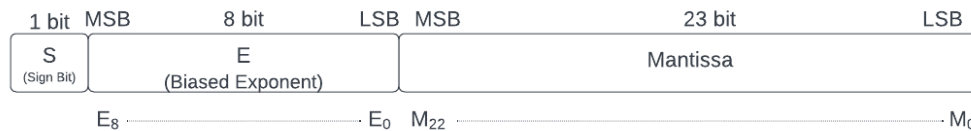


Figure 4 32-bit Floating Point Format

To explain the figure, the single precision (32 bit) binary format has been divided in 3 parts:

- Sign: The most significant bit of the 32-bit number (31st bit) is the bit that conveys information regarding sign of the number being represented. (0 to denote positive numbers, 1 to denote negative numbers).
- Exponent: The next part is the 8-bit biased exponent that ranges from bit 23 to bit 30.
- Mantissa: The final part is the 23-bit long mantissa that ranged from bit 22 to bit 0.

Exponent

The exponent value for the 32-bit binary number system is stored in 8 bits. This has the bias value of 127 added to it. This dictates the final range of exponent to fall between -126_{10} (10000010_2) and $+127_{10}$ (00000001_2), being zero at value (01111111_2).

To find the true value of the exponent, a fixed value (bias) is subtracted from it. In the case of 8 bits, this value yields a true exponent range from 0_{10} to 255_{10} [8].

The biased exponent has a range that is divided into three sections based on type:

- Normal: For 32-bit number system, the range for normal numbers consists of every integer from 2^0 (1_{10}) to $2^8 - 2$ (254_{10}). Given the 8 bits of exponents.

- Subnormal & Zero: For 32-bit number system, the value 0 is used for subnormal numbers and the zero-type number [9].
- Special Cases: The reserved value $2^8 - 1$ (255_{10}) is used to encode special types of number cases like Infinity or NaN.

Significand

The final part of this floating-point binary number representation is the 23-bit long significand which starts from the least significant bit of the number representation. A floating-point number can be represented in variety of different ways.

For Example:

$$0.110 \times 2^5 \qquad 1100 \times 2^1 \qquad 0.00110 \times 2^7$$

Figure 5 Binary Significands

All the values in the above examples (Figure 5) are equivalent in value and expressed with difference significand values. The thesis has emphasized the importance of a unique representations. In order to achieve that objective, finite floating-point numbers are to be normalized for choosing a representation with minimum possible value of the exponent. A normal number is the kind where the most significant bit of the significand is nonzero value when represented in binary or base 2 format. In a typical convention, the radix point is to the right of the last bit of the number which is represented in form of [10]:

$$\pm 1 . b b b b \dots b \times 2^{+E}$$

Figure 6 Significand Representation

The mantissa is reality is a 24 bit number but as shown in figure above, the most significant bit is always 1. This kind of convention deems it unnecessary for the mantissa to

store this most significant bit in their representation. The mantissa thus became a 23 bit value which in reality contains an implicit bit depending on the type of data being represented.

The mantissa is appended with 1 as the most significant bit (24th bit) for all the normal numbers (Figure 6), and appended with 0 for all subnormal number type (Figure 7). Both these most significant bits in different data types will be implied and taken into account when making arithmetic operations to get the correct decimal value.

$$\pm 0 . b b b b \dots b \times 2^0$$

Figure 7 Subnormal Representation

1.2.3 Examples of Floating Point Representation

The Figure 8 below shows examples of four numbers stored in binary floating point format in a 32 bit number system. On the very left for each example is decimal value of a number, in the middle there is the same number in binary format, and on the right is that number in 32 bit floating point binary format as defined by IEEE 754 [11].

$$\begin{aligned}
 1.6328125 \times 2^{-20} &= 1.1010001 \times 2^{-10100} &= 0 \ 01101011 \ 101000100000000000000000 \\
 1.6328125 \times 2^{20} &= 1.1010001 \times 2^{10100} &= 0 \ 10010011 \ 101000100000000000000000 \\
 -1.6328125 \times 2^{-20} &= -1.1010001 \times 2^{-10100} &= 1 \ 01101011 \ 101000100000000000000000 \\
 -1.6328125 \times 2^{20} &= -1.1010001 \times 2^{10100} &= 1 \ 10010011 \ 101000100000000000000000
 \end{aligned}$$

Figure 8 Examples of Binary Floating Format

The binary pattern on the right side has four important features of note:

- Sign of value is stored in the first bit of the word.
- First bit of significand is 1 and is not stored in the significand field.
- The exponent in binary is true exponent with 127 added to it as bias.
- The base of the format is 2 which signifies binary.

1.2.4 Floating Point Parameters

Parameter	Format		
	Binary 32	Binary 64	Binary 128
Storage Bits	32	64	128
Exponent Bits	08	11	15
Exponent Bias	127	1023	16383
Max Exponent	127	1023	16383
Min Exponent	-126	-1022	-16382
Significand Bits	23	52	112
Sign Bits	1	1	1
No. of Exponents	254	2046	32766
No. of Fractions	2^{23}	2^{52}	2^{112}
No. of Values	1.98×2^{31}	1.99×2^{63}	1.99×2^{128}

Table 1 Floating Point Parameters

1.2.5 Range of Floating-Point Number

There is a finite range of values that can be represented with the finite number of bits in the floating-point number system. In the twos complement number system, all the integer values from -2^{31} to $2^{31} - 1$ can be represented in this system. However, when we use the representation as shown in Figure 4 allows for a broader range of numbers [12]:

- Negative Numbers: Range from $-(2 - 2^{-23}) \times 2^{128}$ and -2^{-127}
- Positive Numbers: Range from 2^{-127} and $(2 - 2^{-23}) \times 2^{128}$.

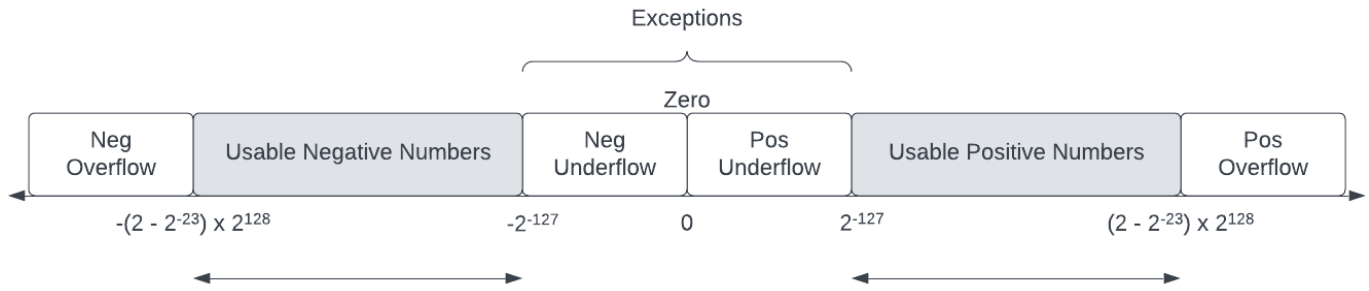


Figure 9 Range of Binary 32-bit Format

Figure 9 shows the range of binary 32 bit floating point number format. The range is divided into a total of seven different regions. The ranges are [13]:

- Negative Overflow: Negative numbers that are less than $-(2 - 2^{-23}) \times 2^{128}$
- Negative Numbers: Negative numbers range from $-(2 - 2^{-23}) \times 2^{128}$ to -2^{-127}
- Negative Underflow: Negative numbers that are greater than -2^{-127}
- Zero
- Positive Underflow: Positive numbers that are less than 2^{-127}
- Positive Numbers: Positive numbers range from 2^{-127} and $(2 - 2^{-23}) \times 2^{128}$.
- Positive Overflow: Negative numbers that are greater than $(2 - 2^{-23}) \times 2^{128}$

1.2.6 Rounding of Floating-Point Number

According to IEEE 754 standard, the result of floating-point operation needs to be unique irrespective of method of computation. This gives birth to the need for absolute precision which can be reached by employing the process of rounding the results [14].

There are four major rounding operations that are recommended by the standard:

- Rounding the result to the nearest representable number.
- Rounding the result towards $+\infty$.
- Rounding the result towards $-\infty$.
- Rounding the result towards 0.

In this thesis, we will be employing Rounding to nearest policy for all our operation results, as it is the default rounding mode listed in the standard. In this policy, the representable value nearest to the infinity precise result will be used.

1.2.7 Data Types of 32 bit Floating-Point Number

	Sign	Biased Exponent	Fraction	Value
Positive Zero	0	0	0	0
Negative Zero	1	0	0	-0
Plus Infinity	0	All 1s	0	∞
Minus Infinity	1	All 1s	0	$-\infty$
Quiet NaN	0 or 1	All 1s	$\neq 0$; MSB = 1	qNaN
Signal NaN	0 or 1	All 1s	$\neq 0$; MSB = 0	sNaN
Positive Normal	0	$0 < e < 255$	F	2^{e-127}
Neg Normal	1	$0 < e < 255$	F	-2^{e-127}
Pos Subnormal	0	0	$F \neq 0$	2^{e-126}
Neg Subnormal	1	0	$F \neq 0$	-2^{e-126}

Table 2 Data Types

1.3 Floating-Point Arithmetic

Floating-point arithmetic is about the basic operations that can be carried out between two different floating point binary numbers. For addition and subtraction, the arithmetic is trickier than multiplication and division as it requires shifting radix point and ensuring that the operands are in alignment with each other [15].

The following table 3 summarizes the basic operations for floating point arithmetic:

Floating Point Numbers	Arithmetic Operations
$X = X_S \times B^{X_E}$ $Y = Y_S \times B^{Y_E}$	$X + Y = (X_S \times B^{X_E - Y_E} + Y_S) \times B^{Y_E}$ $X - Y = (X_S \times B^{X_E - Y_E} - Y_S) \times B^{Y_E}$ $X \times Y = (X_S \times Y_S) \times B^{X_E + Y_E}$ $X / Y = (X_S / Y_S) \times B^{X_E - Y_E}$

Table 3 Arithmetic Operations

1.3.1 Exponent Overflow

A positive exponent exceeds the maximum possible value which is any value more than 127.

1.3.2 Exponent Underflow

A negative exponent that is less than the minimum possible value which is any value less than the -127 value. This implies that the exponent number is too small to be represented, and it may be reported as zero value.

1.3.3 Significand Overflow

On addition of two significands, the result might carry out from the MSB and exceed the maximum value allowed. This can be fixed during the normalization process.

1.3.4 Significand Underflow

In the process of aligning significands, digits may flow off the right end of the value.

Chapter 2: 32-bits Floating Point Adder

In this chapter, we describe an efficient implementation of an IEEE 754 single precision floating point adder targeted for DE-1 Cyclone V FPGA. Verilog is used to implement a technology-independent pipelined design. The adder implementation handles the overflow and underflow cases. Rounding is implemented to give more precision when using the Carry Look Ahead Adder for faster calculations. The Floating-Point Adder was verified by testbench simulations on ModelSim.

In this chapter we will dive deeper into the floating-point adder algorithm, architecture, code design, RTL diagram, and simulation results.

We will talk about the procedure in addition operations and a first look at the code design in a block diagram way followed by deeper understanding of code development. Out of four arithmetic operations, floating point addition is the most complicated operation.

Floating point addition is done by extracting signs, subtracting exponents, adding mantissa values, and shifting the mantissa for normalization.

There are five basic phases of designing a Floating-Point Adder:

- 1) Check for Zeroes.
- 2) Isolate the sign bits.
- 3) Align the Significands.
- 4) Add the Significands.
- 5) Normalize the Significand
- 6) Normalize the Exponent if needed.

2.1 Floating Point Addition Algorithm

As described in the above topics, floating point number is in the format of:

$$Z = (-1)^S * 2^{(E - \text{Bias})} * (1.M)$$

To add two floating point numbers A & B the different steps to follow are [16]:

- 1) Extracting signs, exponents and mantissas of both A and B numbers.
- 2) Calculating the output sign.
- 3) Treating the special cases.
- 4) Finding out the data types of numbers given
- 5) Subtracting the two exponents.
- 6) Shifting the lower exponent number mantissa to the right.
- 7) Addition of the mantissa values
- 8) Normalizing mantissa by bit shifting.
- 9) Detecting exception, overflow, and underflow.

2.1.1 Floating-Point Addition Example

A = 9.75 (base 10)

B = 0.5625 (base 10)

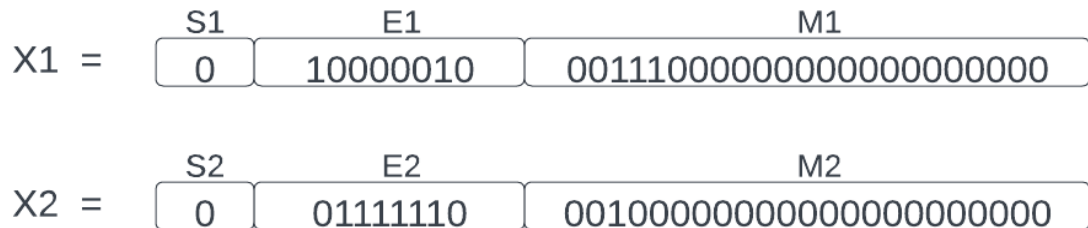


Figure 10 Binary Representation for Add

1) S1 = 0, E1 = 10000010, M1 = 001110000000000000000000

S2 = 0, E2 = 01111110, M2 = 001000000000000000000000

2) Exponent Subtraction

$$\begin{array}{r}
 10000010 \\
 - 01111110 \\
 \hline
 00000100
 \end{array}$$

Figure 11 Adder Exponent Subtraction

E = 00000100₂ = 4₁₀

3) Right Shift mantissa M2 by E1 – E2 (4)

1.M2 = 1.001000000000000000000000

Shifted Mantissa = 0.000100100000000000000000

4) Add the mantissa

$$\begin{array}{r}
 1.001110000000000000000000 \\
 + 0.000100100000000000000000 \\
 \hline
 1.010010100000000000000000
 \end{array}$$

Figure 12 Adder Mantissa Addition

M = 1.010010100000000000000000

5) No normalization needed.

6) No exponent incrementation needed.

7) Result



Figure 13 FPA Example Result

2.2 Floating Point Adder Flowchart

The below, Figure 14, showcases a typical flowchart that is used to design a floating point adder. The figure shows a step by step narrative and displays the high level functions that is required to compute floating point addition [17]. The flowchart shows block level diagram and each block or element is implemented in hardware and is described in detail in the following topics of the thesis .

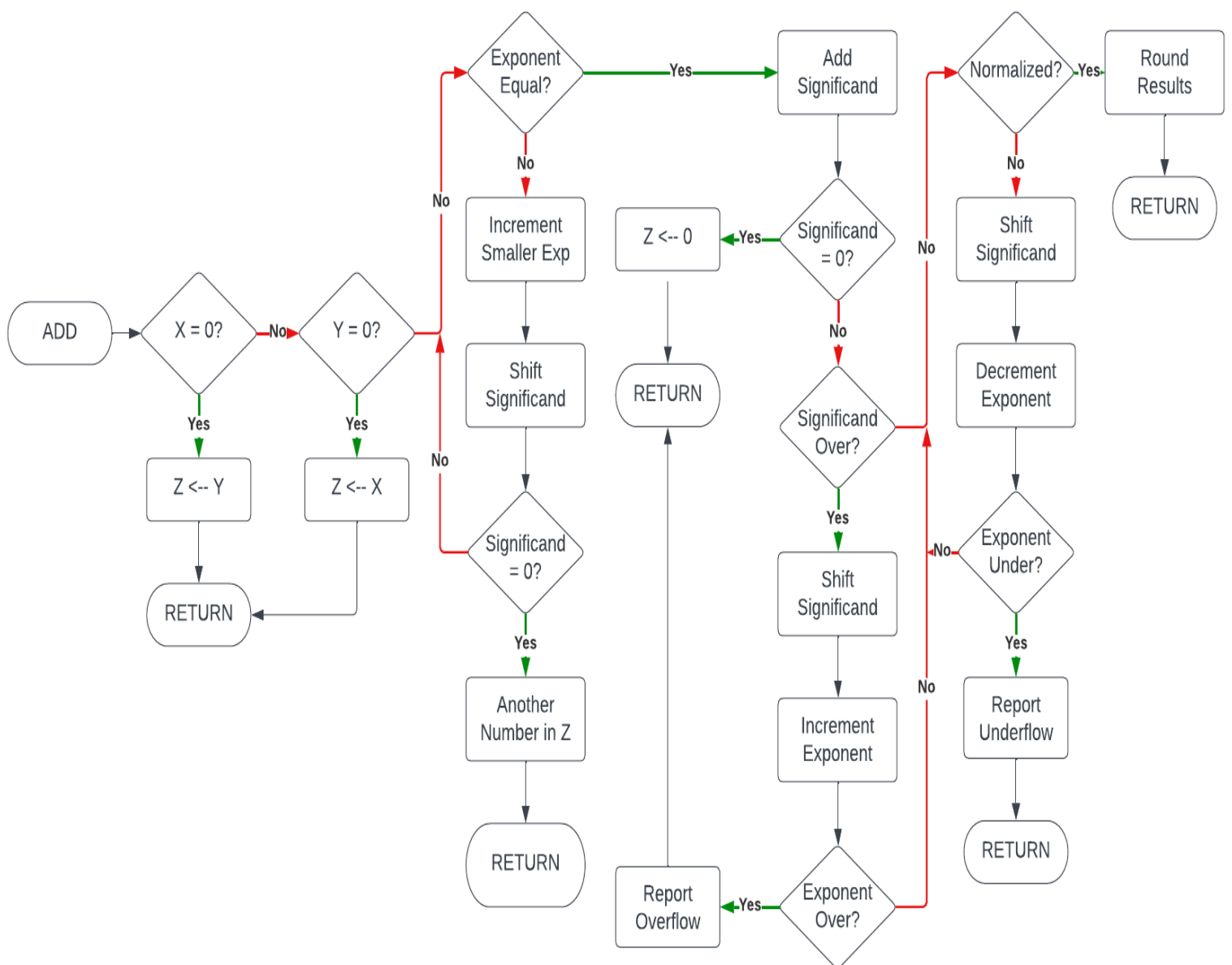


Figure 14 Floating Point Adder Flowchart

2.3 Floating Point Adder Hardware

In this section of the thesis we will start explaining and diving deeper into the hardware implementation of the floating point adder. This section will start by elaborating the flowchart further with help of showcasing the hardware architecture used to design the module followed by detailed description of each module used in the architecture.

After understanding the theory of hardware implementation and the architecture of floating point adder the thesis will show the code development that achieved out final objective of building this floating point unit.

2.3.1 Floating Point Adder Hardware Architecture

The below figure, Figure 15, showcases the hardware architecture that was designed and coded to implement synthesizable 32-bit floating point adder using Verilog.

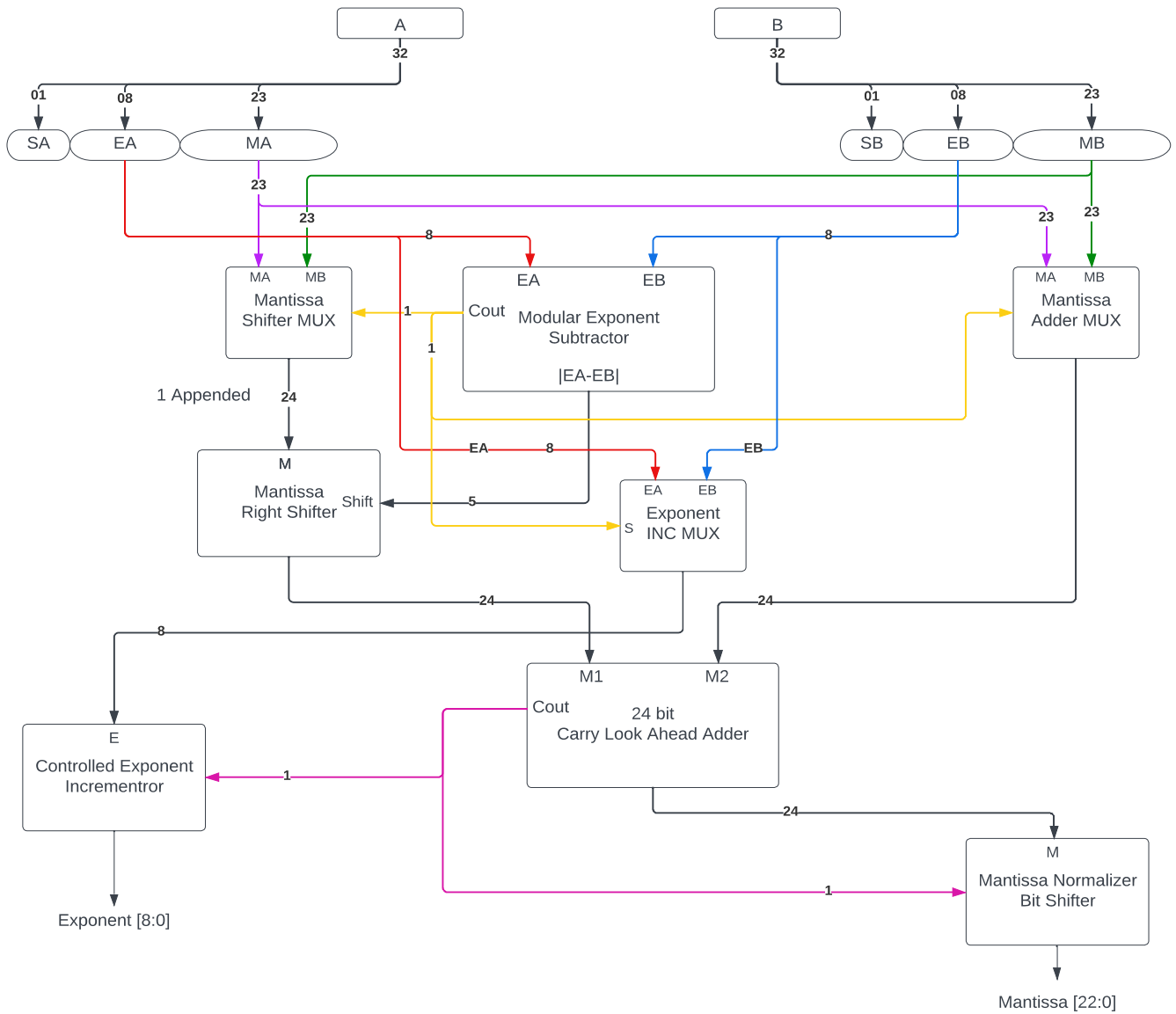


Figure 15 Floating Point Adder Architecture

This floating point architecture uses a total of eight modules that serve various unique purposes in making the design work. The modules are:

- Mantissa Shifter Multiplexer
- Modular Exponent Subtractor
- Exponent Increment Multiplexer
- Controlled Incrementor
- Mantissa Adder Multiplexer
- Mantissa Right Shifter
- Carry Look Ahead Adder
- Mantissa Normalizer

2.3.2 Floating Point Adder Hardware Implementation

In this section, we will discuss the hardware implementation designed for the floating point adder and explain each module and each algorithm step in detail.

2.3.2.1 Sign Bit Calculation

Adding two positive numbers will result in a positive number which makes this section easy for us since there will be another module to take care of subtraction. The table below shows sign operations for various cases:

A's Sign	Symbol	B's Sign	Operation
+	+	+	+
+	+	-	-
-	+	-	+
-	+	+	-

Table 4 Sign Operations

2.3.2.2 Modular Exponent Subtractor

This modular exponent subtractor is responsible for subtracting the exponent of the second input from the exponent of the first input. This module of hardware description language ensures that the exponent difference value is absolute in nature. Before the subtraction operation is performed the program doesn't know which exponent is higher in value. The modular exponent subtractor allows us to not just compute the absolute exponent difference, it also allows us to identify the larger exponent which further identifies the exponent that will be used for the incrementor module and ultimately computing the result of the entire operation.

Modular Exponent Subtractor RTL Diagram

This module further has a total of two modules that facilitate the two tasks. The figure below, Figure 16, shows the RTL of the module consisting of the two modules.

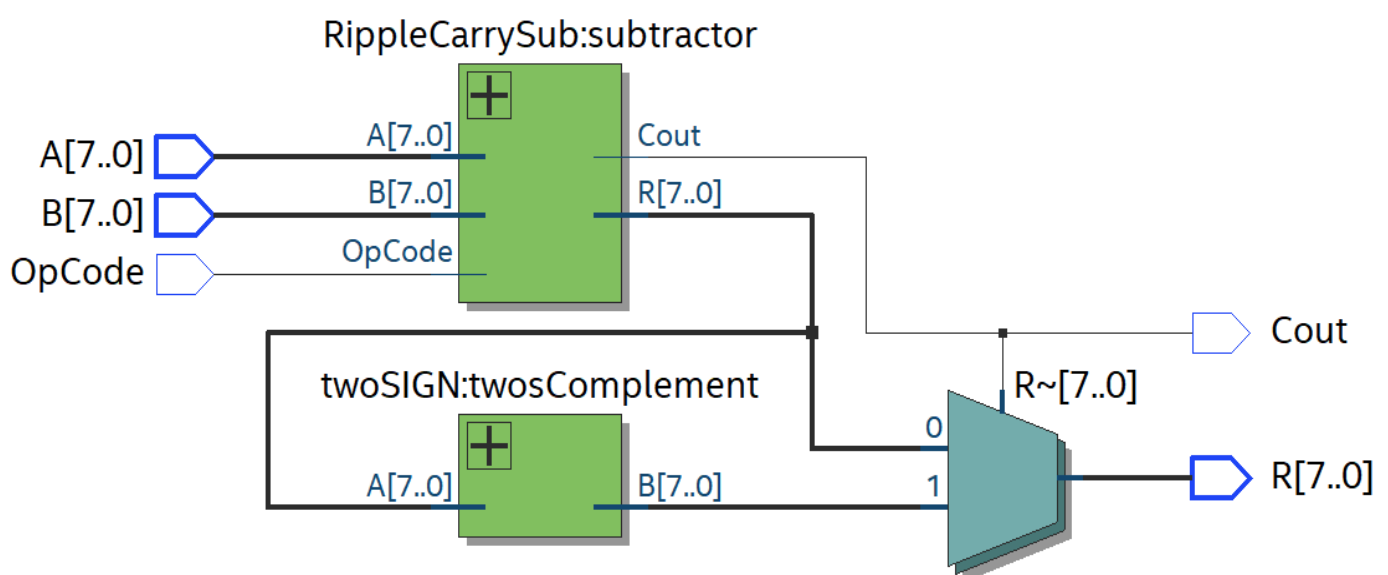


Figure 16 Modular Subtractor RTL

Modular Exponent Subtractor Verilog Code

The figure below, Figure 17, shows the top level design's Verilog code for the implementation of the Modular Exponent Subtractor discussed in the previous section.

```
1  module ModSubtractor #(parameter W = 8)
2  (
3      input [W-1:0] A,
4      input [W-1:0] B,
5      input OpCode,
6      output [W-1:0] R,
7      output Cout
8  );
9
10     wire [W-1:0] out;
11     wire [W-1:0] outFinal;
12
13     RippleCarrySub #(.N(W)) subtractor
14     (
15         .A(A),
16         .B(B),
17         .OpCode(OpCode),
18         .R(out),
19         .Cout(Cout)
20     );
21
22     twoSIGN #(.N(W)) twosComplement
23     (
24         .A(out),
25         .B(outFinal)
26     );
27
28     assign R = (Cout == 1'b1) ? outFinal : out;
29
30 endmodule
```

Figure 17 Exponent Subtractor Code

Modular Exponent Subtractor Block Diagram

The following figure shows the block diagram for the modular exponent subtractor used in the floating point adder algorithm.

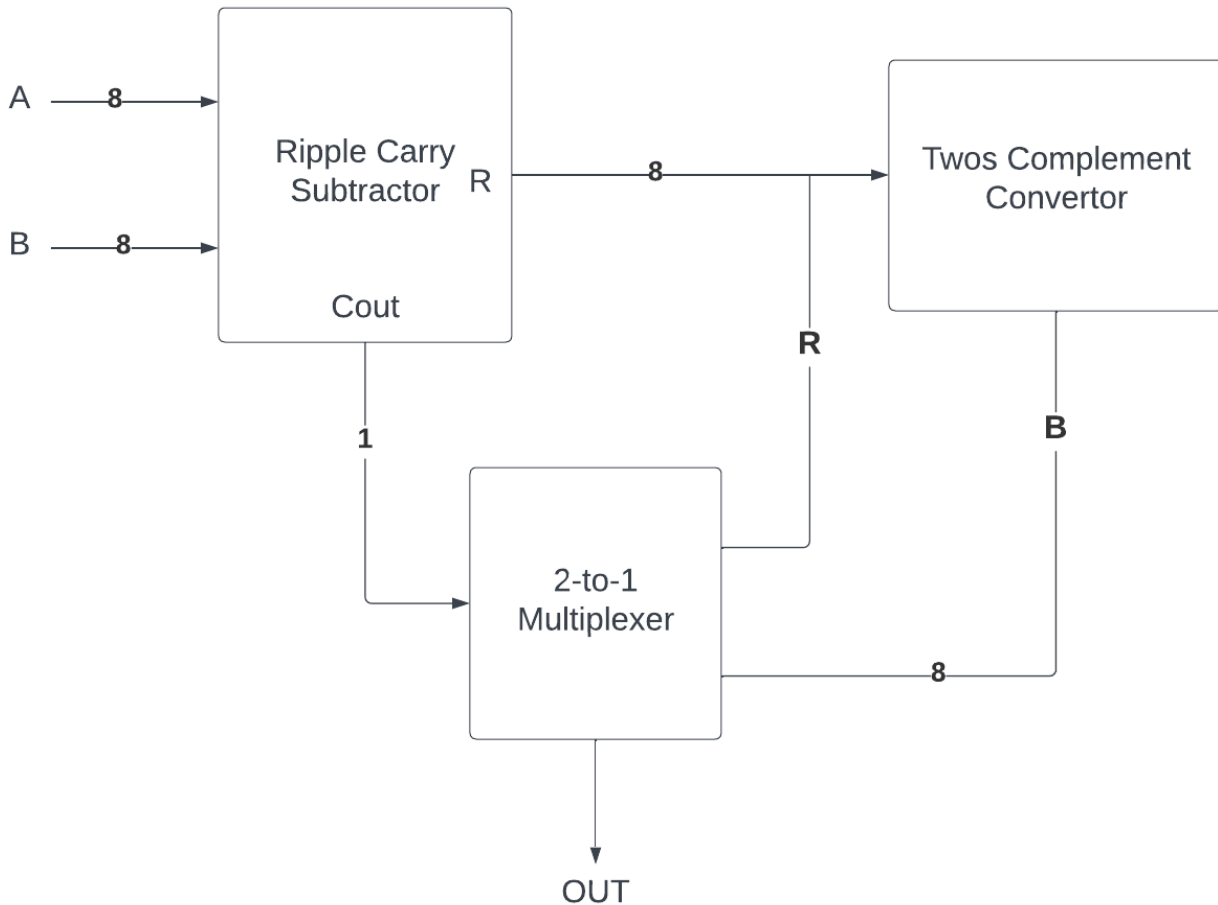


Figure 18 Modular Exponent Sub Block

The three modules inside the modular exponent subtractor are:

- 8-bit Ripple Carry Subtractor
- Twos Complement Converter
- Output Multiplexer

8-bit Ripple Carry Subtractor

To compute the result, an 8-bit ripple carry subtractor was used to subtract the exponent of input A and the exponent of input B. As shown in Figure 19, it is shown that a ripple carry subtractor is a chain of cascaded full adders. Each full adder of this ripple carry adder has three total inputs described as A, B, & C. The full adders also has two outputs namely R and Cout. The carry output of each full adder is fed into the C input of the next full adder, you can also say that the carry output bit ripples to the next full adder [18].

The following figure also shows XOR gates with B and C as inputs and feeding into the input. The XOR gates achieve the purpose of turning the B input into a negative, followed by a regular addition results into the subtraction of the two inputs.

8-bit Ripple Carry Subtractor Block Diagram

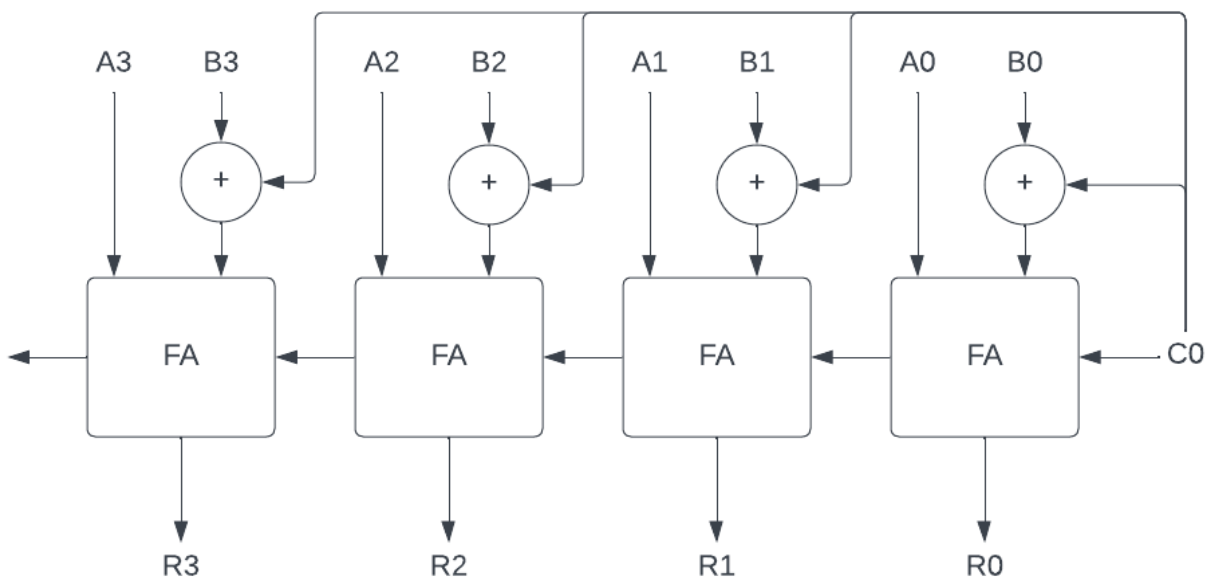


Figure 19 Ripple Carry Subtractor Block

8-bit Ripple Carry Subtractor RTL Diagram

The following figure, Figure 20, shows an RTL diagram of the 8 bit ripple carry subtractor that was used to compute the exponent and the carry out.

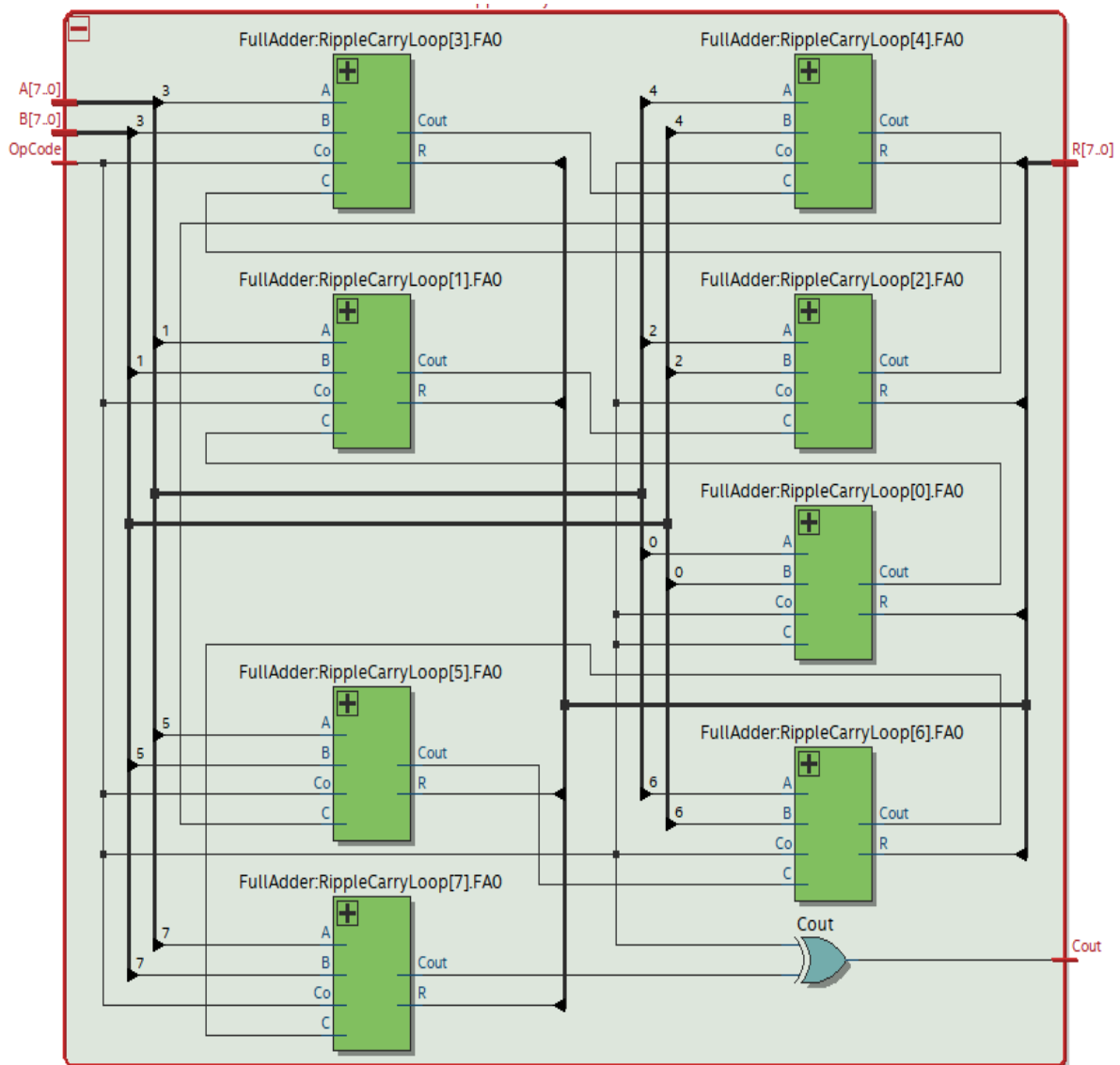


Figure 20 Ripple Carry RTL Diagram

As it can be seen in the RTL diagram the Cout output is computed based on the Carry output of the last full adder and the opcode (1 in case of subtraction, 0 in case of addition) input for the ripple carry subtractor. The Cout is low when Exponent of input A is greater than exponent of input B, and Cout is high for the other way around.

8-bit Ripple Carry Subtractor Verilog Code

The figure below, Figure 21, shows the Verilog code for the ripple carry subtractor. The module consists of two 8-bit inputs, opcode (1 for negative), 8-bit output, and a carry output. The code mainly consists of one for loop that synthesizes a full adder 8 times for every bit of input and output.

```

1  module RippleCarrySub #(parameter N = 8)
2  (
3      input [N-1:0] A,
4      input [N-1:0] B,
5      input  opcode,
6      output[N-1:0] R,
7      output Cout
8  );
9      wire [N:0] C;
10     wire [N-1:0] SUM;
11     assign C[0] = opcode;
12
13     genvar i;
14     generate
15         for(i=0; i<N; i=i+1)
16             begin: RippleCarryLoop
17                 FullAdder FA0
18                 (
19                     .A(A[i]),
20                     .B(B[i]),
21                     .C(C[i]),
22                     .Co(opcode),
23                     .R(SUM[i]),
24                     .Cout(C[i+1])
25                 );
26             end
27     endgenerate
28
29     assign Cout = C[0]^C[N];
30     assign R = SUM;
31
32 endmodule

```

Figure 21 Ripple Carry Subtractor Code

Twos Complement Converter

To take into account the absolute value part of the subtractor, a twos complement converter has been employed. This module takes in the output of the ripple carry subtractor as an input and outputs the conversion.

As shown in Figure 22, the module takes in an 8 bit input called input A, each bit of input A is fed into an XOR gate which other input of XOR gate being MSB of input A. The output of the XOR gate then feeds into an half adder, and the other input of that half adder is also the MSB of input A [19].

The output of the half adder is fed into the input of the next half adder. This module is a cascade of 8 half adders accounting for each bit of the 8 bit input and output. This is how an 8 bit output B is computed that is two's complement of input A.

Twos Complement Converter Block Diagram

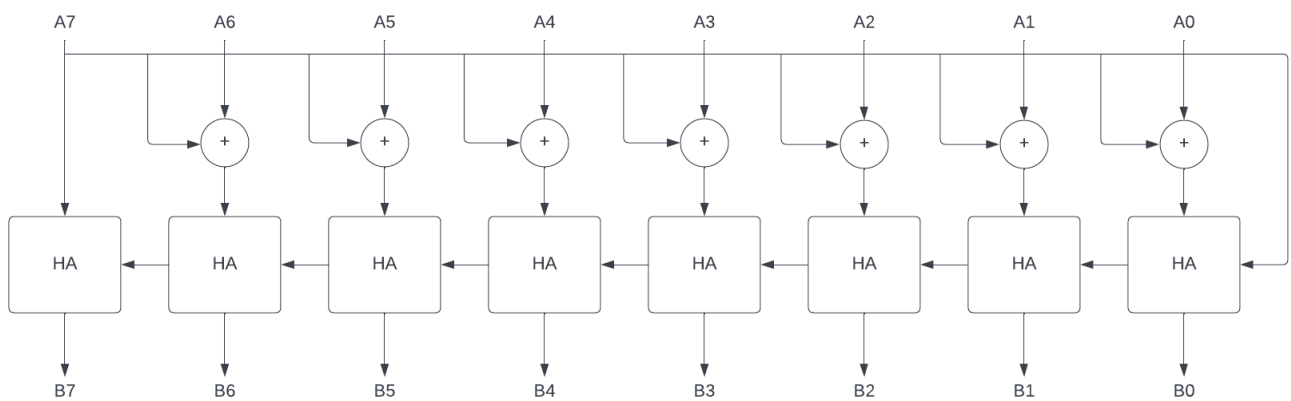


Figure 22 Twos Complementor Block Diagram

Twos Complement Convertor RTL Diagram

The figure below, Figure 23, shows the RTL diagram the twos complement convertor. As shown in the RTL diagram it can be seen that there are total of 7 XOR gates being used to feed into eight half adders.

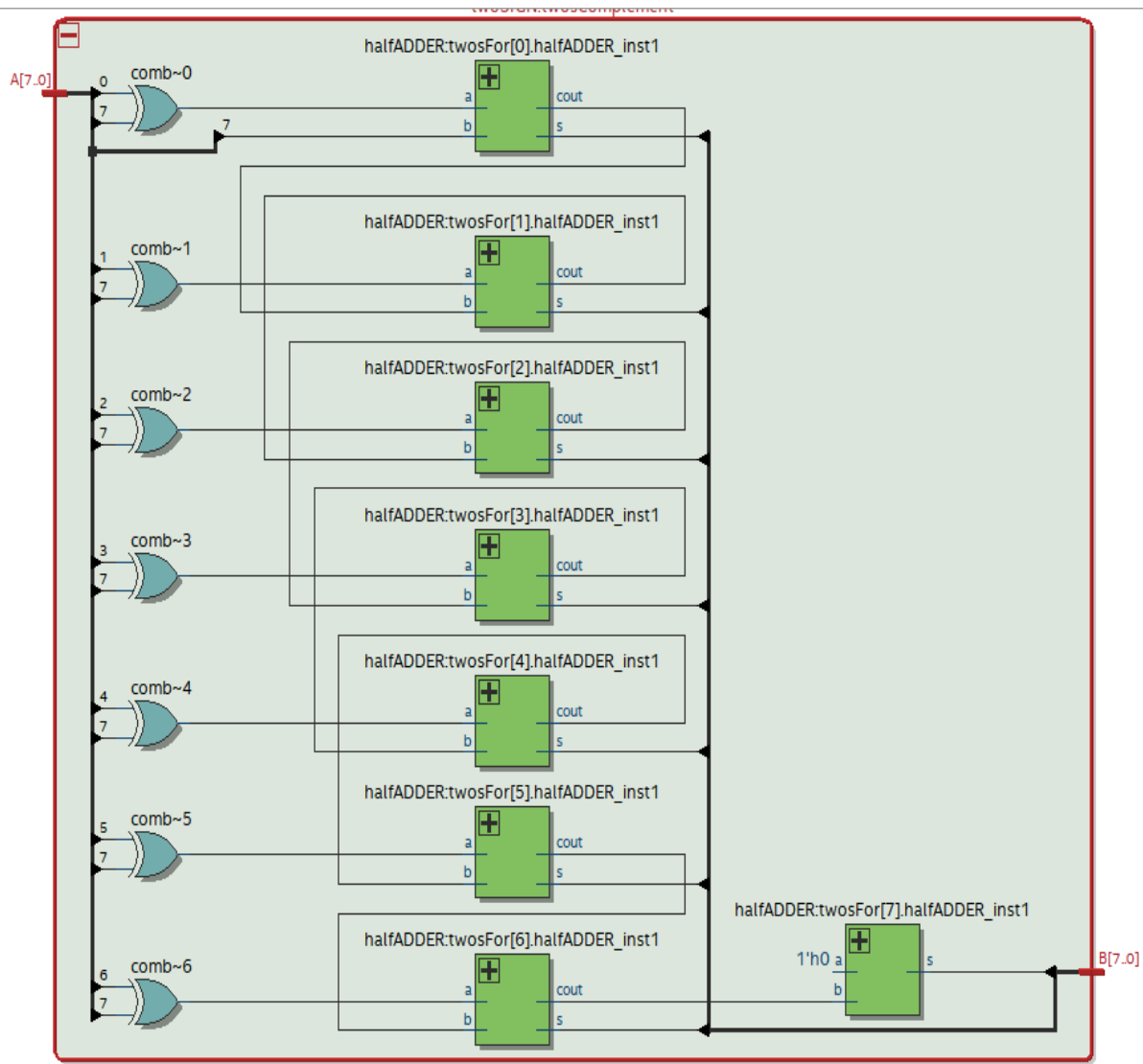


Figure 23 Twos Complementor RTL Diagram

Twos Complement Converter Verilog Code

The figure below, Figure 24, shows the Verilog code for the ripple carry subtractor. The module consists of two 8-bit inputs, opcode (1 for negative), 8-bit output, and a carry output. The code mainly consists of one for loop that synthesizes a full adder 8 times for every bit of input and output.

```

1  module twoSIGN #(parameter N = 8)
2  (
3      input [N-1:0] A,
4      output [N-1:0] B
5  );
6
7      wire [N:0] ha;
8      assign ha[0] = A[N-1];
9
10     genvar i;
11     generate
12     for(i=0; i<N; i=i+1)
13     begin: twosFor
14         halfADDER halfADDER_inst1
15         (
16             .s(B[i]), // output s_sig
17             .cout(ha[i+1]), // output cout_sig
18             .a((A[i])^(A[N-1])), // input a_sig
19             .b(ha[i]) // input b_sig
20         );
21     end
22     endgenerate
23 endmodule
24

```

Figure 24 Twos Complementor Code

Output Multiplexer

The output multiplexer is a module with three inputs R, A, & B. A & B are two 8-bit inputs that feeds into the module. The R input acts as a selector that switches the output between input A and input B.

The A input of this module is the output of the ripple carry subtractor module, and the B input of the module is the output of twos complementor module. The R input is defined by the Cout output coming from the ripple carry subtractor module.

As mentioned in previous section, The Cout is low when Exponent of input A is greater than exponent of input B, and Cout is high for the other way around. When the Cout is low the output of this multiplexer is the output of ripple carry subtractor, and when the Cout is high the output of the module and the multiplexer is the output of the twos complement convertor module.

The code and RTL diagram for this module is included in the RTL diagram and code section from the modular exponent subtractor section.

2.3.2.3 Mantissa Shifter Multiplexer

The next module used to design a floating point adder is a 2-to-1 multiplexer. As shown in the block diagram in Figure 25, this kind of multiplexer consists of two inputs A and B, and one select input called S, and finally one output labelled R.

Mantissa Shifter Multiplexer Block Diagram

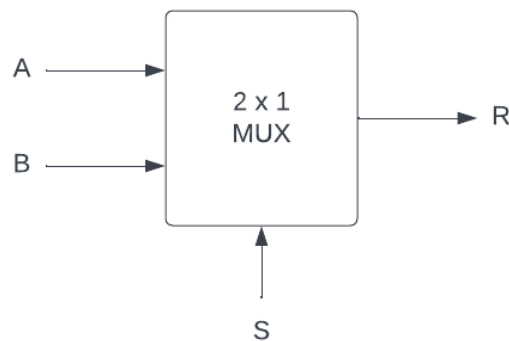


Figure 25 Mux Block Diagram

The output of the multiplexer is dependent on the select input to the multiplexer which connects one of the inputs to the output at a time. Since there are only two input signals we only require a 1 bit select signal to link one of the inputs to the output.

The objective of this Mantissa Shifter Multiplexer is to connect the lower input out of two inputs to the input of a right shifter unit that will be discussed in the next section. The decision to select which input is the lower one is taken by Cout of the exponent subtractor module.

Mantissa Shifter Multiplexer RTL Diagram

The figure below, Figure 26, is the RTL diagram for the multiplexer designed to output the mantissa to be feed to the shifter unit.

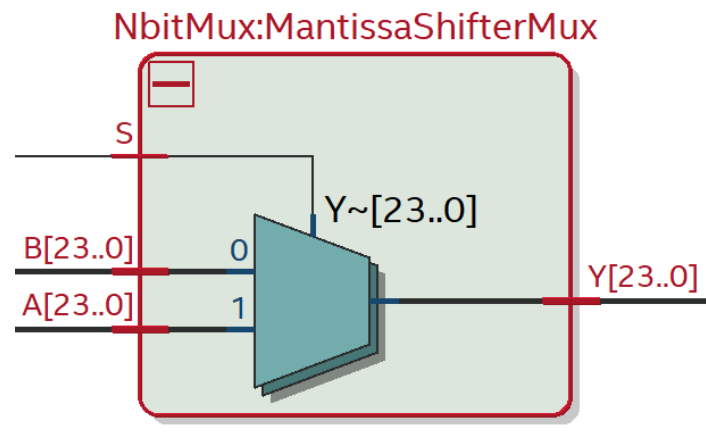


Figure 26 Mantissa MUX RTL

As seen in the RTL diagram above, the multiplexer takes two 24 bit inputs, one bit of a select input, and a 24 bit output. The two inputs are 23-bit mantissa of input A and 23 bit mantissa for input B, both the inputs are appended with a 1 as their most significant bit. The select input for this multiplexer comes from the Cout output of the exponent subtractor module. The appended 1 to the mantissa inputs are to take care of the hidden bit implied in the IEEE 754 standard of floating point binary format.

Mantissa Shifter Multiplexer Verilog Code

```

1  module NbitMux #(parameter N)
2  (
3      input [N-1:0] A, B, //declare data inputs
4      input S, //declare select input
5      output [N-1:0] Y //declare output
6  );
7      assign Y = S==1 ? A : B; //select input
8  endmodule
9  |

```

Figure 27 Mantissa MUX Code

Figure 27, shows the Verilog code for the mantissa shifter multiplexer. The select input coming in from Cout output of the exponent subtractor decides which input goes through the multiplexer and gets inputted into the shifter unit. If the Cout value is high, mantissa A gets selected for shifter unit and if Cout is low, mantissa B gets selected for the output of the shifter unit.

This operation is described in the truth table below:

Input 1	Input 2	Select	Output
A	B	1	A
A	B	0	B

Table 5 MUX Truth Table

2.3.2.4 Mantissa Right Shifter

For the next module of the floating point adder, we implemented a Barrel Shifter type of right shifter unit. A barrel shifter is a combinational circuit that facilitates right shift for this adder. Unlike regular shifter a barrel shifter is a sequential circuit [20].

If we were to use a regular register based shifter, a 24 bit data shift would take around 24 clock cycles to process the shift. However, with this barrel shifter the module will only need one clock cycle to compute the shift.

Mantissa Right Shifter Block Diagram

The block diagram for the mantissa barrel shifter is shown down below in Figure 28. The first level showcases a 4 bit right shift, the second level shows a 2 bit right shift, and the last level shows a 1 bit right shift.

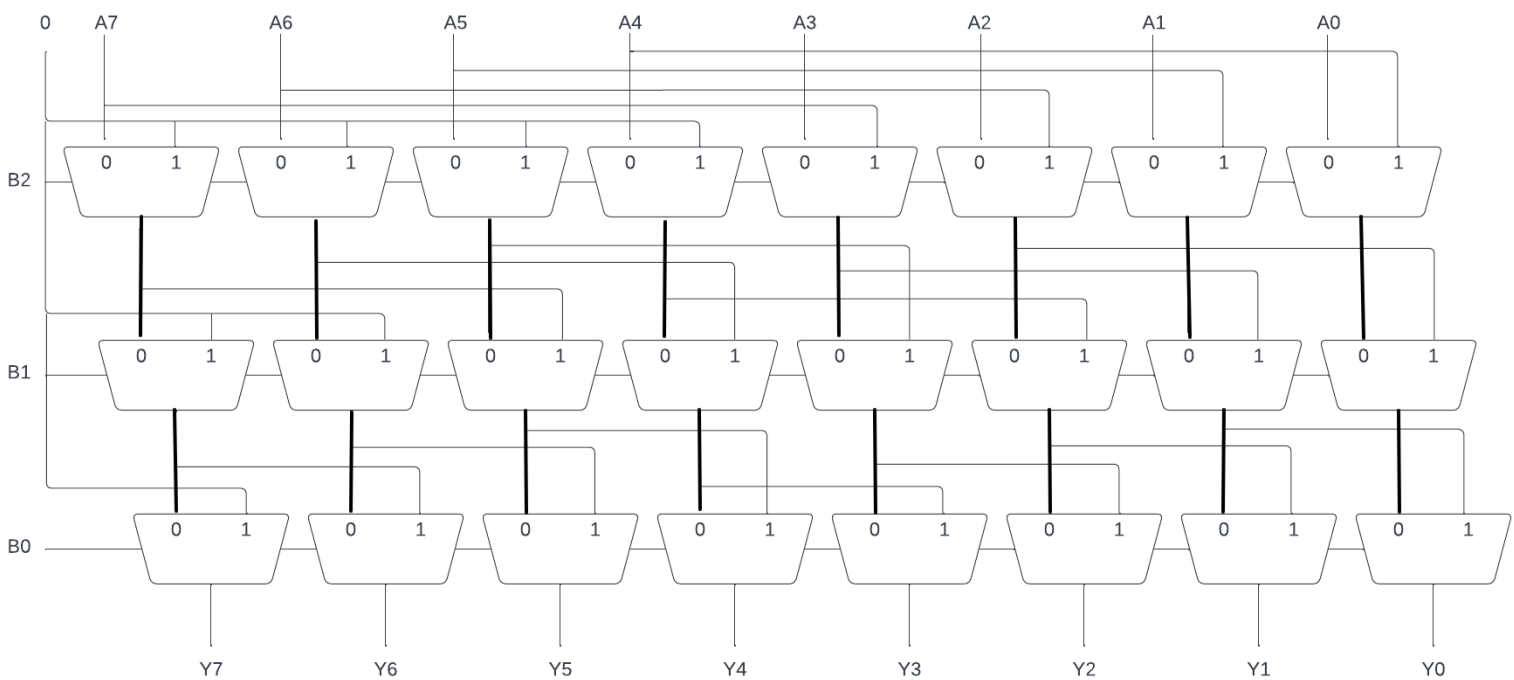


Figure 28 Right Shifter Block Diagram

Mantissa Right Shifter Verilog Code

The following figure, Figure 29, shows the code for a mantissa right shifter unit used to implement the floating point adder [21].

The mantissa right shifter unit takes in the smaller mantissa from two inputs, which is being selected by the multiplexer described above. The outputted mantissa from multiplexer then gets inputted into this barrel shifter that uses multiplexers to shift the mantissa by the desired shift value.

This shift value is achieved from the exponent subtractor unit module. The mantissa is shifted the same value as of the difference between the two exponents.

```

1  module BarrelShifter(
2      input [23:0] In,
3      output [23:0] out,
4      input [4:0] shift
5  );
6
7      wire [23:0] a;
8      genvar i;
9
10     generate
11     begin:b1
12         for(i=0; i<23; i=i+1)
13             begin:b2
14                 Mux M(In[i] , In[i+1] , shift[0] , a[i]);
15             end
16             Mux M1(In[23] , 1'b0 , shift[0] , a[23]);
17         end
18     endgenerate
19
20     wire [23:0] a1;
21     genvar j , k;
22
23     generate
24     begin:b3
25         for(j=0; j<22; j=j+1)
26             begin:b4
27                 Mux M2(a[j] , a[j+2] , shift[1] , a1[j]);
28             end
29             for(k=22; k<24; k=k+1)
30                 begin:b5
31                     Mux M3(a[k] , 1'b0 , shift[1] , a1[k]);
32                 end
33         end
34     endgenerate
35
36     genvar p , q;
37     wire [23:0] a2;
38
39     generate
40     begin:b6
41         for(p=0; p<20; p=p+1)
42             begin:b7
43                 Mux M4(a1[p] , a1[p+4] , shift[2] , a2[p]);
44             end
45             for(k=20; k<24; k=k+1)
46                 begin:b8
47                     Mux M5(a1[k] , 1'b0 , shift[2] , a2[k]);
48                 end
49         end
50     endgenerate

```

Figure 29 Right Shifter Code

2.3.2.5 Mantissa Adder Multiplexer

Similar to its predecessor, this module is also a two-to-one multiplexer. The two inputs of this multiplexer are mantissa of input A and input B. The selector for this multiplexer is the same select for its predecessor, the Cout output from the exponent subtractor module.

The purpose of this multiplexer is to select the mantissa out of two input mantissas with the higher exponent value. It essentially does the opposite of the previous mantissa multiplexer as it selects the mantissa with higher exponent value. The output of this multiplexer feeds into the mantissa adder.

This mantissa multiplexer has the same Verilog code, block diagram, and RTL diagram as the multiplexer explained in the section above. However, the truth table would be inverted for the desired operation as shown in table below:

Input 1	Input 2	Select	Output
A	B	0	A
A	B	1	B

Table 6 Multiplexer 2 Table

2.3.2.6 Mantissa Carry Look Ahead Adder

The next step in the floating point adder algorithm is adding the mantissa of input A with the mantissa of input B. The first mantissa for this adder operation will come directly from the input's mantissa. While the other input for this addition will come from the right shifter unit that we just described in the section above.

To implement the mantissa adder module in hardware implementation, we make use of a carry lookahead adder in Verilog language. A Carry Lookahead Adder is composed of a variable number of full adders cascaded together, similar to the ripple carry adder hardware construction. The number of full adders cascaded in this design, depends on the number of input bits to be added.

The difference between a ripple carry adder and a carry lookahead adder is that the carry lookahead adder is able to compute the Cout value using the input values. This makes sure that the Cout is produced before the full adder finishes its operation. The advantage of carry look ahead adder is the speed with which it performs these calculations. Since the following full adder doesn't need to wait for the previous full adder to finish operation, all the full adders can work in parallel and save a lot of computing time.

The drawback of using a carry lookahead adder over a ripple carry adder is that it utilizes a lot more logic than a simple ripple carry adder. Using a carry look ahead adder is a good lesson to showcase the balance between speed of execution and resources used when designing a module on FPGA [22].

Mantissa Carry Look Ahead Adder Block Diagram

The following figure, Figure 30, shows the block diagram for the carry lookahead logic. As seen in the block diagram, each bit of both the inputs feed into individual full adders which in turn return a S bit that computes the sum for that bit [23].

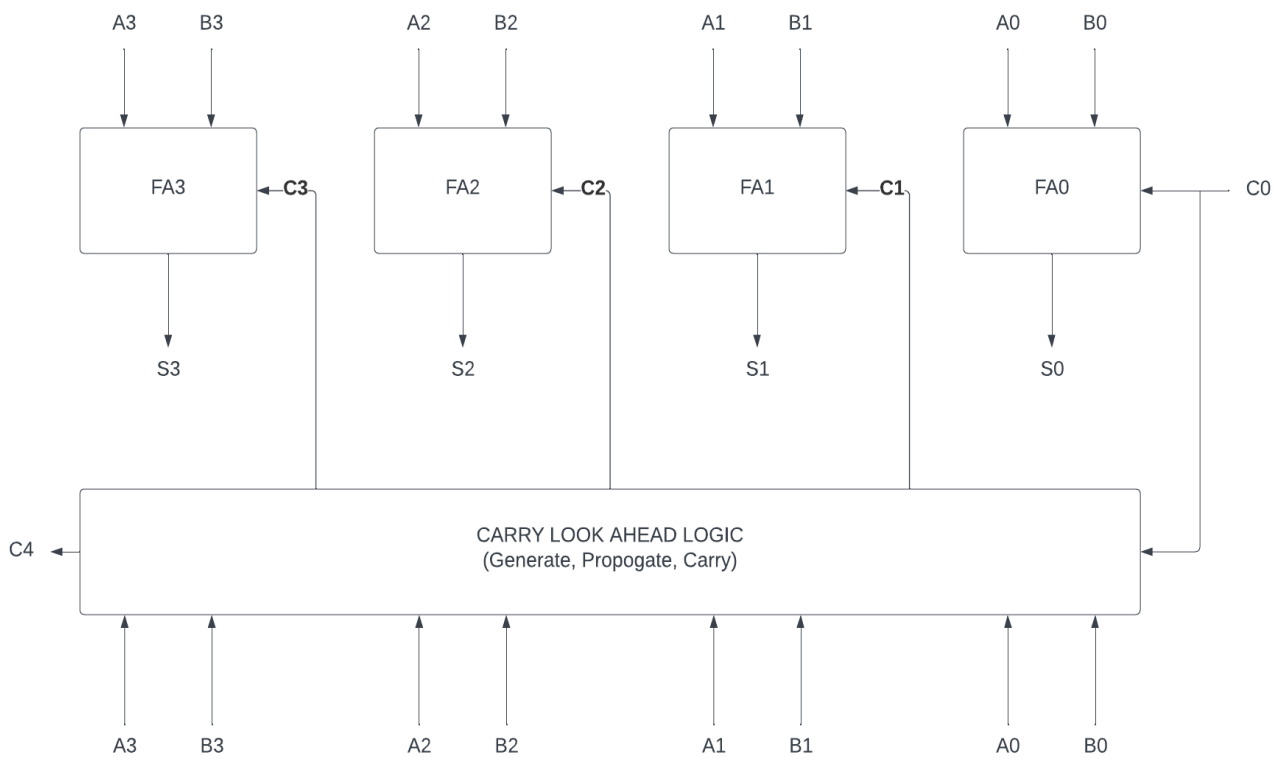


Figure 30 Mantissa CLA Block Diagram

The second module seen in that block diagram is the carry look ahead logic block. This block takes in the two inputs, the block's inside mechanism can be explained in two parts and two logic equation that ultimately computes a carry output. The three logic steps inside the carry look ahead logic block are:

- Compute generate variable:

We compute the generate variable by putting the two input bits through an AND gate and the output is the generate variable.

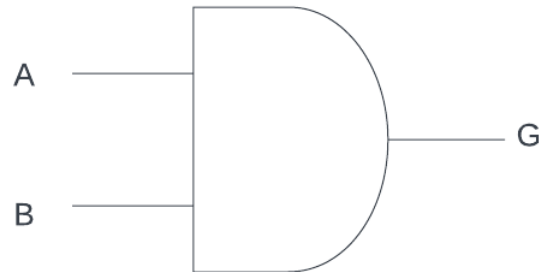


Figure 31 Generate AND

- Compute propagate variable:

We compute the propagate variable by putting the two input bits through an XOR gate and the output is the propagate variable.

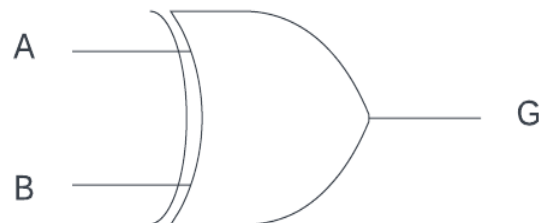


Figure 32 Propagate OR

- Compute Carry out.

The circuit schematic shows the way to compute the Carry output that is being computed by the carry look ahead logic block.

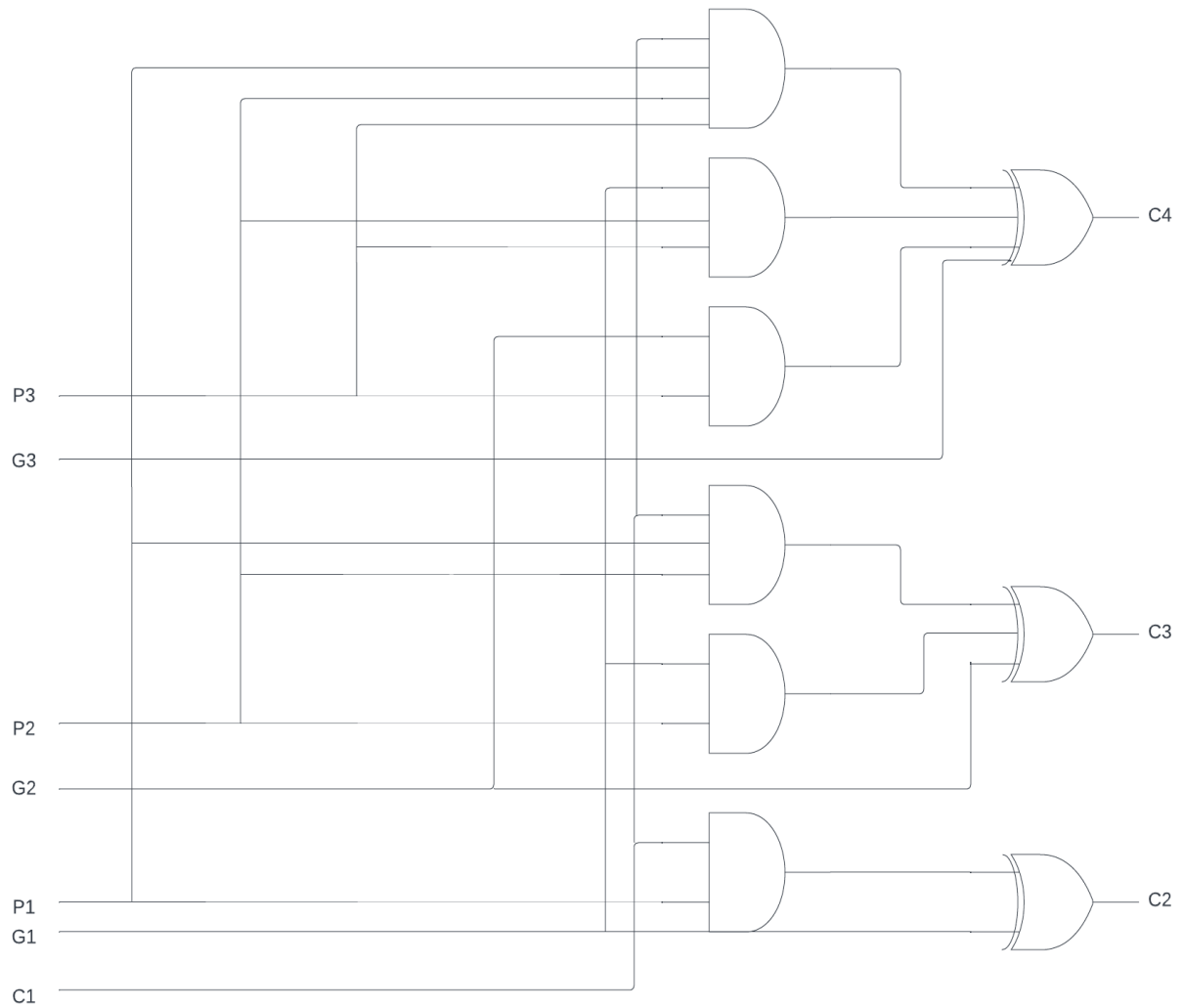


Figure 33 Carry Out Logic

Mantissa Carry Look Ahead Adder Verilog Code

The following figure, Figure 34, showcases the Verilog code for a parameterized carry look ahead adder that is implemented in this floating point adder design.

The carry look ahead adder takes in two 24 bit input labelled A and B, it also shows R as a 24 bit output and Cout as a carry out output. Input A of the adder comes from the right shifter unit explained in previous section that contains the bit shifted mantissa from one of the inputs. Input B contains the value from the mantissa adder multiplexer also explained in the previous section. The Cout output will further be used to normalize the mantissa result and increment the exponent which will be explained in further sections.

Lastly, the module contains Opcode input which defines whether the module does addition or subtraction (0 for addition, 1 for subtraction).

```

1  module CLAPParameter #(parameter N = 24)
2  (
3      input [N-1:0] A,
4      input [N-1:0] B,
5      input Opcode,
6      output [N-1:0] R,
7      output Cout
8  );
9
10 wire [N:0] C;
11 wire [N-1:0] G, P, SUM;
12
13 assign C[0] = Opcode; // opCode = 0 for addition, opCode = 1 for subtraction
14
15 // Create the Full Adders
16 genvar i;
17 generate
18     for (i=0; i<N; i=i+1)
19     begin: FullAdderFor
20         FullAdder FullAdder_inst
21         (
22             .A(A[i]),
23             .B(B[i]),
24             .C(C[i]),
25             .Co(Opcode),
26             .R(SUM[i]),
27             .Cout()
28         );
29     end
30 endgenerate
31
32 genvar j;
33 generate
34     for (j=0; j<N; j=j+1)
35     begin: TermGenerator
36         assign G[j] = A[j] & B[j]; // Create the Generate (G) Terms: Gi=Ai*Bi
37         assign P[j] = (A[j] | B[j]); // Create the Propagate Terms: Pi=Ai+Bi
38         assign C[j+1] = G[j] | (P[j] & C[j]); // Create the Carry Terms
39     end
40 endgenerate
41
42 assign R = SUM;
43 assign Cout = C[N];
44
45 endmodule
46

```

Figure 34 Carry Lookahead Adder Code

2.3.2.7 Exponent Increment Multiplexer

Similar to its predecessor, this module is also a two-to-one multiplexer. The two inputs of this multiplexer are exponents of input A and input B. The selector for this multiplexer is the same select for its predecessor, the Cout output from the exponent subtractor module.

The purpose of this multiplexer is to select the exponent out of two input exponents with the higher exponent value. The output of this multiplexer feeds into the exponent incrementor module that we will discuss in later section.

This exponent multiplexer has the same Verilog code, block diagram, and RTL diagram as the multiplexer explained in the sections above. However, the truth table would be inverted for the desired operation as shown in table below:

Input 1	Input 2	Select/Cout	Output
EA	EB	1	EA
EA	EB	0	EB

Table 7 Multiplexer 3 Table

2.3.2.8 Controlled Incrementor

In this section of the thesis, we will move on to the next module of the floating point adder unit, the controlled exponent incrementor. This module is similar to the ripple carry adder we discussed earlier in the thesis, with slight modifications for our purposes.

The controlled exponent incrementor module is built of eight different full adders cascaded together with the Cout of each full adder being outputted to the input of the next full adder. Additionally, the first input of each full adder is connected to the input and the other input of those full adders are all hard coded with zeroes. Except for the very first full adder which takes its second input from the other input to the module.

Controlled Incrementor Block Diagram

The following figure, Figure 35, shows the block diagram used for implementing the controlled incrementor module. The block diagram shown below shows the block diagram for incrementing a 4-bit number based on select input. The modification to make this 8-bit incrementor is shown and explained in the sections below.

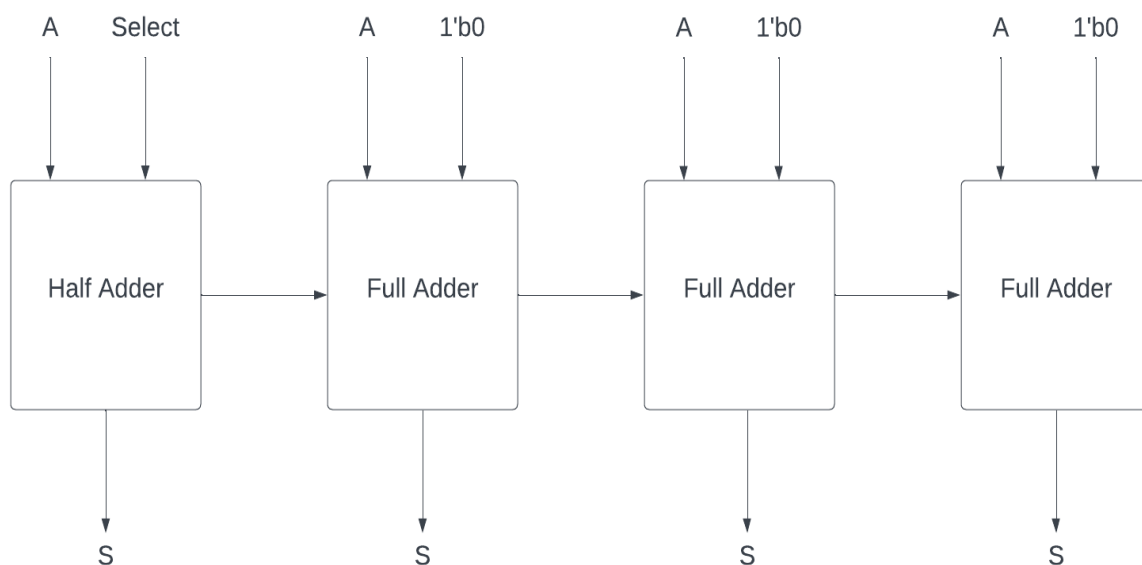


Figure 35 Controlled Incrementor Block

Controlled Incrementor RTL Diagram

As shown in Figure 36, the RTL diagram of the controlled exponent incrementor shows the 8-bit input labelled E, and another 1-bit input labelled select, in addition there is an 8-bit output labelled as Out.

Each individual bit of the 8-bit input comes directly from the output of the exponent incrementor multiplexer that was discussed in the previous section. Each bit of this 8-bit input feeds into seven different full adder and one half adder. The other 1-bit input called select goes into the first half adder. The output of the first half adder gets cascaded through to the next full adders and the outputs are all concatenated together to form the 8-bit output that is shown coming out of this RTL diagram.

The output of this controlled incrementor depends of the select input. The select input comes from the Cout output of the carry look ahead adder. If the select is high it signifies that the exponent must be incremented to be normalized for final output. If the select is low the exponent is outputted as it was inputted. This output makes up for the final exponent part of the 32-bit result.

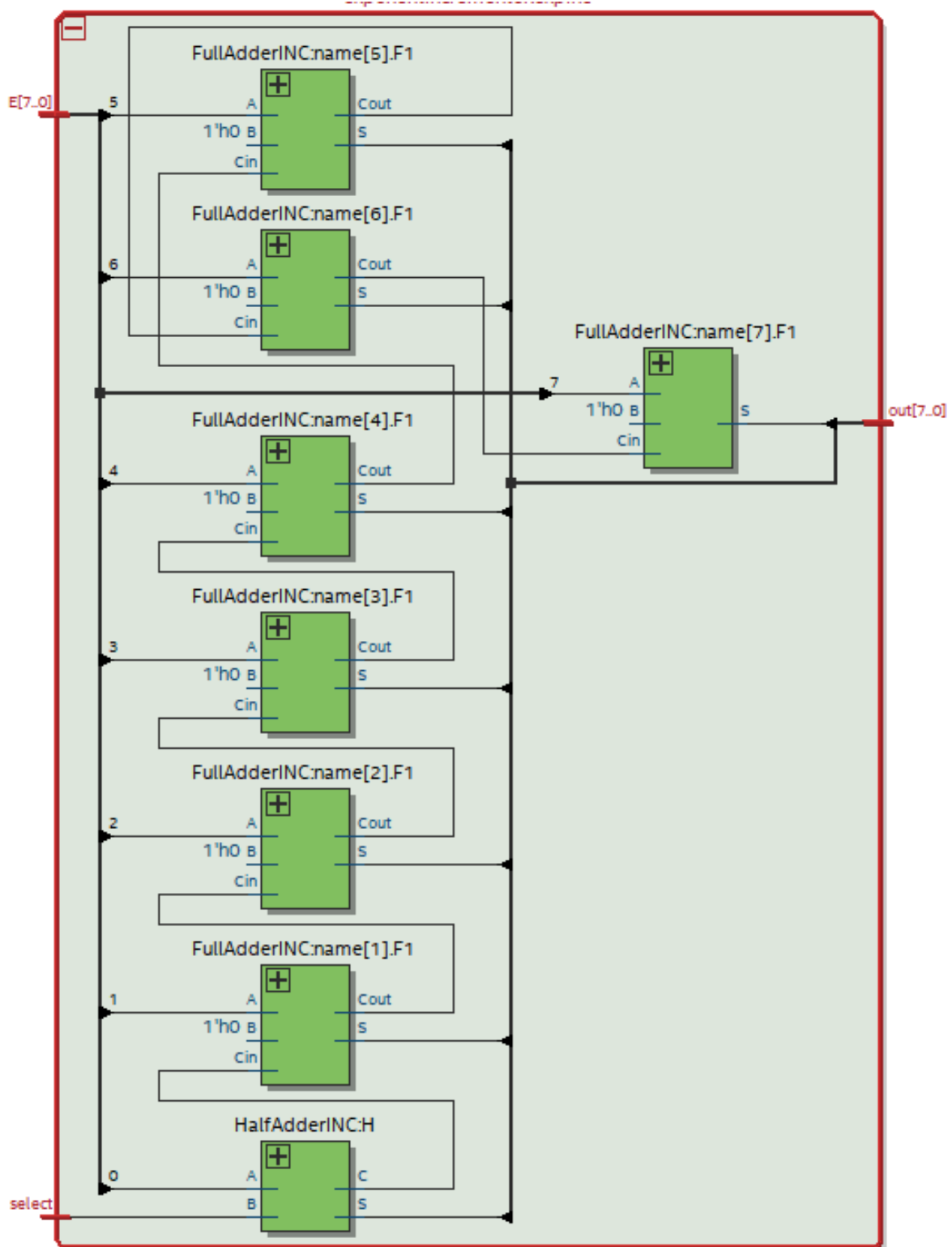


Figure 36 Controlled Incrementor RTL

Controlled Incrementor Verilog Code

The following figure, Figure 37, shows the Verilog code used to design the controlled incrementor. The module is named exponent incrementor as it takes in the lower exponent as input and increments it depending on the select input but shown in the code below.

```
1  module exponentIncrementor
2  (
3      input select,
4      input [7:0] E,
5      output [7:0] out
6  );
7
8      wire [7:0]w;
9      wire [7:0]cin;
10
11     assign w =(select==1'b1)?1:0;
12
13     HalfAdderINC H(E[0],w[0],out[0],cin[0]);
14
15     genvar j;
16     generate
17     begin
18
19         for(j=1;j<8;j=j+1)
20
21             begin: name
22
23                 FullAdderINC F1(E[j],w[j],cin[j-1],out[j],cin[j]);
24
25             end
26         end
27     endgenerate
28
29 endmodule
30
```

Figure 37 Controlled Incrementor Code

2.3.2.9 Mantissa Normalizer

The final module for this floating point adder is the mantissa normalizer. The mantissa normalizer is the same module that used before for right shifting the mantissa before the mantissa addition carried out by carry lookahead adder. The mantissa normalizer module is the same module called mantissa right shifter module that was explained in the section above.

The 24-bit input for this particular module comes from the output of the carry look ahead adder module that computes the mantissa addition. The other 5-bit input for this right shifter mantissa normalizer input comes from the Cout output of the same carry look ahead adder which signifies how much the mantissa must be shifted (0 bits or 1 bit).

The mantissa normalizer shifts the mantissa addition output only when the Cout output of the carry look ahead adder is high. The high value from the adder signifies that the mantissa addition has a carry of 1 and that signifies that the mantissa needs to be shifted to be normalized for the final output.

The output of this right shifter is the final mantissa value used to represent the 32 bit binary floating point result.

Refer to the 'Mantissa Right Shifter' section for Verilog code, block diagram, and the RTL diagram for this final module.

2.4 Floating Point Adder Results

The whole floating point adder unit was tested on Quartus' ModelSim simulation software using testbenches and waveforms. The design simulation involved generating setup scripts for the simulator, compiling simulation models, running the simulation, and viewing the results.

2.4.1 Floating Point Adder Compilation Report


Flow Summary	
 <<Filter>>	
Flow Status	Successful - Mon Mar 27 07:08:50 2023
Quartus Prime Version	18.1.0 Build 625 09/12/2018 SJ Lite Edition
Revision Name	FPUAdder
Top-level Entity Name	FPUAdder
Family	MAX 10
Device	10M08DAF484C8G
Timing Models	Final
Total logic elements	348 / 8,064 (4 %)
Total registers	0
Total pins	111 / 250 (44 %)
Total virtual pins	0
Total memory bits	0 / 387,072 (0 %)
Embedded Multiplier 9-bit elements	0 / 48 (0 %)
Total PLLs	0 / 2 (0 %)
UFM blocks	0 / 1 (0 %)
ADC blocks	0 / 1 (0 %)

Figure 38 FPA Compilation Report

2.4.2 Floating Point Adder Testbench

A testbench is used to generate the stimulus and applies it to the implemented floating point adder and compare the results against our calculations based on the IEEE 754 floating point convertor online. This online easy to use convertor allows us to input a decimal input value and returns a binary or hexadecimal value in 32-bit binary encoding format or vice versa [24]. The design was synthesized using precision synthesis tools targeting the DE-1 SoC Max 10 FPGA machine family.

```

1  `timescale 1ns / 1ps
2  module FPUadder_tb;
3
4      // Inputs
5      reg [31:0] A;
6      reg [31:0] B;
7
8      // Outputs
9      wire [31:0] Out;
10
11     wire Exception;
12     wire Overflow;
13     wire Underflow;
14
15     wire SnanA, QnanA, InfA, ZeroA, SubNA, NormA;
16     wire SnanB, QnanB, InfB, ZeroB, SubNB, NormB;
17
18     // Instantiate the Unit Under Test (UUT)
19     FPUadder fpuadderTB
20     (
21         .A(A),
22         .B(B),
23         .Out(Out),
24         .Exception(Exception),
25         .Overflow(Overflow),
26         .Underflow(Underflow),
27         .SnanA(SnanA),
28         .QnanA(QnanA),
29         .InfA(InfA),
30         .ZeroA(ZeroA),
31         .SubNA(SubNA),
32         .NormA(NormA),
33         .SnanB(SnanB),
34         .QnanB(QnanB),
35         .InfB(InfB),
36         .ZeroB(ZeroB),
37         .SubNB(SubNB),
38         .NormB(NormB)
39     );
40

```

Figure 39 FPA Testbench

2.4.3 Floating Point Adder Simulation Results

Case A:

A:



B:



R:



Simulation Results:

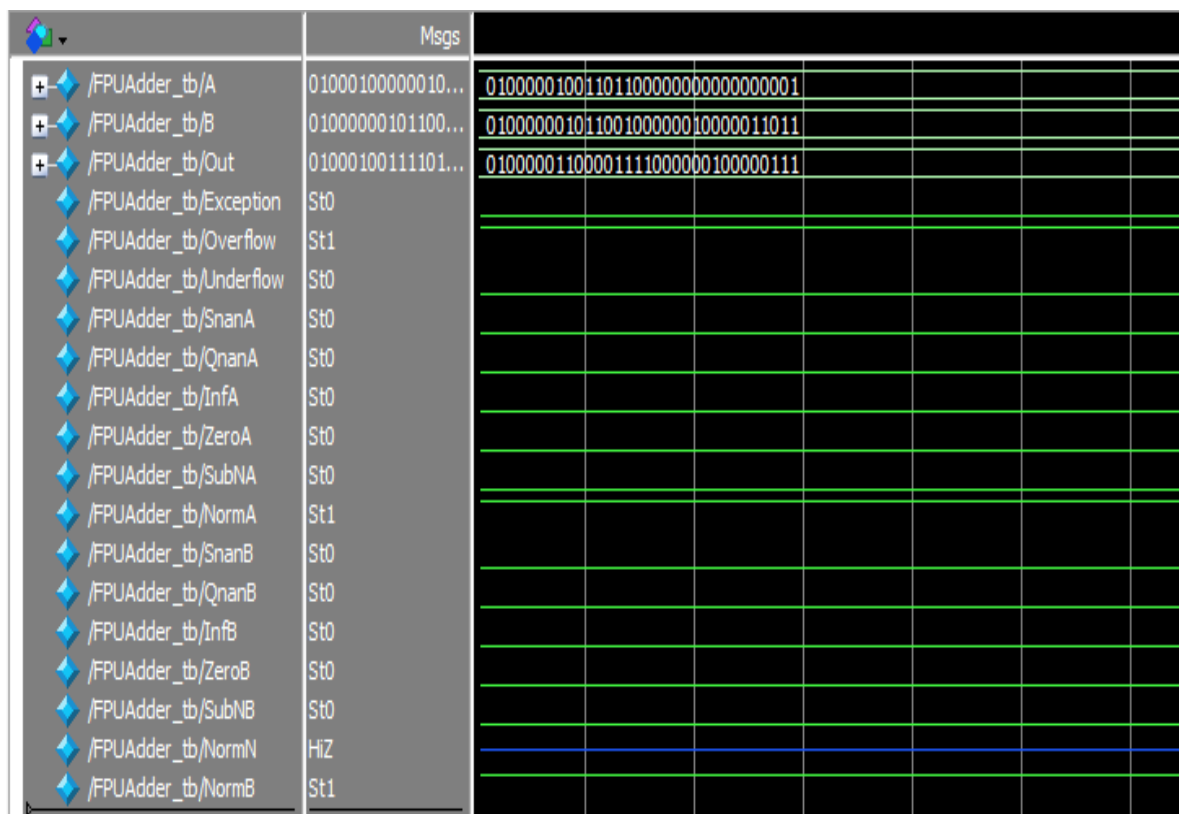
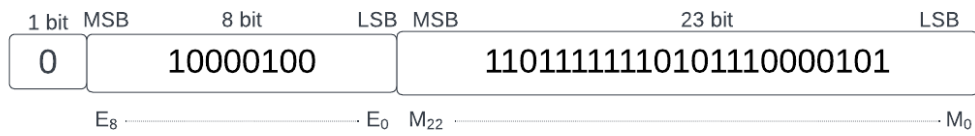


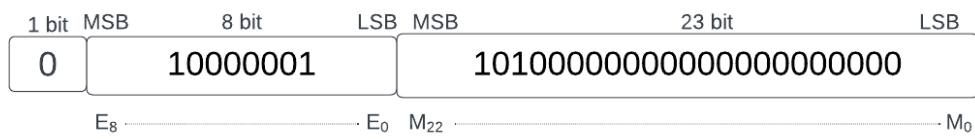
Figure 40 Case A Result

Case B:

A:



B:



R:



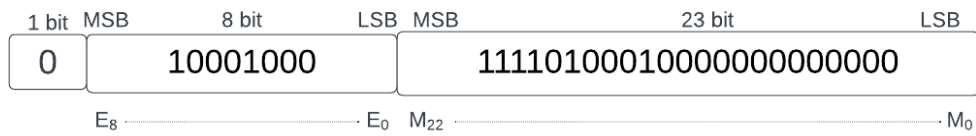
Simulation Result:

	Msgs				
/FPUAdder_tb/A	01000100000010...	01000010011011111110101110000101			
/FPUAdder_tb/B	01000000101100...	01000000110100000000000000000000			
/FPUAdder_tb/Out	01000100111101...	01000010100001001111010111000010			
/FPUAdder_tb/Exception	St0				
/FPUAdder_tb/Overflow	St1				
/FPUAdder_tb/Underflow	St0				
/FPUAdder_tb/SnanA	St0				
/FPUAdder_tb/QnanA	St0				
/FPUAdder_tb/InfA	St0				
/FPUAdder_tb/ZeroA	St0				
/FPUAdder_tb/SubNA	St0				
/FPUAdder_tb/NormA	St1				
/FPUAdder_tb/SnanB	St0				
/FPUAdder_tb/QnanB	St0				
/FPUAdder_tb/InfB	St0				
/FPUAdder_tb/ZeroB	St0				
/FPUAdder_tb/SubNB	St0				
/FPUAdder_tb/NormN	HiZ				
/FPUAdder_tb/NormB	St1				

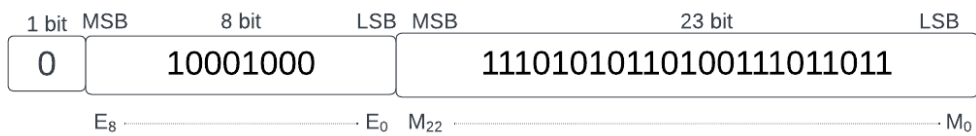
Figure 41 Case B Result

Case C:

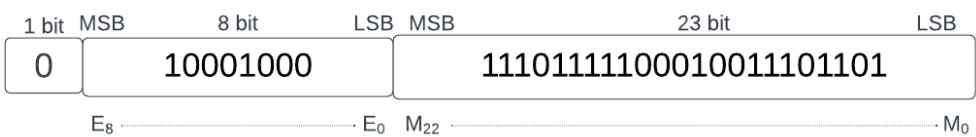
A:



B:



R:



Simulation Result:

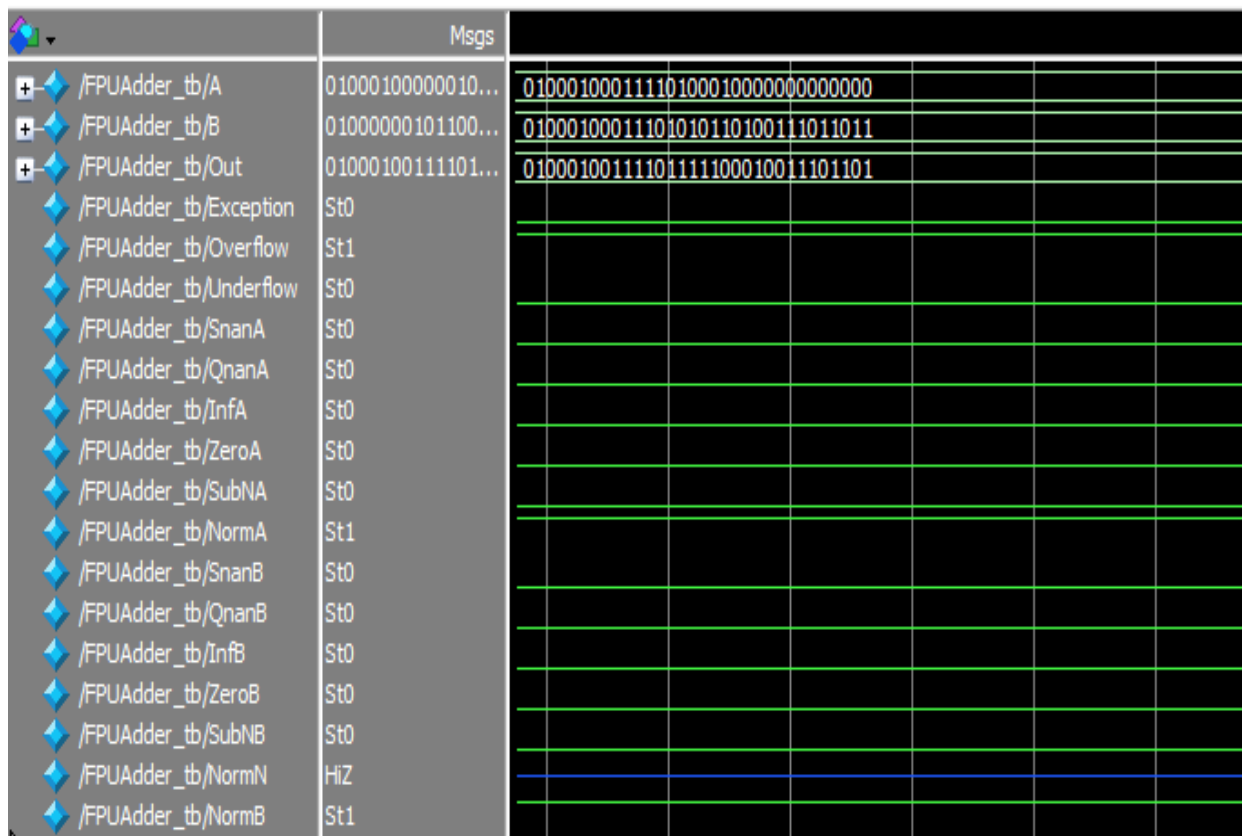


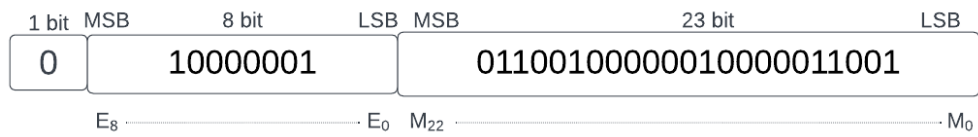
Figure 42 Case C Result

Case D:

A:



B:



R:



Simulation Result:

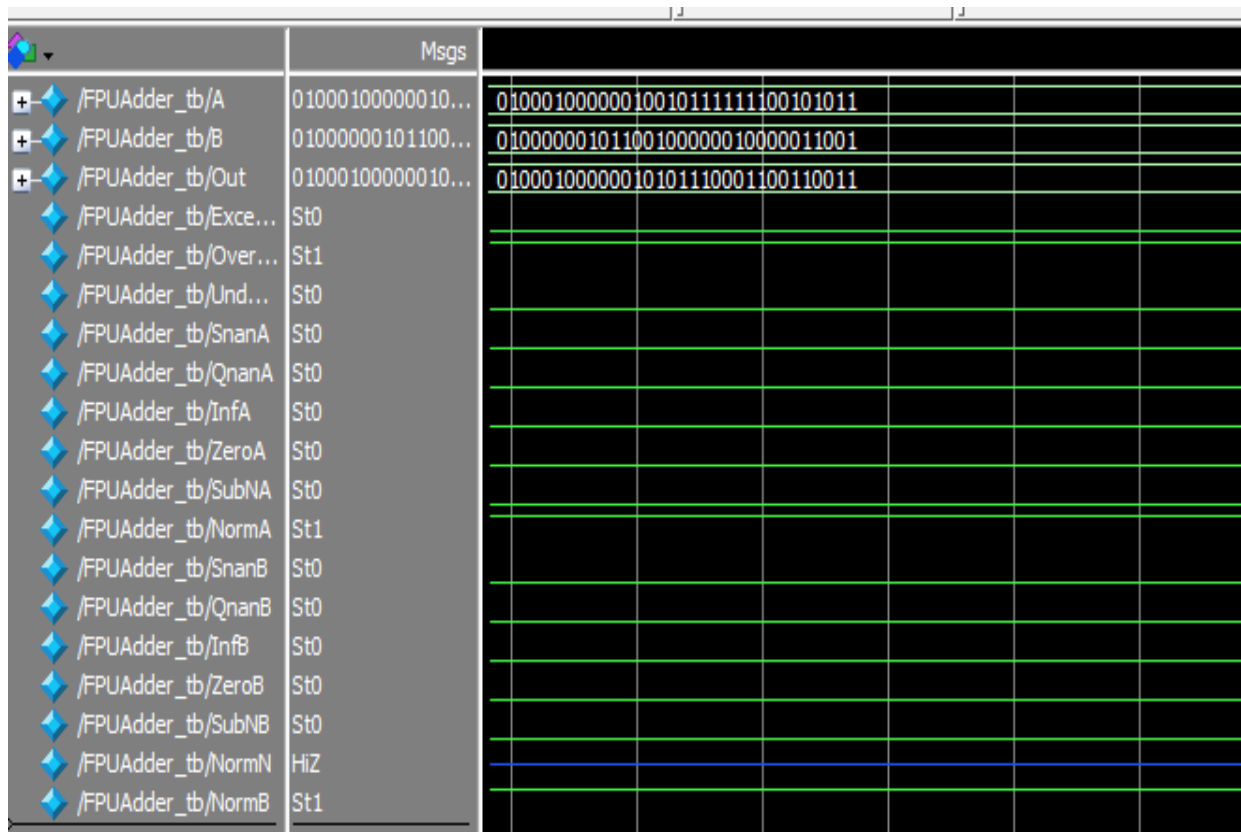


Figure 43 Case D Result

2.5 Conclusion

This section of the thesis presented an implementation of a floating point adder that supports the IEEE 754-2008 binary interchange format. The adder implements this algorithm using a carry look ahead adder for faster computation and used various different modules to compute the final output.

Chapter 3: Floating Point Subtractor

In this chapter, we describe an efficient implementation of an IEEE 754 single precision floating point subtractor targeted for DE-1 Cyclone V FPGA. Verilog is used to implement a technology-independent pipelined design. The subtractor implementation handles the overflow and underflow cases. Rounding is implemented to give more precision when using the Ripple Carry Subtractor for faster calculations. The Floating-Point Subtractor was verified by testbench simulations on ModelSim.

In this chapter we will dive deeper into the floating-point subtractor algorithm, architecture, code design, RTL diagram, and simulation results.

We will talk about the procedure in subtraction operations and a first look at the code design in a block diagram way followed by deeper understanding of code development.

Floating point subtraction is done by extracting signs, subtracting exponents, subtracting mantissa values, and shifting the mantissa for normalization.

Floating-Point Subtraction is a mirror image of floating-point addition which is why a lot of algorithm steps and modules would be similar or mirror of the previous chapter.

There are five basic phases of designing a Floating-Point Subtractor:

- 1) Check for Zeroes.
- 2) Align the Significands.
- 3) Subtract the Significands.
- 4) Normalize the Significand
- 5) Normalize the Exponent if needed.

3.1 Floating Point Subtraction Algorithm

As described in the above topics, floating point number is in the format of:

$$Z = (-1^S) * 2^{(E - \text{Bias})} * (1.M)$$

To subtract two floating point numbers A & B the different steps to follow are [25]:

- 1) Extracting signs, exponents and mantissas of both A and B numbers.
- 2) Calculating the output sign.
- 3) Treating the special cases.
- 4) Finding out the data types of numbers given
- 5) Subtracting the two exponents.
- 6) Shifting the lower exponent number mantissa to the right.
- 7) Subtraction of the mantissa values
- 8) Normalizing mantissa by bit shifting.
- 9) Detecting exception, overflow, and underflow.

3.1.1 Floating-Point Subtraction Example

A = 50.5 (base 10)

B = 21.25 (base 10)

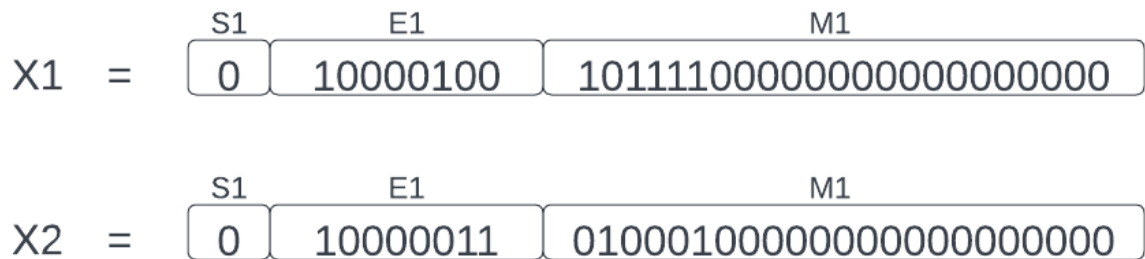


Figure 44 Binary Representation Sub Example

1) $S1 = 0, E1 = 10000100, M1 = 101111000000000000000000$

$S2 = 0, E2 = 10000011, M2 = 010001000000000000000000$

2) Exponent Subtraction

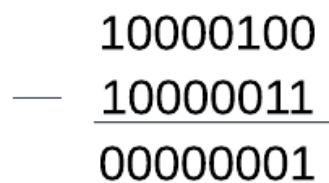


Figure 46 Sub Exponent Subtraction

$E = 00000001 = 1_{10}$

3) Right Shift Mantissa M2 by $E1 - E2 (1)$

$1.M2 = 1.010010000000000000000000$

Shifted Mantissa = $0.100100000000000000000000$

4) Subtract the Mantissa

$$\begin{array}{r}
 1.101111000000000000000000 \\
 - 0.100100000000000000000000 \\
 \hline
 1.001011000000000000000000
 \end{array}$$

Figure 47 Subtract Mantissa Subtraction

5) No normalization needed

6) No exponent incrementation needed.

7) Result

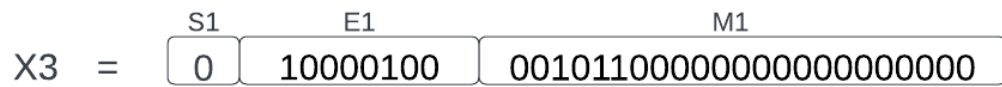


Figure 48 FPS Example Result

3.2 Floating Point Subtractor Flowchart

The below, Figure 49, showcases a typical flowchart that is used to design a floating point subtractor. The figure shows a step by step narrative and displays the high level functions that is required to compute floating point subtraction. The flowchart shows block level diagram and each block or element is implemented in hardware and is described in detail in the following topics of the thesis [26].

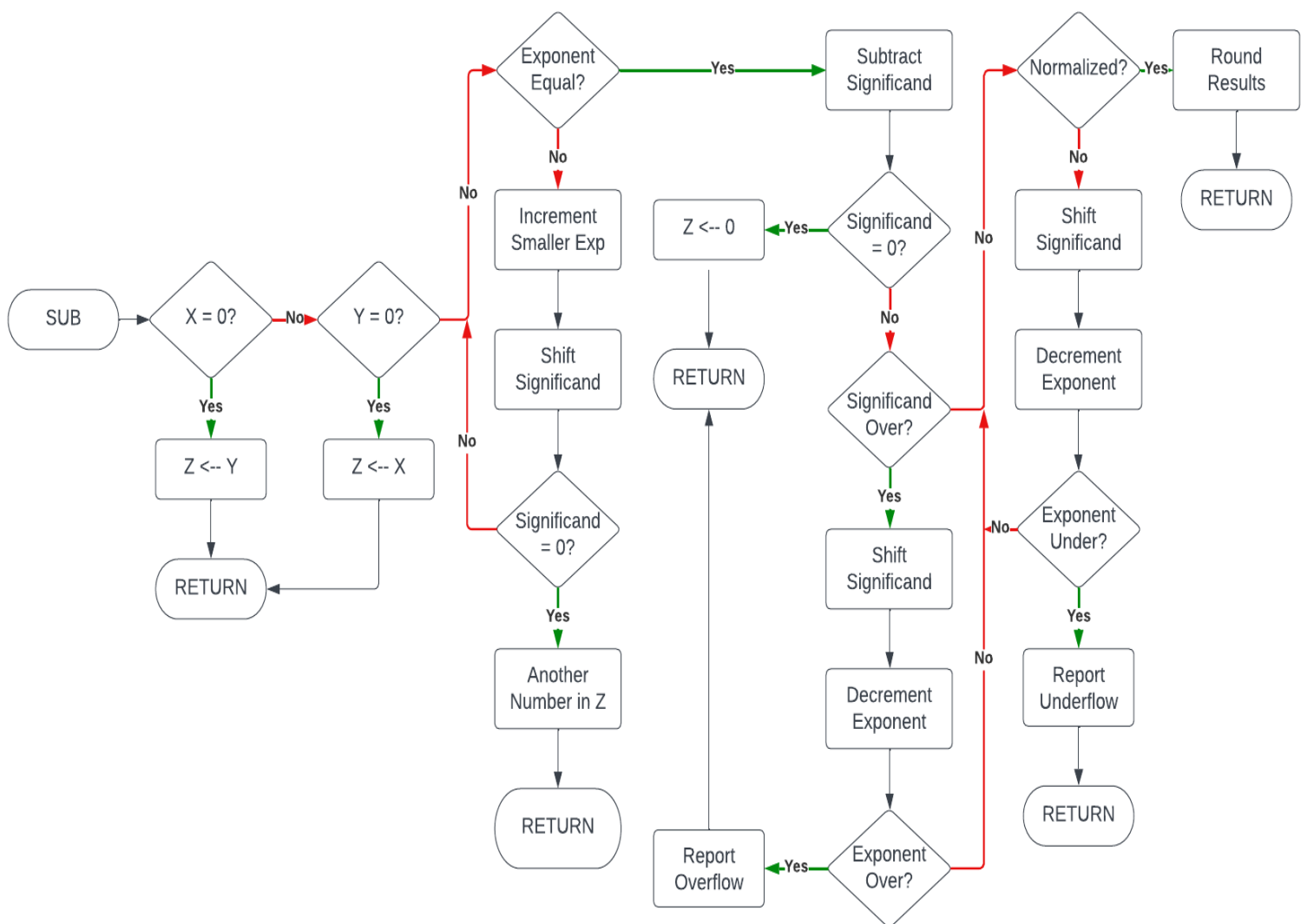


Figure 49 Floating Point Subtractor Flowchart

3.3 Floating Point Subtractor Hardware

In this section of the thesis we will start explaining and diving deeper into the hardware implementation of the floating point subtractor. This section will start by elaborating the flowchart further with help of showcasing the hardware architecture used to design the module followed by detailed description of each module used in the architecture.

After understanding the theory of hardware implementation and the architecture of floating point subtractor the thesis will show the code development that achieved out final objective of building this floating point unit [27].

3.3.1 Floating Point Subtractor Hardware Architecture

The below figure, Figure 50, showcases the hardware architecture that was designed and coded to implement synthesizable 32-bit floating point subtractor using Verilog.

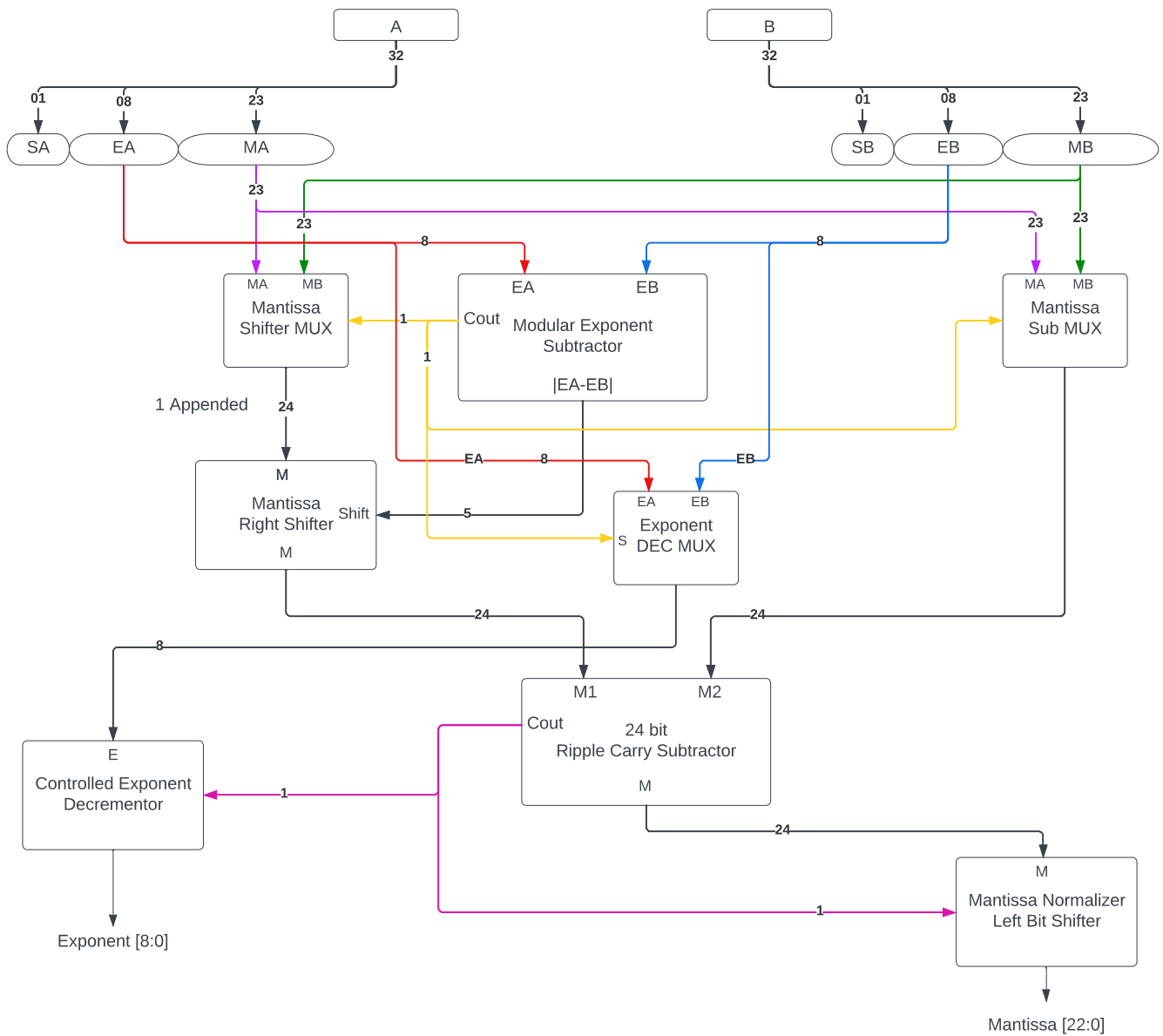


Figure 50 Floating Point Subtractor Architecture

This floating point architecture uses a total of eight modules that serve various unique purposes in making the design work. The modules are:

- Mantissa Shifter Multiplexer
- Modular Exponent Subtractor
- Exponent Decrement Multiplexer
- Controlled Decrement
- Mantissa Subtract Multiplexer
- Mantissa Right Shifter
- Ripple Carry Subtractor
- Mantissa Normalizer

3.3.2 Floating Point Subtractor Hardware Implementation

In this section, we will discuss the hardware implementation designed for the floating point subtractor and explain each module and each algorithm step in detail.

3.3.2.1 Sign Bit Calculation

Subtracting two positive numbers will result in a positive number which makes this section easy for us since there will be another module to take care of subtraction. The table below shows sign operations for various cases:

A's Sign	Symbol	B's Sign	Operation
+	-	+	+
+	-	-	+
-	-	-	-
-	-	+	-

Table 8 Sign Operations

3.3.2.2 Modular Exponent Subtractor

This modular exponent subtractor is responsible for subtracting the exponent of the second input from the exponent of the first input. This module of hardware description language ensures that the exponent difference value is absolute in nature. Before the subtraction operation is performed the program doesn't know which exponent is higher in value. The modular exponent subtractor allows us to not just compute the absolute exponent difference, it also allows us to identify the larger exponent which further identifies the exponent that will be used for the incrementor module and ultimately computing the result of the entire operation.

To get detailed description of the modular exponent subtractor, and understand all the components of this module by help of block diagrams, RTL diagrams, and code snippets, please refer to section 2.3.2.2 in chapter 2: 32-bit Floating Point Adder.

3.3.2.3 Mantissa Shifter Multiplexer

The next module used to design a floating point adder is a 2-to-1 multiplexer. As shown in the block diagram in Figure 51, this kind of multiplexer consists of two inputs A and B, and one select input called S, and finally one output labelled R.

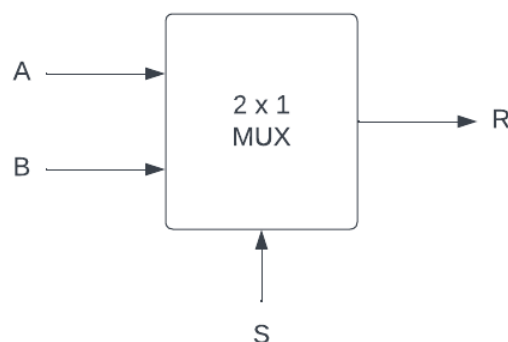


Figure 51 Mux Block Diagram 2

The output of the multiplexer is dependent on the select input to the multiplexer which connects one of the inputs to the output at a time. Since there are only two input signals we only require a 1 bit select signal to link one of the inputs to the output.

The objective of this Mantissa Shifter Multiplexer is to connect the lower input out of two inputs to the input of a right shifter unit that will be discussed in the next section. The decision to select which input is the lower one is taken by Cout of the exponent subtractor module.

The select input coming in from Cout output of the exponent subtractor decides which input goes through the multiplexer and gets inputted into the shifter unit. If the Cout value is high, mantissa A gets selected for shifter unit and if Cout is low, mantissa B gets selected for the output of the shifter unit.

This operation is described in the truth table below:

Input 1	Input 2	Select	Output
A	B	1	A
A	B	0	B

Table 9 MUX Truth Table

To get detailed description of the Mantissa Shifter Multiplexer and understand all the components of this module by help of block diagrams, RTL diagrams, and code snippets, please refer to section 2.3.2.3 in chapter 2: 32-bit Floating Point Adder.

3.3.2.4 Mantissa Right Shifter

For the next module of the floating point adder, we implemented a Barrel Shifter type of right shifter unit. A barrel shifter is a combinational circuit that facilitates right shift for this adder. Unlike regular shifter a barrel shifter is a sequential circuit.

If we were to use a regular register based shifter, a 24 bit data shift would take around 24 clock cycles to process the shift. However, with this barrel shifter the module will only need one clock cycle to compute the shift.

To get detailed description of the Mantissa Right Shifter and understand all the components of this module by help of block diagrams, RTL diagrams, and code snippets, please refer to section 2.3.2.4 in chapter 2: 32-bit Floating Point Adder.

3.3.2.5 Mantissa Subtractor Multiplexer

Similar to its predecessor, this module is also a two-to-one multiplexer. The two inputs of this multiplexer are mantissa of input A and input B. The selector for this multiplexer is the same select for its predecessor, the Cout output from the exponent subtractor module.

The purpose of this multiplexer is to select the mantissa out of two input mantissas with the higher exponent value. It essentially does the opposite of the previous mantissa multiplexer as it selects the mantissa with higher exponent value. The output of this multiplexer feeds into the mantissa adder.

This mantissa multiplexer has the same Verilog code, block diagram, and RTL diagram as the multiplexer explained in the section above. However, the truth table would be inverted for the desired operation as shown in table below:

Input 1	Input 2	Select	Output
A	B	0	A
A	B	1	B

Table 10 Multiplexer 2 Table

3.3.2.6 Mantissa Ripple Carry Subtractor

The next step in the floating point subtractor algorithm is subtracting the mantissa of input A with the mantissa of input B. The first mantissa for this subtractor operation will come directly from the input's mantissa. While the other input for this subtraction will come from the right shifter unit that we just described in the section above.

To implement the mantissa subtractor module in hardware implementation, we make use of a ripple carry subtractor in Verilog language. A ripple carry subtractor is composed of a variable number of full adders cascaded together. The number of full adders cascaded in this design, depends on the number of input bits to be added.

The 8-bit version of the ripple carry subtractor is described by using block diagram, RTL diagram, and Verilog code in the modular subtractor section above. In this section we will dive deeper into the ripple carry subtractor and show modifications needed to make the subtractor operate on 24 bit numbers.

Mantissa 24-bit Ripple Carry Subtractor RTL Diagram

The following figure, Figure 52, shows an RTL diagram of the 24 bit ripple carry subtractor that was used to compute the mantissa subtraction and the carry out.

As it can be seen in the RTL diagram, there are a total of 24 full adders to compute the operation for each bit of input and compute a total of 24 bit output.

As can also be seen in the RTL diagram the Cout output is computed based on the Carry output of the last full adder and the opcode (1 in case of subtraction, 0 in case of addition) input for the ripple carry subtractor. The Cout is low when Exponent of input A is greater than exponent of input B, and Cout is high for the other way around.

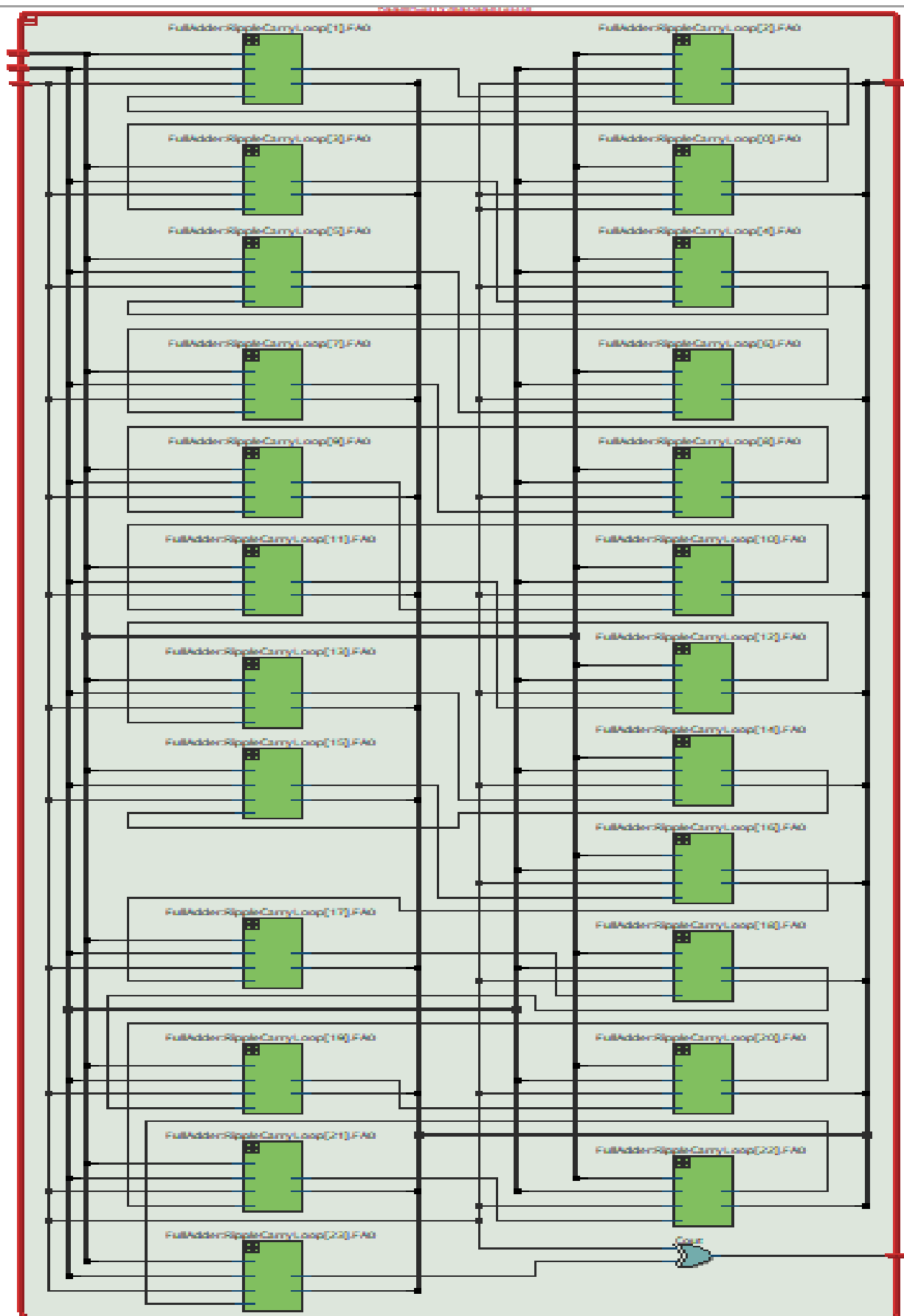


Figure 52 24 bit Ripple Carry Sub RTL

Mantissa 24-bit Ripple Carry Subtractor Verilog Code

The figure below, Figure 53, shows the Verilog code for the 24-bit ripple carry subtractor. The module consists of two 24-bit inputs, opcode (1 for negative), 24-bit output, and a carry output. The code mainly consists of one for loop that synthesizes a full adder 24 times for every bit of input and output.

```

1  module RippleCarrySub #(parameter N = 24)
2  (
3      input [N-1:0] A,
4      input [N-1:0] B,
5      input OpCode,
6      output [N-1:0] R,
7      output Cout
8  );
9      wire [N:0] C;
10     wire [N-1:0] SUM;
11     assign C[0] = OpCode;
12
13     genvar i;
14     generate
15     for(i=0; i<N; i=i+1)
16     begin: RippleCarryLoop
17         FullAdder FA0
18         (
19             .A(A[i]),
20             .B(B[i]),
21             .C(C[i]),
22             .Co(OpCode),
23             .R(SUM[i]),
24             .Cout(C[i+1])
25         );
26     end
27     endgenerate
28
29     assign Cout = C[0]^C[N];
30     assign R = SUM;
31
32 endmodule
33

```

Figure 53 24-bit Ripple Cary Sub Code

3.3.2.7 Exponent Decrement Multiplexer

Similar to its predecessor, this module is also a two-to-one multiplexer. The two inputs of this multiplexer are exponents of input A and input B. The selector for this multiplexer is the same select for its predecessor, the Cout output from the exponent subtractor module.

The purpose of this multiplexer is to select the exponent out of two input exponents with the higher exponent value. The output of this multiplexer feeds into the exponent decrement module that we will discuss in later section.

This exponent multiplexer has the same Verilog code, block diagram, and RTL diagram as the multiplexer explained in the sections above. However, the truth table would be inverted for the desired operation as shown in table below:

Input 1	Input 2	Select/Cout	Output
EA	EB	1	EA
EA	EB	0	EB

Table 11 Multiplexer 3 Table

3.3.2.8 Controlled Decrement

In this section of the thesis, we will move on to the next module of the floating point adder unit, the controlled exponent decrement. This module is similar to the ripple carry subtractor we discussed earlier in the thesis, with slight modifications for our purposes.

The controlled exponent decrement module is built of eight different full adders cascaded together with the Cout of each full adder being outputted to the input of the next full adder. Additionally, the first input of each full adder is connected to the input and the other input of those full adders are all hard coded with zeroes. Except for the very first full adder which takes its second input from the other input to the module. The second input of this decrement module is passed through an XOR gate along with an opcode input.

The XOR gate converts the select input into a twos complement version and performing the addition operation on the twos complement number would give us the decremented result.

Controlled Decrement Block Diagram

The following figure, Figure 54 shows the block diagram used for implementing the controlled decrement module. The block diagram shown below shows the block diagram for decrementing a 4-bit number based on select input. The modification to make this 8-bit decrement is shown and explained in the sections below.

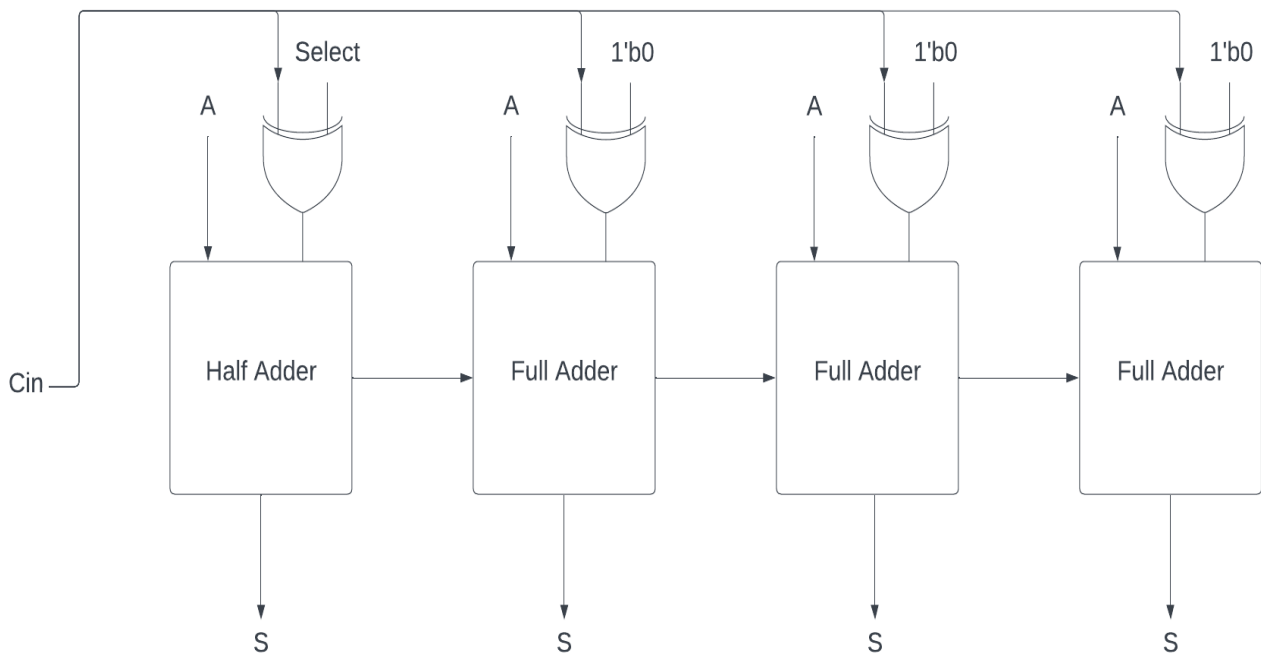


Figure 54 Controlled Decrement Block

Controlled Decrement RTL Diagram

As shown in Figure 55, the RTL diagram of the controlled exponent decrement shows the 8-bit input labelled E, and another 1-bit input labelled select, in addition there is an 8-bit output labelled as Out, and finally a 1-bit Cin input.

Each individual bit of the 8-bit input comes directly from the output of the exponent decrement multiplexer that was discussed in the previous section. Each bit of this 8-bit input feeds into seven different full adder and one half adder. The other 1-bit input called select goes into the first half adder. The output of the first half adder gets cascaded through to the next full adders and the outputs are all concatenated together to form the 8-bit output that is shown coming out of this RTL diagram.

The output of this controlled decrement depends of the select input. The select input comes from the Cout output of the carry look ahead adder. If the select is high it signifies that the exponent must be decremented to be normalized for final output. If the select is low the exponent is outputted as it was inputted. This output makes up for the final exponent part of the 32-bit result.

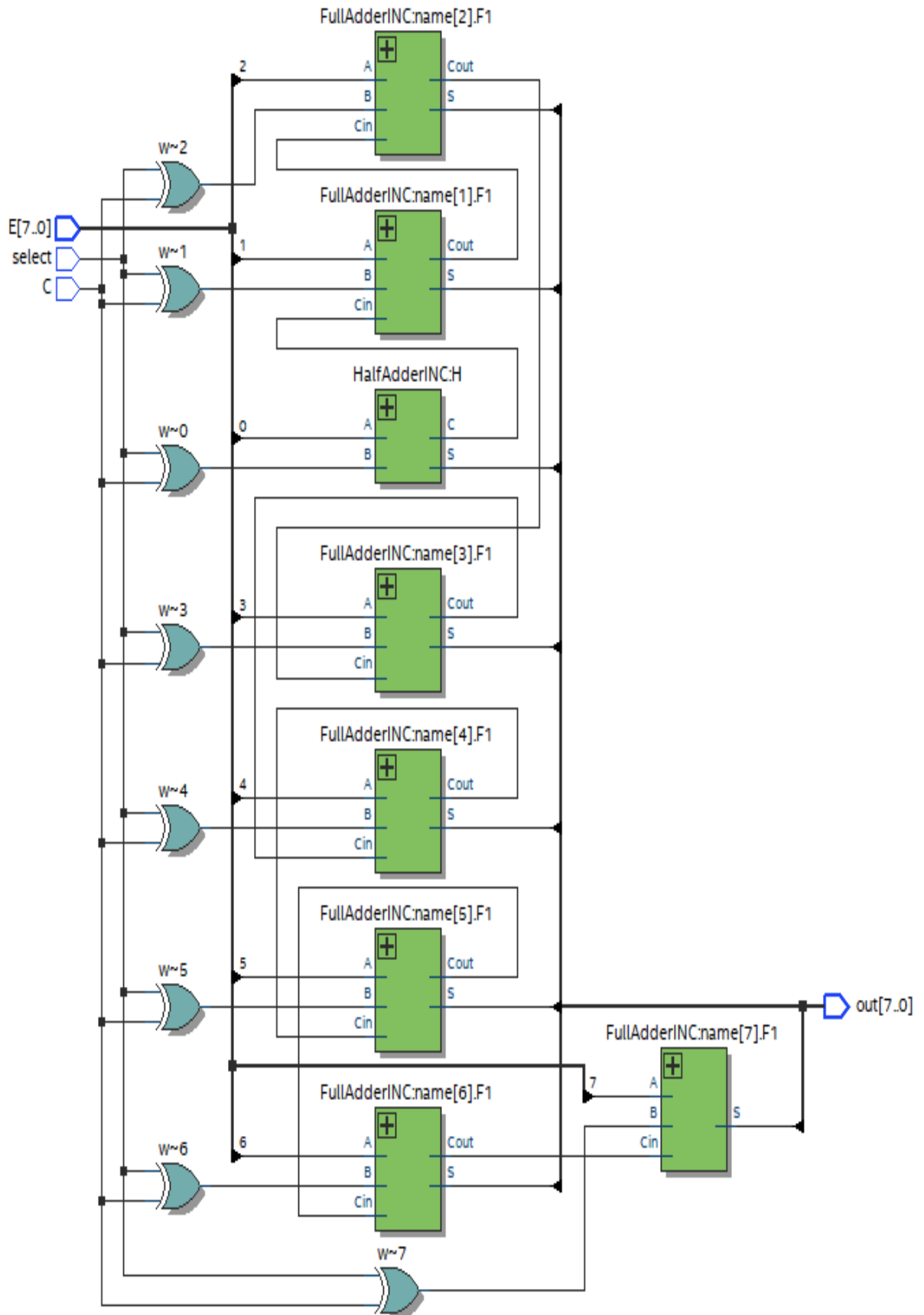


Figure 55 Controlled Decrement RTL

Controlled Decrement Verilog Code

The following figure, Figure 56, shows the Verilog code used to design the controlled decrement. The module is named exponent decrement as it takes in the lower exponent as input and decrements it depending on the select input but shown in the code below.

```
1  module decrementor
2  (
3      input select,
4      input [7:0] E,
5      input C,
6      output [7:0] out
7  );
8
9
10     wire [7:0]x;
11     wire [7:0]w;
12     wire [7:0]cin;
13
14     assign x =(select==1'b1)?1:0;
15     assign w[0] = x^C;
16     HalfAdderINC H(E[0],w[0],out[0],cin[0]);
17
18     genvar j;
19     generate
20     begin
21
22         for(j=1; j<8; j=j+1)
23
24             begin: name
25
26                 assign w[j] = x^C;
27
28                 FullAdderINC F1(E[j],w[j],cin[j-1],out[j],cin[j]);
29
30             end
31         end
32     endgenerate
33
34 endmodule
```

Figure 56 Controlled Decrement Code

3.3.2.9 Mantissa Normalizer

The final module for this floating point subtractor is the mantissa normalizer. The mantissa normalizer is a variation of the module that used before for right shifting the mantissa before the mantissa subtraction was carried out by ripple carry subtractor. The mantissa normalizer module is a module called mantissa left shifter module which is the opposite of the right shifter module discussed above.

For this module of the floating point subtractor, we implemented a Barrel Shifter type of left shifter unit. A barrel shifter is a combinational circuit that facilitates left shift for this subtractor. Unlike regular shifter a barrel shifter is a sequential circuit.

Mantissa Left Shifter Block Diagram

The block diagram for the mantissa barrel shifter is shown down below in Figure 57. The first level showcases a 4 bit left shift, the second level shows a 2 bit left shift, and the last level shows a 1 bit left shift [28].

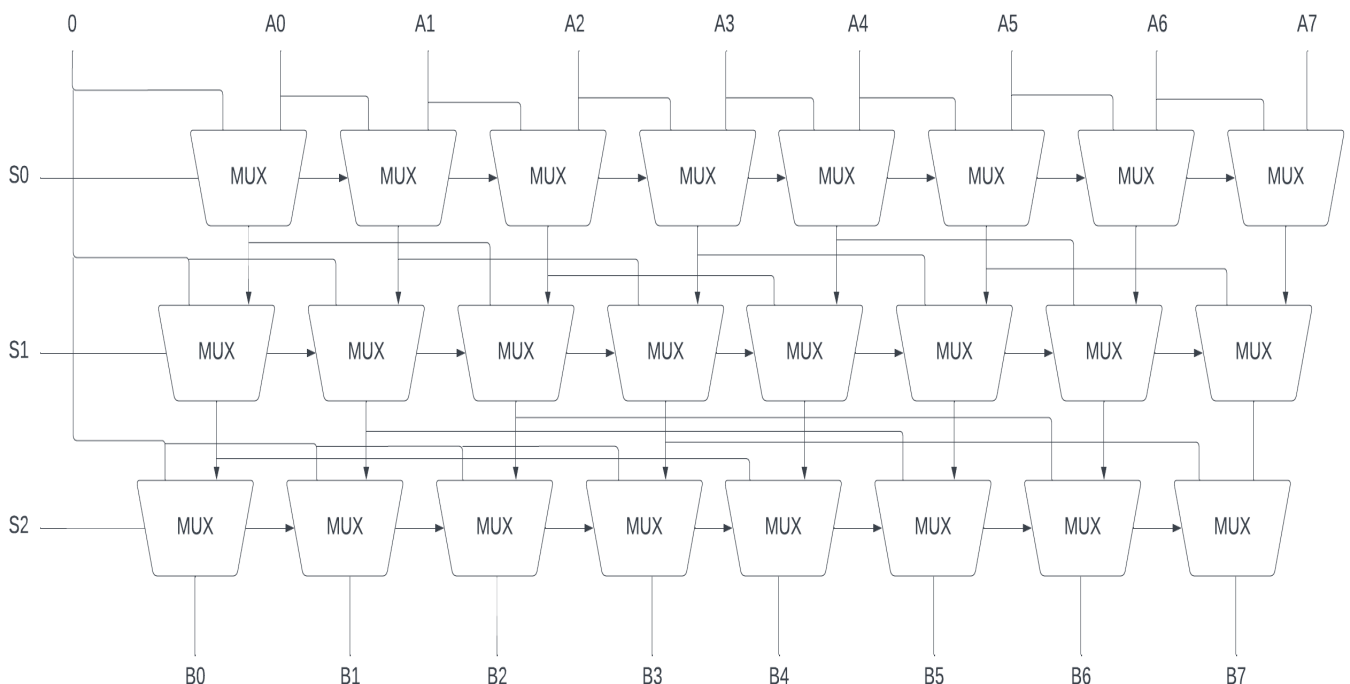


Figure 57 Left Shifter Block Diagram

Mantissa Left Shifter Verilog Code

The following figure, Figure 58, shows the code for a mantissa left shifter unit used to implement the floating point subtractor [29].

The mantissa left shifter unit takes in the smaller mantissa from two inputs, which is being selected by the multiplexer described above. The outputted mantissa from multiplexer then gets inputted into this barrel shifter that uses multiplexers to shift the mantissa by the desired shift value.

This shift value is achieved from the exponent subtractor unit module. The mantissa is shifted the same value as of the difference between the two exponents.

```

1
2  module barrelLeftshifter
3  (
4      input  [7:0] In; //The 8-bit Input line
5      output [7:0] Out; //The 8-bit Output line
6      input  [2:0] shift; //The 3-bit shift magnitude selection Input
7  );
8
9      wire [7:0] Level1, Level2; //Two 8-bit intermediate lines
10
11     mux_2to1 m0 (1'b0, In[0], Level1[0], shift[0]);
12     mux_2to1 m1 (1'b0, In[1], Level1[1], shift[0]);
13     mux_2to1 m2 (In[1], In[2], Level1[2], shift[0]);
14     mux_2to1 m3 (In[2], In[3], Level1[3], shift[0]);
15     mux_2to1 m4 (In[3], In[4], Level1[4], shift[0]);
16     mux_2to1 m5 (In[4], In[5], Level1[5], shift[0]);
17     mux_2to1 m6 (In[5], In[6], Level1[6], shift[0]);
18     mux_2to1 m7 (In[6], In[7], Level1[7], shift[0]);
19
20     mux_2to1 m00 (1'b0, Level1[0], Level2[0], shift[1]);
21     mux_2to1 m11 (1'b0, Level1[1], Level2[1], shift[1]);
22     mux_2to1 m22 (Level1[0], Level1[2], Level2[2], shift[1]);
23     mux_2to1 m33 (Level1[1], Level1[3], Level2[3], shift[1]);
24     mux_2to1 m44 (Level1[2], Level1[4], Level2[4], shift[1]);
25     mux_2to1 m55 (Level1[3], Level1[5], Level2[5], shift[1]);
26     mux_2to1 m66 (Level1[4], Level1[6], Level2[6], shift[1]);
27     mux_2to1 m77 (Level1[5], Level1[7], Level2[7], shift[1]);
28
29     mux_2to1 m000 (1'b0, Level2[0], Out[0], shift[2]);
30     mux_2to1 m111 (1'b0, Level2[1], Out[1], shift[2]);
31     mux_2to1 m222 (1'b0, Level2[2], Out[2], shift[2]);
32     mux_2to1 m333 (1'b0, Level2[3], Out[3], shift[2]);
33     mux_2to1 m444 (Level2[0], Level2[4], Out[4], shift[2]);
34     mux_2to1 m555 (Level2[1], Level2[5], Out[5], shift[2]);
35     mux_2to1 m666 (Level2[2], Level2[6], Out[6], shift[2]);
36     mux_2to1 m777 (Level2[3], Level2[7], Out[7], shift[2]);
37
38  endmodule
39

```

Figure 58 Left Shifter Code

The 24-bit input for this particular module comes from the output of the ripple carry subtractor module that computes the mantissa subtraction. The other 5-bit input for this left shifter mantissa normalizer input comes from the Cout output of the same carry look ahead adder which signifies how much the mantissa must be shifted (0 bits or 1 bit).

The mantissa normalizer shifts the mantissa addition output only when the Cout output of the ripple carry subtractor is high. The high value from the subtractor signifies that the mantissa subtraction has a carry of 1 and that signifies that the mantissa needs to be shifted to be normalized for the final output.

The output of this left shifter is the final mantissa value used to represent the 32 bit binary floating point result.

3.4 Floating Point Subtractor Results

The whole floating point subtractor unit was tested on Quartus' ModelSim simulation software using testbenches and waveforms. The design simulation involved generating setup scripts for the simulator, compiling simulation models, running the simulation, and viewing the results.

3.4.1 Floating Point Subtractor Compilation Report

Flow Summary	
<input type="text" value="Filter"/>	
Flow Status	Successful - Tue Mar 28 04:20:41 2023
Quartus Prime Version	18.1.0 Build 625 09/12/2018 SJ Lite Edition
Revision Name	FPUSubtractor
Top-level Entity Name	FPUSubtractor
Family	MAX 10
Device	10M08DAF484C8G
Timing Models	Final
Total logic elements	331 / 8,064 (4 %)
Total registers	0
Total pins	127 / 250 (51 %)
Total virtual pins	0
Total memory bits	0 / 387,072 (0 %)
Embedded Multiplier 9-bit elements	0 / 48 (0 %)
Total PLLs	0 / 2 (0 %)
UFM blocks	0 / 1 (0 %)
ADC blocks	0 / 1 (0 %)

Figure 59 FPS Compilation Report

3.4.2 Floating Point Subtractor Testbench

A testbench is used to generate the stimulus and applies it to the implemented floating point subtractor and compare the results against our calculations based on the IEEE 754 floating point calculator online [30]. The design was synthesized using precision synthesis tools targeting the DE-1 SoC Max 10 FPGA machine family.

```

1  |`timescale 1ns / 1ps
2  |module FPUSubtractor_tb;
3  |
4  |    // Inputs
5  |    reg [31:0] A;
6  |    reg [31:0] B;
7  |
8  |    // Outputs
9  |    wire [31:0] out;
10 |
11 |    wire Exception;
12 |    wire Overflow;
13 |    wire Underflow;
14 |
15 |    wire SnanA, QnanA, InfA, ZeroA, SubNA, NormA;
16 |    wire SnanB, QnanB, InfB, ZeroB, SubNB, NormB;
17 |
18 |    // Instantiate the Unit Under Test (UUT)
19 |    FPUSubtractor fpuSubrTB
20 |    (
21 |        .A(A),
22 |        .B(B),
23 |        .out(out),
24 |        .Exception(Exception),
25 |        .Overflow(Overflow),
26 |        .Underflow(Underflow),
27 |        .SnanA(SnanA),
28 |        .QnanA(QnanA),
29 |        .InfA(InfA),
30 |        .ZeroA(ZeroA),
31 |        .SubNA(SubNA),
32 |        .NormA(NormA),
33 |        .SnanB(SnanB),
34 |        .QnanB(QnanB),
35 |        .InfB(InfB),
36 |        .ZeroB(ZeroB),
37 |        .SubNB(SubNB),
38 |        .NormB(NormB)
39 |    );
40 |

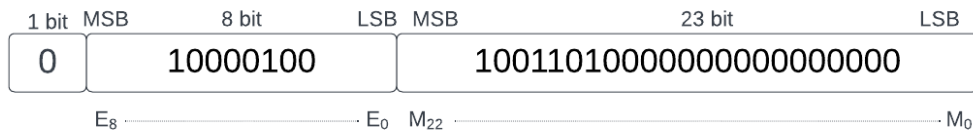
```

Figure 60 FPS Testbench

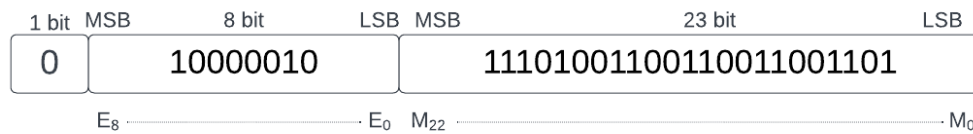
3.4.3 Floating Point Subtractor Simulation Results

Case A:

A:



B:



R:



Simulation Results:

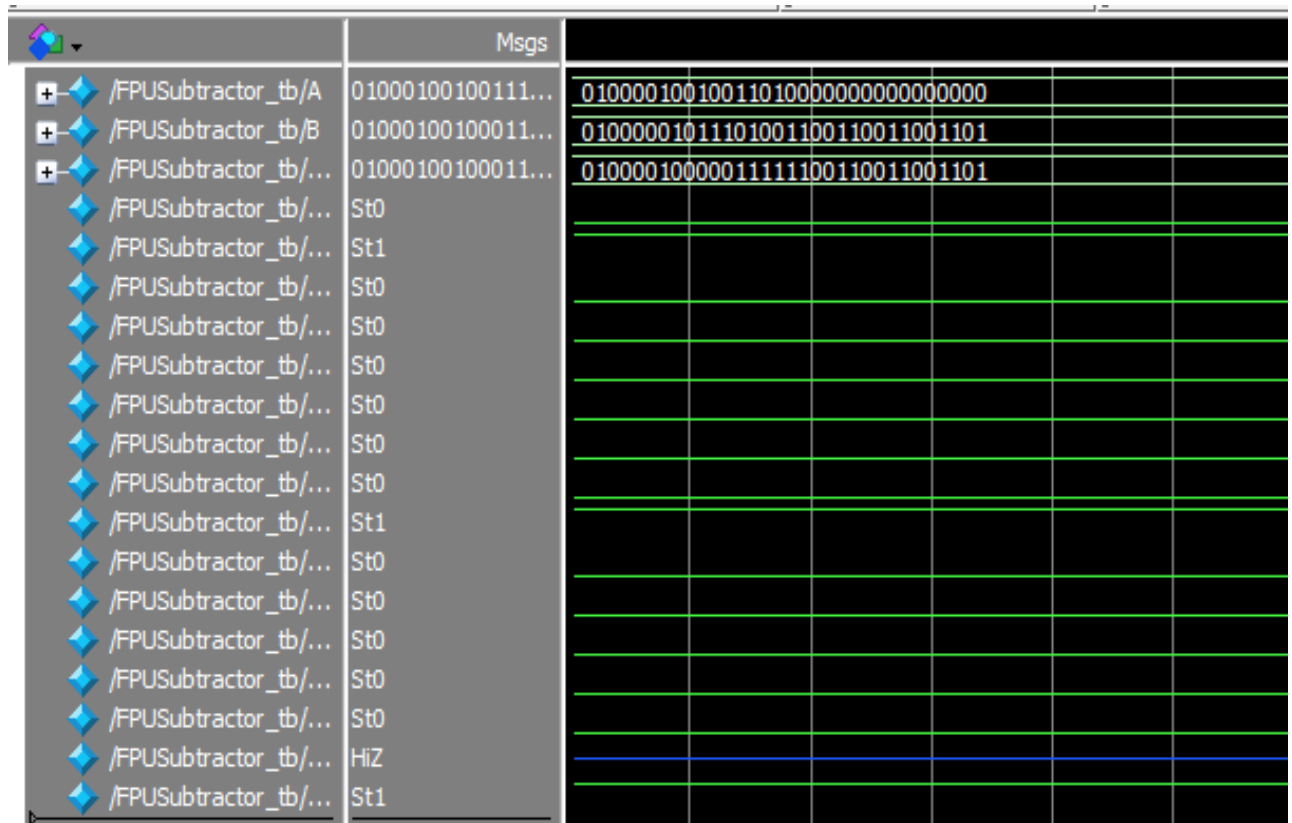
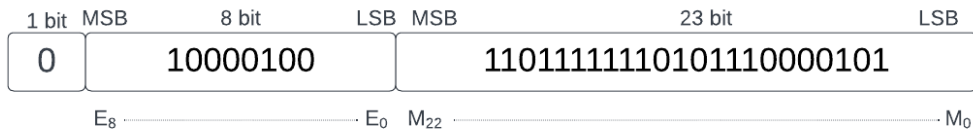


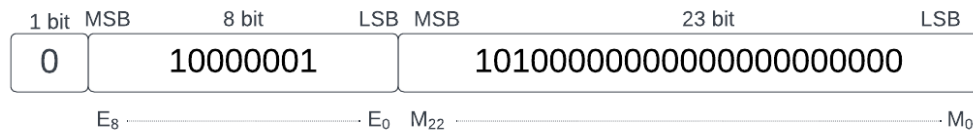
Figure 61 FPS Case A Result

Case B:

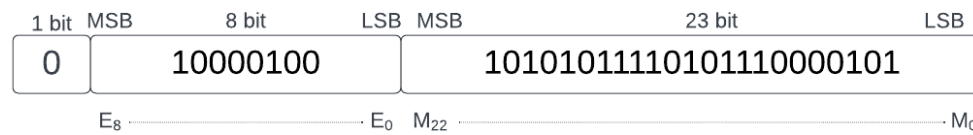
A:



B:



R:



Simulation Result:

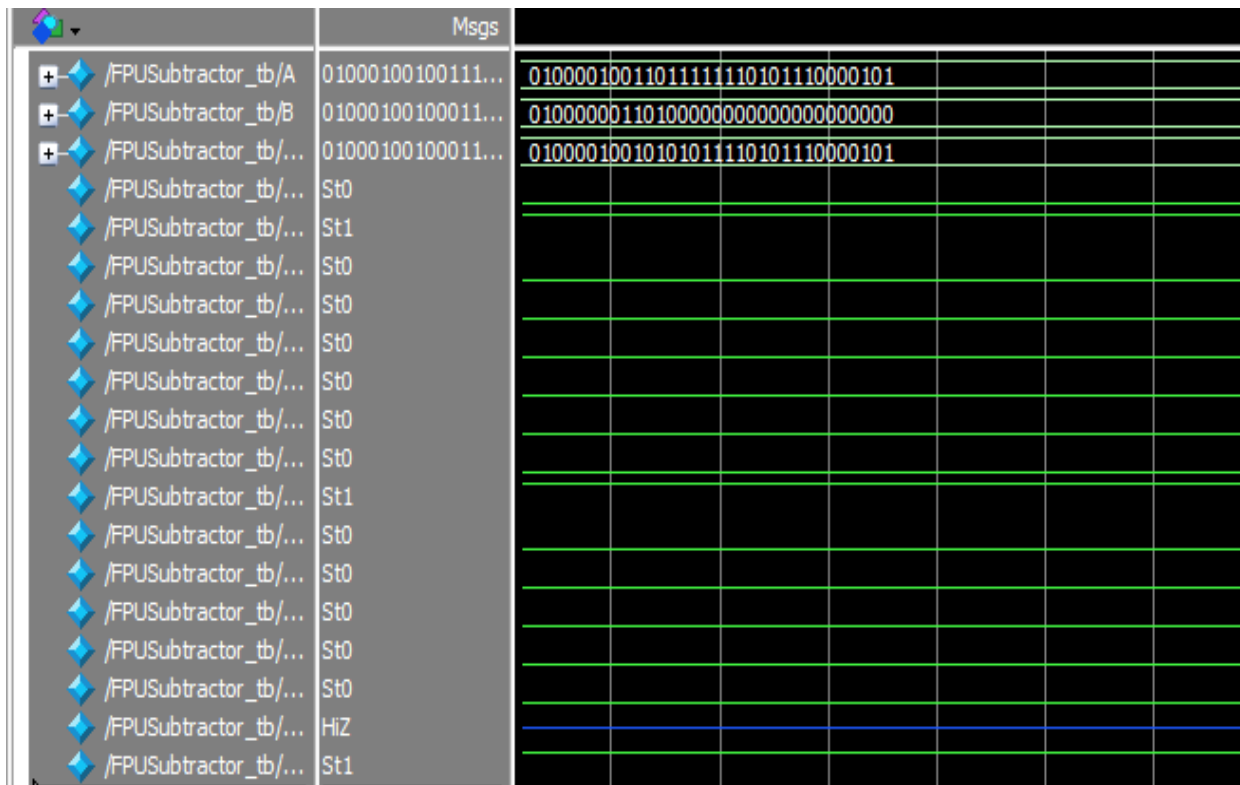
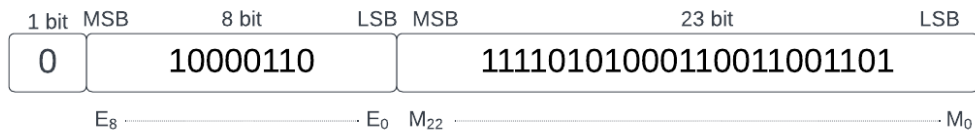


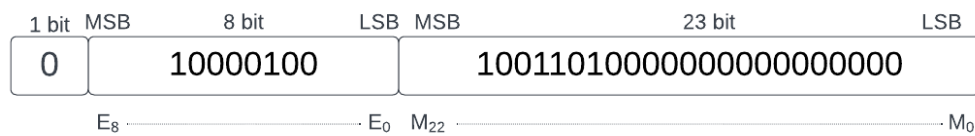
Figure 62 FPS Case B Result

Case C:

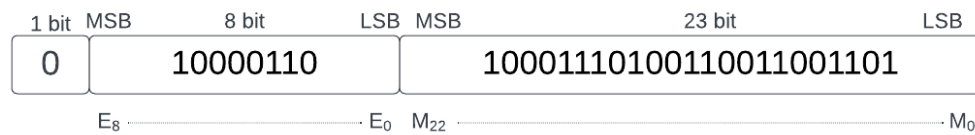
A:



B:



R:



Simulation Result:

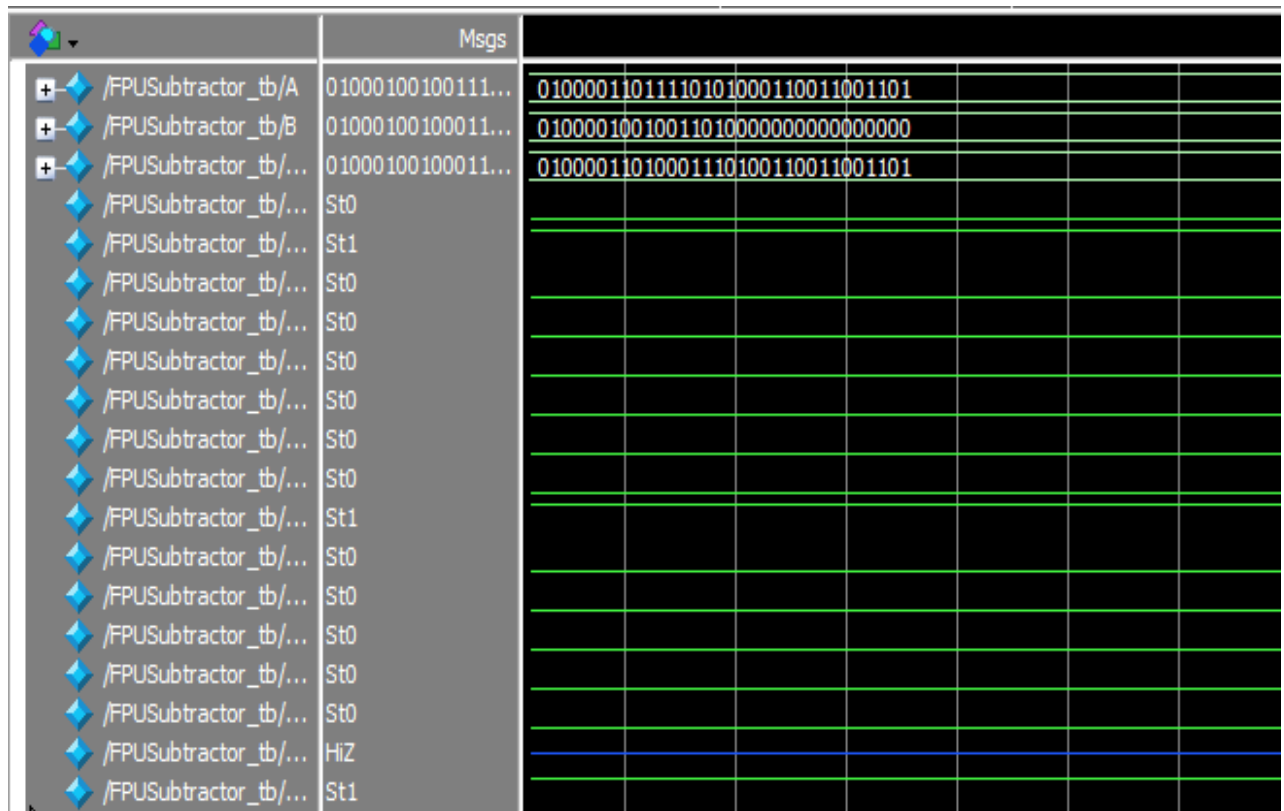


Figure 63 Case C Result

Case D:

A:



B:



R:



Simulation Result:

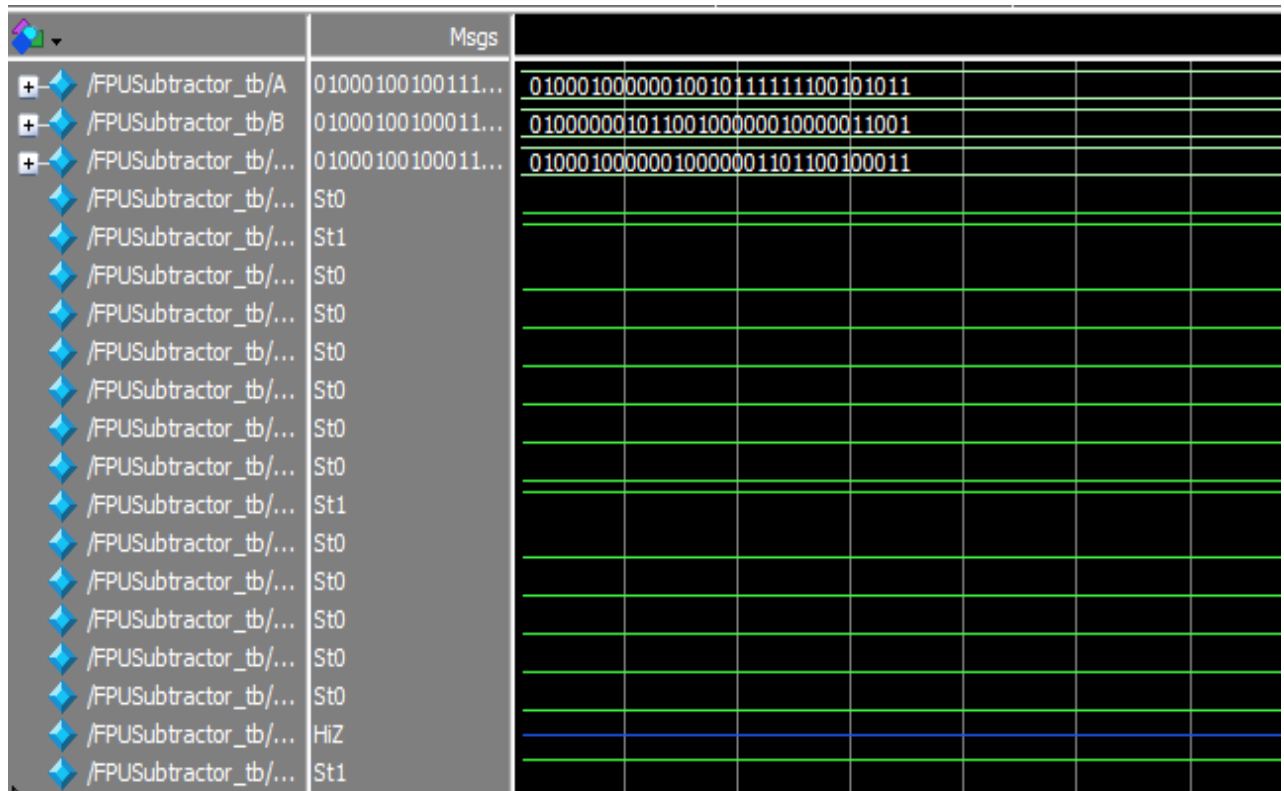


Figure 64 Case D Result

3.5 Conclusion

This section of the thesis presented an implementation of a floating point subtractor that supports the IEEE 754-2008 binary interchange format. The subtractor implements this algorithm using a ripple carry subtractor for faster computation and used various different modules to compute the final output.

Chapter 4: 32-bits Floating Point Multiplier

In this chapter, we describe an efficient implementation of an IEEE 754 single precision floating point multiplier targeted for DE-1 Cyclone V FPGA. Verilog is used to implement a technology-independent pipelined design. The multiplier implementation handles the overflow and underflow cases. Rounding is implemented to give more precision when using the Wallace Tree Multiplier for faster calculations. The Floating-Point Multiplier was verified by testbench simulations on ModelSim.

In this chapter we will dive deeper into the floating-point multiplier algorithm, architecture, code design, RTL diagram, and simulation results. Floating-point multiplication is much less complicated than addition and subtraction as the following discussion showcases:

We will talk about the procedure in multiplication operations and a first look at the code design in a block diagram way followed by deeper understanding of code development.

Floating point multiplication is done by extracting signs, adding exponents, multiplying mantissa values, and shifting the mantissa for normalization [31].

There are five basic phases of designing a Floating-Point Multiplier:

- 1) Check for Zeroes.
- 2) Add exponents.
- 3) Subtract Bias.
- 4) Multiply the Significands.
- 5) Normalize the Significand.
- 6) Normalize the Exponent if needed.

4.1 Floating Point Multiplication Algorithm

As described in the above topics, floating point number is in the format of:

$$Z = (-1^S) * 2^{(E - \text{Bias})} * (1.M)$$

To multiply two floating point numbers A & B the different steps to follow are [32]:

- 1) Extracting signs, exponents and mantissas of both A and B numbers.
- 2) Calculating the output sign.
- 3) Treating the special cases.
- 4) Finding out the data types of numbers given
- 5) Adding the two exponents.
- 6) Subtracting the bias from exponent addition.
- 7) Multiplying the mantissa values
- 8) Normalizing mantissa by bit shifting.
- 9) Normalizing exponent if necessary.
- 10) Detecting exception, overflow, and underflow.

4.1.1 Floating-Point Multiplication Example

A = 125.125 (base 10)

B = 12.0625 (base 10)

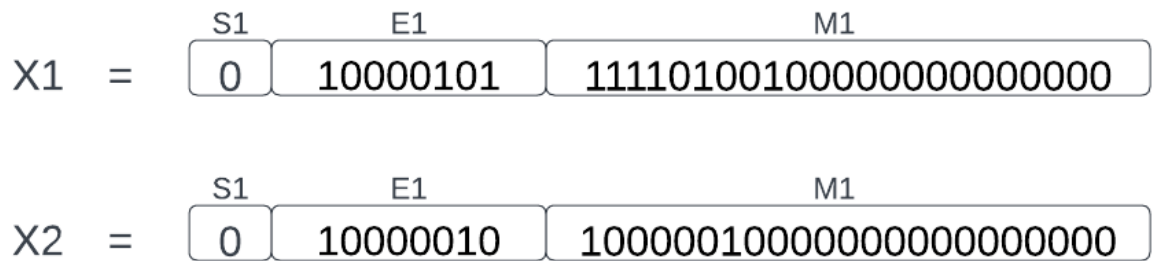


Figure 65 Binary Presentation Mul Example

- 1) S1 = 0, E1 = 10000101, M1 = 111101001000000000000000
- S2 = 0, E2 = 10000010, M2 = 10000010000000000000000000

2) Sign bit calculation

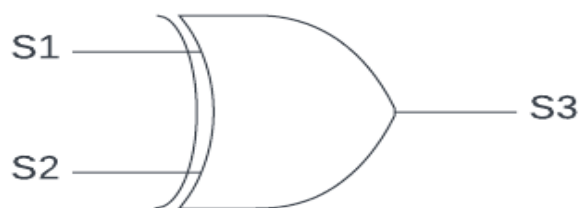


Figure 66 XOR Sign Subtraction

S3 = 0

3) Exponent Addition

$$\begin{array}{r}
 10000101 \\
 + 10000010 \\
 \hline
 100000111
 \end{array}$$

Figure 67 Mul Exponent Addition

Unbiased Exponent = 100000111

4) Subtract Bias

$$\begin{array}{r}
 100000111 \\
 - 01111111 \\
 \hline
 010001000
 \end{array}$$

Figure 68 Mul Bias Subtraction

Biased Exponent = 010001000

5) Multiply the Mantissa

$$\begin{array}{r} 1.111101001000000000000000 \\ \times 1.100000100000000000000000 \\ \hline 10.111100101010100100000000000000000000000000000000000000 \end{array}$$

Figure 69 Mantissa Multiplication

1.M3 = 10.11110010101010010000000000000000000000000000000000000

6) Left Shift the Mantissa for normalization

Left Shifted Mantissa = 1.11110010101010010000000000000000000000000000000000000

7) Increment the exponent

$$\begin{array}{r} 010001000 \\ + 000000001 \\ \hline 010001000 \end{array}$$

Figure 70 Incrementing Exponent

E = 10001000

8) Result

X3	=	<table style="display: inline-table; border-collapse: collapse;"> <tr> <td style="padding: 2px 5px;">S3</td> <td style="padding: 2px 5px;">E3</td> <td style="padding: 2px 5px;">M3</td> </tr> <tr> <td style="border: 1px solid black; padding: 2px 5px;">0</td> <td style="border: 1px solid black; padding: 2px 5px;">10001000</td> <td style="border: 1px solid black; padding: 2px 5px;">111100101010100100000000</td> </tr> </table>	S3	E3	M3	0	10001000	111100101010100100000000
S3	E3	M3						
0	10001000	111100101010100100000000						

Figure 71 FPM Example Result

4.2 Floating Point Multiplier Flowchart

The below, Figure 72, showcases a typical flowchart that is used to design a floating point multiplier. The figure shows a step by step narrative and displays the high level functions that is required to compute floating point multiplication. The flowchart shows block level diagram and each block or element is implemented in hardware and is described in detail in the following topics of the thesis [33].

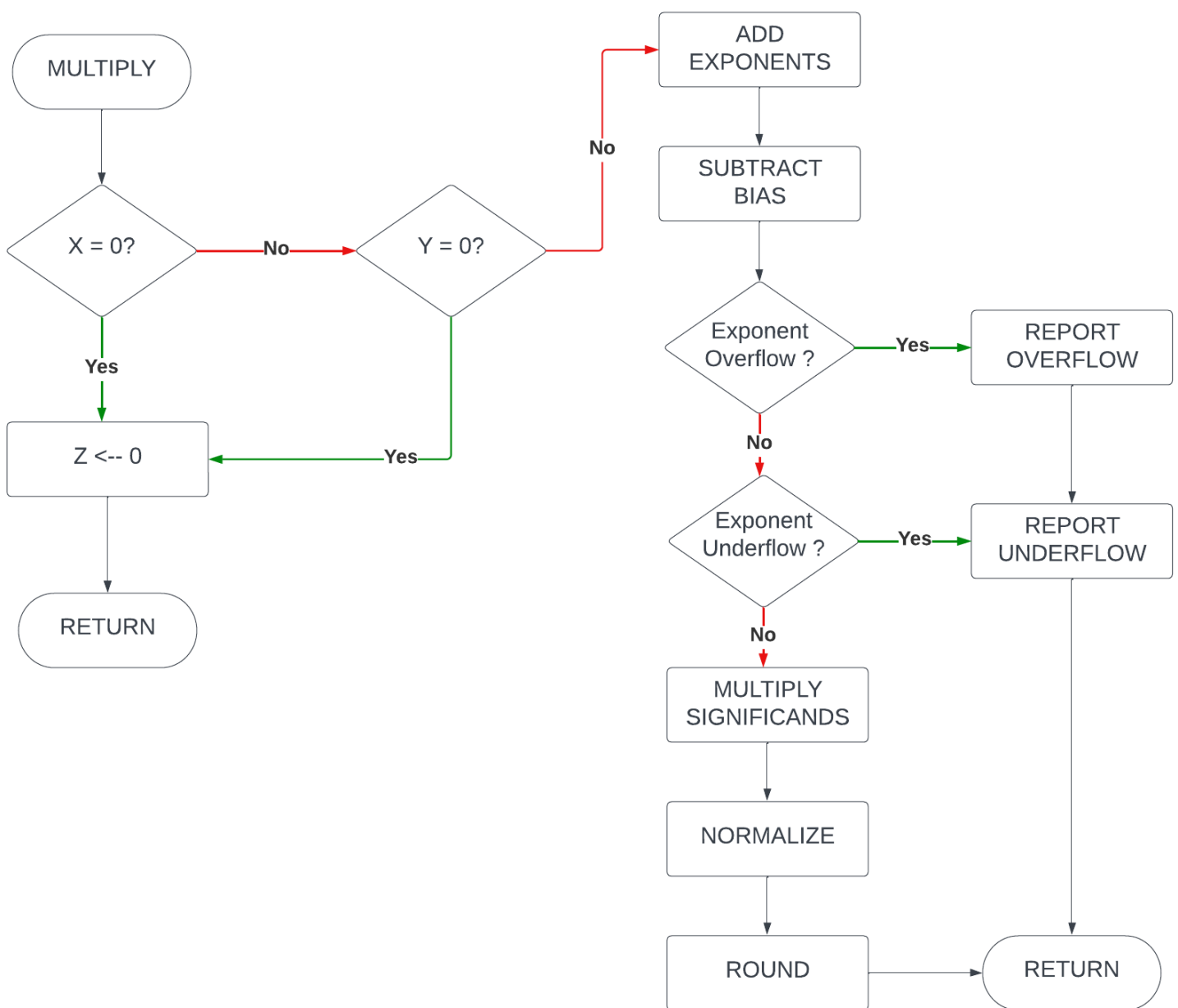


Figure 72 Floating Point Multiplier

4.3 Floating Point Multiplier Hardware

In this section of the thesis we will start explaining and diving deeper into the hardware implementation of the floating point multiplication. This section will start by elaborating the flowchart further with help of showcasing the hardware architecture used to design the module followed by detailed description of each module used in the architecture.

After understanding the theory of hardware implementation and the architecture of floating point multiplication the thesis will show the code development that achieved out final objective of building this floating point unit.

4.3.1 Floating Point Multiplier Hardware Architecture

The below figure, Figure 73, showcases the hardware architecture that was designed and coded to implement synthesizable 32-bit floating multiplier adder using Verilog

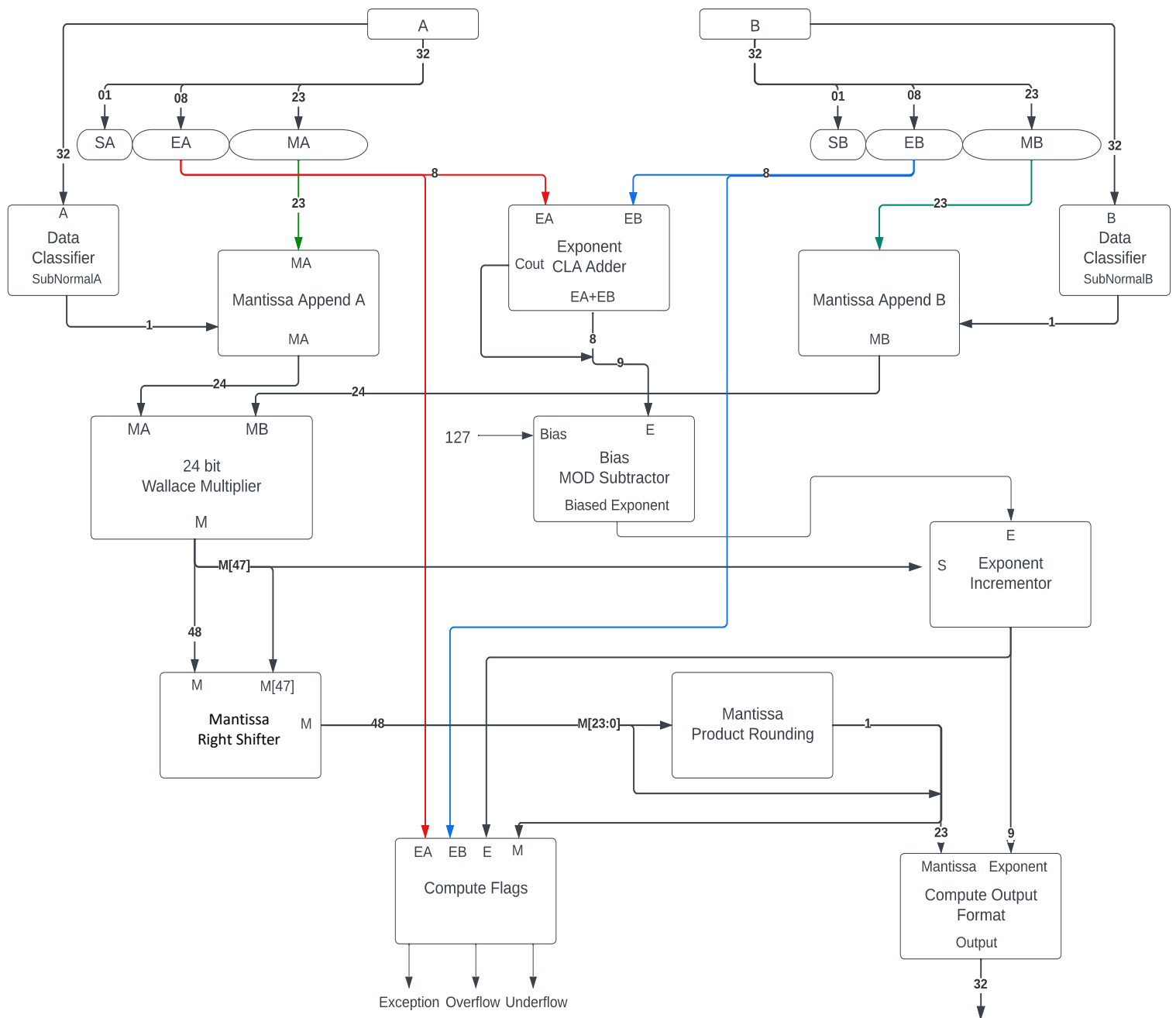


Figure 73 Floating Point Multiplication Architecture

This floating point architecture uses a total of ten modules that serve various unique purposes in making the design work. The modules are:

- Exponent CLA Adder
- Mantissa Append Module
- 24-bit Wallace Multiplier
- Exponent Incrementor
- Compute Flags
- Data Classifier
- Modular Subtractor
- Mantissa Right Shifter
- Mantissa Product Rounding
- Compute Output

4.3.2 Floating Point Multiplier Hardware Implementation

In this section, we will discuss the hardware implementation designed for the floating point multiplier and explain each module and each algorithm step in detail.

4.3.2.1 Sign Bit Calculation

Multiplying two positive numbers will result in a positive number. Multiplying two negative numbers will result in a negative number. Multiplying one positive number and one negative number will result in a negative number. The sign bit calculation for this floating point multiplication unit is done using an XOR gate. The table below shows sign operations for various cases:

A's Sign	Symbol	B's Sign	Operation
+	x	+	+
+	x	-	-
-	x	+	-
-	x	-	-

Table 13 Sign Operations Mul

4.3.2.2 Data Classification Module

A 32-bit binary floating point number can be encoded to form a total of six different cases based on the value of each data bit. The six different data types and the criteria that must be met for their encoding are [34]:

1) Signalling NaN (sNaN)



Figure 74 sNaN Format

If all eight exponent bits are 1.

And MSB of mantissa is 0.

And at least one bit from the rest of mantissa is 1.

2) Quiet NaN (qNaN)

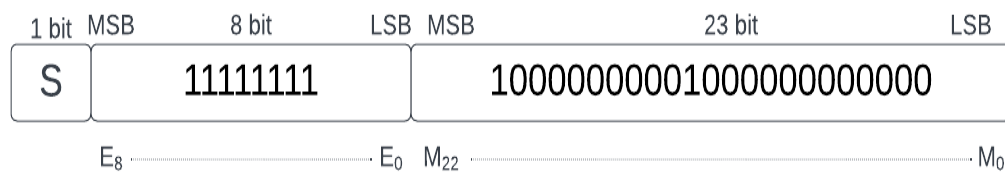


Figure 75 qNaN Format

If all eight exponent bits are 1.

And MSB of mantissa is 1.

And rest of mantissa be zero or non-zero.

3) Negative Infinity ($-\infty$)

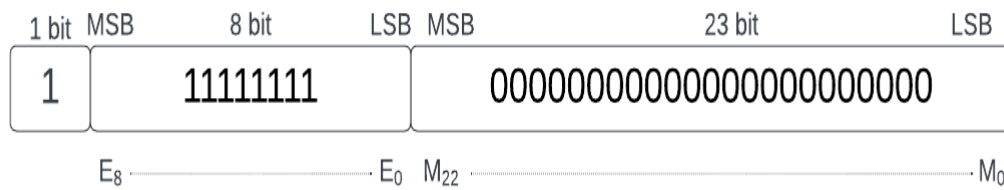


Figure 76 Plus Infinity Format

If sign bit of the number is 1.

And all eight exponent bits are 1.

And MSB of mantissa is 0.

And rest of mantissa bits are also zero.

4) Positive Infinity ($+\infty$)

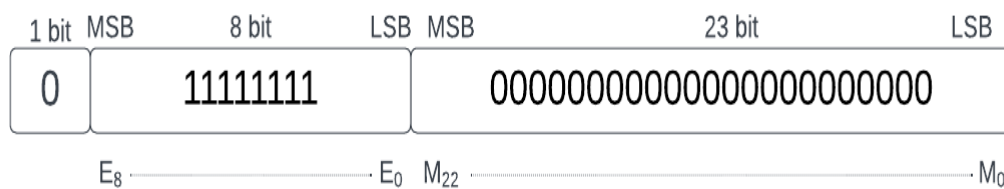


Figure 77 Negative Infinity Format

If sign bit of the number is 0.

And all eight exponent bits are 1.

And MSB of mantissa is 0.

And rest of mantissa bits are also zero.

5) Positive Zero (+ Z)

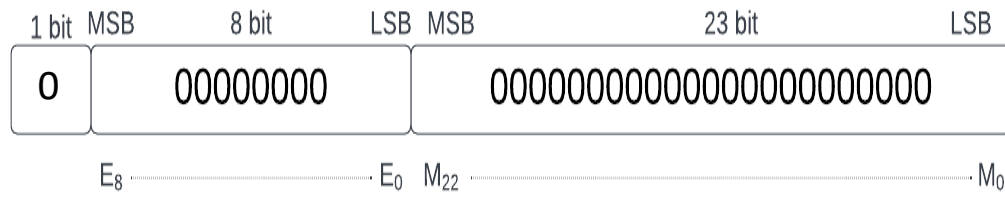


Figure 78 Positive Zero Format

If sign bit of the number is 0.

If all eight exponent bits are 0.

And MSB of mantissa is 0.

And rest of mantissa bits are also 0.

6) Negative Zero (- Z)



Figure 79 Negative Zero Format

If sign bit of the number is 1.

If all eight exponent bits are 0.

And MSB of mantissa is 0.

And rest of mantissa bits are also 0.

7) Subnormal



Figure 80 Subnormal Format

If all eight exponent bits are 0.

At least one bit from the rest of mantissa is 1.

8) Normal



Figure 81 Normal Format

If at least one exponent bit is 1.

At least one bit from the rest of mantissa is 1.

Data Classification Module Verilog Code

The following figure, Figure 82, shows the Verilog code used for designing the data classification model with all the criteria described above:

```

1  module sp_class
2  (
3      input[31:0] A,
4      output snan, qnan, infinity, zero, subnormal, normal
5  );
6
7      wire expHigh, expLow, sigHigh, sigLow;
8
9      //AND reduction operator to check if all exponent bits are 1
10     assign expHigh    = &A[30:23];
11     //NOR reduction operator to check if all exponent bits are 0
12     assign expLow     = ~|A[30:23];
13
14     //AND reduction operator to check if all significand bits are 1
15     assign sigHigh    = &A[22:0];
16     //AND reduction operator to check if all significand bits are 0
17     assign sigLow     = ~|A[22:0];
18
19     assign snan        = expHigh & ~A[22] & ~sigLow;
20     assign qnan        = expHigh & A[22];
21
22     assign infinity    = expHigh & sigLow;
23
24     assign zero        = expLow & sigLow;
25
26     assign subnormal   = expLow & ~sigLow;
27     assign normal      = ~expHigh & ~expLow;
28
29     endmodule
30

```

Figure 82 Data Classification Verilog Code

The Verilog code utilizes AND & NOR reduction operators to check all the bits of exponents and significands for 1s and 0s respectively. The reduced variable is then used to check different cases in accordance with all the criteria defined by IEEE 754 standards.

This data classification module is used twice for both A and B inputs for the multiplication operation. Input A and Input B are both 32-bit inputs to the two instantiations of this module. The output of this module are the various data types, the Zero data type output is later used in another module to compute the hidden mantissa bit.

Data Classification Module RTL Diagram

The following figure, Figure 83, shows the RTL diagram that was outputted when the Verilog code shown above was compiled. The RTL diagram's first level shows the AND & NOR reduction operator at work, followed by which there are AND gates at level two that finally computes the various data types as high or low.

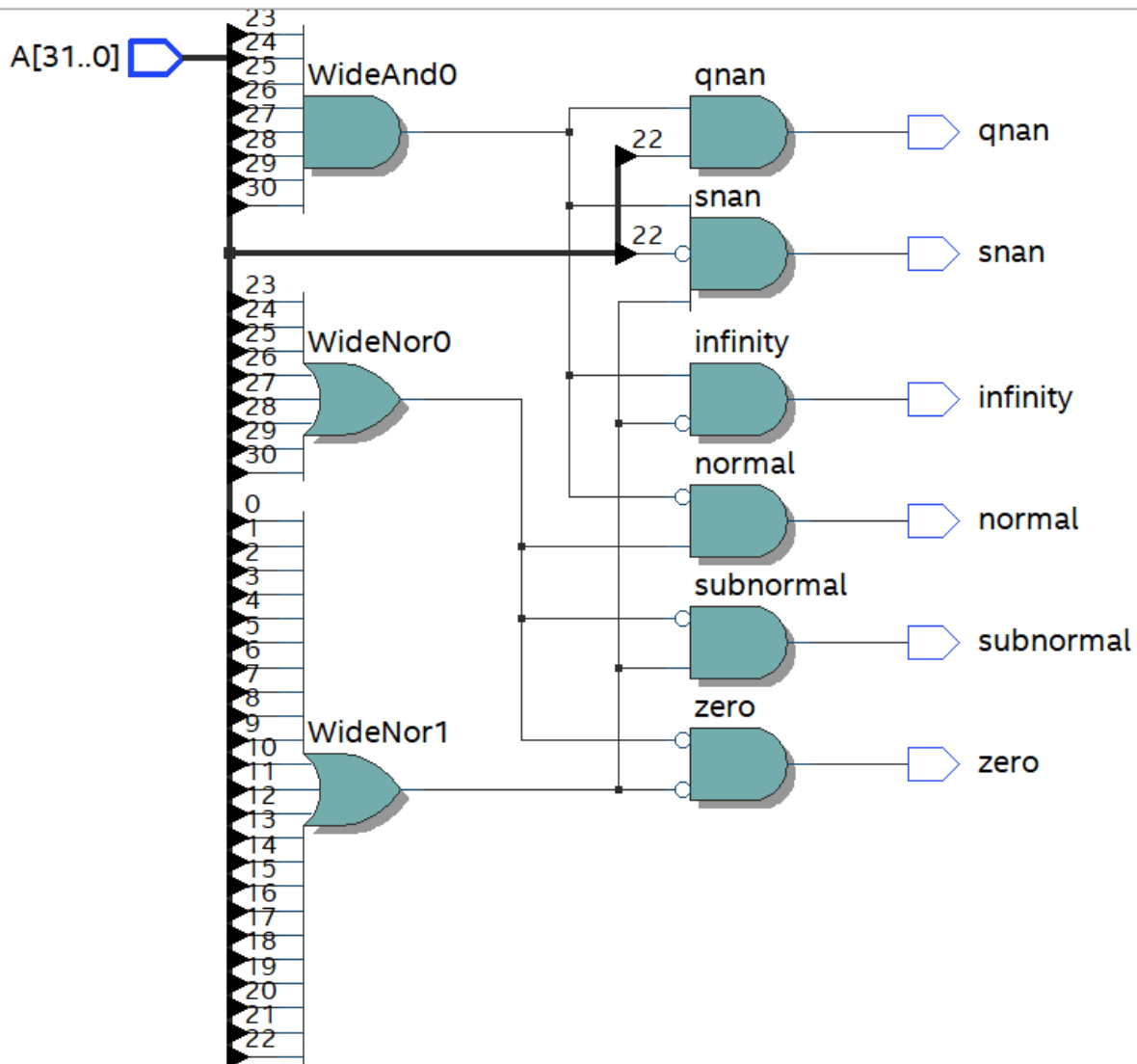


Figure 83 Data Classification RTL Diagram

This RTL diagram is generated twice in the entire floating point multiplication unit as it is used to classify both A and B inputs into different data types described above.

4.3.2.3 Exponent Carry Lookahead Adder

This exponent carry lookahead adder module is the first arithmetic operation module that constitutes the floating point multiplication algorithm described in section 4.1 above in this chapter.

This module's basic task is to add the exponent of input A and exponent of input B. The module used to carry out this addition operation is the Carry Lookahead Adder that is described in great detail in section 2.3.2.6.

The only difference between the carry lookahead adder implemented in section 2.3.2.6 and this section is that the previous adder worked with 24-bits inputs, whereas this adder operates on 8-bits inputs. The carry look ahead adder described in section 2.3.2.6 was a parametrized implementation, hence there was only one modification necessary to change the input parameter from 24-bits to 8-bits.

```

70  CLAParameter #(N(8)) ExponentAdder
71  □ (
72  .A(EA),
73  .B(EB),
74  .OpCode(1'b0),
75  .Cout(coutExpOp),
76  .R(CLAExpOutTemp)
77  );

```

Figure 84 Exponent Adder Instantiation

Figure 84 shows the instantiation of the exponent carry look ahead adder modified for 8-bits operation by changing the value that is assigned to variable N. The output of this exponent adder feeds into the next module that is the modular bias subtractor.

Please refer to section 2.3.2.6 for details about Carry Look Ahead Adder including Verilog code, RTL Diagram, and Block Diagram.

4.3.2.4 Modular Bias Subtractor

This modular bias subtractor module is the next arithmetic operation module that constitutes the floating point multiplication algorithm described in section 4.1 above in this chapter.

This module's primary task is to subtract the fixed bias value of 127_{10} from the result of the exponent carry look ahead adder module. When our design added the two exponent values with each other, the bias of those two exponents got doubled. This module subtracts the extra bias value and normalizes the exponent back to its correct magnitude.

The modular bias subtractor makes use of the same module that was used in section 2.3.2.2 of chapter two to carry out exponent subtraction. The instantiation for this module is shown in Figure 85 below:

```

80
81 //Subtractig bias from EA+EB
82 //subtraction is done to find the absolute value
83 ModeSubtractor #(.w(9)) EAEBSub
84 (
85     .A(CLAExpOut),
86     .B(9'b00111111),
87     .OpCode(1'b1),
88     .R(unbiasedExponent),
89     .Cout(coutSub)
90 );
91

```

Figure 85 Bias Subtraction Instantiation

As shown in the instantiation above in Figure 85, the inputs to the modular subtractor is the output of the exponent addition, along with 00111111_2 which is 127 in decimal. The modular subtractor is also a parameterized module which has been modified to operate on 9-bits for this operation. The output of this module is absolute value and it will be fed to the exponent incrementor module discussed in coming sections.

Please refer to section 2.3.2.2 for details about the modular subtractor and all its constituting elements including Verilog code, RTL Diagram, and Block Diagram.

4.3.2.5 Mantissa Append Module

This module acts as a preparation step before we get to the most crucial step of the floating point multiplication algorithm which is mantissa multiplication.

This module's primary task is to compute the hidden/implied bit of the mantissa that exists at the most significant bit spot but hidden for representation purposes. The hidden bit of a mantissa depends on the data type of each input. If the data type of the input, as computed by the data classification module, is of type Subnormal, then it is computed that the most significant bit of the 24-bit mantissa is a 0. If the data type of an input is computed as non-Subnormal then it is decided that the most significant bit of the mantissa is a 1.

Mantissa Append Module Block Diagram

The following figure, Figure 86, shows the block diagram for Mantissa Append module discussed above. The mantissa append module is essentially a two-to-one multiplexer with two inputs and one output.

The module has another input labelled as S which selects which input gets linked through and outputted out of the module and in turn gets fed into the Wallace multiplier.

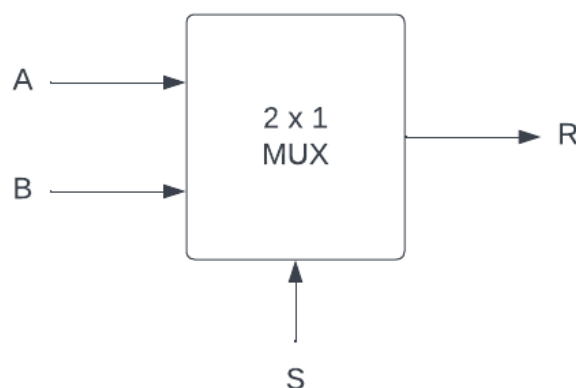


Figure 86 Mantissa Append Block Diagram

Mantissa Append Module Verilog Code

The following figure, Figure 87, shows the code for the mantissa append module.

```
1  module appendMantissa
2  (
3      input [22:0] A,
4      input S,
5      output [23:0] R
6  );
7
8      assign R =(S==1'b1)? {1'b0,A} : {1'b1,A};
9
10 endmodule
11
12
```

Figure 87 Mantissa Append Verilog Code

This module takes in two 23-bits inputs which come directly from the mantissa of input A. The input S comes from the Subnormal output of the data classification module. As shown in the code, if the select is high (i.e. input is of type subnormal) then the output of the module is the input of the module appended with value of 0. If the select is low the output of the module is the input of the module appended with value of 1.

The mantissa append module is instantiated and used twice in this operation. The first use of this module is deciding on the hidden bit of mantissa of input A as it takes in mantissa of input A as its input. The second use of this module is deciding on the hidden bit of mantissa of input B as it takes in mantissa of input B as its input.

The outputs of both the modules feeds into the two different inputs of the Wallace multiplier as the mantissa has now been prepped for multiplication. The output of those modules are 24-bits.

Mantissa Append Module RTL Diagram

The figure below, Figure 88, shows the RTL diagram for the mantissa append module designed to output the mantissa to be feed to the multiplier unit.

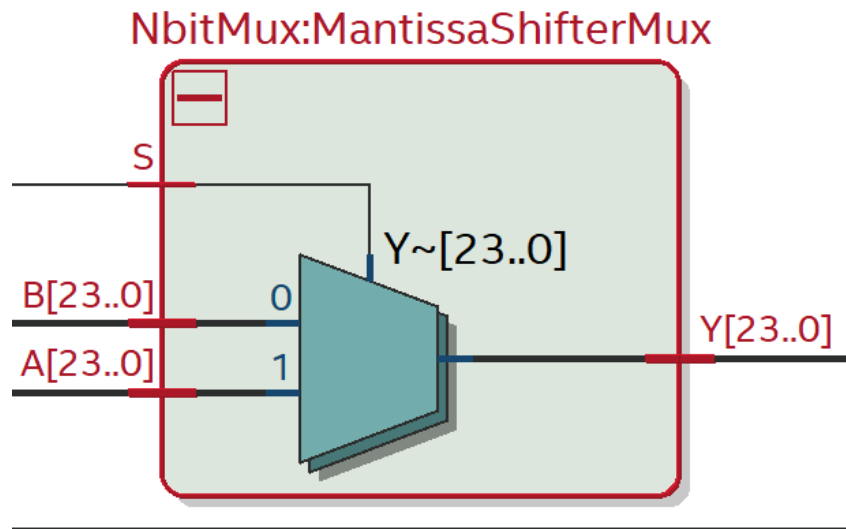


Figure 88 Mantissa MUX RTL

The operation of this module is described in the table below:

Input 1	Input 2	Select	Output
A	B	1	{1'b0,A}
A	B	0	{1'b1,B}

Table 14 Append Mantissa Truth Table

4.3.2.6 Mantissa 32-bit Wallace Multiplier

The 32-bit Wallace tree mantissa multiplier module is the next arithmetic operation module that constitutes the floating point multiplication algorithm described in section 4.1 above in this chapter.

This module takes in two 24-bits input A and B and produce a 48-bits output that is the multiplication result of inputs A and B. The input A to this module comes from the first mantissa append module and the second input, input B, comes from the second mantissa append module as described in the section above.

This modules carries out the mantissa multiplication operation using a Wallace tree multiplier algorithm that facilitates fast calculation and results in an efficient system.

Wallace tree multiplier is a multiplication algorithm that uses a tree structure to add partial products to obtain the product and carry two numbers. Wallace Tree Multiplier is a multiplier that works in parallel by making use of the Wallace tree algorithm. This algorithm allows for a fast and efficient multiplication of two integers.

Wallace tree multiplier is a fast multiplier with medium complexity which can be described as its biggest advantage. Although this multiplier does require a large chip area due to a large amount of logic in terms of AND gates and full adders.

Mantissa 32-bit Wallace Multiplier Computation

In Wallace multiplier, any three wires with the exact same weights and input into a full adder. The result will be an output wire of the same weight and an output wire with a higher weight for each of the three input wires. Furthermore, If there are two wires of the same weight left, input them into a half adder. And finally, If there is just one wire left, connect it to the next layer of full adder or half adder depending on what is available in the next level to the immediate adjacent of the result.

For a 8 by 8 Wallace multiplier the computation steps are provided below [35]:

Step 1: Partial product obtained after multiplication is taken at the first stage. The data is taken with 3 wires and added using adders and the carry of each stage is added with next two data in the same stage.

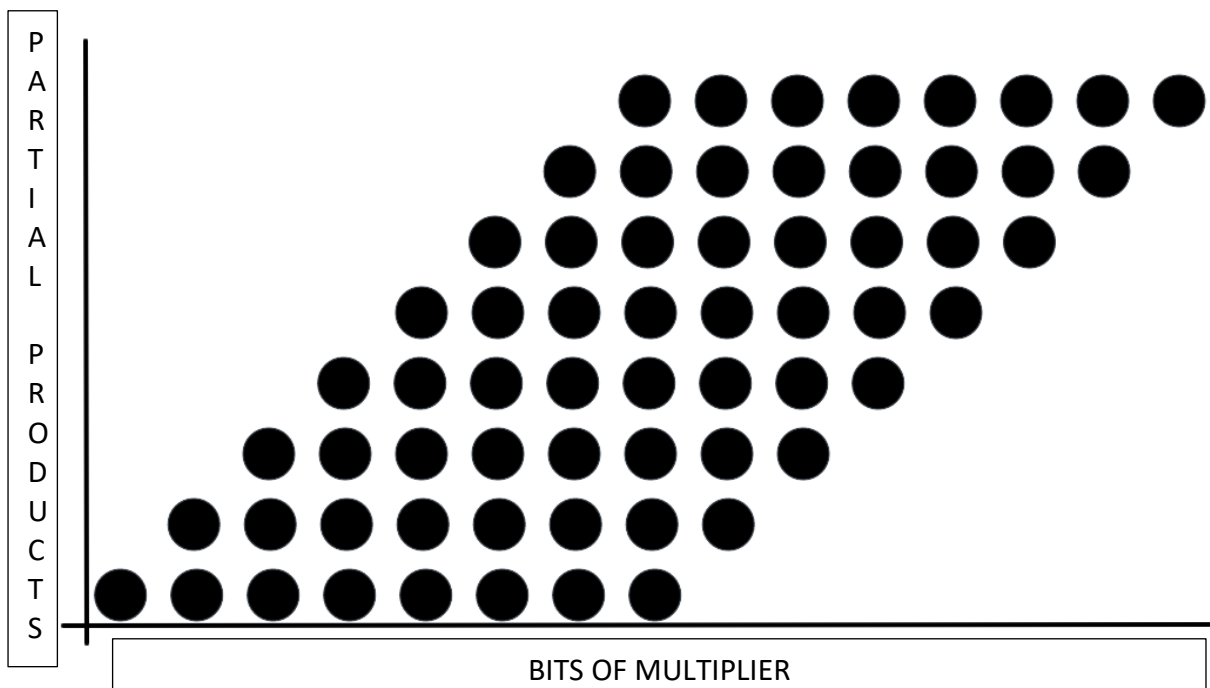


Figure 89 Wallace Multiplication Stages

Step 2: Partial products reduced to two layers of full adders with same procedure.

Stage 0:

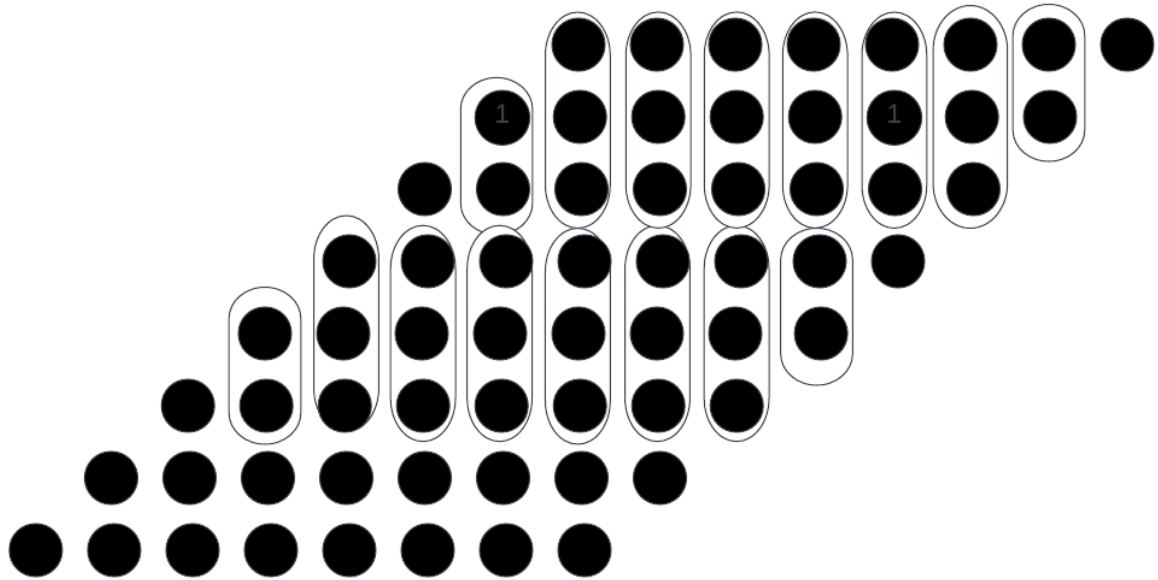


Figure 90 Wallace Multiplication Stage 0

Stage 1:

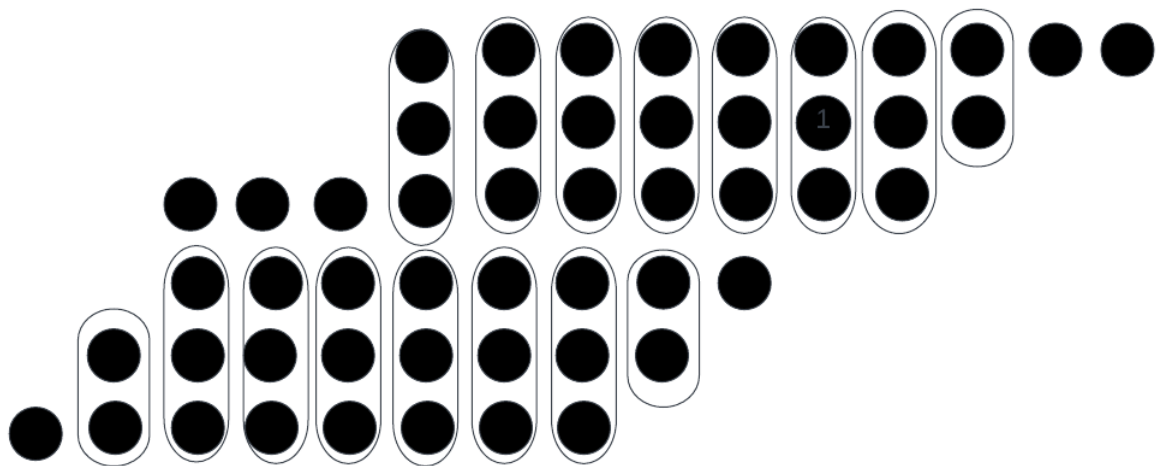


Figure 91 Wallace Multiplication Stage 1

Stage 2:

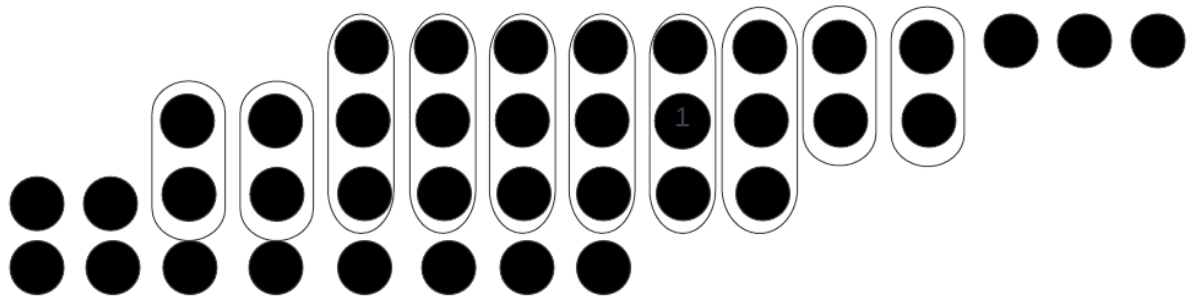


Figure 92 Wallace Multiplication Stage 2

Stage 3:

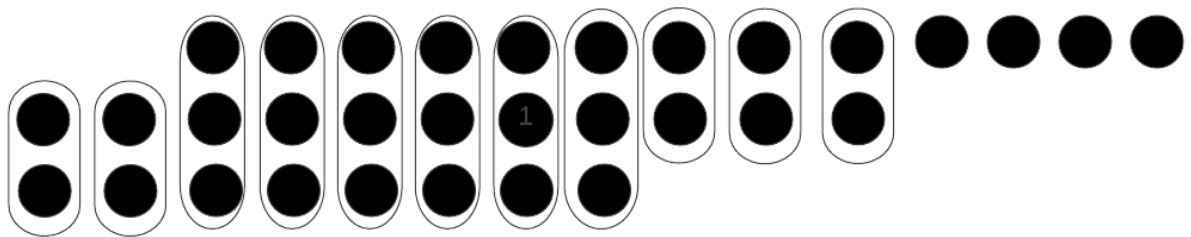


Figure 93 Wallace Multiplication Stage 3

Step 3: Use Ripple carry adder or Carry look ahead adder to compute final addition

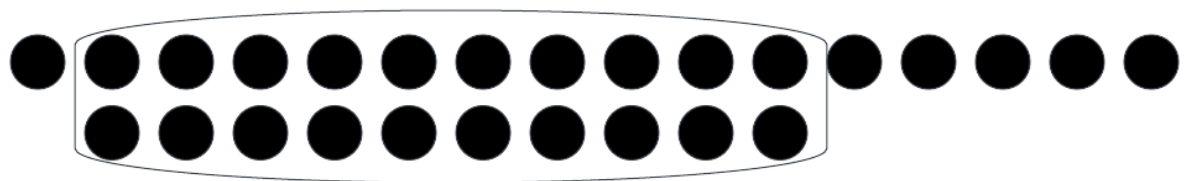


Figure 94 Wallace Multiplication Step 3

Mantissa 32-bit Wallace Multiplier Flow Diagram

The following figure, Figure 95, shows the flow diagram for a Wallace tree multiplier [36].

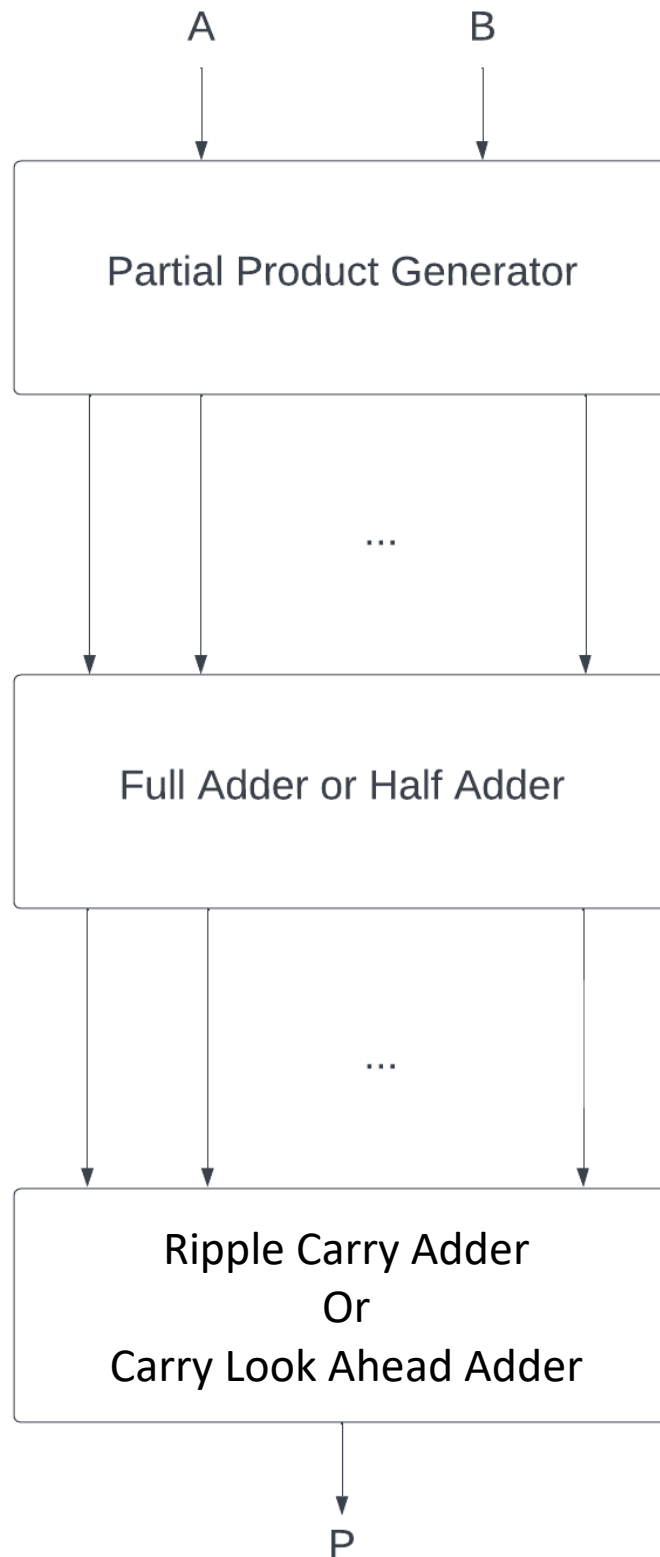


Figure 95 Wallace Multiplier Flow Diagram

Mantissa 32-bit Wallace Multiplier Block Diagram

The following figure, Figure 96, elaborates further on the flow diagram shown in the section just above. There are three primary levels to this Wallace tree multiplier algorithm as apparent from the data flow diagram [37].

- 1) The partial product generator generates partial products using a simple two-input AND gate that is fed to the Wallace tree adder.
- 2) Multiple half and a full adders that does additions in multiple levels and also considers carry generated by a previous level adder.
- 3) The last level of the Wallace tree adder can be implemented ripple carry adder. To improve computation latency, a carry look-ahead adder can also be used.

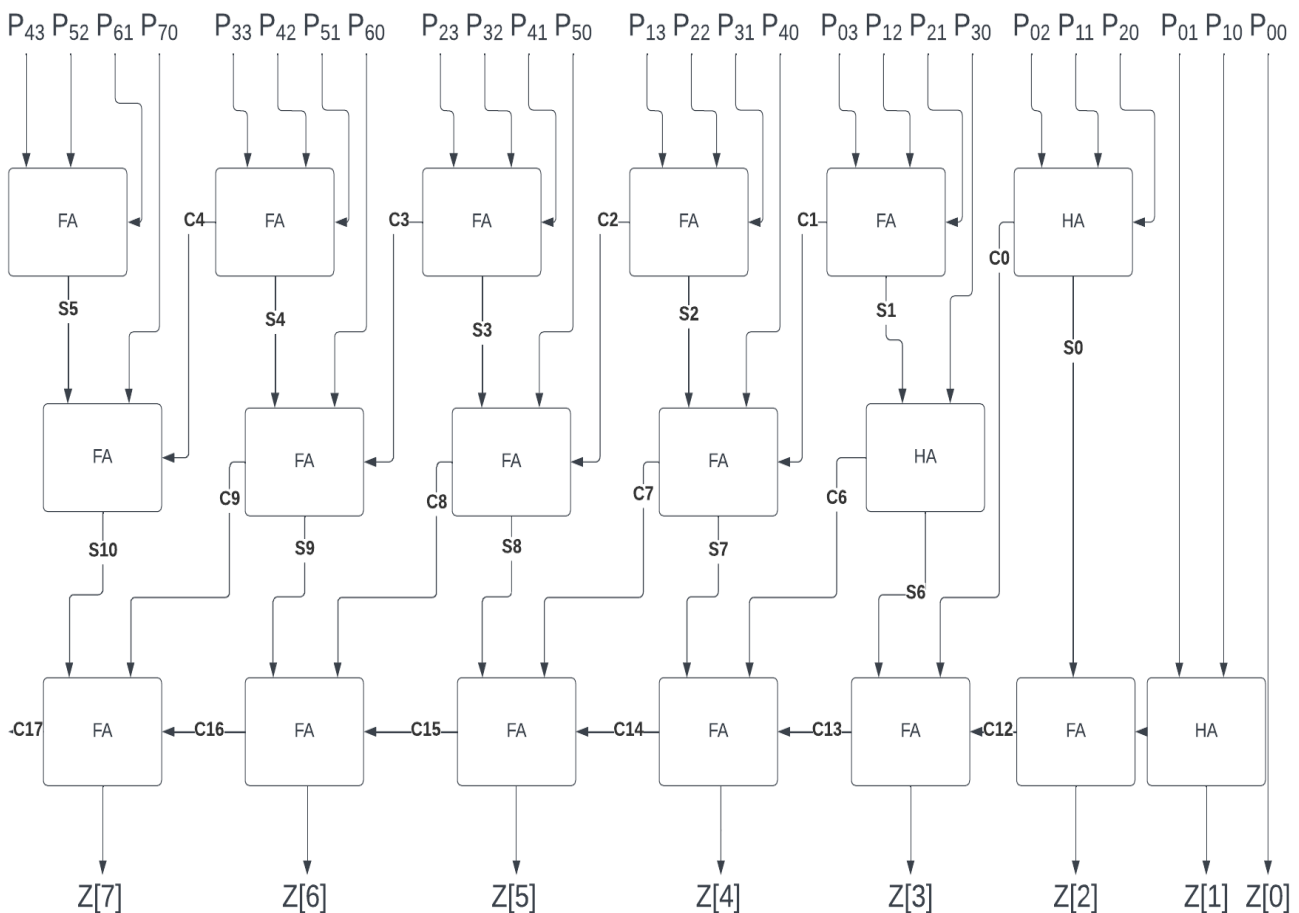


Figure 96 Wallace Tree Mul Block Diagram

Mantissa 32-bit Wallace Multiplier Verilog Code

The following figure, Figure 97, shows a code snippet from the 32-bit Wallace tree multiplier code. The code shows two 32 inputs being taken in the Wallace multiplier and a 64-bit output being outputted.

This design has been achieved by instantiating four 16-bit Wallace tree multiplier which in turn was written using the base 8-bit Wallace multiplier module. The code for all these different modules can be found in the annex of this thesis.

```

1  module wallace32bit (a,b,out,asn);
2
3  input  [31:0] a;
4  input  [31:0] b;
5  output [63:0] asn;
6
7  wire  [31:0] in1,tmp1,tmp2,tmp3,tmp4;
8  wire  [47:0] in2,in3;
9  wire  [63:0] in4;
10
11 wallace16bit w12(a[15:0],b[15:0],tmp1);
12 wallace16bit w22(a[15:0],b[31:16],tmp2); //m1 nh
13 wallace16bit w33(a[31:16],b[15:0],tmp3); //mh n1
14 wallace16bit w44(a[31:16],b[31:16],tmp4);
15
16 assign in1 = tmp1;
17 assign in2 = tmp2<<16;
18 assign in3 = tmp3<<16;
19 assign in4 = tmp4<<32;
20
21 assign asn= in1+in2+in3+in4;
22
23 endmodule
24
25

```

Figure 97 Wallace Multiplier Code

For our purposes, the floating point multiplier unit passed two 24 bit inputs to this multiplier while appending the rest of the bits with 0s to satisfy the 32-bit data requirement. Similarly, the 64 bit output was truncated to be 48 bits to get the required length of data that is essential to the computation of this multiplication module.

Mantissa 32-bit Wallace Multiplier RTL Diagram

The following figure, Figure 98, shows the RLT Diagram for a 32-bit Wallace tree multiplier.

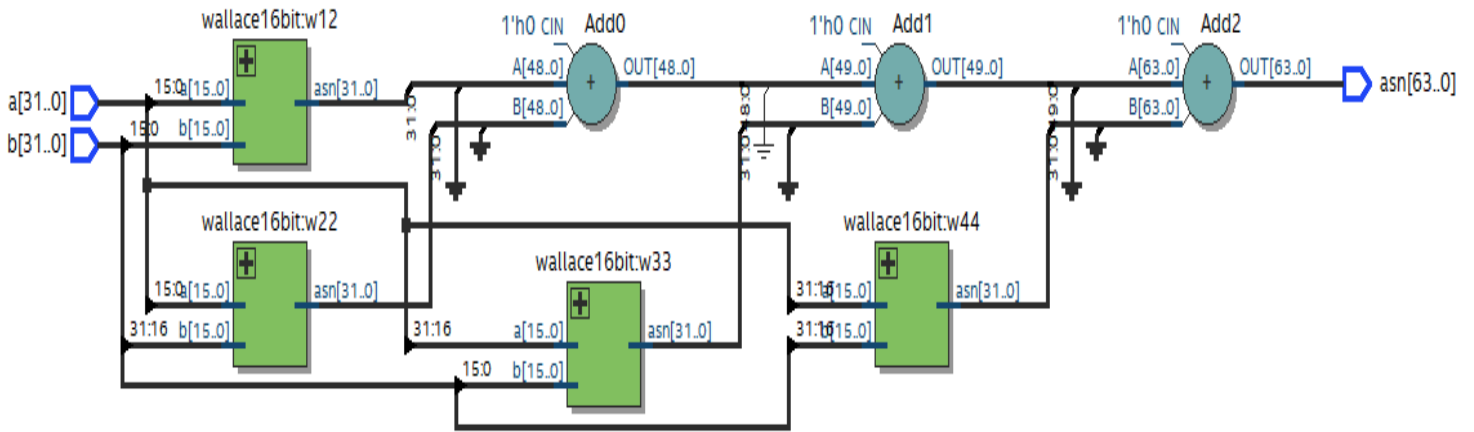


Figure 98 Wallace Tree 32-bit RTL

The following figure, Figure 99, shows the RLT Diagram for a 16-bit Wallace tree multiplier.

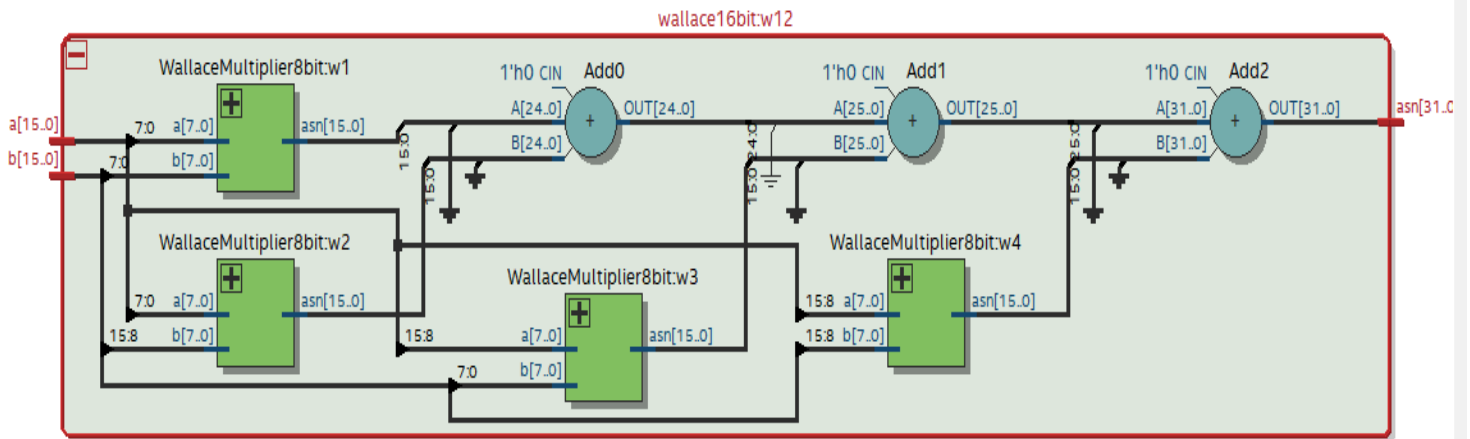


Figure 99 Wallace Tree 16-bit RTL

The following figure, Figure 100, shows the RTL Diagram for a 8-bit Wallace multiplier.

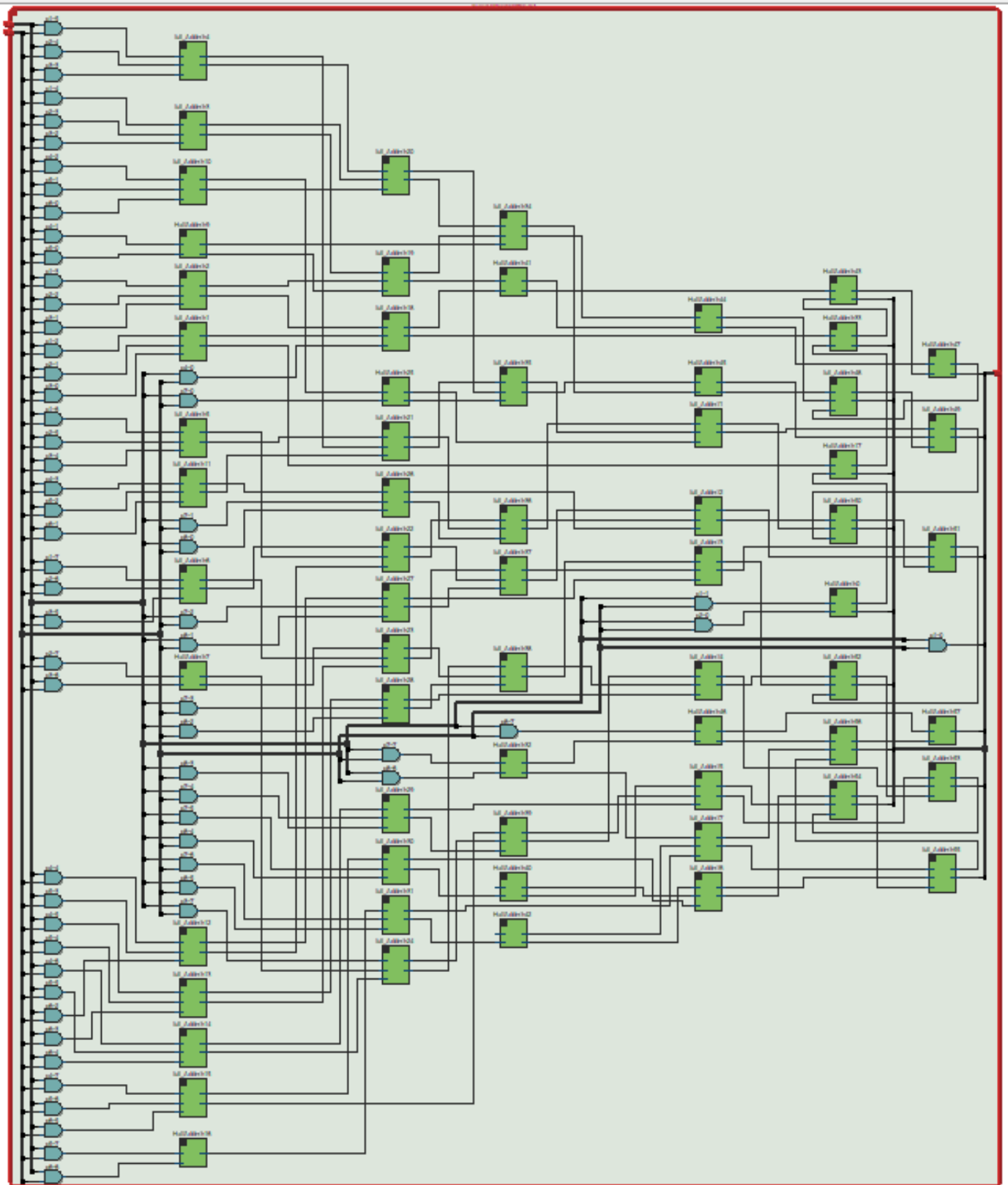


Figure 100 Wallace Tree 8-bit RTL

4.3.2.7 Mantissa Right Shifter

The next module for the floating point multiplication is used to normalize the output coming out from the previous module which is the mantissa Wallace multiplier. This module takes in the 48-bit mantissa product that is outputted from the previous module and checks the most significant bit of the mantissa product to decide for shifting operation. The mantissa left shifter shifts the 48-bit product by 1-bit if the most significant bit of the product is low which is binary 0.

If most significant bit of the mantissa product is 0 then the product is already normalized and next 23 bits after most significant bits are taken into consideration for further operations by consequent modules.

if most significant bit of the mantissa product is 1 then it is safe to assume that the next bit of the multiplication product is always 1, so starting from next to next bit, next 23 bits are taken into consideration for further operations by consequent modules.

```

136
137   assign shiftMantissa[4:0] = {4'b0000,productMantissa[47]};
138
139   Rightshifter #(.w(48)) ManittsashiftRightFinal
140   □ (
141     .in(productMantissa),
142     .shift(shiftMantissa),
143     .out(normalizedProductMantissa)
144   );
145
146

```

Figure 101 Right Shifter Instantiation

The figure above, Figure 101, shows the instantiation for left shifter module. As shown in the figure above, the shift variable depends on the most significant bit of the product.

Please refer to section 3.3.2.9 labelled Mantissa Normalizer to look at detailed description of the Left Shifter module including its workings, Code, & RTL Diagrams.

4.3.2.8 Mantissa Product Rounding

The next module for the floating point multiplication is used to round the output coming out from the previous module which is the mantissa left shifter module. The general rule when rounding binary fractions to the n^{th} place prescribes to check the digit following the n^{th} place in the number. If it's 0, then the number should always be rounded down. If, instead, the digit is 1 and any of the following digits are also 1, then the number should be rounded up.

Mantissa Product Rounding Verilog Code

The figure below, Figure 102, shows the Verilog code for the mantissa product rounding module. The code takes in a 24-bit input and returns a 1-bit output that is the product rounded value. The input for this module comes from the output of the previous left shifter module. The 24-bit input is sliced from the least significant bit to 24th bit of the 48-bit output coming from the left shifter module.

```

1  module productRounding
2  Ⓜ(
3     input [23:0] A,
4     output R
5  );
6
7     wire round;
8
9     assign round = |(A[22:0]); ///Ending 22 bits are OR'ed for rounding operation.
10    assign R = (A[23] & round);
11
12 endmodule
13

```

Figure 102 Product Rounding Verilog Code

As shown in the code, we are performing a reductive OR operation on 23-bits of the input and then performing an AND operation between the reduced bit and the 24th bit of the input. This performs the algorithm described in the above paragraph. The output of this module is then used as the least significant bit of the mantissa product value.

Mantissa Product Rounding RTL Diagram

As shown in Figure 103, the RTL diagram shows the logic that is used to design the product rounding module for our purposes. The product rounding module takes in a 24-bit input and has a one bit output that is used as the least significant bit for our final mantissa value used in computing the final results. 23-bits out of the 24-bits input excluding the most significant bit is fed together to an OR gate and reduced to a single bit as the output from the OR gate. This output is then inputted to an AND gate along with the most significant bit of the input. The output of this AND gate is our rounding result.

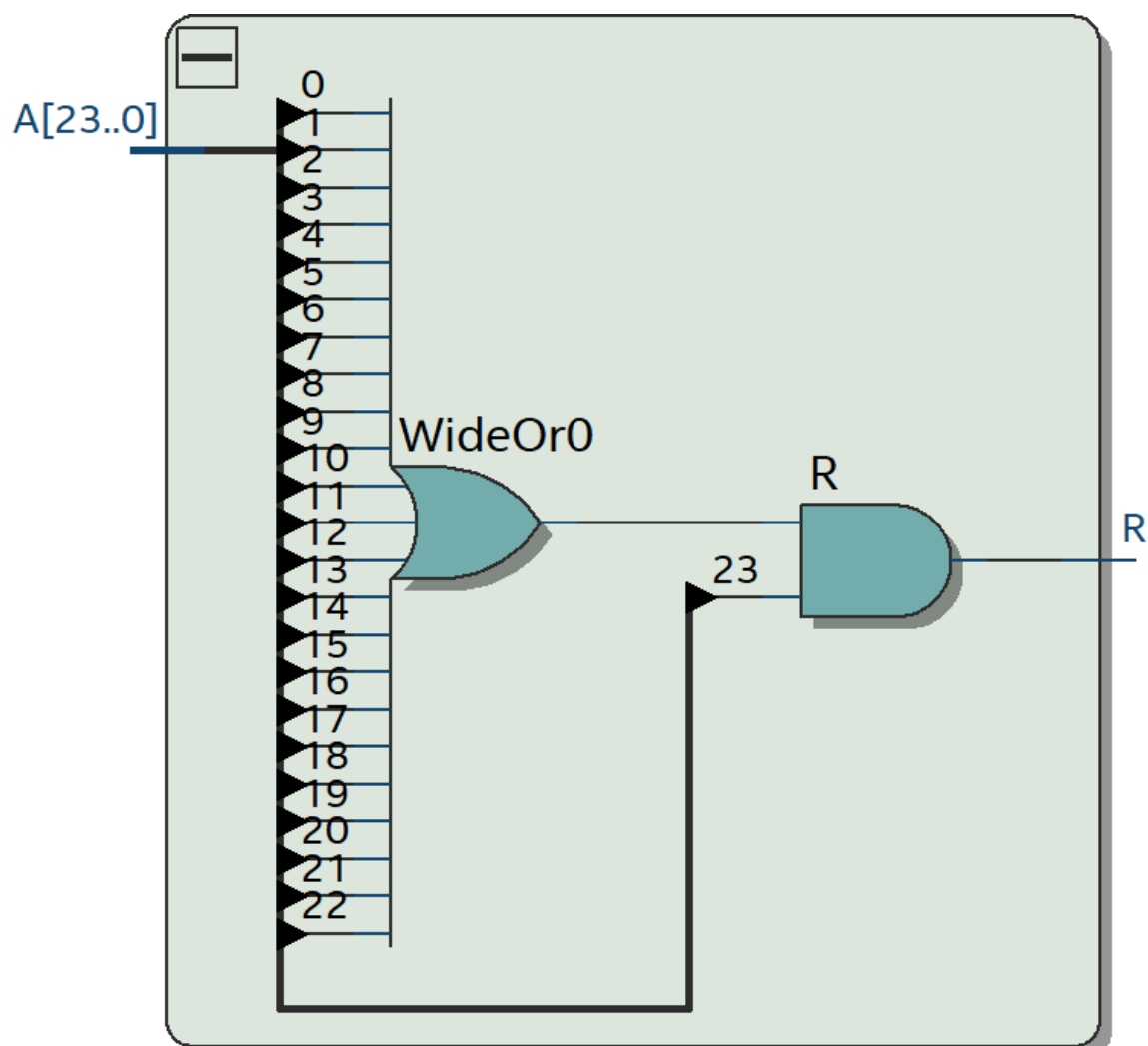


Figure 103 Product Rounding RTL Diagram

4.3.2.9 Exponent Incrementor

In this section of the thesis, we will move on to the next module of the floating point multiplier unit, the controlled exponent incrementor. The exponent incrementor is discussed in great detail in section 2.3.2.7 of chapter two of this thesis. As discussed in the section mentioned, the controlled exponent incrementor has an 8-bit input labelled E, and another 1-bit input labelled select, in addition there is an 8-bit output labelled as Out.

Each individual bit of the 8-bit input comes directly from the output of the modular bias subtractor module that was discussed previously in this chapter. Each bit of this 8-bit input feeds into seven different full adder and one half adder. The other 1-bit input called select goes into the first half adder. The output of the first half adder gets cascaded through to the next full adders and the outputs are all concatenated together to form the 8-bit output that is talked about in section 2.3.2.7.

The output of this controlled incrementor depends on the select input. The select input comes from the most significant bit of the Wallace tree multiplier result. If most significant bit of the Wallace tree multiplier result is 1 then the product is of the form 2^b11 , and we need to shift the decimal point to left to make the product normalized and therefore we add 1 to resultant exponent. If most significant bit of the Wallace tree multiplier result is 0 then the product is of the form 2^b01 and the product is already normalized and nothing is added or subtracted to exponent.

```

168
169 exponentIncrementorMUL expInc
170   (
171     .E(unbiasedExponent[8:0]),
172     .select(productMantissa[47]),
173     .out(finalExponent)
174   );
175

```

Figure 104 Exponent Incrementor Instantiation

4.3.2.10 Compute Flags

In this section of the thesis, we will move on to the next module of the floating point multiplier unit, the compute flags module. As discussed in chapter 1 of this thesis there are certain error flags that must be computed in accordance with the IEEE 754 standard when performing binary floating point arithmetic.

The flags that are expected to be computed during a floating point arithmetic operations are Zero, Exception, Underflow, and Overflow flags. The compute flags achieves this desired objective using logical operations on the final exponent and final mantissa value computed from modules discussed above [38].

- Exception: The exception flag is set to high if either of the two initial exponent values are 255 which is an error in the value of the initial exponent.
- Zero: If the exception flag is set to low and all the final mantissa bits are low in value then the mantissa equals a value of zero. This is when the Zero flag is set high.
- Overflow: If the Zero flag is set to high and the final exponent value in binary is a number greater than the upper bound limit of 255, the overflow flag is set to high.
- Underflow: If the Zero flag is set to high and the final exponent value in binary is a number lesser than the lower bound limit of 127, the underflow flag is set to high.

These four flags are computed in this module using AND and OR gates in association with NOT gates that compute the flags required following all required guidelines by the IEEE 754 standard released in the year 2008.

Compute Flags RTL Diagram

The figure on following page, Figure 105, shows the RTL diagram that was resulted upon designing the compute flags module.

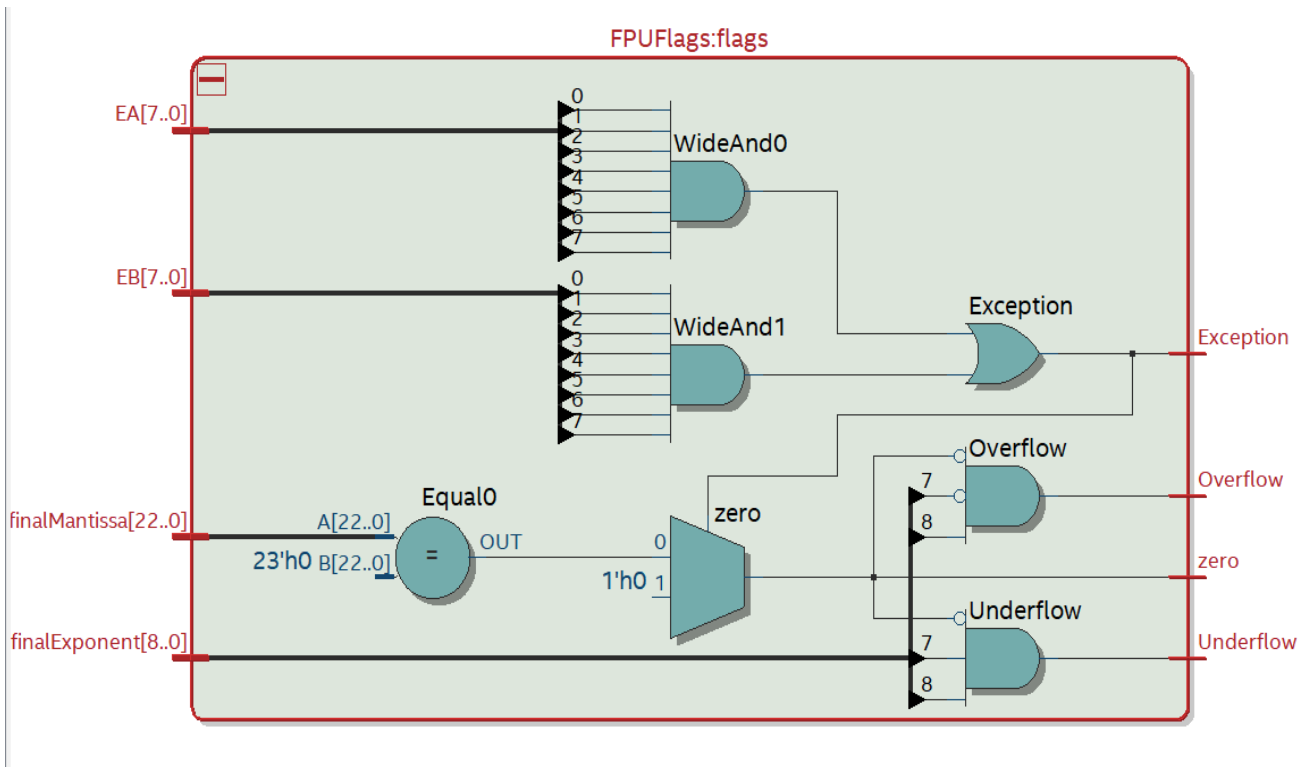


Figure 105 Compute Flags RTL Diagram

Exception

As shown in the RTL diagram above in Figure 105, The exception value is driven using the 8-bit exponent values from input A and input B. These two 8-bit inputs are sent to two individual 8 input wide AND gates which perform a reductive operation on these bits. The output coming from this AND gate is single bit and if each bit of the 8-bit input is 1 the AND gate output shows a 1 to signify that the input is 255 in value (8'b11111111).

Next, the outputs of the two individual AND gates are fed into the inputs of a 2 input wide OR gate. If either of the input of the OR gate is a 1 the output is also a 1 which then sets the exception flag high signifying at least one of the initial exponent value is 255.

Zero

As shown in RTL diagram zero flag is simply a combination of an equal to condition statement that compares the final mantissa value received from the left shifter module (section 4.3.2.6) with the zero value to check if it is equal to zero.

Additionally, we employ a multiplexer which selects between the result of this equal statement and another zero value. The select to this multiplexer comes from the exception value computed in the previous paragraph.

Overflow

As shown in RTL diagram the overflow flag is computed using the most significant bit and second most significant bit of the final exponent value computed using the exponent incrementor module. The 2-bits of the exponents are fed to two individual 3-input AND gate. The most significant bit of exponent is fed as inputted whereas the 7th bit of the final exponent is sent through a NOT gate before being fed to the AND gate..

Additionally, the third input value in the AND gate is the zero flag set in previous section. If the value of the output from this AND gate is set as high, it signifies that the exponent value is more than 255 and hence there is an overflow flag displayed.

Underflow

As shown in RTL diagram the underflow flag is computed using the most significant bit and second most significant bit of the final exponent value computed using the exponent incrementor module. The 2-bits of the exponents are fed to two individual 3-input AND gate. Additionally, the third input value in the AND gate is the zero flag set in previous section. If the value of the output from this AND gate is set as high, it signifies that the exponent value is less than 127 and hence there is an underflow flag displayed.

Compute Flags Verilog Code

The following figure, Figure 106, shows the Verilog code for a compute flag module used to implement the floating point multiplier. The code shows all the assign statements that computes all the flags using logic gates discussed in previous sections pertaining to the RTL diagram of the module.

```

module FPUFlags
(
    input [7:0] EA, EB,
    input [22:0] finalMantissa,
    input [8:0] finalExponent,

    output wire Exception, Overflow, Underflow,
    output wire zero
);
    //ZERO and Exception flag

    //Exception flag sets 1 if either one of the exponent is 255.
    assign Exception = (&EA) | (&EB);

    //If exception is true and final mantissa is 0
    assign zero = Exception ? 1'b0 : (finalMantissa == 22'd0) ? 1'b1 : 1'b0;

    //Overflow Flag
    //If final exponent is greater than 255 then overflow.
    assign Overflow = ((finalExponent[8] & !finalExponent[7]) & !zero) ;

    //Underflow Flag
    //If sum of both exponents is less than 127 then Underflow.
    assign Underflow = ((finalExponent[8] & finalExponent[7]) & !zero) ? 1'b1 : 1'b0;
endmodule

```

Figure 106 Compute Flags Verilog Code

As seen in the figure above, the module takes in two 8-bit inputs in form of exponent values from input A and input B. Module also takes in the 23-bit final mantissa value, and the 8-bit final exponent value as inputs. The outputs of this module are different error flags such as Exception, Overflow, Underflow, and finally the Zero flag.

4.3.2.11 Compute Output

The final module of the floating point multiplication unit is the compute output modules. As suggested by the name this final module outputs the result of the entire floating point multiplication arithmetic. In this module we are not simply concatenating the final exponent value with the final mantissa value along with the sign bit to get the final result.

This module has been designed in accordance with the IEEE 754 standard that dictates what the output of the arithmetic should look like based on the error flags that was computed in the previous module. The output for each computed error flag is defined by IEEE 754 standard is [39]:

- Exception: 32-bits output with all bits being 0 in value.
- Zero: The most significant bit of the result will be the sign bit computed previously. The rest of the 31-bits of the result will be all 0.
- Overflow: The most significant bit of the result will be the sign bit computed previously. The next 8-bits of the result will be the exponent which will have all 8-bits set to 1. The final 23 bits will be set to 0.
- Underflow: The most significant bit of the result will be the sign bit computed previously. The rest of the 31-bits of the result will be all 0.
- No Error: The most significant bit of the result will be the sign bit computed previously. The next 8-bits of the result will be the exponent as outputted from the exponent incrementor module. used previously in the design. The final 23-bits of the result is as outputted from the left shifter module used previously in the design.

These are standard output result values in case an error flag is noted by the floating point multiplication unit.

Compute Output RTL Diagram

The figure on following page, Figure 107, shows the RTL diagram that was resulted upon designing the compute output module.

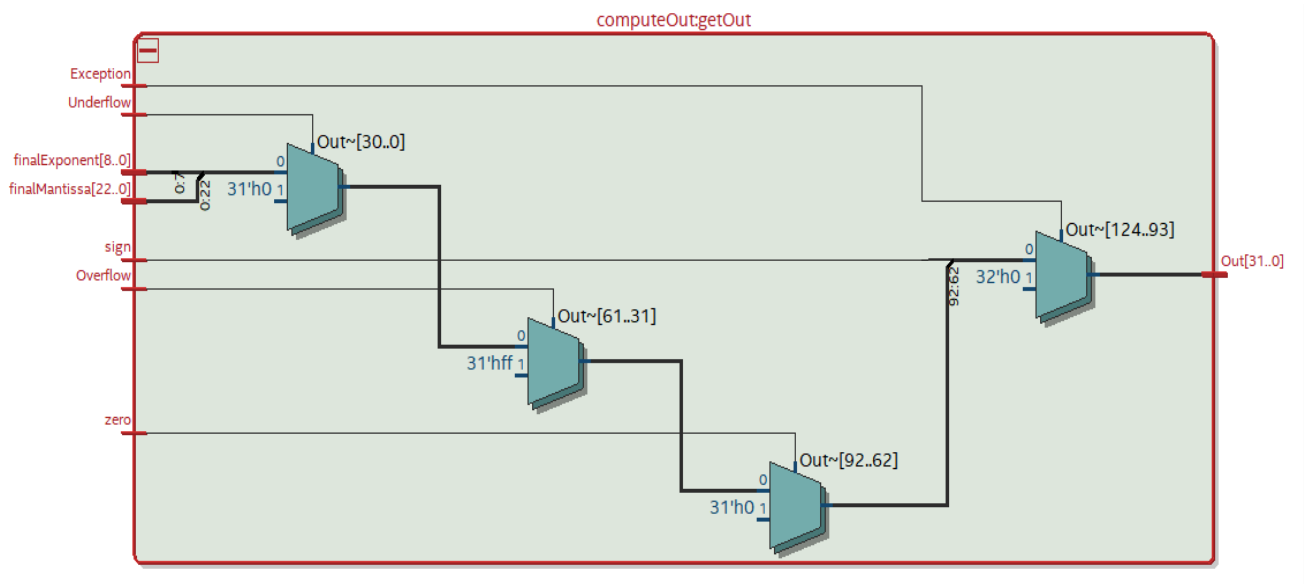


Figure 107 Compute Output RTL Diagram

As shown in the RTL diagram the module uses four multiplexers hardcoded with the IEEE 754 standard as outputs. The select for these multiplexers come from the error flag module discussed in previous section.

Compute Output Verilog Code

The figure on following page, Figure 108, shows the Verilog code that was used for designing the compute output module.

```

1  module computeout
2  (
3      input Exception, zero, sign, overflow, Underflow,
4      input [8:0] finalExponent,
5      input [22:0] finalMantissa,
6
7      output [31:0] out
8  );
9
10 assign out = Exception ? 32'd0 : zero ? {sign,31'd0} : overflow ? {sign,8'hFF,23'd0} :
11                Underflow ? {sign,31'd0} : {sign,finalExponent[7:0],finalMantissa};
12
13 endmodule
14
15

```

Figure 108 Compute Output Verilog Code

4.4 Floating Point Multiplier Results

The whole floating point multiplier unit was tested on Quartus' ModelSim simulation software using testbenches and waveforms. The design simulation involved generating setup scripts for the simulator, compiling simulation models, running the simulation, and viewing the results.

4.4.1 Floating Point Multiplier Compilation Report


Flow Summary	
 <<Filter>>	
Flow Status	Successful - Wed Apr 05 22:23:58 2023
Quartus Prime Version	18.1.0 Build 625 09/12/2018 SJ Lite Edition
Revision Name	FPM
Top-level Entity Name	FPM
Family	MAX 10
Device	10M08DAF484C8G
Timing Models	Final
Total logic elements	243 / 8,064 (3 %)
Total registers	0
Total pins	111 / 250 (44 %)
Total virtual pins	0
Total memory bits	0 / 387,072 (0 %)
Embedded Multiplier 9-bit elements	7 / 48 (15 %)
Total PLLs	0 / 2 (0 %)
UFM blocks	0 / 1 (0 %)
ADC blocks	0 / 1 (0 %)

Figure 109 FPM Compilation Report

4.4.2 Floating Point Multiplier Testbench

A testbench is used to generate the stimulus and applies it to the implemented floating point multiplier and compare the results against our calculations based on the IEEE 754 floating point calculator online [40]. The design was synthesized using precision synthesis tools targeting the DE-1 SoC Max 10 FPGA machine family.

```

`timescale 1ns / 1ps
module FPM_tb;

    // Inputs
    reg [31:0] A;
    reg [31:0] B;

    // Outputs
    wire [31:0] out;

    wire Exception;
    wire Overflow;
    wire Underflow;

    wire SnanA, QnanA, InfA, ZeroA, SubNA, NormA;
    wire SnanB, QnanB, InfB, ZeroB, SubNB, NormB;

    // Instantiate the Unit Under Test (UUT)
    FPM fpuAdderTB
    (
        .A(A),
        .B(B),
        .Out(out),
        .Exception(Exception),
        .Overflow(Overflow),
        .Underflow(Underflow),
        .SnanA(SnanA),
        .QnanA(QnanA),
        .InfA(InfA),
        .ZeroA(ZeroA),
        .SubNA(SubNA),
        .NormA(NormA),
        .SnanB(SnanB),
        .QnanB(QnanB),
        .InfB(InfB),
        .ZeroB(ZeroB),
        .SubNB(SubNB),
        .NormB(NormB)
    );

```

Figure 110 FPM Testbench

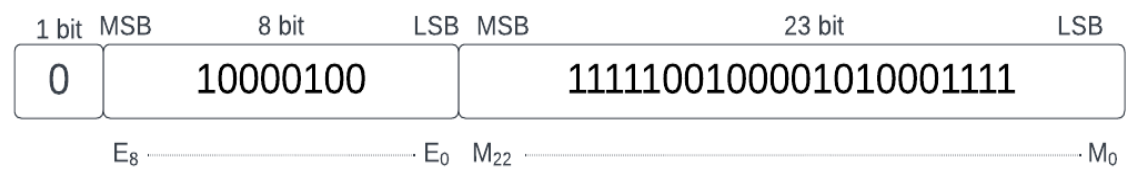
4.4.3 Floating Point Multiplier Simulation Results

Case A:

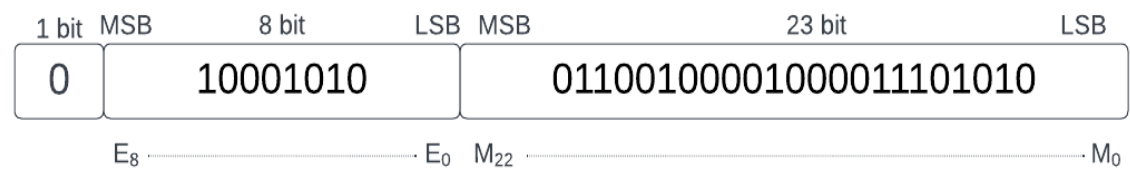
A:



B:



R:



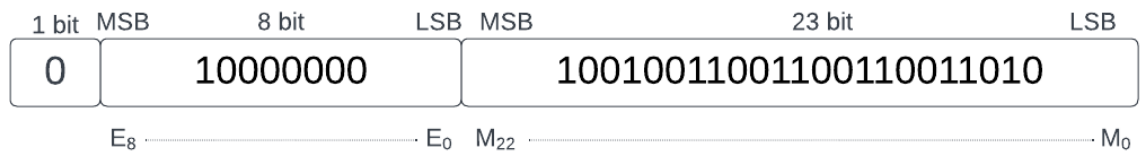
Simulation Results:

	Msgs				
/FPM_tb/A	11000001010100...	0100001000110100	1000010100011111		
/FPM_tb/B	00000000000000...	0100001001111100	1000010100011111		
/FPM_tb/Out	10000001110100...	01000101001100	100001000011101010		
/FPM_tb/Exception	St0				
/FPM_tb/Overflow	St0				
/FPM_tb/Underflow	St0				
/FPM_tb/SnanA	St0				
/FPM_tb/QnanA	St0				
/FPM_tb/InfA	St0				
/FPM_tb/ZeroA	St0				
/FPM_tb/SubNA	St0				
/FPM_tb/NormA	St1				
/FPM_tb/SnanB	St0				
/FPM_tb/QnanB	St0				
/FPM_tb/InfB	St0				
/FPM_tb/ZeroB	St1				
/FPM_tb/SubNB	St0				
/FPM_tb/NormN	HiZ				
/FPM_tb/NormB	St0				

Figure 111 FPM Case A Result

Case B:

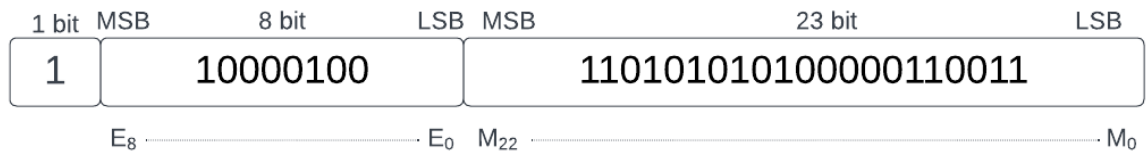
A:



B:



R:



Simulation Result:

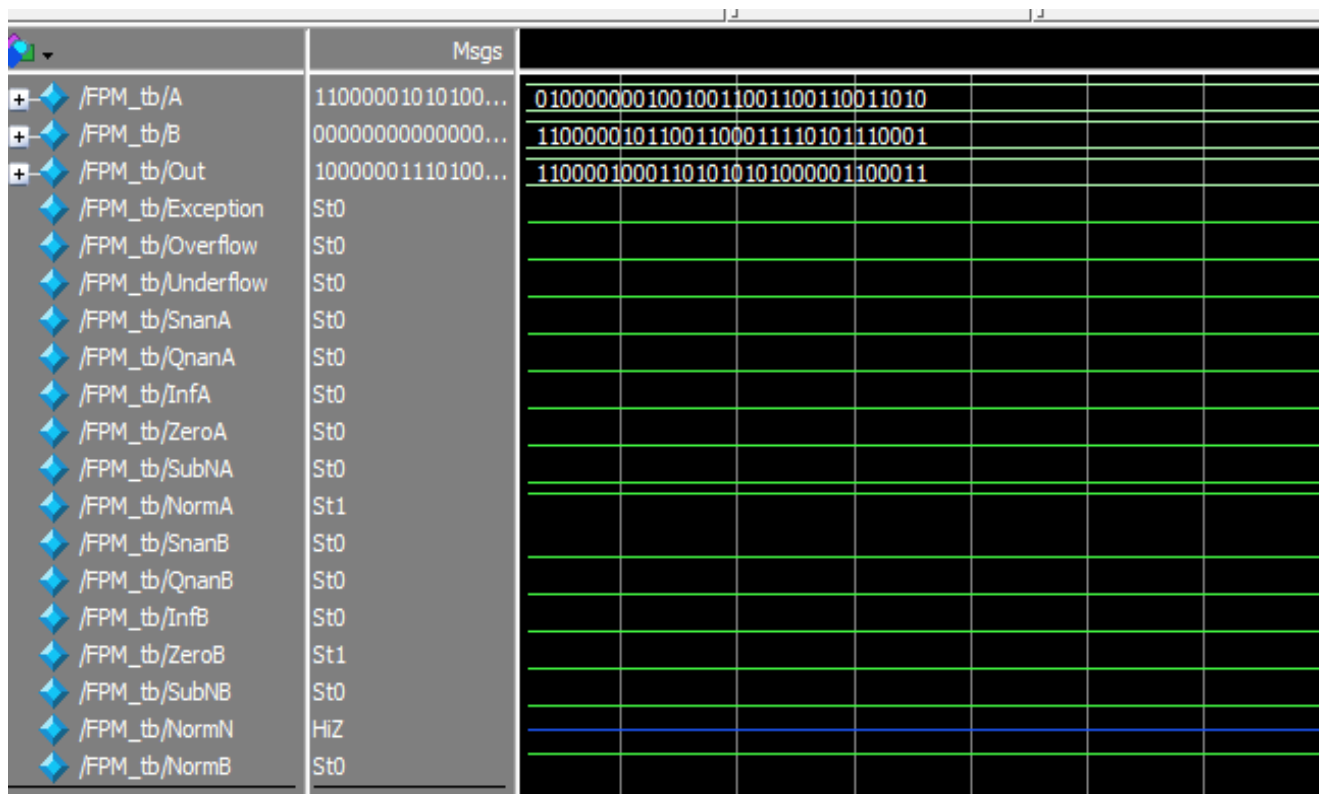


Figure 112 FPM Case B

Case C:

A:



B:



R:



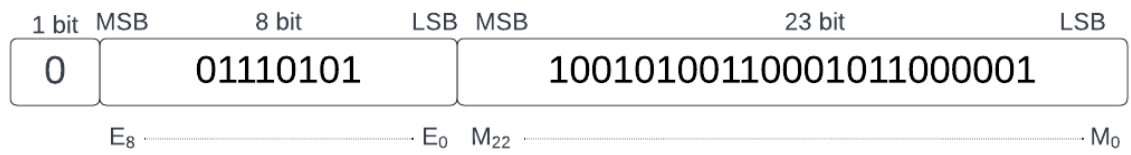
Simulation Results:

	Msgs				
/FPM_tb/A	11000001010100...	11000001010100100110011001100110			
/FPM_tb/B	0000000000000000...	11000010010000001010001111010111			
/FPM_tb/Out	10000001110100...	01000100000111100101001101110100			
/FPM_tb/Exception	St0				
/FPM_tb/Overflow	St0				
/FPM_tb/Underflow	St0				
/FPM_tb/SnanA	St0				
/FPM_tb/QnanA	St0				
/FPM_tb/InfA	St0				
/FPM_tb/ZeroA	St0				
/FPM_tb/SubNA	St0				
/FPM_tb/NormA	St1				
/FPM_tb/SnanB	St0				
/FPM_tb/QnanB	St0				
/FPM_tb/InfB	St0				
/FPM_tb/ZeroB	St1				
/FPM_tb/SubNB	St0				
/FPM_tb/NormN	HiZ				
/FPM_tb/NormB	St0				

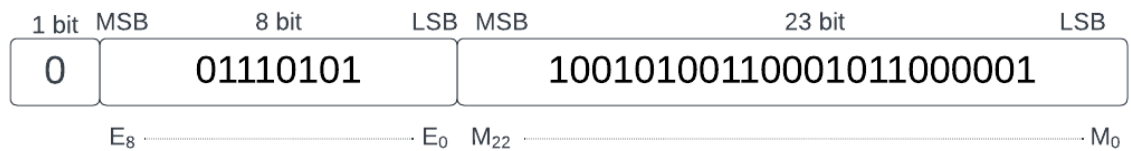
Figure 113 FPM Case C

Case D:

A:



B:



R:



Simulation Result:

	Msgs	
/FPM_tb/A	11000001010100...	00111010110010100110001011000001
/FPM_tb/B	0000000000000000...	00111010110010100110001011000001
/FPM_tb/Out	10000001110100...	00110110000111111111111111111111
/FPM_tb/Exception	St0	
/FPM_tb/Overflow	St0	
/FPM_tb/Underflow	St0	
/FPM_tb/SnanA	St0	
/FPM_tb/QnanA	St0	
/FPM_tb/InfA	St0	
/FPM_tb/ZeroA	St0	
/FPM_tb/SubNA	St0	
/FPM_tb/NormA	St1	
/FPM_tb/SnanB	St0	
/FPM_tb/QnanB	St0	
/FPM_tb/InfB	St0	
/FPM_tb/ZeroB	St1	
/FPM_tb/SubNB	St0	
/FPM_tb/NormN	HiZ	
/FPM_tb/NormB	St0	

Figure 114 FPM Case D

Case E:

A:



B:



R:



Simulation Result:

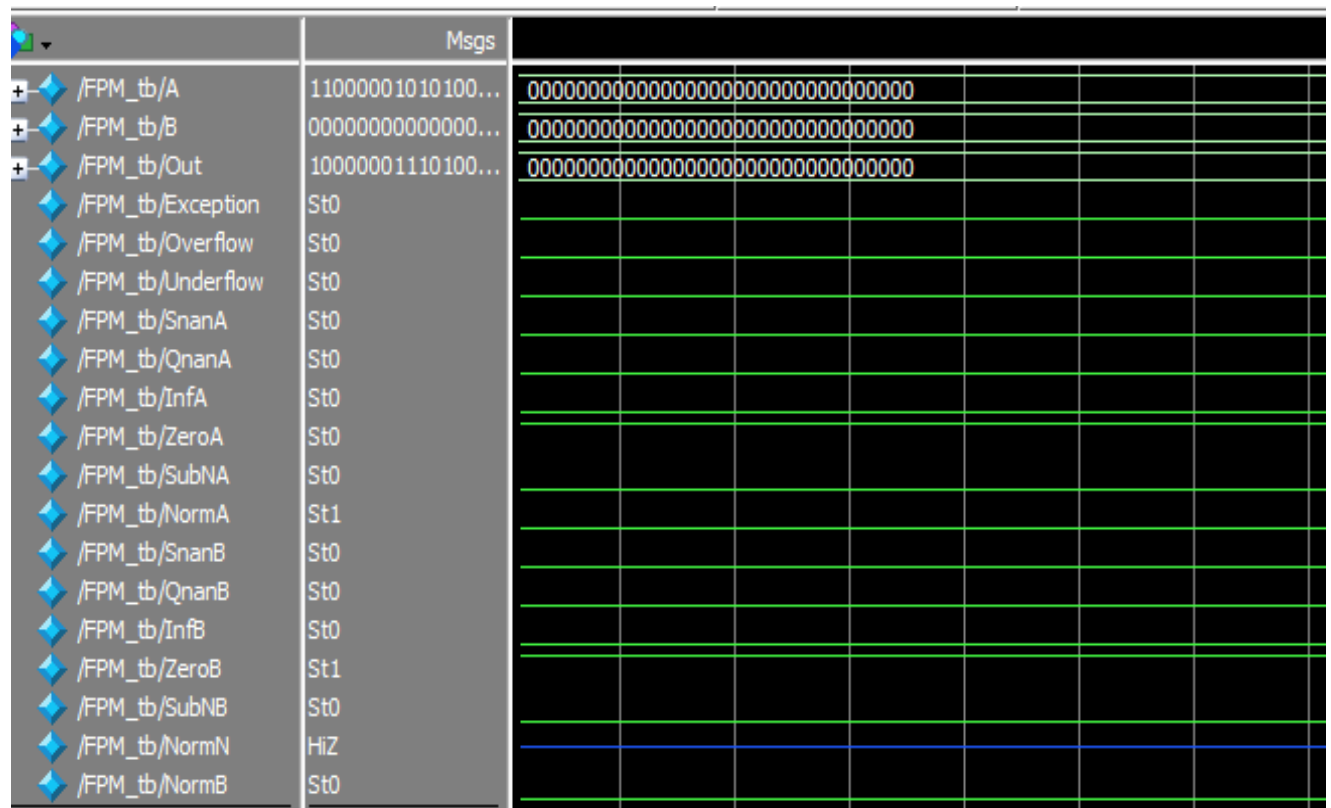


Figure 115 FPM Case E

4.5 Conclusion

This section of the thesis presented an implementation of a floating point multiplier that supports the IEEE 754-2008 binary interchange format. The multiplier implements this algorithm using a Wallace tree multiplier for faster computation and used various different modules to compute the final output.

Chapter 5: 32-bits Floating Point Divider

In this chapter, we describe an efficient implementation of an IEEE 754 single precision floating point divider targeted for DE-1 Cyclone V FPGA. Verilog is used to implement a technology-independent pipelined design. The divider implementation handles the overflow and underflow cases. Rounding is implemented to give more precision to the output of the divider operation. The Floating-Point Divider was verified by testbench simulations on ModelSim.

In this chapter we will dive deeper into the floating-point divider algorithm, architecture, code design, RTL diagram, and simulation results. Floating-point multiplication is much less complicated than addition and subtraction as the following discussion showcases:

We will talk about the procedure in division operations and a first look at the code design in a block diagram way followed by deeper understanding of code development.

Floating point division is done by extracting signs, subtracting exponents, dividing mantissa values, and shifting the mantissa for normalization [41].

There are six basic phases of designing a Floating-Point Multiplier:

- 1) Check for Zeroes.
- 2) Subtract exponents.
- 3) Add Bias.
- 4) Divide the Significands.
- 5) Normalize the Significand.
- 6) Normalize the Exponent if needed.

5.1 Floating Point Division Algorithm

As described in the above topics, floating point number is in the format of:

$$Z = (-1)^S * 2^{(E - \text{Bias})} * (1.M)$$

To divide two floating point numbers A & B the different steps to follow are [42]:

- 1) Extracting signs, exponents and mantissas of both A and B numbers.
- 2) Calculating the output sign.
- 3) Treating the special cases.
- 4) Finding out the data types of numbers given
- 5) Subtracting the two exponents.
- 6) Adding the bias from exponent subtraction.
- 7) Dividing the mantissa values
- 8) Normalizing mantissa by bit shifting.
- 9) Normalizing exponent if necessary.
- 10) Detecting exception, overflow, and underflow.

5.1.1 Floating-Point Division Example

A = 127.03125 (base 10)

B = 16.9375 (base 10)

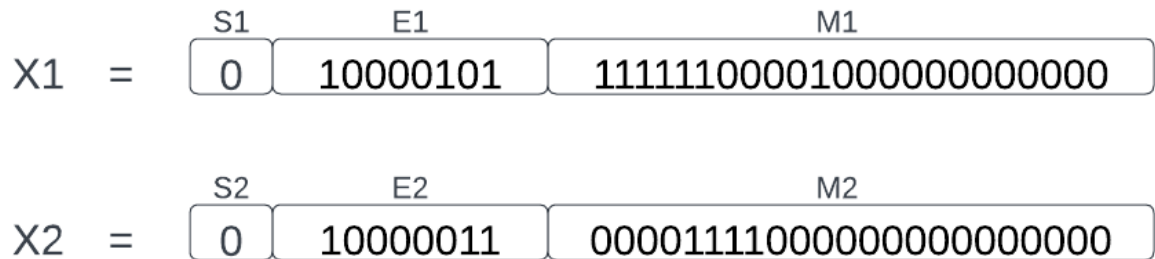


Figure 116 Binary Presentation Divide Example

9) $S_1 = 0, E_1 = 10000101, M_1 = 111111000010000000000000$

$S_2 = 0, E_2 = 10000011, M_2 = 000011110000000000000000$

10) Sign bit calculation



Figure 117 XOR Sign Division

$S_3 = 0$

11) Exponent Subtraction

$$\begin{array}{r}
 10000101 \\
 - 10000011 \\
 \hline
 00000010
 \end{array}$$

Figure 118 Divide Exponent Subtraction

Unbiased Exponent = 00000010

12) Add Bias

$$\begin{array}{r}
 00000010 \\
 + 01111111 \\
 \hline
 10000001
 \end{array}$$

Figure 119 Divide Bias Subtraction

Biased Exponent = 10000001

13) Divide the Mantissa

$$\begin{array}{r}
 1.111111000010000000000000 \\
 / 1.000011110000000000000000 \\
 \hline
 1.111000000000000000000000
 \end{array}$$

Figure 120 Mantissa Division

1.M3 = 1.111000000000000000000000

14) Right Shift the Mantissa for normalization

No right shift needed.

Right Shifted Mantissa = 1.111000000000000000000000

15) Decrement the exponent

No decrement for exponent needed.

16) Result

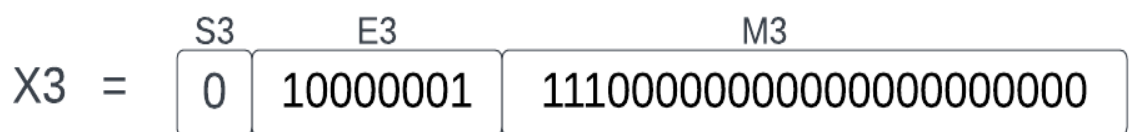


Figure 121 FPD Example Result

5.2 Floating Point Divider Flowchart

The below, Figure 122, showcases a typical flowchart that is used to design a floating point divider. The figure shows a step by step narrative and displays the high level functions that is required to compute floating point division. The flowchart shows block level diagram and each block or element is implemented in hardware and is described in detail in the following topics of the thesis [43].

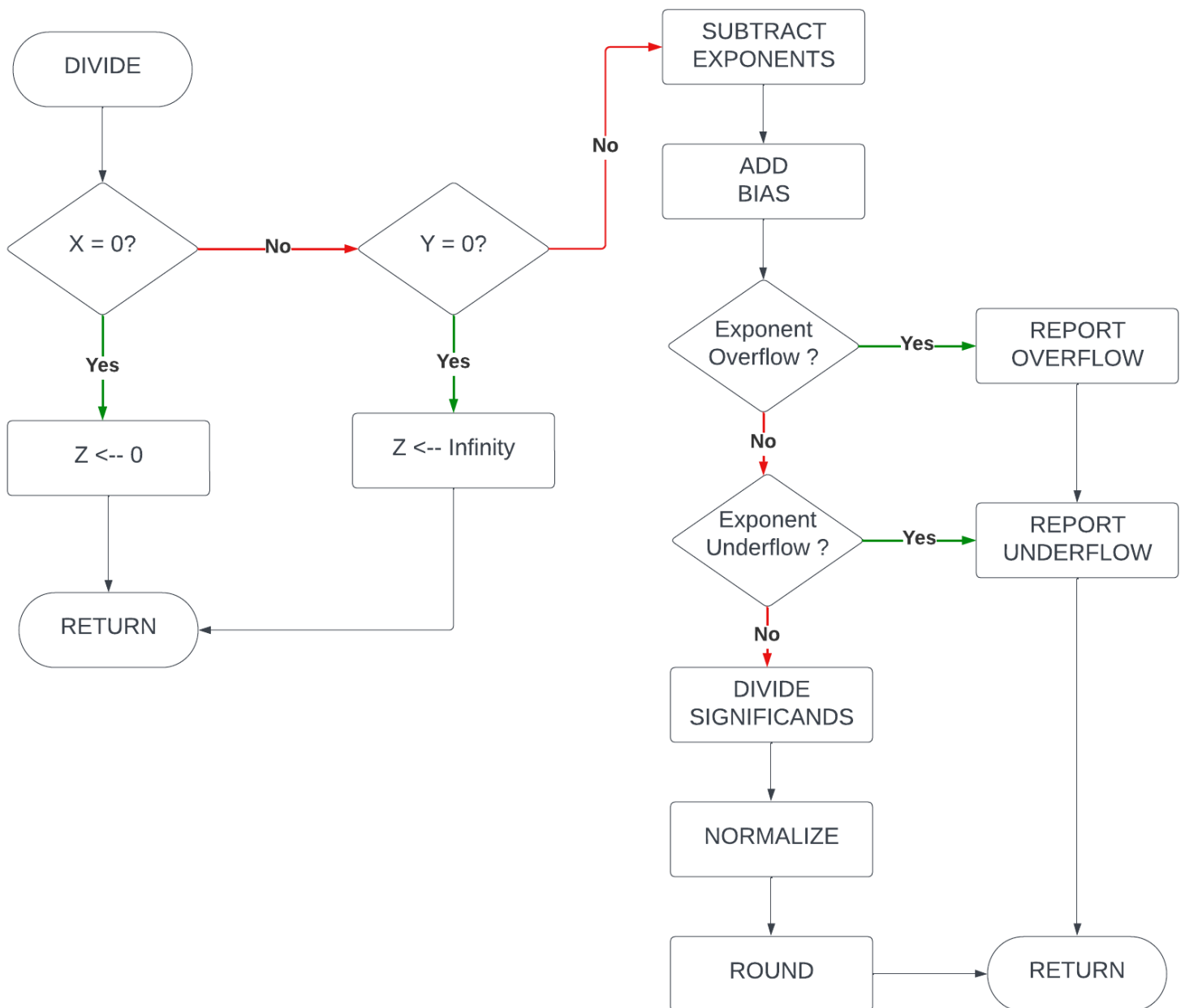


Figure 122 Floating Point Divider

5.3 Floating Point Divider Hardware

In this section of the thesis we will start explaining and diving deeper into the hardware implementation of the floating point division. This section will start by elaborating the flowchart further with help of showcasing the hardware architecture used to design the module followed by detailed description of each module used in the architecture.

After understanding the theory of hardware implementation and the architecture of floating point division the thesis will show the code development that achieved out final objective of building this floating point unit.

5.3.1 Floating Point Divider Hardware Architecture

The below figure, Figure 123, showcases the hardware architecture that was designed and coded to implement synthesizable 32-bit floating multiplier adder using Verilog following the IEEE 754 standard.

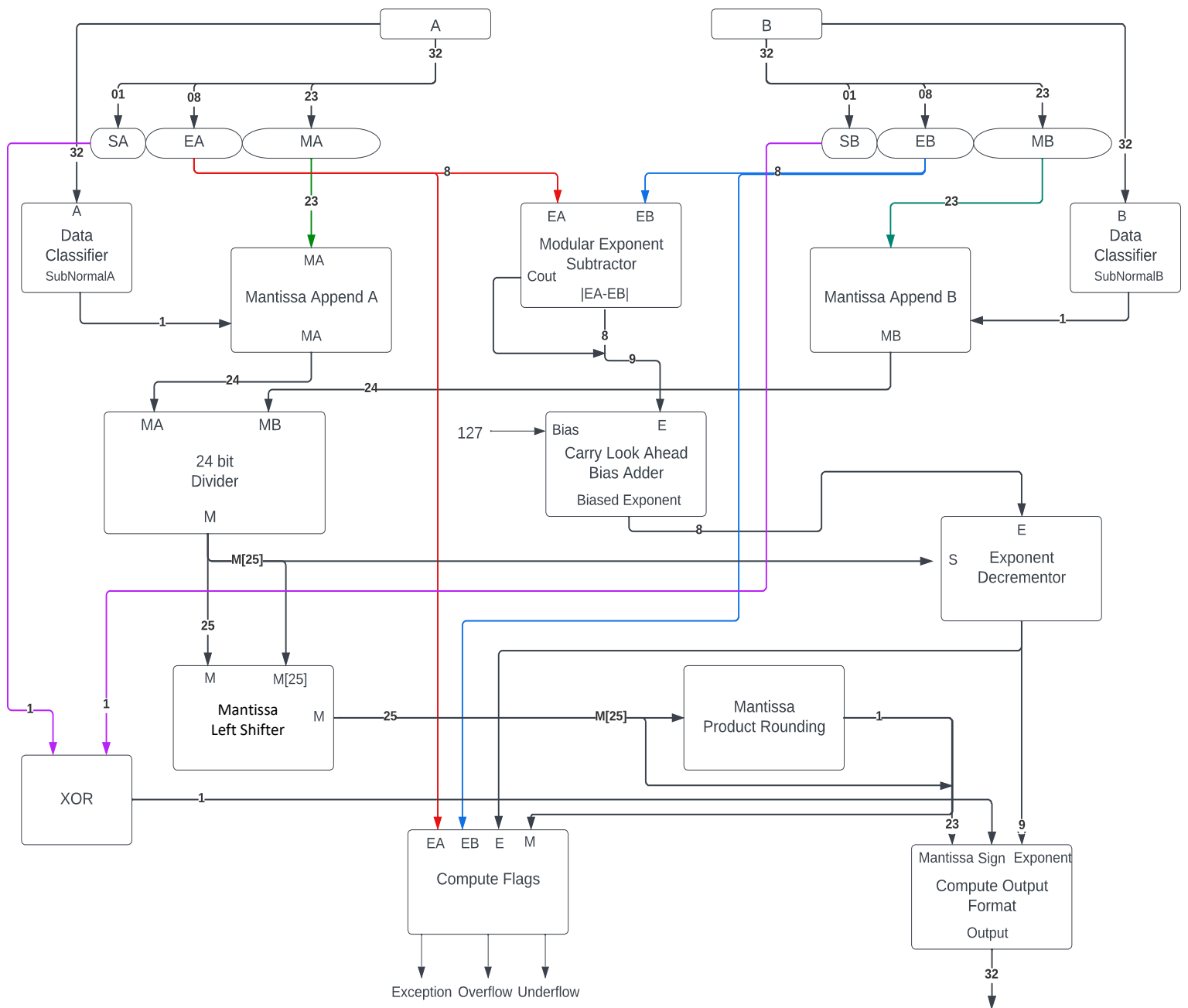


Figure 123 Floating Point Division

This floating point architecture uses a total of ten modules that serve various unique purposes in making the design work. The modules are:

- Modular Exponent Subtractor
- Mantissa Append Module
- 24-bit Divider
- Exponent Decrement
- Compute Flags
- Data Classifier
- CLA Bias Adder
- Mantissa Left Shifter
- Mantissa Division Rounding
- Compute Output

5.3.2 Floating Point Divider Hardware Implementation

In this section, we will discuss the hardware implementation designed for the floating point divider and explain each module and each algorithm step in detail.

5.3.2.1 Sign Bit Calculation

Dividing two positive numbers will result in a positive number. Dividing two negative numbers will result in a negative number. Dividing one positive number and one negative number will result in a negative number. The sign bit calculation for this floating point division unit is done using an XOR gate. The table below shows sign operations for various cases:

A's Sign	Symbol	B's Sign	Operation
+	/	+	+
+	/	-	-
-	/	+	-
-	/	-	-

Table 15 Sign Operations Divide

5.3.2.2 Data Classification Module

A 32-bit binary floating point number can be encoded to form a total of six different cases based on the value of each data bit. The six different data types and the criteria that must be met for their encoding are:

- 1) Signalling NaN (sNaN)
- 2) Quiet NaN (qNaN)
- 3) Negative Infinity ($-\infty$)
- 4) Positive Infinity ($+\infty$)
- 5) Positive Zero (+0)
- 6) Negative Zero (-0)
- 7) Subnormal
- 8) Normal

The Data classification module takes in the two inputs A and B as inputs to the module and computes the type of input into 8 different data types. The data types are defined in great detail and the criteria necessary for a data to be classified as each type in section 4.3.2.2 in the multiplication chapter.

The data classification module outputs 8-bits of data with each line of data carrying one classification for the data input. The output is high for output that the input is classified as and low for all the other outputs as an input cannot be classified into more than one data type. The Subnormal bit of output is then fed into the mantissa append module discussed in one of the sections below.

The data classification module has been described in great detail in section 4.3.2.2 of the multiplication chapter using Verilog code, RTL diagram, and classification examples.

The figure below, Figure 124, shows the instantiation of the data classification module used to classify the data input B.

```

30  sp_class classB
31  (
32      .A(B),
33      .snan(SnanB),
34      .qnan(QnanB),
35      .infinity(InfB),
36      .zero(ZeroB),
37      .subnormal(SubNB),
38      .normal(NormB)
39  );
40

```

Figure 124 Data Classification Instantiation B

The figure below, Figure 125, shows the instantiation of the data classification module used to classify the data input A.

```

18
19  sp_class classA
20  (
21      .A(A),
22      .snan(SnanA),
23      .qnan(QnanA),
24      .infinity(InfA),
25      .zero(ZeroA),
26      .subnormal(SubNA),
27      .normal(NormA)
28  );
29

```

Figure 125 Data Classification Instantiation

The above two figures shows the data classification module instantiation used in the floating point division module. As shown in figures the two modules have the inputs A and B for each module. The figure also shows six different outputs coming out of the data classification module. Each of these 6 bits depict each of the data class discussed in the paragraph above. These outputs are wires that goes into next modules to assist with operations done in consecutive modules.

5.3.2.3 Modular Exponent Subtractor

This modular exponent subtractor is responsible for subtracting the exponent of the second input from the exponent of the first input. This module of hardware description language ensures that the exponent difference value is absolute in nature. Before the subtraction operation is performed the program doesn't know which exponent is higher in value. The modular exponent subtractor allows us to not just compute the absolute exponent difference, it also allows us to identify the larger exponent. This exponent difference will further be sent to the bias addition module, the output of which, will be used for the exponent decrement module and ultimately computing the result of the entire operation.

The figure below, Figure 126, shows the instantiation of the modular exponent subtractor used for the floating point divider algorithm. As shown in the figure, the two inputs of the modules are exponent of A and B, the output of the module is a wire called exponent diff that is fed into the next module discussed in the next section.

```

67
68
69   ModeSubtractor #(.w(8)) EAEBSub
70   □ (
71     .A(EA),
72     .B(EB),
73     .OpCode(1'b1),
74     .R(exponentDiff),
75     .Cout(coutSub)
76   );
77   |
78

```

Figure 126 Mode Subtractor Instantiation

The modular exponent subtractor module has been described in great detail in section 2.3.2.2 of the adder chapter using Verilog code, RTL diagram, Block diagram and detailed explanation of each module inside the top level subtractor module.

5.3.2.4 Carry Lookahead Bias Adder

This carry lookahead bias adder module is the next arithmetic operation module that constitutes the floating point division algorithm described in section 5.1 above in this chapter. This module makes use of the same carry lookahead adder module that was used in section 2.3.2.6 of chapter two to carry out exponent addition.

This module's primary task is to add the fixed bias value of 127_{10} to the result of the modular exponent subtractor module. When our design subtracted the two exponent values with each other, the bias of those two exponents also got subtracted and cancelled out. This module adds the negated bias value and normalizes the exponent back to its correct magnitude. The instantiation for this module is shown in Figure 127 below:

```

85
86
87     CLAParameter #(.N(8)) ExponentBiasAdder
88     (
89         .A(exponentDiff),
90         .B(8'd127),
91         .opCode(1'b0),
92         .Cout(coutExpOp),
93         .R(biasedExponentTemp)
94     );
95
96

```

Figure 127 Bias Addition Instantiation

As shown in the instantiation above in Figure 127, the inputs to the carry lookahead bias adder is the output of the exponent subtraction, along with 00111111_2 which is 127 in decimal. This module is also a parameterized module which has been modified to operate on 8-bits for this operation. The output of this module is absolute value and it will be fed to the exponent decrement module discussed in coming sections.

Please refer to section 2.3.2.6 for details about carry lookahead bias adder and all its constituting elements including Verilog code, RTL Diagram, and Block Diagram.

5.3.2.5 Mantissa Append Module

This module acts as a preparation step before we get to the most crucial step of the floating point division algorithm which is mantissa division.

This module's primary task is to compute the hidden/implied bit of the mantissa that exists at the most significant bit spot but hidden for representation purposes. The hidden bit of a mantissa depends on the data type of each input.

This module takes in two 23-bits inputs which come directly from the mantissa of input A. The input S comes from the Subnormal output of the data classification module. As shown in the code, if the select is high (i.e. input is of type subnormal) then the output of the module is the input of the module appended with value of 0. If the select is low the output of the module is the input of the module appended with value of 1.

The operation of this module is described in the table below:

Input 1	Input 2	Select	Output
A	B	1	{1'b0,A}
A	B	0	{1'b1,B}

Table 16 Append Mantissa Truth Table

The figure below, Figure 128, shows the instantiation of the mantissa append module used to append the mantissa of input A.

```
107
108
109 appendMantissa appendA
110 (
111     .A(MA),
112     .S(SubNA),
113     .R(AppendedMantissaA)
114 );
115
116
```

Figure 128 Append Mantissa A Instantiation

The figure below, Figure 129, shows the instantiation of the mantissa append module used to append the mantissa of input B.

```
115
116
117 appendMantissa appendB
118 (
119     .A(MB),
120     .S(SubNB),
121     .R(AppendedMantissaB)
122 );
123
124
```

Figure 129 Append Mantissa A Instantiation

The above two figures shows the mantissa append module instantiation used in the floating point division module. As shown in figures the two modules have the inputs A and B for each module. The figure also shows one output coming out of the mantissa append module. These outputs are wires that goes into next modules to assist with operations done in consecutive module of mantissa division.

5.3.2.6 Mantissa 24-bit Divider

The 24-bit mantissa divider module is the next arithmetic operation module that constitutes the floating point division algorithm described in section 5.1 above in this floating point division chapter.

This module takes in two 24-bits input A and B and produce a 24-bits output that is the quotient result of inputs A and B. The input A to this module comes from the first mantissa append module and the second input, input B, comes from the second mantissa append module as described in the section above [44].

Mantissa 24-bit Divider Verilog Code

```

1  module dividerTest(A,B,Res);
2      parameter WIDTH = 24;
3      input  [WIDTH*2-1:0] A;
4      input  [WIDTH-1:0] B;
5      output [WIDTH:0] Res;
6      reg   [WIDTH:0] Res = 0;
7      reg   [WIDTH*2-1:0] a1;
8      reg   [WIDTH-1:0] b1;
9      reg   [WIDTH:0] p1;
10     reg   [WIDTH*2-1:0] Div_Test;
11     integer i;
12
13     always@ (A or B)
14     begin
15         a1 = A;
16         b1 = B;
17         p1 = 0;
18         start = 0;
19         for(i=0; i < WIDTH; i=i+1) begin //start the for loop
20             p1 = {p1[WIDTH-2:0], a1[WIDTH-1]};
21             a1[WIDTH-1:1] = a1[WIDTH-2:0];
22             p1 = p1-b1;
23             if(p1[WIDTH-1] == 1) begin
24                 a1[0] = 0;
25                 p1 = p1 + b1; end
26             else
27                 a1[0] = 1;
28         end
29         Res = a1;
30     end
31 endmodule
32
33

```

Figure 130 24-bit Divider Code

The following figure, Figure 130, shows a code snippet from the 24-bit mantissa divider code. The code shows two 24 inputs being taken in the mantissa divider and a 24-bit quotient being outputted.

5.3.2.7 Mantissa Left Shifter

The next module for the floating point division is used to normalize the output coming out from the previous module which is the mantissa divider module. This module takes in the 25-bit mantissa division value that is outputted from the previous module and checks the most significant bit of the mantissa division value to decide for shifting operation. The mantissa left shifter shifts the 25-bit division result by 1-bit, if the most significant bit of the division result is low which is binary 0.

If most significant bit of the mantissa division result is 1 then the division value is already normalized and next 23 bits after most significant bits are taken into consideration for further operations by consequent modules.

if most significant bit of the mantissa division is 0 then it is safe to assume that the next bit of the division value is always 1, so starting from next to next bit, next 23 bits are taken into consideration for further operations by consequent modules.

```
//If the MSB of the product is 0 then shift the result to the left by 1-bit.
assign shiftMantissa[4:0] = {4'b0000,~quotientMantissa[24]};

Leftshifter #(.w(25)) ManitssaShiftLeftFinal
  (
    .in(quotientMantissa),
    .shift(shiftMantissa),
    .out(normalizedQuotientMantissa)
  );
```

Figure 131 Left Shifter Instantiation

The figure above, Figure 131, shows the instantiation for right shifter module. As shown in the figure above, the shift variable depends on the most significant bit of the product.

Please refer to section 2.3.2.4 labelled Mantissa Right Shifter to look at detailed description of the Right Shifter module including its workings, Code, & RTL Diagrams.

5.3.2.8 Mantissa Division Rounding

The next module for the floating point division is used to round the output coming out from the previous module which is the mantissa right shifter module.

The working of the Mantissa Product Rounding module is discussed in great detail in section 4.3.2.8 of the multiplication chapter. Please refer to that section to understand the inner workings of this module by the help of Verilog code, RTL diagram, and explanation

The figure below, Figure 132, shows the instantiation of the division result rounding module that was used in this floating point division unit.

```
155
156   wire quotientRound;
157
158   productRounding #(.N(0)) round
159   (
160     .A(normalizedQuotientMantissa[0]),
161     .R(quotientRound)
162   );
163
164
```

Figure 132 Division Rounding Instantiation

As shown in the figure above, the input of this module is the least significant bit from the result of the mantissa division computed by the module explained in the section above. The output of this module is a one bit value which is then concatenated with the rest of the quotient value to form the final mantissa value of the floating point division arithmetic result.

5.3.2.9 Exponent Decrement

In this section of the thesis, we will move on to the next module of the floating point divider unit, the controlled exponent decrement module. The exponent decrement module is discussed in great detail in section 3.3.2.8 of chapter three of this thesis. As discussed in the section mentioned, the controlled exponent decrement module has an 8-bit input labelled E, and another 1-bit input labelled select, in addition there is an 8-bit output.

Each individual bit of the 8-bit input comes directly from the output of the carry lookahead bias adder module that was discussed previously in this chapter. Each bit of this 8-bit input feeds into seven different full adder and one half adder. The other 1-bit input called select goes into the first half adder. The output of the first half adder gets cascaded through to the next full adders and the outputs are all concatenated together to form the 8-bit output that is talked about in section 3.3.2.8.

The output of this controlled decrement module depends on the select input. The select input comes from the most significant bit of the mantissa divider result. If most significant bit of the mantissa divider result is 1 then the divider result is of the form 2^b11 , and we need to shift the decimal point to right to make the divider result normalized and therefore we subtract 1 to resultant exponent. If most significant bit of the mantissa divider result is 0 then the divider result is of the form 2^b01 and the divider result is already normalized and nothing is added or subtracted to exponent.

```

175
176   exponentDecrementor expDec
177   □ (
178     .E(biasedExponent[8:0]),
179     .select(quotientMantissa[24]),
180     .out(finalExponent)
181   );
182

```

Figure 133 Exponent Decrement Instantiation

5.3.2.10 Compute Flags

In this section of the thesis, we will move on to the next module of the floating point divider unit, the compute flags module. As discussed in chapter 1 of this thesis there are certain error flags that must be computed in accordance with the IEEE 754 standard when performing binary floating point arithmetic.

The flags that are expected to be computed during a floating point arithmetic operations are Zero, Exception, Underflow, and Overflow flags. The compute flags achieves this desired objective using logical operations on the final exponent and final mantissa value computed from modules discussed above.

The figure below, Figure 134, shows the instantiation of the compute flags module used to compute error flags for the floating point division module.

```

187 .....
188 FPUFlags flags
189 (
190     .EA(EA),
191     .EB(EB),
192     .finalMantissa(finalMantissa),
193     .finalExponent(finalExponent),
194     .Exception(Exception),           //ZERO and Exception flag
195     .Overflow(Overflow),           //Overflow Flag
196     .Underflow(Underflow),        //Underflow Flag
197     .zero(zero)
198 );
199 .....

```

Figure 134 Compute Flags Instantiation

As shown in the figure above, the compute flags module has two 8-bit inputs. The first input is the exponent of input A and the second input is the exponent of input B. And the figure also shows another two 23-bits input in terms of the two mantissa values. The module outputs the four error flags as discussed.

Please refer to section 4.3.2.10 from chapter 4 of the thesis for detailed explanation into working of this module using block diagram, RTL diagram, and Verilog code.

5.3.2.11 Compute Output

The final module of the floating point division unit is the compute output modules. As suggested by the name this final module outputs the result of the entire floating point division arithmetic. In this module we are not simply concatenating the final exponent value with the final mantissa value along with the sign bit to get the final result.

This module has been designed in accordance with the IEEE 754 standard that dictates what the output of the arithmetic should look like based on the error flags that was computed in the previous module.

The figure below, Figure 135, shows the instantiation of the compute output module used to compute output for the floating point division module.

```
201
202
203 computeout getout
204 (
205     .Exception(Exception),
206     .zero(zero),
207     .sign(sign),
208     .Overflow(Overflow),
209     .Underflow(Underflow),
210     .finalExponent(finalExponent),
211     .finalMantissa(finalMantissa),
212     .out(out)
213 );
214
```

Figure 135 Compute Out Instantiation

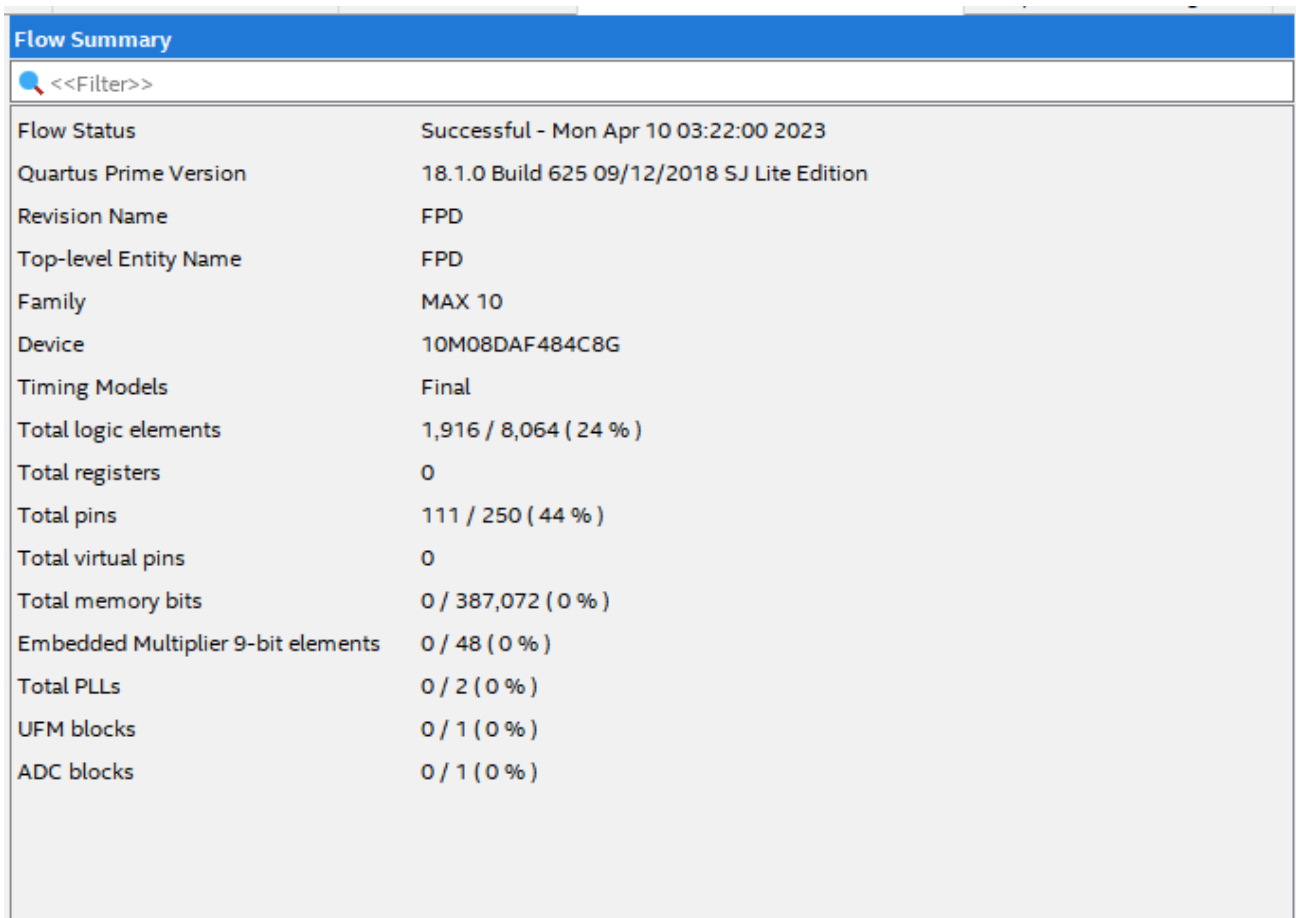
As shown in the figure above, the compute output module has inputs in the form of final mantissa and exponent, along with error flags computed in previous module. T

Please refer to section 4.3.2.11 from chapter 4 of the thesis for detailed explanation into working of this module using block diagram, RTL diagram, and Verilog code.

5.4 Floating Point Divider Results

The whole floating point divider unit was tested on Quartus' ModelSim simulation software using testbenches and waveforms. The design simulation involved generating setup scripts for the simulator, compiling simulation models, running the simulation, and viewing the results.

5.4.1 Floating Point Divider Compilation Report



Flow Summary	
Flow Status	Successful - Mon Apr 10 03:22:00 2023
Quartus Prime Version	18.1.0 Build 625 09/12/2018 SJ Lite Edition
Revision Name	FPD
Top-level Entity Name	FPD
Family	MAX 10
Device	10M08DAF484C8G
Timing Models	Final
Total logic elements	1,916 / 8,064 (24 %)
Total registers	0
Total pins	111 / 250 (44 %)
Total virtual pins	0
Total memory bits	0 / 387,072 (0 %)
Embedded Multiplier 9-bit elements	0 / 48 (0 %)
Total PLLs	0 / 2 (0 %)
UFM blocks	0 / 1 (0 %)
ADC blocks	0 / 1 (0 %)

Figure 136 FPD Compilation Report

4.4.2 Floating Point Divider Testbench

A testbench is used to generate the stimulus and applies it to the implemented floating point divider and compare the results against our calculations based on the IEEE 754 floating point calculator online [45]. The design was synthesized using precision synthesis tools targeting the DE-1 SoC Max 10 FPGA machine family.

```

1  module FPD_tb;
2
3  // Inputs
4  reg [31:0] A;
5  reg [31:0] B;
6
7  // Outputs
8  wire [31:0] out;
9
10 wire Exception;
11 wire Overflow;
12 wire Underflow;
13
14 wire SnanA, QnanA, InfA, ZeroA, SubNA, NormA;
15 wire SnanB, QnanB, InfB, ZeroB, SubNB, NormB;
16
17 // Instantiate the Unit Under Test (UUT)
18 FPD fpuDivTB
19 (
20     .A(A),
21     .B(B),
22     .out(out),
23     .Exception(Exception),
24     .Overflow(Overflow),
25     .Underflow(Underflow),
26     .SnanA(SnanA),
27     .QnanA(QnanA),
28     .InfA(InfA),
29     .ZeroA(ZeroA),
30     .SubNA(SubNA),
31     .NormA(NormA),
32     .SnanB(SnanB),
33     .QnanB(QnanB),
34     .InfB(InfB),
35     .ZeroB(ZeroB),
36     .SubNB(SubNB),
37     .NormB(NormB)
38 );
39

```

Figure 137 FPD Testbench

5.4.3 Floating Point Divider Simulation Results

Case A:

A:



B:



R:



Simulation Results:

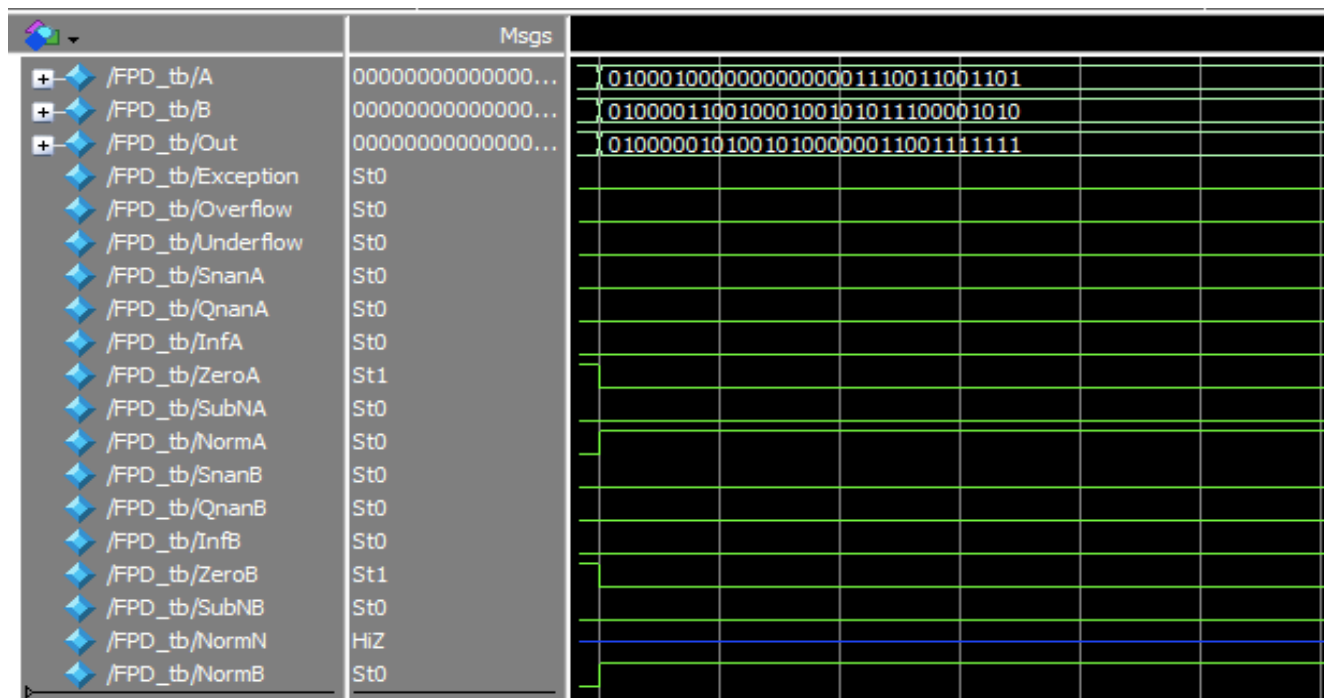
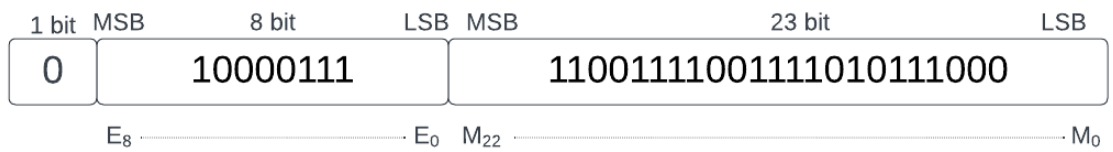


Figure 138 FPD Case A Result

Case B:

A:



B:



R:



Simulation Result:

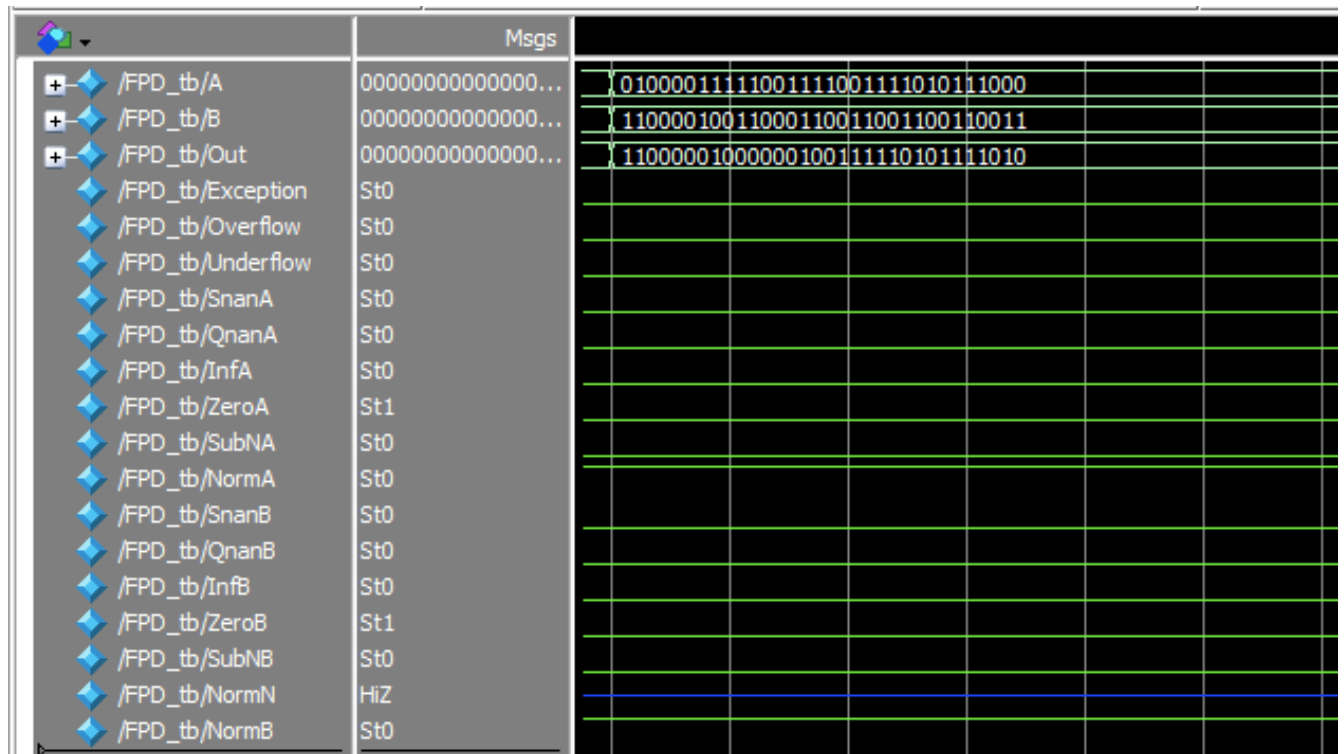


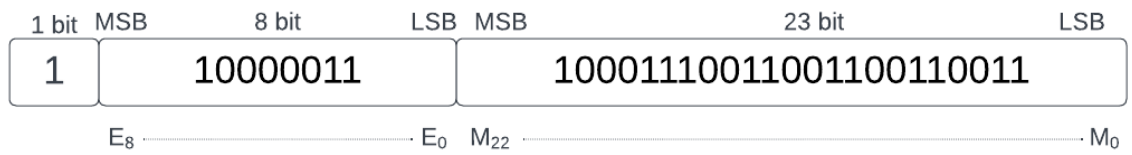
Figure 139 FPD Case B

Case C:

A:



B:



R:



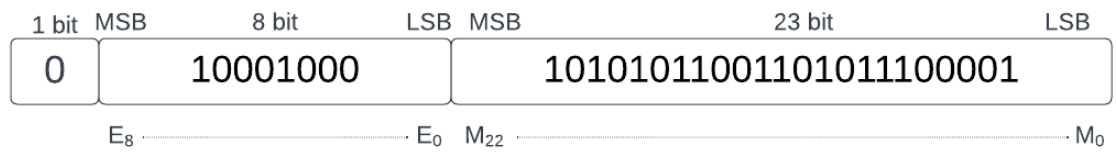
Simulation Result:

		Msgs					
+	/FPD_tb/A	00000000000000...	11000010	10001000	10011001	10011010	
+	/FPD_tb/B	00000000000000...	11000001	11000111	10011001	10011001	
+	/FPD_tb/Out	00000000000000...	01000001	00101111	10001100	11011010	
	/FPD_tb/Exception	St0					
	/FPD_tb/Overflow	St0					
	/FPD_tb/Underflow	St0					
	/FPD_tb/SnanA	St0					
	/FPD_tb/QnanA	St0					
	/FPD_tb/InfA	St0					
	/FPD_tb/ZeroA	St1					
	/FPD_tb/SubNA	St0					
	/FPD_tb/NormA	St0					
	/FPD_tb/SnanB	St0					
	/FPD_tb/QnanB	St0					
	/FPD_tb/InfB	St0					
	/FPD_tb/ZeroB	St1					
	/FPD_tb/SubNB	St0					
	/FPD_tb/NormN	HiZ					
	/FPD_tb/NormB	St0					

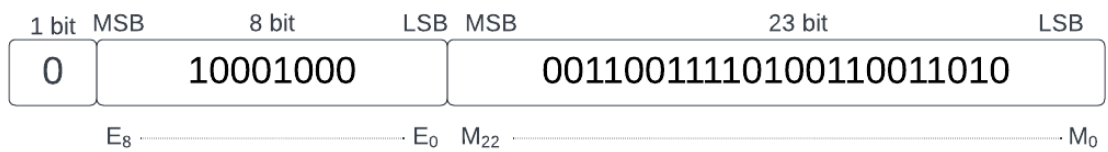
Figure 140 FPD Case C

Case D:

A:



B:



R:



Simulation Result:

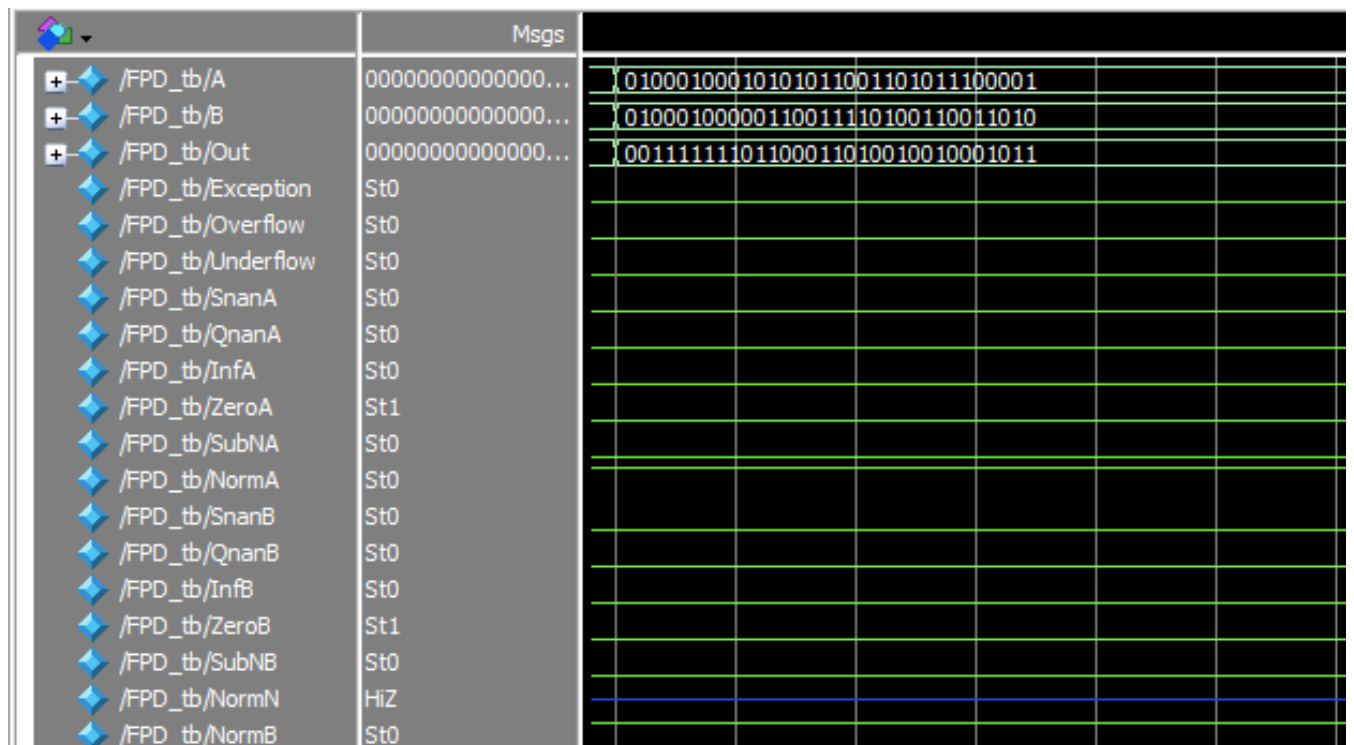


Figure 141 Case D

Case E:

A:



B:



R:



Simulation Result:

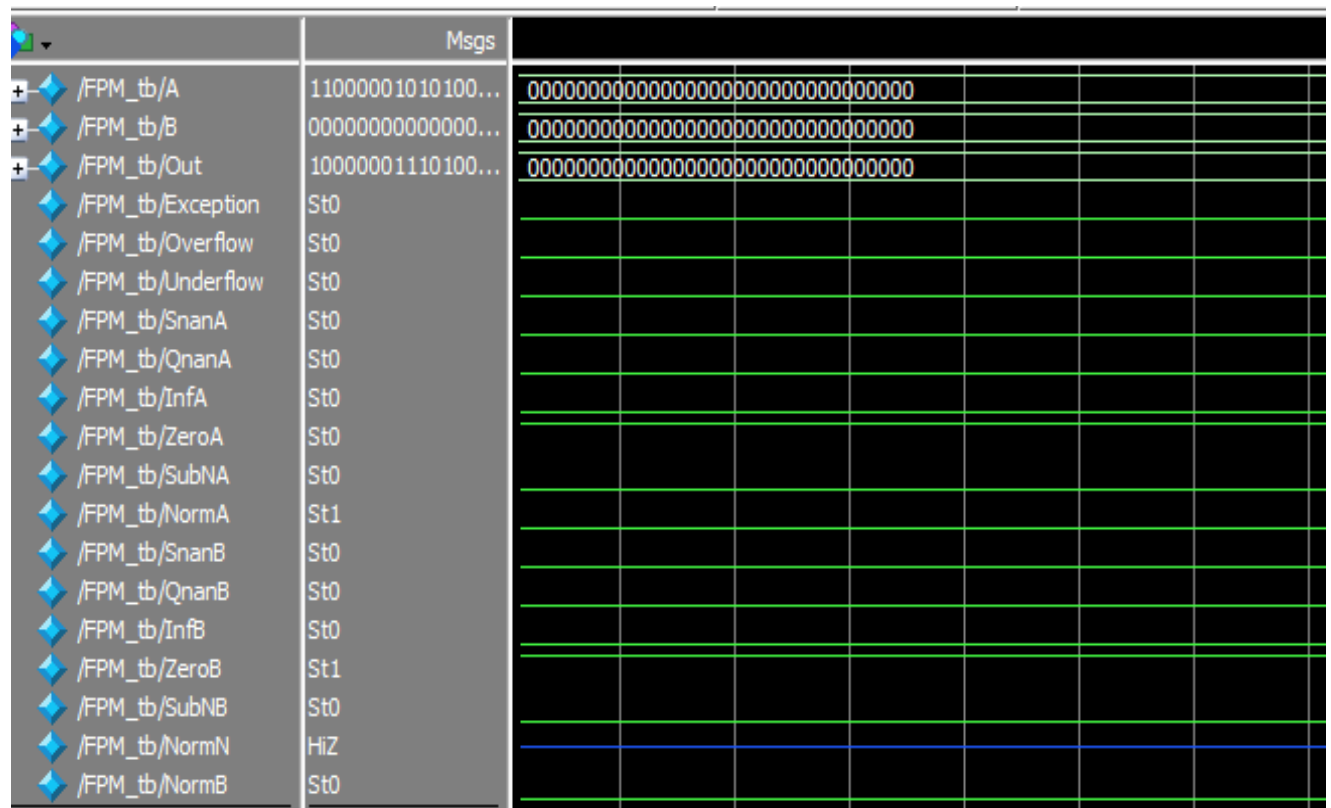


Figure 142 FPD Case E

5.5 Conclusion

This section of the thesis presented an implementation of a floating point divider that supports the IEEE 754-2008 binary interchange format. The divider implements this algorithm using a shift divider for faster computation and used various different modules to compute the final output.

Chapter 6: Floating Point Unit

In this chapter, we describe an efficient implementation of an IEEE 754 single precision floating point unit targeted for DE-1 Cyclone V FPGA. Verilog is used to implement a technology-independent pipelined design. The floating-point unit implementation instantiates all the previous operation modules and makes use of an opcode to display one of the four outputs. The Floating-Point Unit was verified by testbench simulations on ModelSim. In this chapter we will look at floating-point unit design, architecture, code design, and RTL diagram.

6.1 Floating Point Unit Block Diagram

The figure below, Figure 143, shows the block diagram that was used as an architecture for the final floating-point unit. The block diagram shows all the previous operation modules instantiated along with multiple multiplexers to choose input and output wires that are fed to input and outputs of the operation modules respectively.

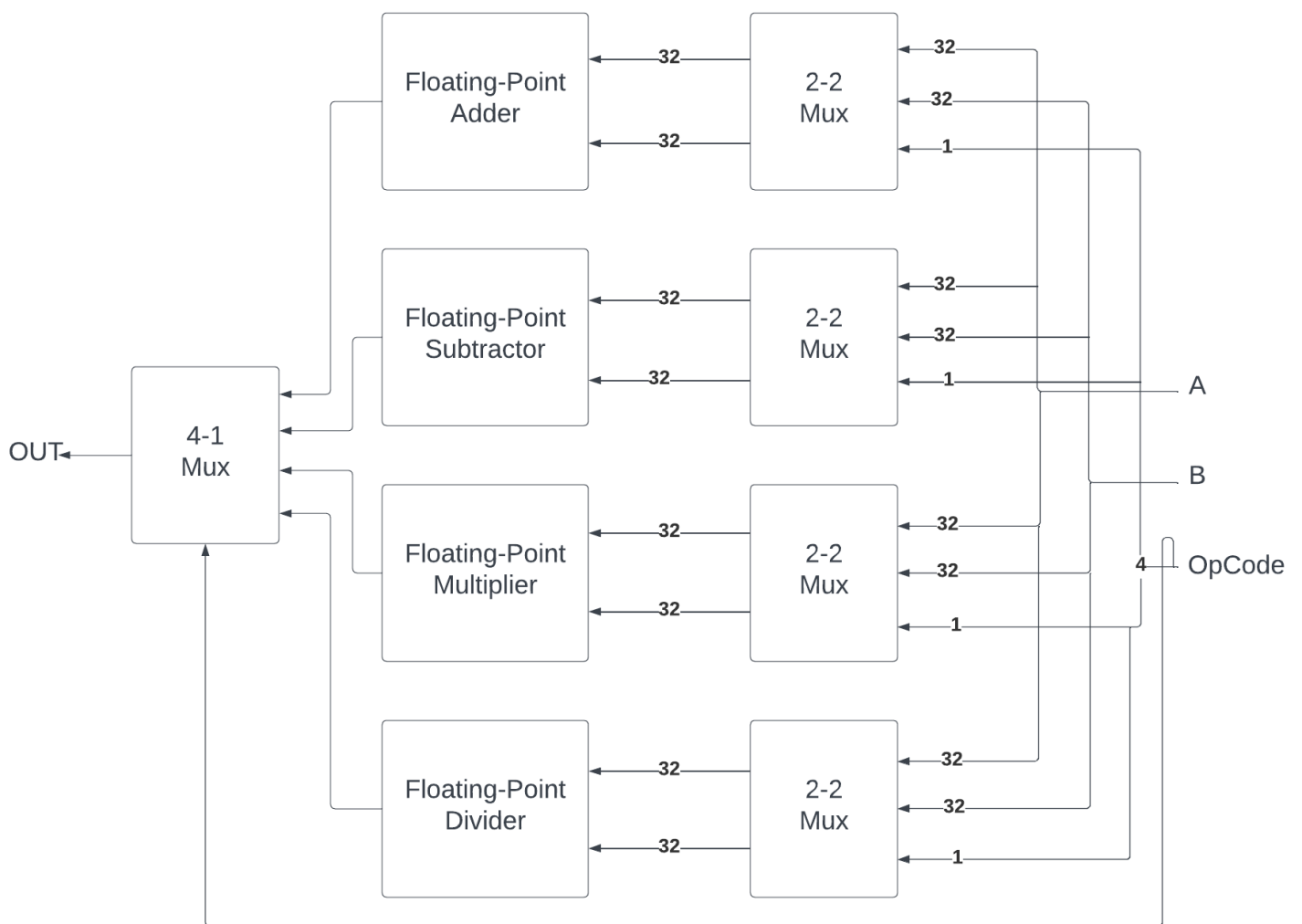


Figure 143 Floating Point Unit Block Diagram

6.2 Floating Point Unit Verilog Code

The figure below, Figure 144, shows a snippet from the Verilog code that was used to program and run the final floating-point unit. The Verilog shows all the previous operation modules instantiated in a top-level design along with multiple multiplexers that are connected to the operation modules using wires. The final multiplexer takes in use of an input called the Opcode to compute the final output of the floating-point unit.

```

1  module FPU
2  (
3      input [31:0] A,
4      input [31:0] B,
5      input [1:0] opcode,
6      output Overflow, Underflow, Exception,
7      output [31:0] out
8  );
9
10 wire [31:0] out_add, out_mul, out_div;
11 wire Exception_add, Exception_mul, Exception_div;
12 wire Overflow_add, Overflow_mul, Overflow_div;
13 wire Underflow_add, Underflow_mul, Underflow_div;
14 wire [31:0] temp_result,result1,result2,result3,result4,result5,result6;
15
16 FPUAdder add_sub
17 (
18     .A(A),
19     .B(B),
20     .Out(out_add),
21     .Exception(Exception_add),
22     .Overflow(Overflow_add),
23     .Underflow(Underflow_add)
24 );
25
26 FPM mult
27 (
28     .A(A),
29     .B(B),
30     .Out(out_mul),
31     .Exception(Exception_mul),
32     .Overflow(Overflow_mul),
33     .Underflow(Underflow_mul)
34 );
35
36 FPD div
37 (
38     .A(A),
39     .B(B),
40     .Out(out_div),
41     .Exception(Exception_div),
42     .Overflow(Overflow_div),
43     .Underflow(Underflow_div)
44 );
45

```

Figure 144 Floating Point Unit Verilog Code

6.3 Floating Point Unit RTL Diagram

The figure below, Figure 145, shows the RTL diagram that was generated from the Verilog code that was used to program and run the final floating-point unit and discussed in the previous section.

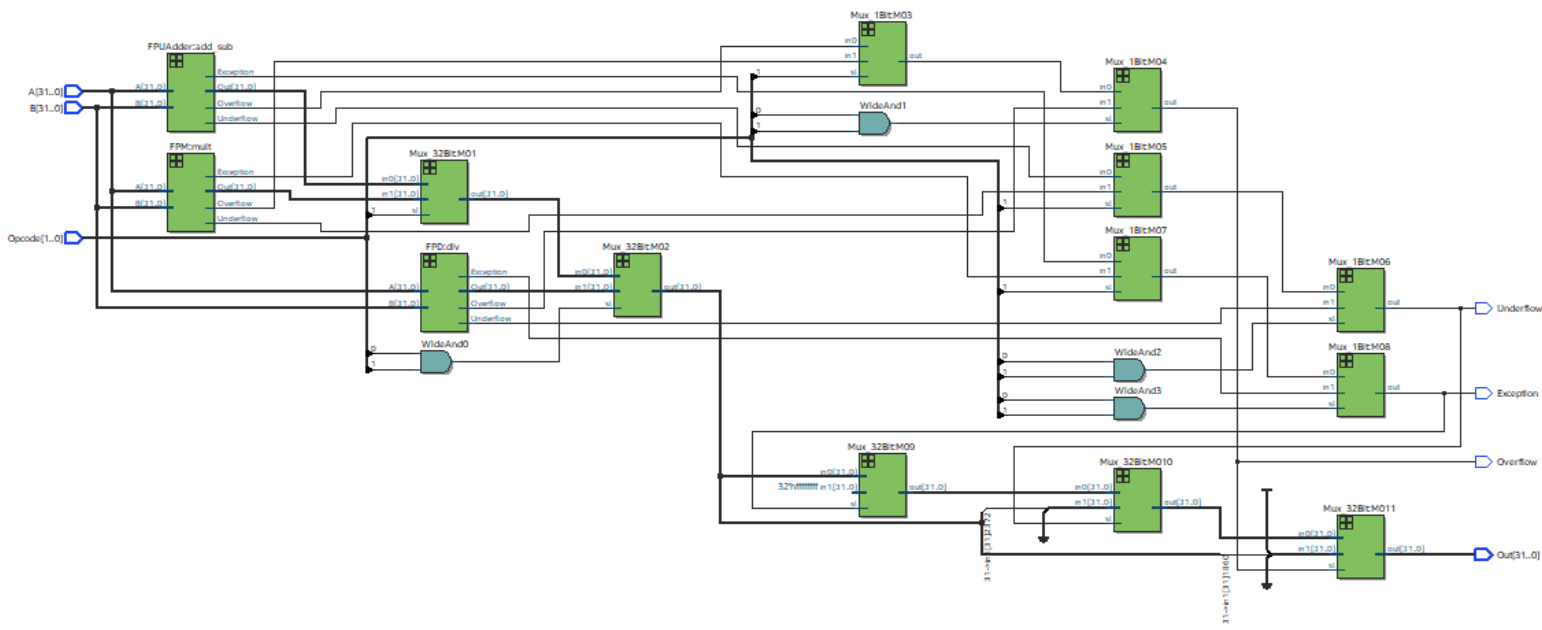


Figure 145 Floating Point Unit RTL Diagram

Chapter 7: Education Module

In this chapter of the thesis, we will shift our focus from designing and constructing the 32-bit floating point unit to creating various education modules. The purpose of this chapter is to provide a road map for a digital logic and design student and facilitate them to build their own floating point unit.

This chapter has been divided into ten different sections. Each section is aimed to act as a laboratory assignment with specific guidelines, and theory of the subject matter to build each module illustrated in the previous thesis chapters.

The first section of each lab, it starts the students with educating them regarding the purpose and outcome of each laboratory, the purpose section introduces the students to the lab assignment and gives them a brief reason as to why the module is being implemented. The next section, provides students with the necessary background needed to implement the module at hand. This section discusses modules implemented in previous labs as well as the computation or algorithm necessary to implement the lab they are working on. The next section, walks the student through the design requirement followed by the design verification. Finally, the lab assignments have a section that asks students important questions regarding their implementation of the design.

7.1 Lab 1

Name: _____ ID# _____	
Date Submitted: _____ Lab Section # _____	
CSE [xxxx] Digital Logic	Fall Semester 2024
Lab Number 1 – Modular Exponent Subtractor	
Perform [Month] [Date], [Year]	
This lab is performed on the DE10-Lite.	

CSE 2441**LABORATORY ASSIGNMENT 1****FALL 2024****VERILOG MODULAR EXPONENT SUBTRACTOR****(100 POINTS)****PURPOSE/OUTCOMES**

To give you experience writing Verilog modules and instantiating these modules to realize more complex designs. In this lab you will implement a modular exponent subtractor. You will construct a Ripple Carry Subtractor to subtract two exponents and compute the difference between the two. Then, you will construct a twos complement to sign magnitude convertor to find the absolute value of the difference. This absolute difference will then be used to perform floating point arithmetic in future labs. After completing this lab, you will have demonstrated an ability to design eight-bit ripple carry subtractor, eight-bit twos complement to sign magnitude convertor, to write Verilog models of adders and subtractors, to capture and verify your designs using Model-Sim on Quartus Prime, and to realize and test your designs on a DE10-Lite.

DESIGN REQUIREMENTS

In this lab you will construct a modular exponent subtractor unit that consists of three individual modules connected together to find the absolute difference between two 8-bits input. The top level block diagram is shown in Figure 1.

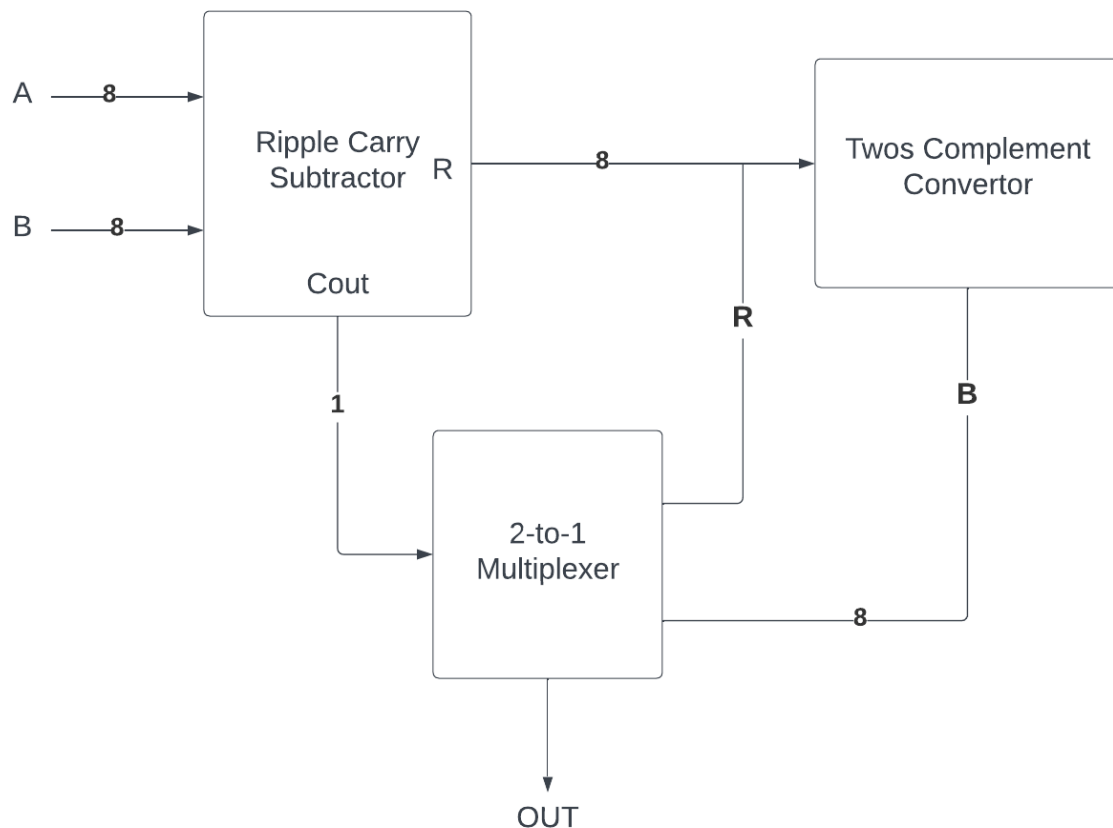


Figure 1 – Modular Exponent Subtractor

You will construct the module shown in Figure 1, in the following steps constructing each of the underlying module in each step of the process:

DESIGN REQUIREMENT

1. Constructing and Testing the Ripple Carry Subtractor

a) For the first module, the Ripple Carry Subtractor, start by writing a Verilog model of a full-adder using the circuit shown in Figure 2. Create an instantiation template for this module.

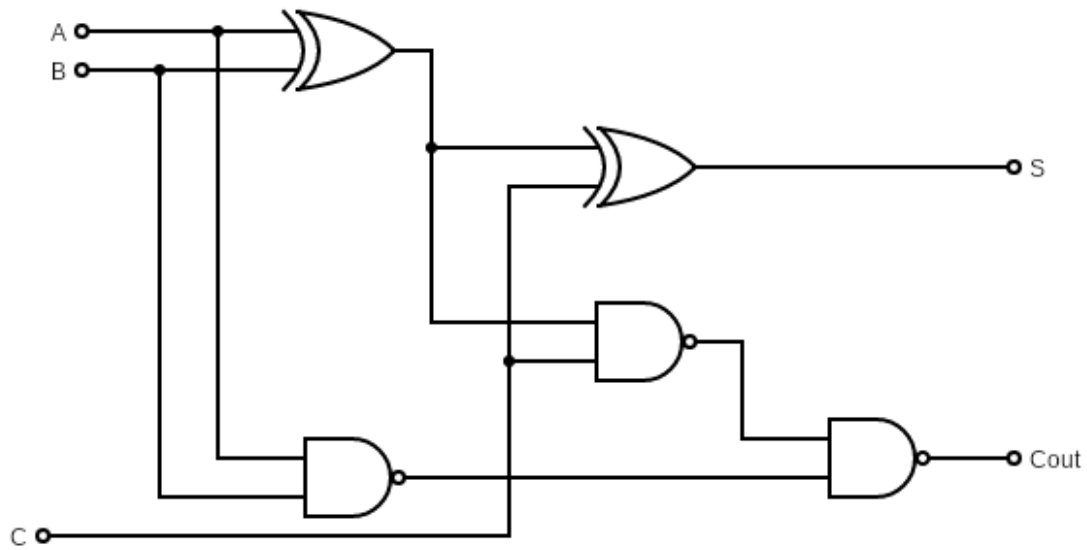


Figure 2 – Full Adder Circuit

b) Secondly, use the instantiation of full adder module created in step 'a' to form a ripple carry subtractor module in Verilog. For a four-bit ripple carry subtractor, four full adders are cascaded together passing the output of the first full adder to the input of the next full adder as shown in Figure 3 below.

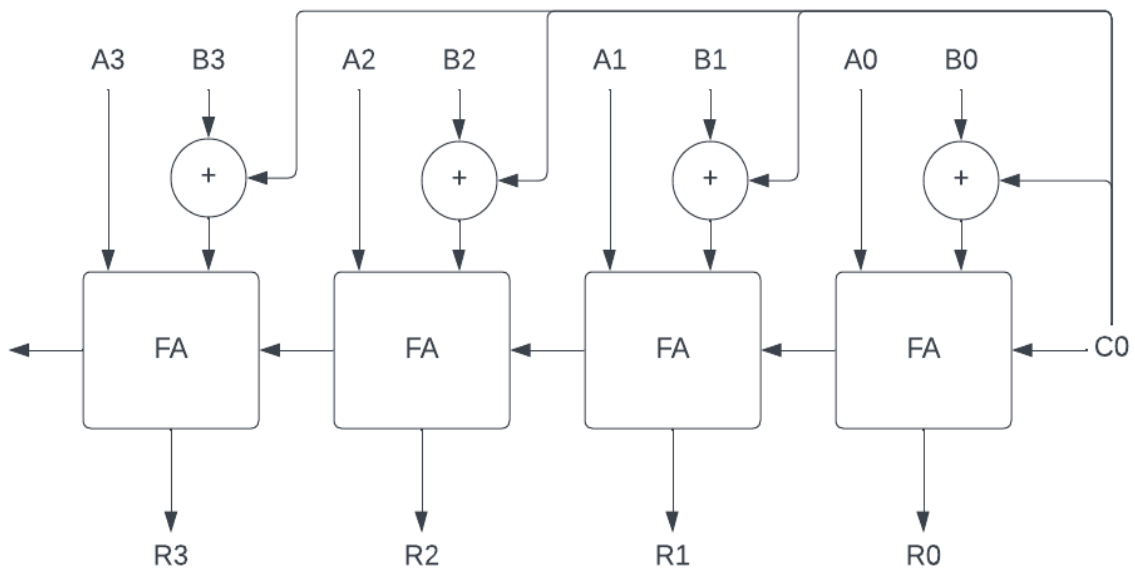


Figure 3 – Four-bit Ripple Carry Subtractor

c) Modify the block diagram shown in Figure 3 for 8-bits subtraction operation. Perform this by instantiating eight full adders in chain in a Verilog module. Remember to pass each bit of Input B through an XOR gate along with C0 which will be high for subtraction operation.

2. Constructing and Testing the Twos Complement to Sign Magnitude Convertor

a) For this second module, start by writing a Verilog model of a half-adder using the circuit shown in Figure 4. Create an instantiation template for this module.

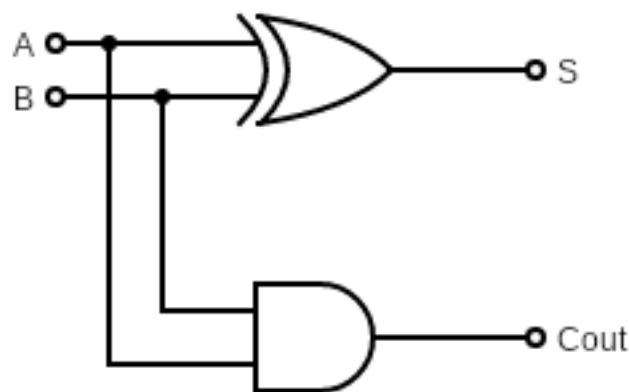


Figure 4 – Half Adder Circuit

b) Secondly, use the instantiation of half adder module created in step 'a' to form a twos complement convertor module in Verilog. For a four-bit twos complement convertor, four half adders are cascaded together as shown in Figure 5 below.

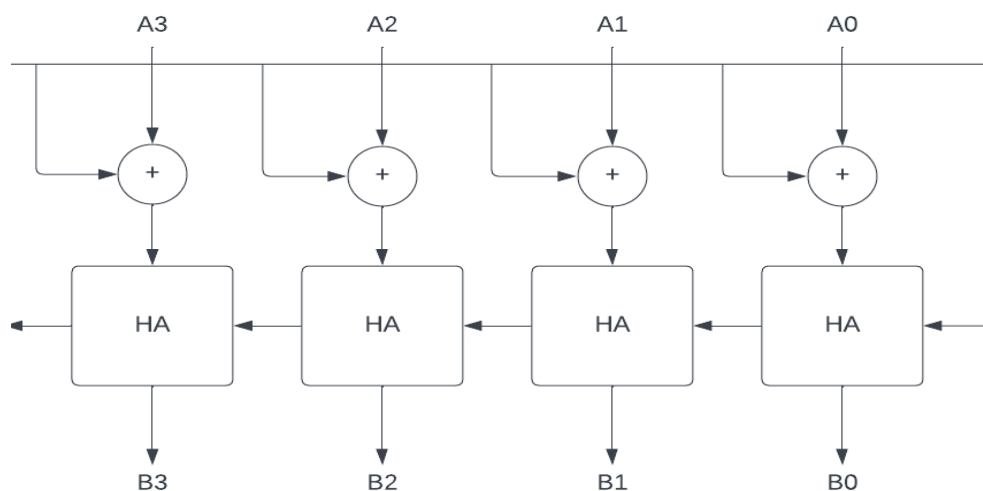


Figure 5 – Twos Complement Convertor

c) Modify the block diagram shown in Figure 4 for 8-bits conversion operation. Perform this by instantiating eight half adders in chain in a Verilog module. Remember to pass each bit of Input A through an XOR gate along with most significant bit of the input. The MSB of input A is also the second input of the first half adder.

3. Constructing Two-to-One Multiplexer

a) For the last element of this module, write a Verilog module for two-to-one multiplexer.

The multiplexer should select the A input as output if S input is high and B input as output if S input is low in value. Use block diagram in Figure 6 for reference.

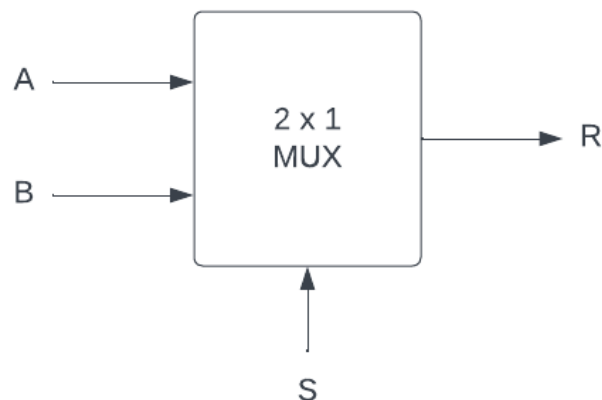


Figure 6 – Two-to-One Multiplexer

b) Create an instantiation template of this module for future use.

4. Constructing the Modular Subtractor Module

a) For the final step of the construction, use the block diagram shown in Figure 1 to connect all three modules constructed in steps above. Open a new project on Quartus and in a new Verilog code file, and declare inputs and outputs as needed. Finally, instantiate each of the three modules discussed above and connect them using wires.

DESIGN VERIFICATION

1. Simulate your design using waveforms to verify its correctness. Use the following values of A and B for your simulation inputs.

(a) 01010101 - 10101010

(b) 01111111 - 00000001

(c) 01111111 - 11111111

(d) 01100110 - 11011101

2. Include screen shots of your simulation waveform in your report.

3. Record the simulation results in the table below for your report.

A	B	$R = A - B $
01010101	10101010	
01111111	00000001	
01111111	11111111	
01100110	11011101	

DE10-Lite IMPLEMENTATION

1. Implement your design on the DE-10 Lite using the following inputs/outputs using pin assignments of your choice.

Inputs: A0, A1, A2, A3, A4, A5, A6, A7, B0, B1, B2, B3, B4, B5, B6, B7

Outputs: R0, R1, R2, R3, R4, R5, R6, R7

2. Include a table of your assignments in your report.

3. Program the DE10-Lite with your design.

CHECK YOUR UNDERSTANDING

1. Explain how the circuit in Figure 3 computes $R = A - B$, where $A = (A_3A_2A_1A_0)_2$, $B = (B_3B_2B_1B_0)_2$ and $D = (R_3R_2R_1R_0)_2$
2. Explain how the circuit in Figure 5 converts twos complement to signed magnitude.
3. Explain the role of multiplexer in figure 1, what does value of input S signify?
4. Instantiate the modular exponent subtractor module for future use.

7.2 Lab 2

Name: _____ ID# _____	
Date Submitted: _____ Lab Section # _____	
CSE [xxxx] Digital Logic	Fall Semester 2024
Lab Number 2 – Right & Left Barrel Shifter	
Perform [Month] [Date], [Year]	
This lab is performed on the DE10-Lite.	

CSE 2441

LABORATORY ASSIGNMENT 2

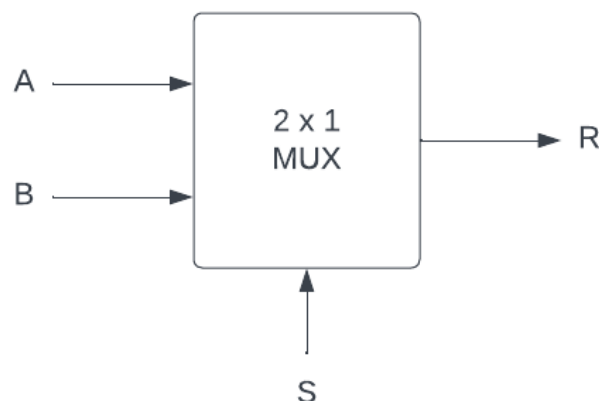
FALL 2024

RIGHT & LEFT BARREL SHIFTER**(100 POINTS)****PURPOSE/OUTCOMES**

To give you experience writing Verilog modules and instantiating these modules to realize more complex designs. In this lab you will implement a left and right barrel shifter module. A regular shift operation done in Verilog using the shift operator (<<,>>) uses a sequential circuit. The register based shift operation takes eight clock cycles to shift eight bits of data. However, a barrel shifter module uses a combinational circuit to shift eight bits of data by only using one clock cycle. The barrel shifter module you will implement in this lab will use several multiplexers for each level of data shift. After completing this lab, you will have demonstrated an ability to design 8-bit and a 24-bit left and right barrel shifter module, to write Verilog models of multiplexers, to capture and verify your designs using Model-Sim on Quartus Prime, and to realize and test your designs on a DE10-Lite.

BACKGROUND

In Lab 1, you designed, constructed, and tested the two-to-one multiplexer that was used to form the modular exponent subtractor module as shown in Figure 1.

**Figure 1 – Two-to-One Multiplexer**

DESIGN REQUIREMENT - 1

1. In the first step of the lab you will write the Verilog code for a right barrel shifter unit with 3-bit shift input and 8-bit data input. This module consists of twenty-four two-to-one multiplexers arranged in three levels. Each level signifies one additional bit of data shift. The block diagram for the right barrel shifter is shown in Figure 2.

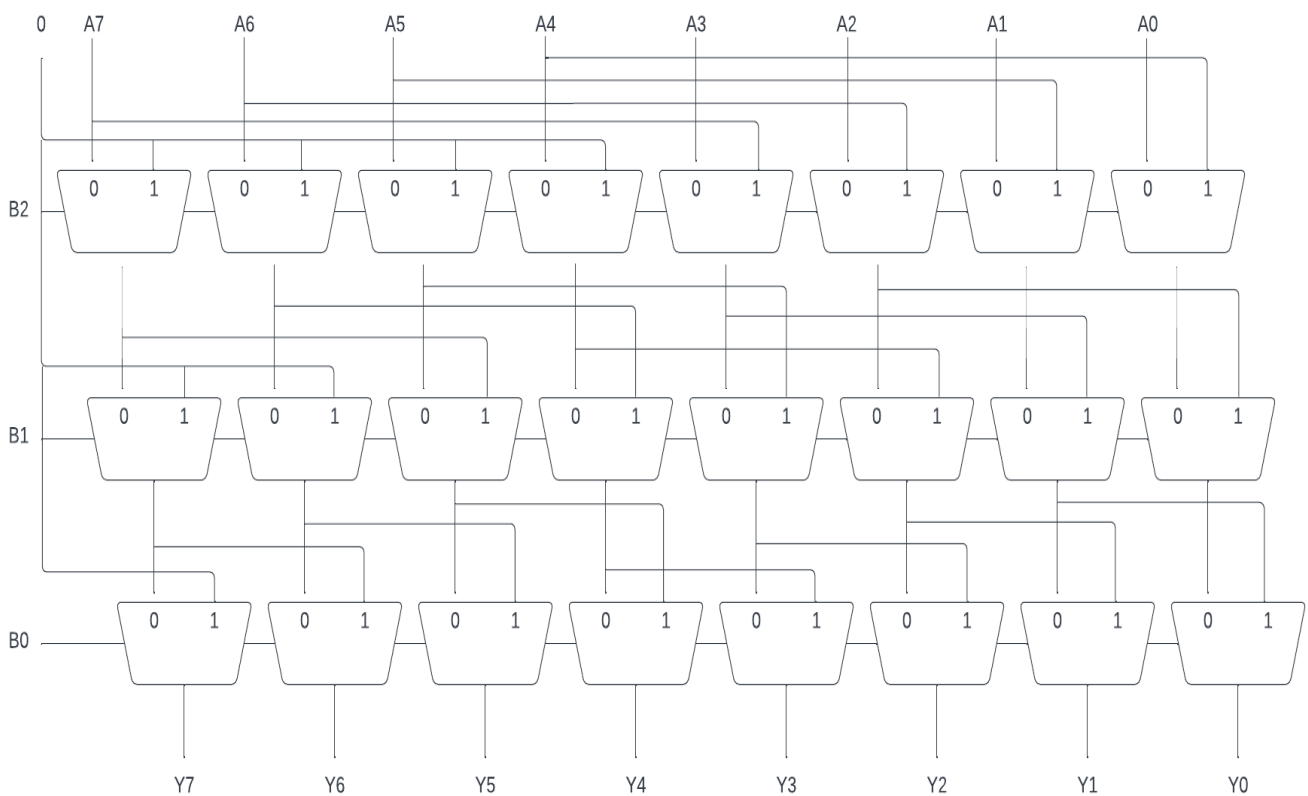


Figure 2 – 8-bit Right Barrel Shifter

2. A_0 through A_7 is the 8-bit data input to the barrel shifter module. Inputs B_0 to B_2 are the 3-bit shift input that defines how many bits does the input need to be shifted.

DESIGN VERIFICATION - 1

1. Simulate your design using waveforms to verify its correctness. Use the following values of A and B for your simulation inputs.

(a) A = 01010101, B = 001

(b) A = 01111111, B = 010

(c) A = 01111111, B = 011

(d) A = 01100110, B = 100

2. Include screen shots of your simulation waveform in your report.

3. Record the simulation results in a table for your report.

DE10-Lite IMPLEMENTATION - 1

1. Implement your design on the DE-10 Lite using the following inputs/outputs using pin assignments of your choice.

Inputs: A0, A1, A2, A3, A4, A5, A6, A7, A8, B0, B1, B2, B3

Outputs: R0, R1, R2, R3, R4, R5, R6, R7, R8

2. Include a table of your pin assignments in your report.

3. Program the DE10-Lite with your design.

DESIGN REQUIREMENT - 2

1. Take the block diagram for the right barrel shifter from Figure 2 and modify it to operate on a 24-bit data input and equip it to perform a 5-bit data shift on the input. Draw the block diagram for the next part of this lab.

2. Use the block diagram created in step 1 of this section and modify the Verilog code written in design requirement section 1 of this lab to operate on 24-bit data input for a 5-bit data shift operation.

DESIGN VERIFICATION - 2

1. Simulate your design using waveforms to verify its correctness. Use the following values of A and B for your simulation inputs.

(a) A = 0101010101010101010101, B = 00001

(b) A = 011111110111111101100110, B = 00010

(c) A = 011111110110011001100110, B = 01000

(d) A = 0110011001010101010101, B = 10100

2. Include screen shots of your simulation waveform in your report.

3. Record the simulation results in a table for your report.

DESIGN REQUIREMENT - 3

1. In the next section, you will write the Verilog code for a left barrel shifter unit with 3-bit shift input and 8-bit data input. This module consists of twenty-four two-to-one multiplexers arranged in three levels. Each level signifies one additional bit of data shift. The block diagram for the left barrel shifter is shown in Figure 3.

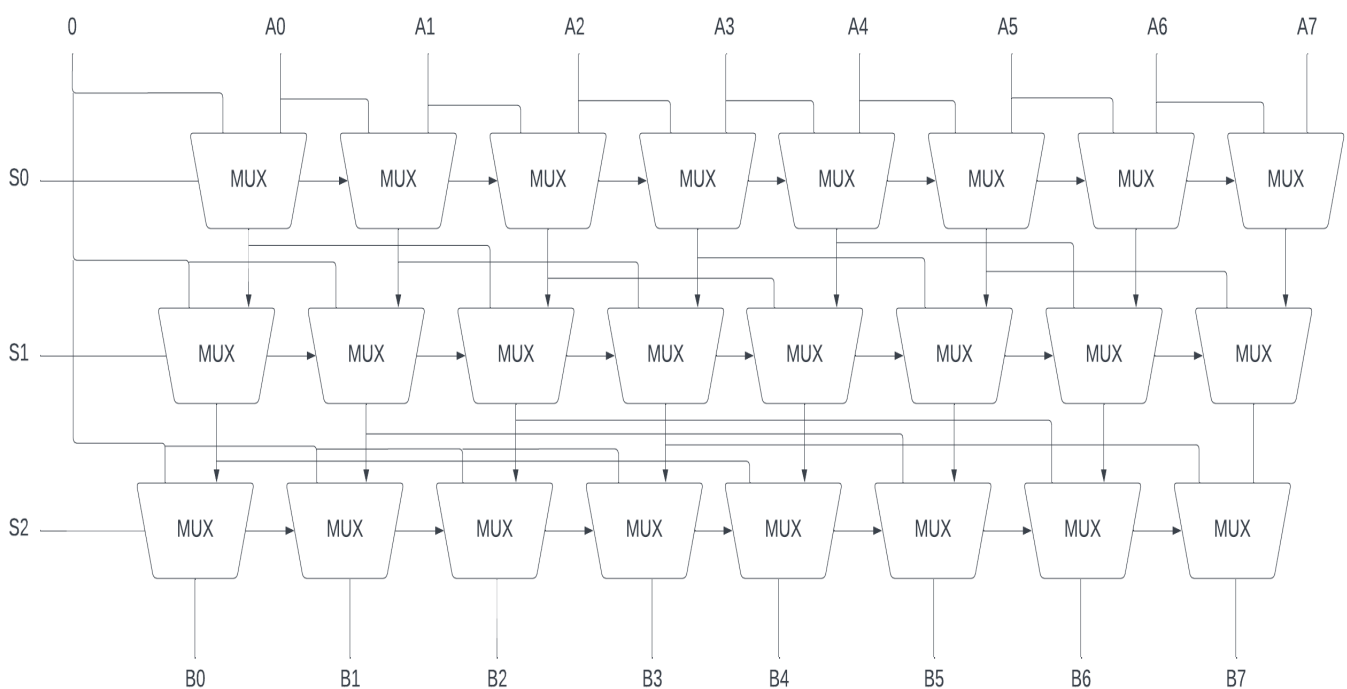


Figure 3 – 8-bit Left Barrel Shifter

DESIGN VERIFICATION - 3

1. Simulate your design using waveforms to verify its correctness. Use the following values of A and B for your simulation inputs.

(a) A = 01010101, B = 001

(b) A = 01111111, B = 010

(c) A = 01111111, B = 011

(d) A = 01100110, B = 100

2. Include screen shots of your simulation waveform in your report.

3. Record the simulation results in a table for your report.

DE10-Lite IMPLEMENTATION - 3

1. Implement your design on the DE-10 Lite using the following inputs/outputs using pin assignments of your choice.

Inputs: A0, A1, A2, A3, A4, A5, A6, A7, A8, B0, B1, B2, B3

Outputs: R0, R1, R2, R3, R4, R5, R6, R7, R8

2. Include a table of your pin assignments in your report.

3. Program the DE10-Lite with your design.

DESIGN REQUIREMENT - 4

1. Take the block diagram for the right barrel shifter from Figure 3 and modify it to operate on a 24-bit data input and equip it to perform a 5-bit data shift on the input. Draw the block diagram for the next part of this lab.

2. Use the block diagram created in step 1 of this section and modify the Verilog code written in section 3 to operate on 24-bit data input for a 5-bit data shift operation.

DESIGN VERIFICATION - 4

1. Simulate your design using waveforms to verify its correctness. Use the following values of A and B for your simulation inputs.

(a) A = 010101010101010101010101, B = 00001

(b) A = 011111110111111101100110, B = 00010

(c) A = 011111110110011001100110, B = 01000

(d) A = 011001100101010101010101, B = 10100

2. Include screen shots of your simulation waveform in your report.

3. Record the simulation results in a table for your report.

CHECK YOUR UNDERSTANDING

1. What is the advantage of using a barrel shifter instead of the Verilog shift operator

(<<,>>) which uses a register based shift module?

2. What do the various levels of multiplexers signify in the different designs?

3. What would be total number of multiplexers used for a 64-bit shifter design?

7.3 Lab 3

Name: _____ ID# _____	
Date Submitted: _____ Lab Section # _____	
CSE [xxxx] Digital Logic	Fall Semester 2024
Lab Number 3 – Controlled Incrementor/Decrementer	
Perform [Month] [Date], [Year]	
This lab is performed on the DE10-Lite.	

CSE 2441

LABORATORY ASSIGNMENT 3

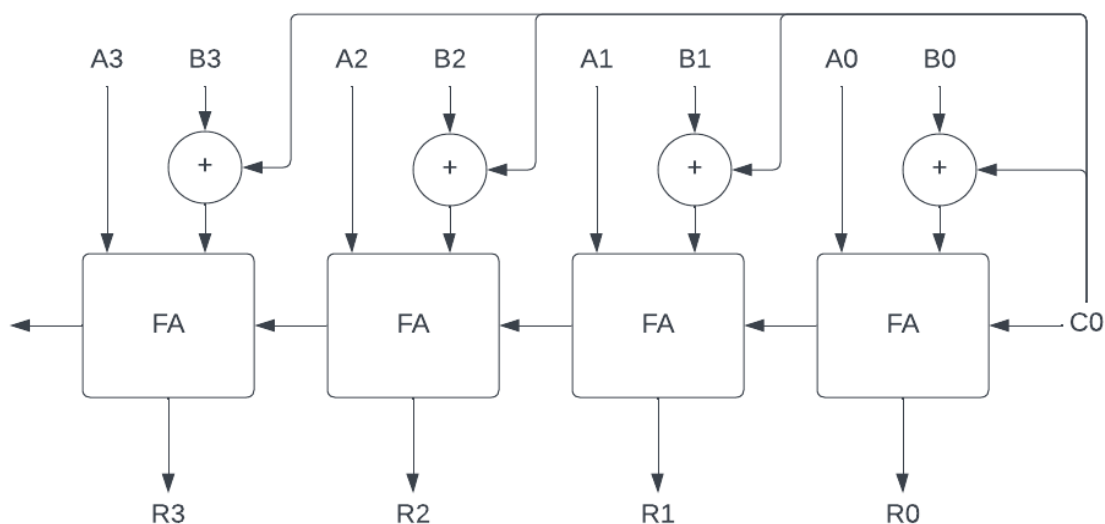
FALL 2024

CONTROLLED INCREMENTOR/DECREMENTER**(100 POINTS)****PURPOSE/OUTCOMES**

To give you experience writing Verilog modules and instantiating these modules to realize more complex designs. In this lab you will implement a modular controlled incrementor and decrementer module. The controlled incrementor module will take an 8-bit input and increment it based on a one bit input. The controlled decrement module will take an 8-bit input and decrement it based on a one bit input. This module makes use of the ripple carry subtractor with minor modification to accomplish this task. After completing this lab, you will have demonstrated an ability to design eight-bit ripple carry subtractor, to write Verilog models of adders and subtractors, to capture and verify your designs using Model-Sim on Quartus Prime, and to realize and test your designs on a DE10-Lite.

BACKGROUND

In Lab 1, you designed, constructed, and tested the 4-bit ripple carry adder/subtractor that was used to form the modular exponent subtractor module as shown in Figure 1.

**Figure 1 – Ripple Carry Adder/Subtractor**

Additionally in previous Labs you also designed, constructed, and tested a half-adder as shown in Figure 2 below.

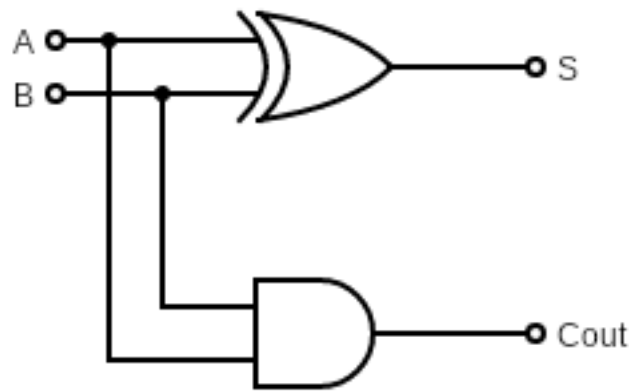


Figure 2 – Half-Adder

DESIGN REQUIREMENT - 1

1. In the first step of the lab you will write the Verilog code for the 4-bit controlled incrementor module by making certain changes to the ripple carry adder that was used in lab 1. The modifications to be made are shown in Figure 3 below.

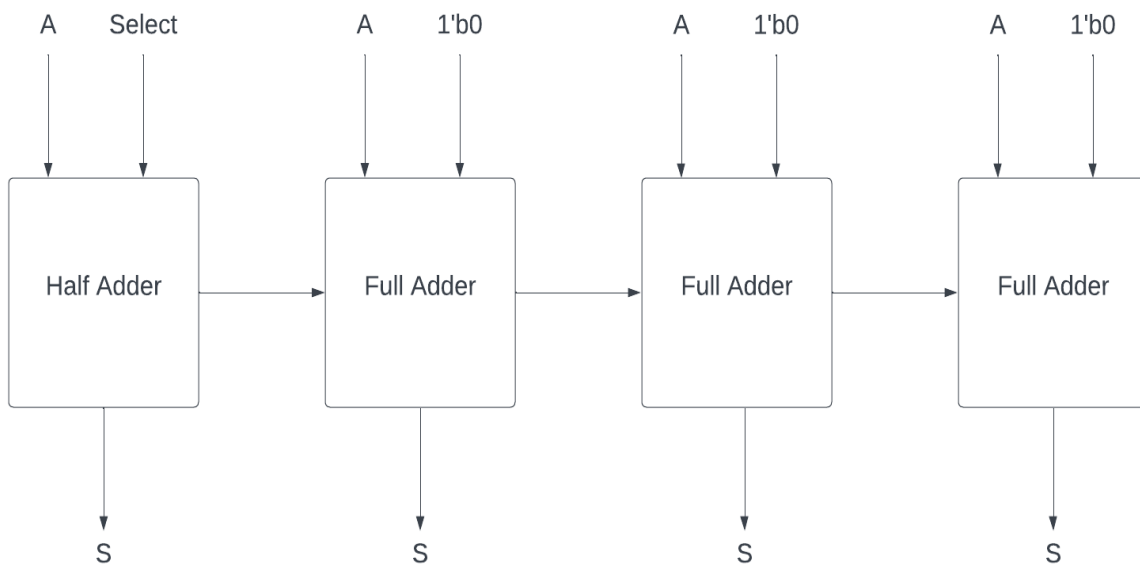


Figure 3 – Controlled Incrementor

2. Figure 3 shows 4-bit input that needs to be incremented labelled as A, 1-bit input labelled as Select that allows the increment to happen when high.

3. For step 3, modify the Verilog code written for 4-bit controlled incrementor in step 1.

Write the Verilog code for 8-bit controlled incrementor with four additional full-adders.

4. Create an instantiation template of this module for future use.

DESIGN VERIFICATION - 1

1. Simulate your design using waveforms to verify its correctness. Use the following values of A and Select for your simulation inputs.

(a) A = 01010101, Select = 1

(b) A = 01111111, Select = 1

(c) A = 01111111, Select = 0

(d) A = 01100110, Select = 1

2. Include screen shots of your simulation waveform in your report.

3. Record the simulation results in a table for your report.

DE10-Lite IMPLEMENTATION - 1

1. Implement your design on the DE-10 Lite using the following inputs/outputs using pin assignments of your choice.

Inputs: A0, A1, A2, A3, A4, A5, A6, A7, Select

Outputs: S0, S1, S2, S3, S4, S5, S6, S7

2. Include a table of your assignments in your report.

3. Program the DE10-Lite with your design.

DESIGN REQUIREMENT - 2

1. In the next section of the lab you will write the Verilog code for the 4-bit controlled decremter module. The modifications to be made are shown in Figure 4 below.

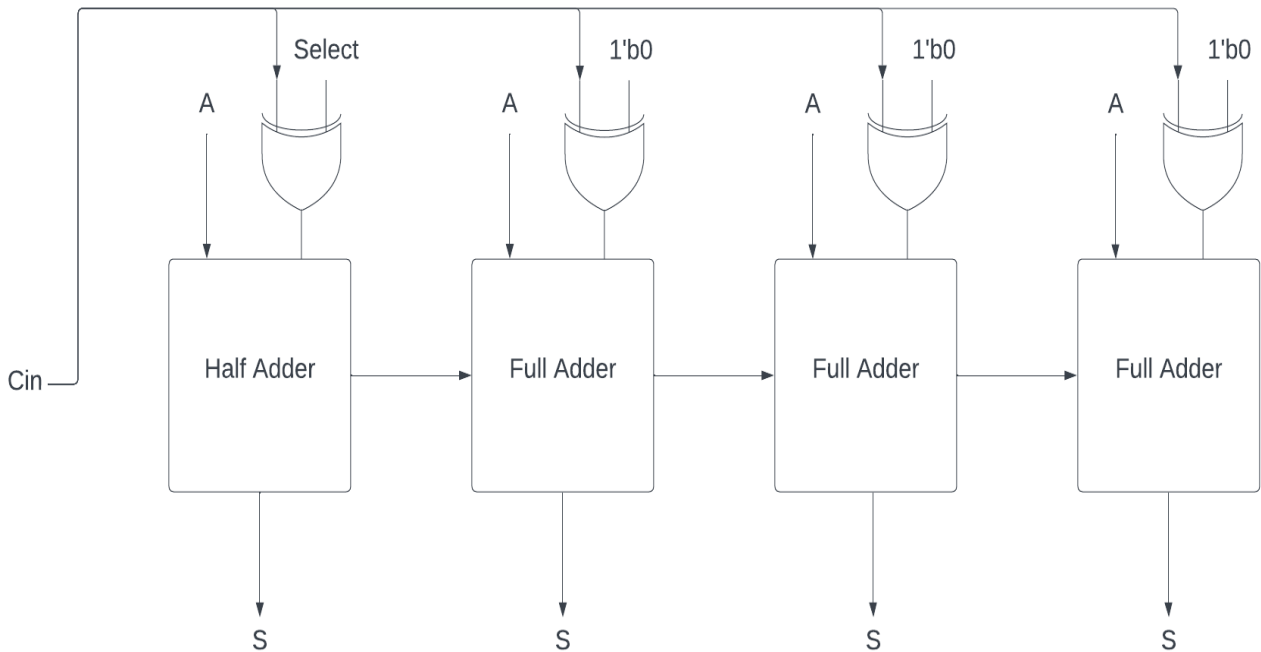


Figure 4 – Controlled Decrementer

2. In this step, modify the Verilog code written for 4-bit controlled decremter in step 1.

Write the Verilog code for 8-bit controlled decremter with four additional full-adders.

3. Create an instantiation template of this module for future use.

DESIGN VERIFICATION - 2

1. Simulate your design using waveforms to verify its correctness. Use the following values of A and Select for your simulation inputs.

(a) A = 01010101, Select = 1

(b) A = 01111111, Select = 1

(c) A = 01111111, Select = 0

(d) A = 01100110, Select = 1

2. Include screen shots of your simulation waveform in your report.
3. Record the simulation results in a table for your report.

DE10-Lite IMPLEMENTATION - 2

1. Implement your design on the DE-10 Lite using the following inputs/outputs using pin assignments of your choice.

Inputs: A0, A1, A2, A3, A4, A5, A6, A7, Select, Cin

Outputs: S0, S1, S2, S3, S4, S5, S6, S7

2. Include a table of your assignments in your report.
3. Program the DE10-Lite with your design.

7.4 Lab 4

Name: _____ ID# _____	
Date Submitted: _____ Lab Section # _____	
CSE [xxxx] Digital Logic	Fall Semester 2024
Lab Number 4 – 24-bit Carry Look Ahead Adder	
Perform [Month] [Date], [Year]	
This lab is performed on the DE10-Lite.	

CSE 3441

LABORATORY ASSIGNMENT 4

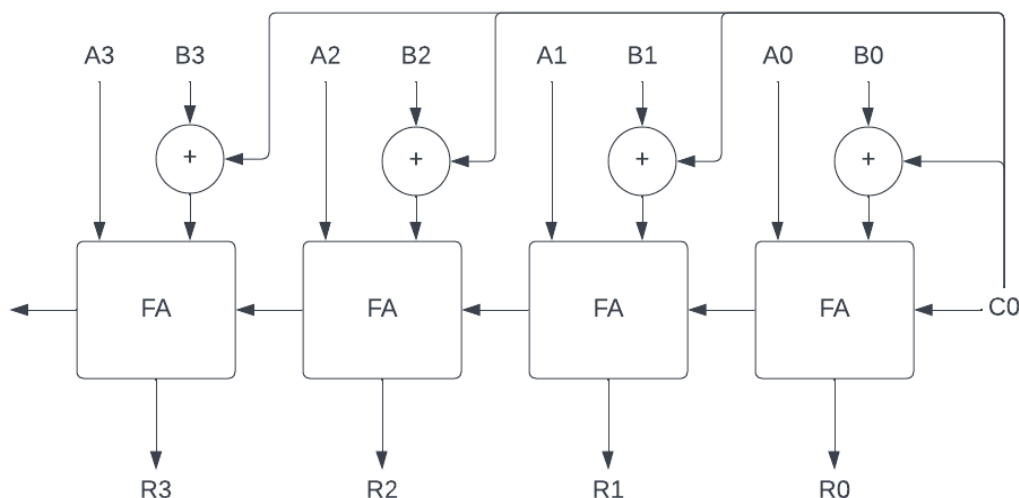
FALL 2024

24-bit Carry Look Ahead Adder**(100 POINTS)****PURPOSE/OUTCOMES**

To give you experience writing Verilog modules and instantiating these modules to realize more complex designs. In this lab you will implement a 24-bit Carry Look Ahead Adder using Verilog hardware description language. The carry lookahead adder is an upgraded version of the ripple carry adder that you have used so far for the addition and subtraction operation. In this lab you will also learn how to design a parameterize model for your Verilog code. The parameterize model will allow you to scale the module up or down by the number of bits that is required in the design. After completing this lab, you will have demonstrated an ability to write Verilog models of adders and subtractors, to capture and verify.

BACKGROUND

Ripple-carry adders (RCA), designed in previous labs, as shown in Figure 1 are constructed using a simple circuit of cascaded full adders. But the RCA is slow due to the necessity for carries to propagate the full length of the chain in the worst case scenario.

**Figure 1 – Ripple Carry Adder/Subtractor**

A carry-lookahead adder (CLA) overcomes the propagation problem by generating all carries at once with two-level logic but at the expense of a much more complex design. The basic elements of the CLA is illustrated below.

Compute generate variable:

You compute the generate variable by putting the two input bits through an AND gate and the output is the generate variable as shown in Figure 2.

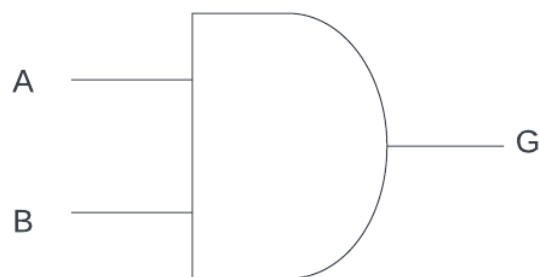


Figure 2 – Compute Generate

Compute propagate variable:

You compute the propagate variable by putting the two input bits through an XOR gate and the output is the propagate variable as shown in Figure 3.

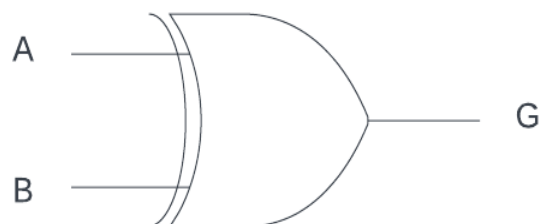


Figure 3 – Compute Propagate

- Compute Carry out.

The circuit schematic shows the way to compute the Carry output that is being computed by the carry look ahead logic block.

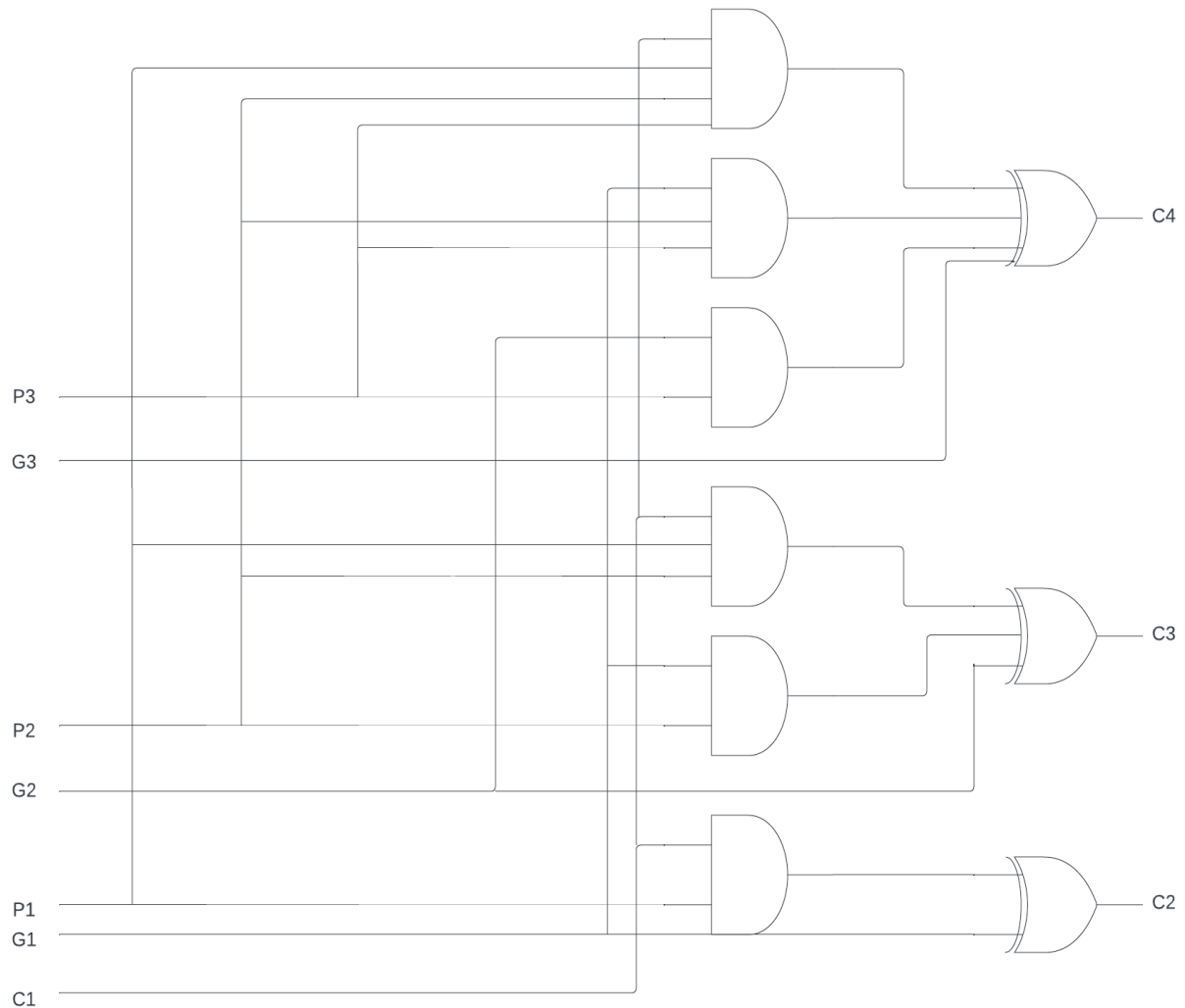


Figure 4 – Compute Carry

The p and g outputs drive carry-generate logic that simultaneously produces the carries for all stages of the CLA.

DESIGN REQUIREMENT – 1

1. In the first part of lab, you will start by constructing a 4-bit carry look ahead adder as shown in the block diagram shown below in Figure 5. The first level of carry look ahead

adder is made of full adders which you designed in previous labs. For the second level of logic refer to the background section for help in constructing that level.

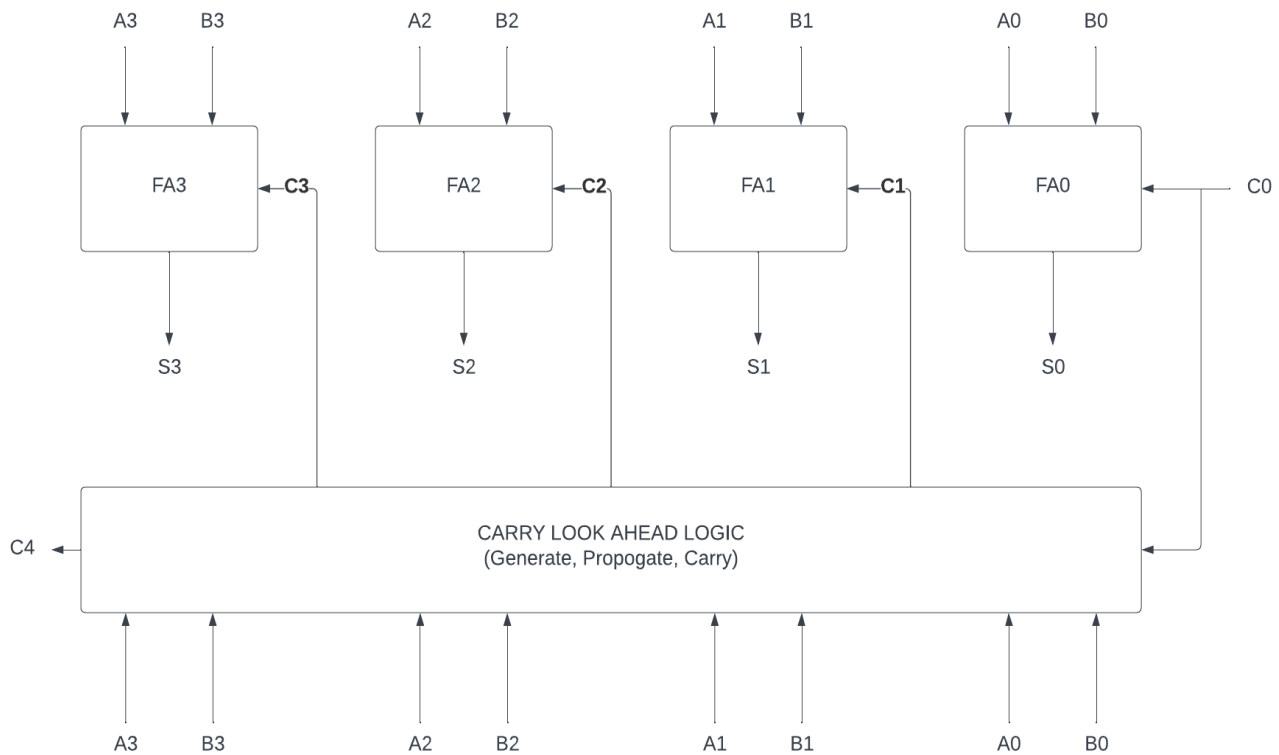


Figure 5 – Carry Look Ahead Adder Block Diagram

Helpful Tip: Use two for loops to construct the two levels of logic. Use first for loop for the cascaded chain of full adders, and the second for loop for computing the carry look ahead logic. Parametrize the for loop to change number of input and output bits.

DESIGN VERIFICATION - 1

1. Simulate your design using waveforms to verify its correctness. Use the following values of A and B for your simulation inputs.

(a) $0101 + 1010$

(b) $0111 + 0001$

(c) $1111 + 1111$

(d) $0110 + 1100$

2. Include screen shots of your simulation waveform in your report.
3. Record the simulation results in the table below for your report.

A	B	R = A + B
0101	1010	
0111	0001	
1111	1111	
0110	1100	

DE10-Lite IMPLEMENTATION

1. Implement your design on the DE-10 Lite using the following inputs/outputs using pin assignments of your choice.

Inputs: A0, A1, A2, A3, B0, B1, B2, B3, C0

Outputs: R0, R1, R2, R3, C4

2. Include a table of your assignments in your report.
3. Program the DE10-Lite with your design.

DESIGN REQUIREMENT – 2

1. Construct a 24-bit carry look ahead adder using the design and block diagram illustrated in the first design requirement section. You should only be changing the iteration value in both your for loops to switch from 4-bits to 24-bits of inputs and outputs.
2. Parameterize your carry look ahead adder module. After this step you should only need to change the variable value at top of design to change the number of bits.
3. Instantiate the 24-bit module for future use.

DESIGN VERIFICATION - 2

1. Simulate your design using waveforms to verify its correctness. Use the following values of A and B for your simulation inputs.

(a) 0101010101010101010101010101 + 101010101010101010101010

(b) 011111110111111101100110 + 000000010101010101010101

(c) 011111110110011001100110 + 11111111010101011010101

(d) 011001101100110100110000 + 011001100110011010101011

2. Include screen shots of your simulation waveform in your report.

3. Record the simulation results in the table below for your report.

A	B	R = A + B
a.1	a.2	
b.1	b.2	
c.1	c.2	
d.1	d.2	

TESTBENCH VERIFICATION

1. Write a test bench for this 24-bit carry look ahead adder module.

2. Simulate the test bench using ModelSim on Quartus.

3. Screenshot your output from the tcl console.

4. Screenshot your ModelSim simulation results.

CHECK YOUR UNDERSTANDING

1. What's the speed up in operating speed achieved by using CLA instead of RCA?

2. What's the increase in complexity, in terms of logic elements, required for CLA compared to RCA?

7.5 Lab 5

Name: _____ ID# _____	
Date Submitted: _____ Lab Section # _____	
CSE [xxxx] Digital Logic	Fall Semester 2024
Lab Number 5 – Floating Point Adder	
Perform [Month] [Date], [Year]	
This lab is performed on the DE10-Lite.	

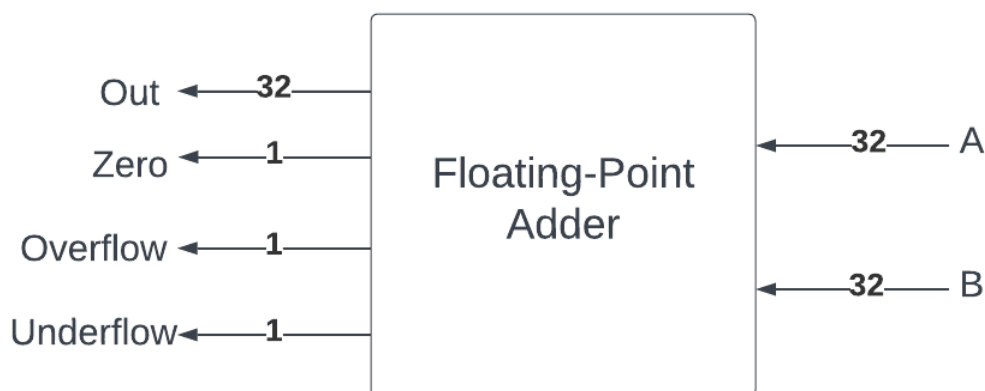
CSE 3441

LABORATORY ASSIGNMENT 5

FALL 2024

Floating Point Adder**(100 POINTS)****PURPOSE/OUTCOMES**

Your purpose in this lab is to design a 32-bit single precision floating point adder that can perform 32-bit floating point addition and that produces exception, underflow, overflow, zero, and the final output. See Figure 1 for the input/output diagram of the floating point adder. The adder component of your floating point adder should use group carry lookahead architecture implemented in the previous lab. You will code your design in System Verilog, simulate to verify its correctness, and test its functionality using testbenches and ModelSim on Quartus. After completing this lab, you will have demonstrated an ability to design a floating point arithmetic unit, to write Verilog models of the floating point adder, to capture and verify your designs using Model-Sim on Quartus Prime.

**Figure 1 – Floating Point Adder**

BACKGROUND

In this lab you will use the instantiations created in all the previous labs from Lab 1 to Lab 5.

The modules you need for this lab are:

1. Lab 1: 8-bit Modular Exponent Subtractor
2. Lab 1: 24-bit Two-to-One Multiplexer
3. Lab 1: 8-bit Two-to-One Multiplexer
4. Lab 2: 24-bit Barrel Right Shifter
5. Lab 3: 8-bit Controlled Incrementor
6. Lab 4: 24-bit Carry Look Ahead Adder

You will use the instantiations for the above mentioned module and connect them together using wires in a top level type module design.

DESIGN REQUIREMENT

1. Write a System Verilog model for a 32-bit single precision floating point adder. In addition to the floating point addition, provide functionality for overflow, underflow, zero, & exception outputs. Use the architecture or block diagram shown in Figure 2 to understand how the modules are interconnected.

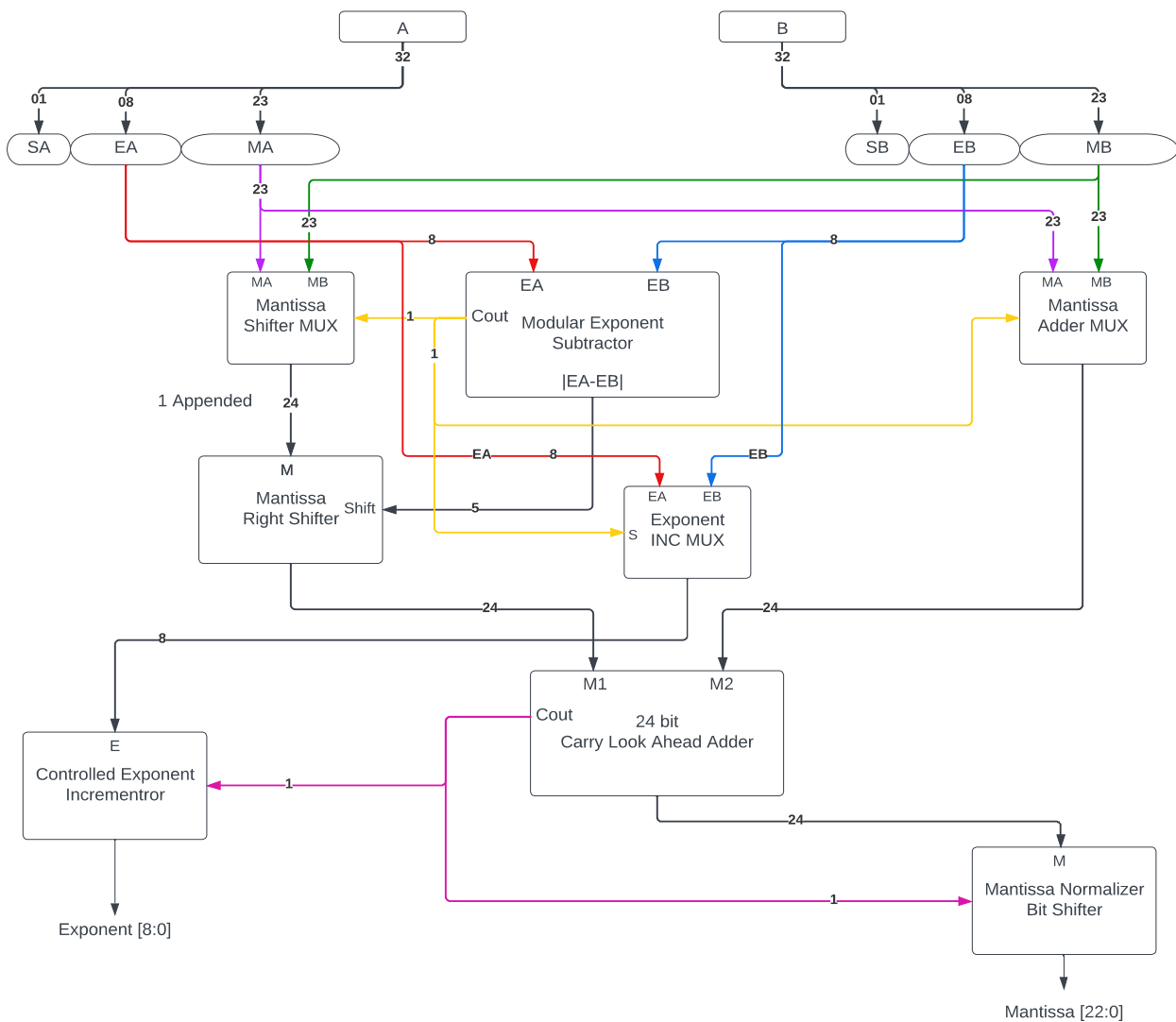


Figure 2 – Floating Point Adder Architecture

2. Add logic to compute the final sign bit output depending on the two different input values and the sign bit of each input value.
3. Add logic to compute the various error flags for this module like overflow, underflow etc.
4. Create an instantiation template for this module for future use.

DESIGN VERIFICATION

1. Simulate your design using waveforms to verify its correctness. Use the following values of A and B for your simulation inputs.

(a) $A = 32'b01000001001101100000000000000001 //11.375$

$B = 32'b01000000101100100000010000011011 //5.56300$

(b) $A = 32'b01000010011011111110101110000101 //59.979$

$B = 32'b01000000110100000000000000000000 //6.5$

(c) $A = 32'b01000100011110100010000000000000 //1000.5$

$B = 32'b01000100011101010110100111011011 //981.654$

(d) $A = 32'b01000100000010010111111100101011 //549.987$

$B = 32'b01000000101100100000010000011001 //5.563$

2. Include screen shots of your simulation waveform in your report.

3. Record the simulation results in the table below for your report.

A	B	R = A + B	Underflow	Overflow	Zero
11.375	5.56300				
59.979	6.5				
1000.5	981.654				
549.987	5.563				

TESTBENCH VERIFICATION

1. Write a test bench for this 32-bit floating point adder module using test cases above.
2. Simulate the test bench using ModelSim on Quartus.
3. Screenshot your output from the tcl console.
4. Screenshot your ModelSim simulation results.

CHECK YOUR UNDERSTANDING

1. Why is the mantissa passed through a mantissa right shifter module, how do you decide which mantissa goes into the mantissa right shifter module?
2. Why is the modular subtractor used to determine the shift value for the right shifter unit?
3. Why is the exponent incremented before the final output, what case would the exponent not be incremented in?
4. Why do we use the mantissa normalizer unit, what case would we not normalize it?

7.6 Lab 6

Name: _____ ID# _____	
Date Submitted: _____ Lab Section # _____	
CSE [xxxx] Digital Logic	Fall Semester 2024
Lab Number 6 – Floating Point Subtractor	
Perform [Month] [Date], [Year]	
This lab is performed on the DE10-Lite.	

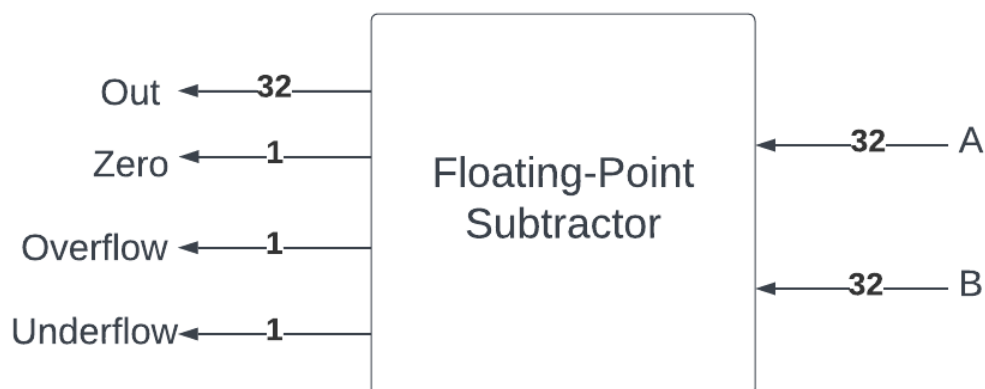
CSE 3441

LABORATORY ASSIGNMENT 6

FALL 2024

Floating Point Subtractor**(100 POINTS)****PURPOSE/OUTCOMES**

Your purpose in this lab is to design a 32-bit single precision floating point subtractor that can perform 32-bit floating point addition and that produces exception, underflow, overflow, zero, and the final output. See Figure 1 for the input/output diagram of the floating point subtractor. The subtractor component of your floating point subtractor should use ripple carry subtractor implemented in previous labs. You will code your design in System Verilog, simulate to verify its correctness, and test its functionality using testbenches and ModelSim on Quartus. After completing this lab, you will have demonstrated an ability to design a floating point arithmetic unit, to write Verilog models of the floating point subtractor, to capture and verify your designs using Model-Sim on Quartus Prime.

**Figure 1 – Floating Point Subtractor**

BACKGROUND

In this lab you will use the instantiations created in all the previous labs from Lab 1 to Lab 5.

The modules you need for this lab are:

1. Lab 1: 8-bit Modular Exponent Subtractor
2. Lab 1: 24-bit Two-to-One Multiplexer
3. Lab 1: 8-bit Two-to-One Multiplexer
4. Lab 1: 24-bit Ripple Carry Subtractor
5. Lab 2: 24-bit Barrel Right Shifter
6. Lab 2: 24-bit Barrel Left Shifter
7. Lab 3: 8-bit Controlled Decrementer

You will use the instantiations for the above mentioned module and connect them together using wires in a top level type module design.

DESIGN REQUIREMENT

1. Write a System Verilog model for a 32-bit single precision floating point subtractor. In addition to the floating point subtraction, provide functionality for overflow, underflow, zero, & exception outputs. Use the architecture or block diagram shown in Figure 2 to understand how the modules are interconnected.

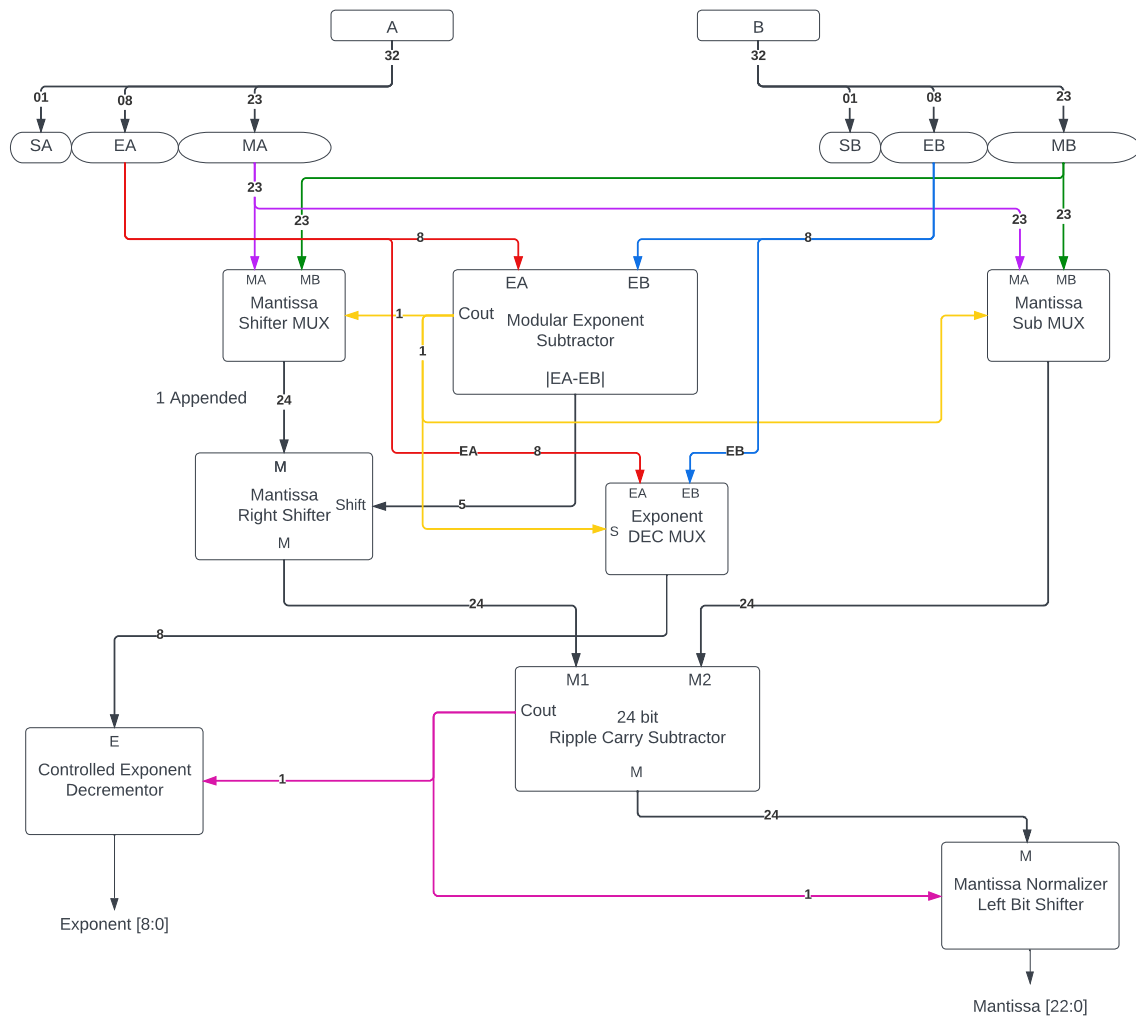


Figure 2 – Floating Point Subtractor Architecture

2. Add logic to compute the final sign bit output depending on the two different input values and the sign bit of each input value.
3. Add logic to compute the various error flags for this module like overflow, underflow etc.
4. Create an instantiation template for this module for future use.

DESIGN VERIFICATION

1. Simulate your design using waveforms to verify its correctness. Use the following values of A and B for your simulation inputs.

(a) $A = 32'b01000010010011010000000000000000 //51.25$

$B = 32'b01000001011101001100110011001101 //15.3$

(b) $A = 32'b0100001001101111110101110000101 //59.979$

$B = 32'b01000000110100000000000000000000 //6.5$

(c) $A = 32'b01000011011110101000110011001101 //250.55$

$B = 32'b01000010010011010000000000000000 //51.25$

(d) $A = 32'b01000100000010010111111100101011 //549.987$

$B = 32'b01000000101100100000010000011001 //5.563$

2. Include screen shots of your simulation waveform in your report.

3. Record the simulation results in the table below for your report.

A	B	R = A - B	Underflow	Overflow	Zero
51.25	15.3				
59.979	6.5				
250.55	51.25				
549.987	5.563				

TESTBENCH VERIFICATION

1. Write a test bench for this 32-bit floating point subtractor module using test cases above.
2. Simulate the test bench using ModelSim on Quartus.
3. Screenshot your output from the tcl console.
4. Screenshot your ModelSim simulation results.

CHECK YOUR UNDERSTANDING

1. Explain all the major differences between the floating point adder and the floating point subtractor architecture.

7.7 Lab 7

Name: _____ ID# _____	
Date Submitted: _____ Lab Section # _____	
CSE [xxxx] Digital Logic	Fall Semester 2024
Lab Number 7 – 32-bit Wallace Tree Multiplier	
Perform [Month] [Date], [Year]	
This lab is performed on the DE10-Lite.	

CSE 3441

LABORATORY ASSIGNMENT 4

FALL 2024

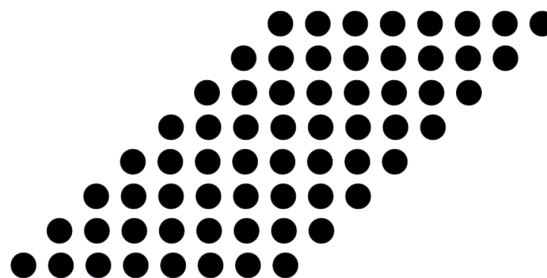
32-bit Wallace Tree Multiplier**(100 POINTS)****PURPOSE/OUTCOMES**

To give you experience writing Verilog modules and instantiating these modules to realize more complex designs. In this lab you will implement a 32-bit Wallace tree multiplier using Verilog hardware description language. The advantage of using Wallace tree multiplier is its faster speed. The Wallace multiplier has $O(\log n)$ reduction layers, but each layer has only $O(1)$ propagation delay. After completing this lab, you will have demonstrated an ability to write Verilog model for the multiplier, half adder, full adder, to capture and verify your designs using Model-Sim on Quartus Prime.

BACKGROUND

Wallace tree multiplier is a multiplication algorithm that uses a tree structure to add partial products to obtain the product and carry two numbers. Wallace Tree Multiplier is a multiplier that works in parallel by making use of the Wallace tree algorithm. This algorithm allows for a fast and efficient multiplication of two integers.

Step 1: Partial product obtained after multiplication is taken at the first stage. The data is taken with 3 wires and added using adders and the carry of each stage is added with next two data in the same stage. Refer to Figure 1.

**Figure 1 – Step 1**

Step 2: Partial products reduced to two layers of full adders with same procedure.

Stage 0:

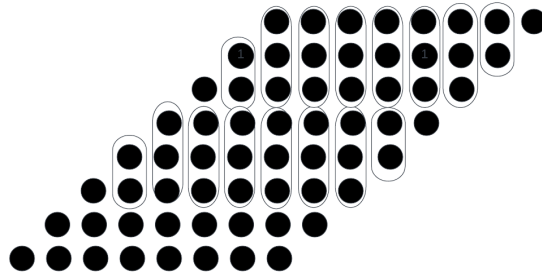


Figure 2 – Step 2 Stage 0

Stage 1:

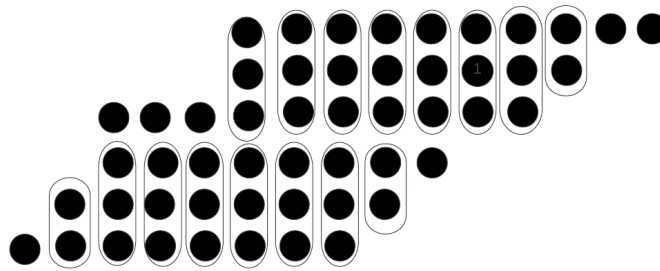


Figure 3 – Step 2 Stage 1

Stage 2:

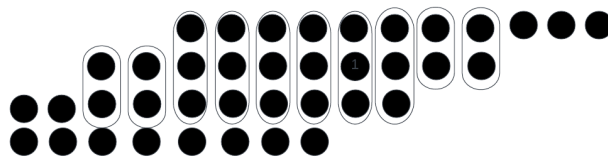


Figure 4 – Step 2 Stage 2

Stage 3:

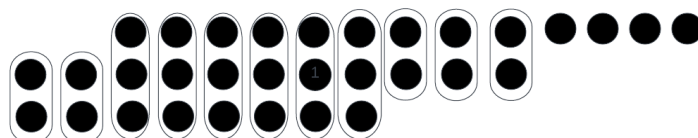


Figure 5 – Step 2 Stage 3

Step 3: Use Ripple carry adder or Carry look ahead adder to compute final addition

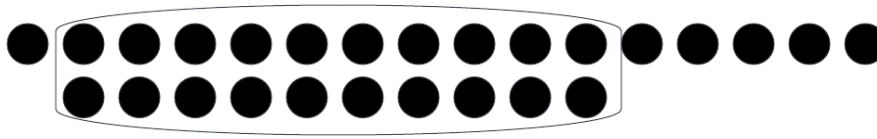


Figure 5 – Step 3

Figure 6 shows the flow diagram for a Wallace tree multiplier.

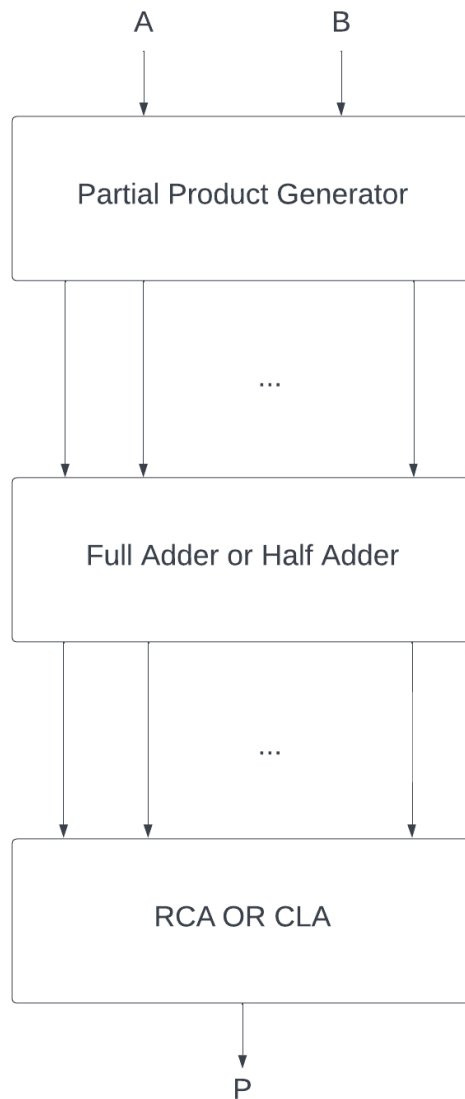


Figure 6 – Data Flow

DESIGN REQUIREMENT

1. Write a System Verilog model for a 8-bit output Wallace tree multiplier. Use the architecture or block diagram shown in Figure 7 to write the Verilog module.

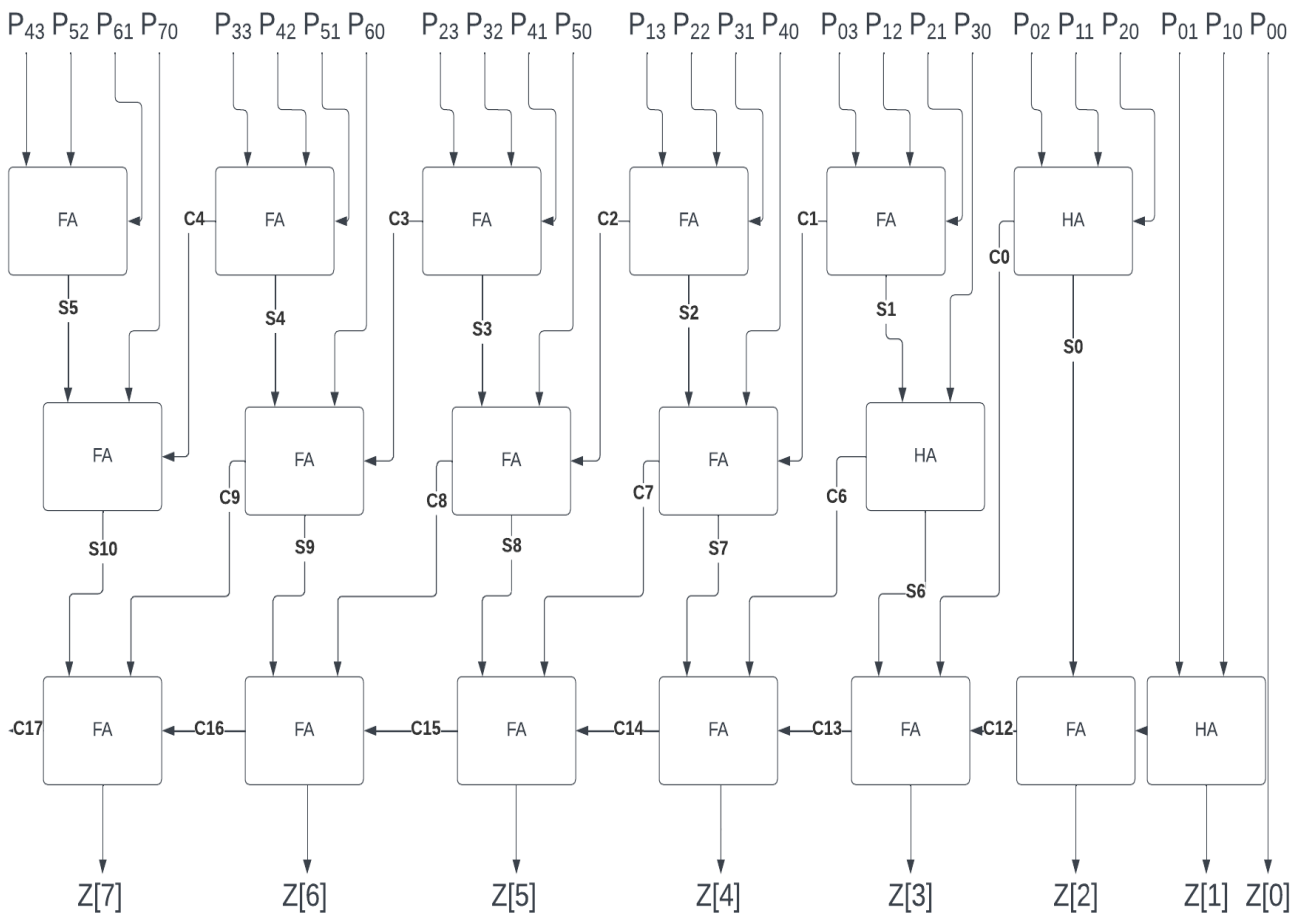


Figure 7 – 4-bit Wallace Tree Multiplier

2. Modify your Verilog module to work with two 8-bit inputs and produce a 16-bit output.
 3. Instantiate your 8-bit Wallace multiplier to produce a two 16-bit input Wallace multiplier.
- Use the RTL diagram in Figure 9 to create this module.

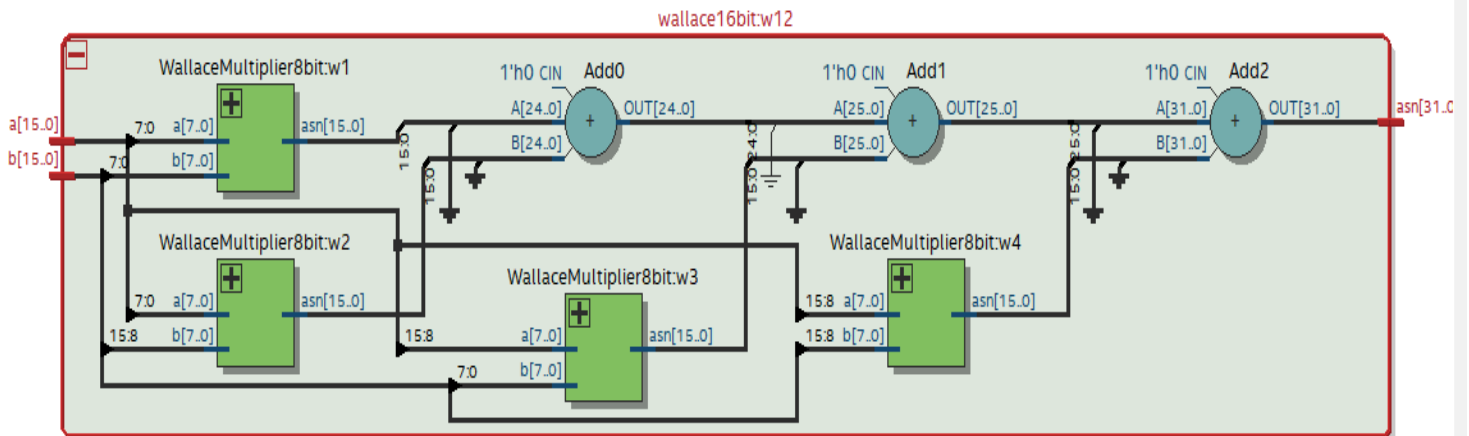


Figure 9 – 16-bit Wallace Multiplier

4. Instantiate your 16-bit Wallace multiplier to produce a two 32-bit input Wallace multiplier. Use the RTL diagram in Figure 10 to create this module.

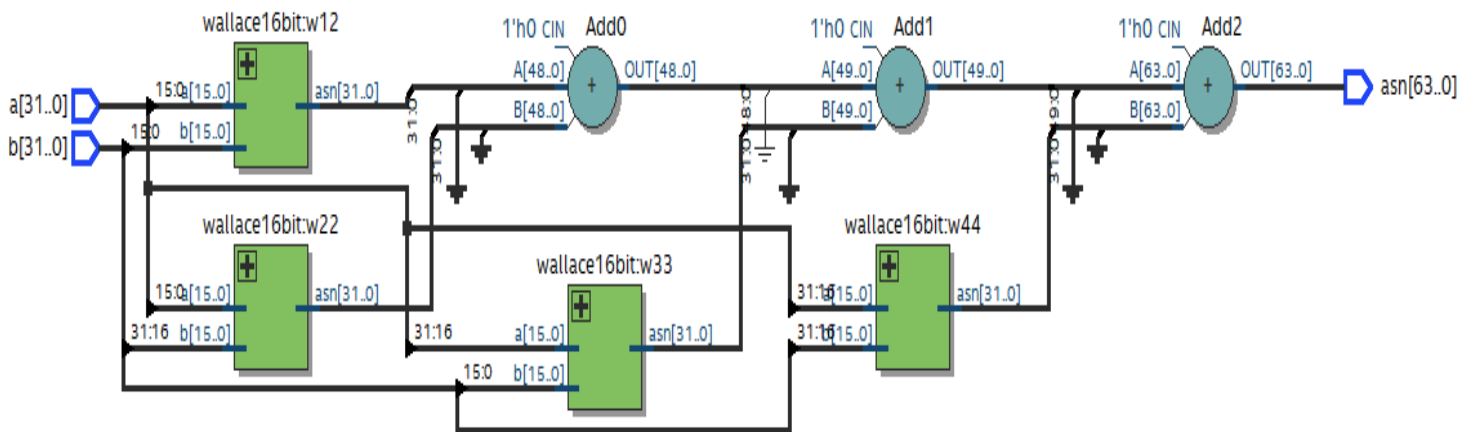


Figure 10 – 32-bit Wallace Multiplier

DESIGN VERIFICATION

1. Simulate your design using waveforms to verify its correctness. Use the following values of A and B for your simulation inputs.

(a) $A = 32'b01000001001101100000000000000001 //11.375$

$B = 32'b01000000101100100000010000011011 //5.56300$

(b) $A = 32'b01000010011011111110101110000101 //59.979$

$B = 32'b01000000110100000000000000000000 //6.5$

(c) $A = 32'b01000100011110100010000000000000 //1000.5$

$B = 32'b01000100011101010110100111011011 //981.654$

(d) $A = 32'b01000100000010010111111100101011 //549.987$

$B = 32'b01000000101100100000010000011001 //5.563$

2. Include screen shots of your simulation waveform in your report.

3. Record the simulation results in the table below for your report.

A	B	R = A * B
11.375	5.56300	
59.979	6.5	
1000.5	981.654	
549.987	5.563	

TESTBENCH VERIFICATION

1. Write a test bench for this 32-bit Wallace tree multiplier module using test cases above.
2. Simulate the test bench using ModelSim on Quartus.
3. Screenshot your output from the tcl console.
4. Screenshot your ModelSim simulation results.

CHECK YOUR UNDERSTANDING

1. What is the one disadvantage of using Wallace tree multiplier over conventional multipliers?

7.8 Lab 8

Name: _____ ID# _____	
Date Submitted: _____ Lab Section # _____	
CSE [xxxx] Digital Logic	Fall Semester 2024
Lab Number 8 – Floating Point Multiplier	
Perform [Month] [Date], [Year]	
This lab is performed on the DE10-Lite.	

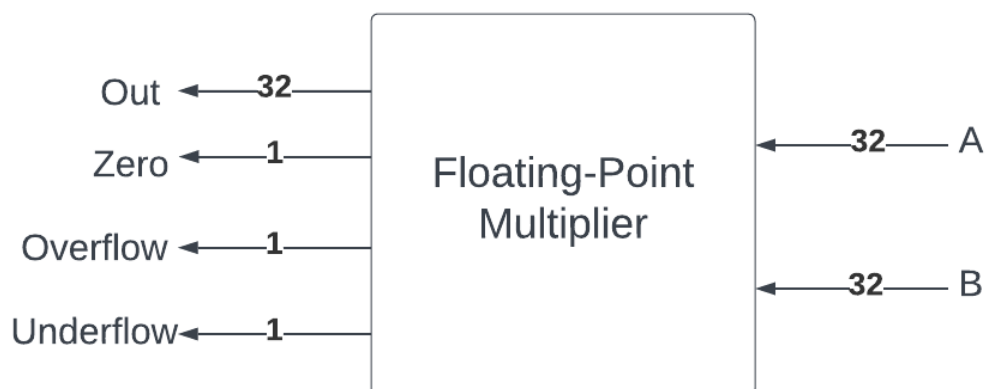
CSE 3441

LABORATORY ASSIGNMENT 8

FALL 2024

Floating Point Multiplier**(100 POINTS)****PURPOSE/OUTCOMES**

Your purpose in this lab is to design a 32-bit single precision floating point multiplier that can perform 32-bit floating point multiplication and that produces exception, underflow, overflow, zero, and the final output. See Figure 1 for the input/output diagram of the floating point multiplier. The multiplier component of your floating point multiplier should use the Wallace tree multiplier implemented in the previous lab. You will code your design in System Verilog, simulate to verify its correctness, and test its functionality using testbenches and ModelSim on Quartus. After completing this lab, you will have demonstrated an ability to design a floating point arithmetic unit, to write Verilog models of the floating point multiplier, to capture and verify your designs using Model-Sim on Quartus Prime.

**Figure 1 – Floating Point Multiplier**

BACKGROUND

In this lab you will use the instantiations created in all the previous labs from Lab 1 to Lab 5.

The modules you need for this lab are:

1. Lab 1: 8-bit Modular Exponent Subtractor
2. Lab 1: 24-bit Two-to-One Multiplexer
3. Lab 1: 8-bit Two-to-One Multiplexer
4. Lab 2: 24-bit Barrel Left Shifter
5. Lab 3: 8-bit Controlled Incrementor
6. Lab 7: Wallace Tree Multiplier

You will use the instantiations for the above mentioned module and connect them together using wires in a top level type module design.

DESIGN REQUIREMENT

1. Write a System Verilog model for a 32-bit single precision floating point multiplier. In addition to the floating point multiplication, provide functionality for overflow, underflow, zero, & exception outputs. Use the architecture or block diagram shown in Figure 2 to understand how the modules are interconnected.

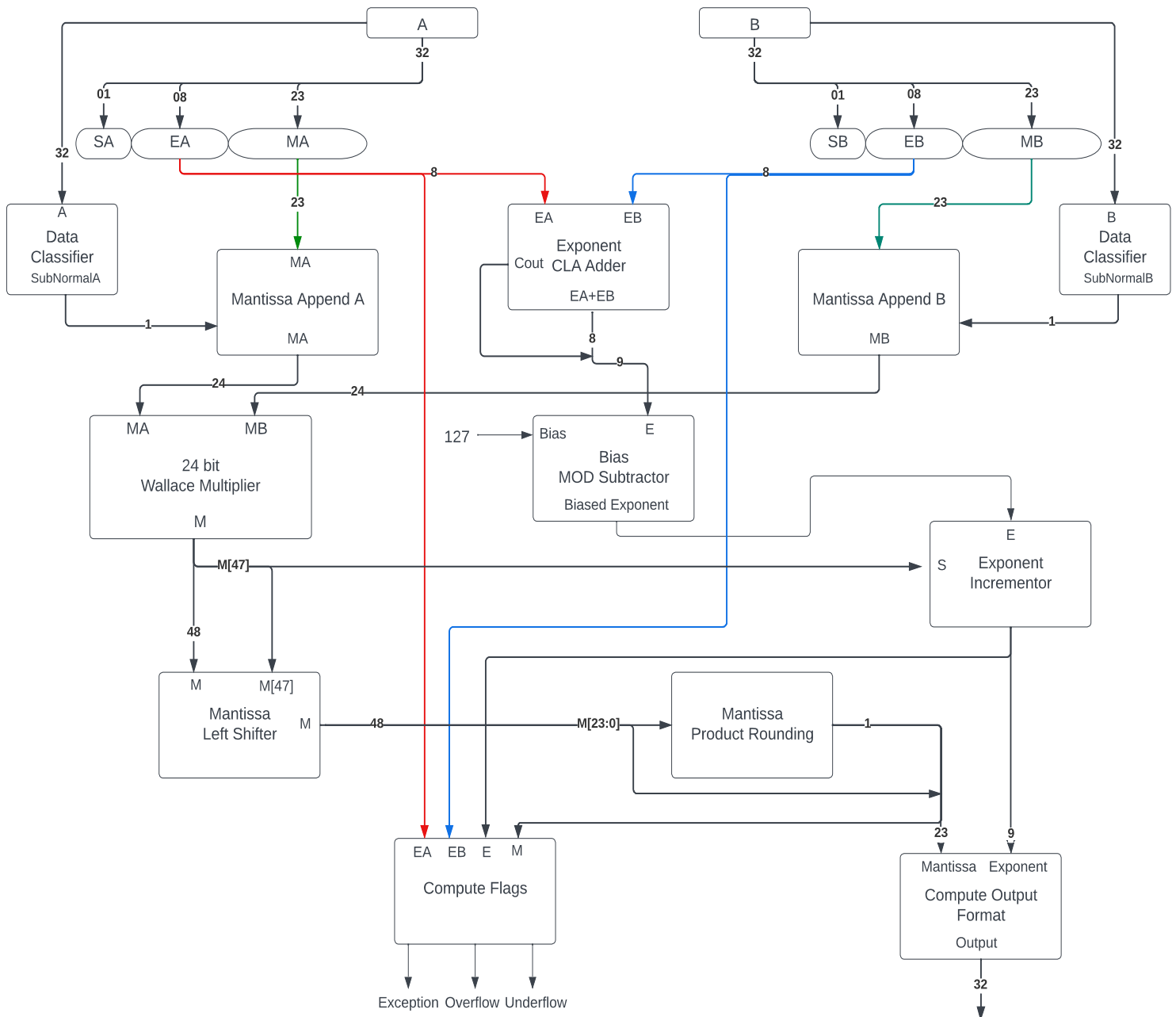


Figure 2 – Floating Point Multiplier Architecture

2. Add logic to compute the final sign bit output depending on the two different input values and the sign bit of each input value.
3. Design and construct a data classifier module to classify inputs into different data types.
4. Design and construct a mantissa append module to append based on data type.
5. Design and construct a product rounding module.
6. Add logic to compute the various error flags for this module like overflow, underflow etc.
7. Create an instantiation template for this module for future use.

DESIGN VERIFICATION

1. Simulate your design using waveforms to verify its correctness. Use the following values of A and B for your simulation inputs.

(a) $A = 32'h4234_851F // 45.13$

$B = 32'h427C_851F // 63.13$

(b) $A = 32'h4049_999A // 3.15$

$B = 32'hC166_3D71 // -14.39$

(c) $A = 32'hC152_6666 // -13.15$

$B = 32'hC240_A3D7 // -48.16$

(d) $A = 32'h3ACA_62C1 // 0.00154408081$

$B = 32'h3ACA_62C1 // 0.00154408081$

2. Include screen shots of your simulation waveform in your report.

3. Record the simulation results in the table below for your report.

A	B	$R = A \times B$	Underflow	Overflow	Zero
45.13	63.13				
3.15	-14.39				
-13.15	-48.16				
0.00154408081	0.00154408081				

TESTBENCH VERIFICATION

1. Write a test bench for this 32-bit floating point multiplier module using test cases above.
2. Simulate the test bench using ModelSim on Quartus.
3. Screenshot your output from the tcl console.
4. Screenshot your ModelSim simulation results.

CHECK YOUR UNDERSTANDING

1. What is a subnormal data type and how does it affect the appending of both your mantissa inputs?
2. What is the significance of subtracting the bias from added exponent value?
3. What kind of IEEE 754 rounding did you perform in the mantissa product rounding module? What influenced your choice?
4. Explain the different error flags used and the method of computing those flags?

7.9 Lab 9

Name: _____ ID# _____	
Date Submitted: _____ Lab Section # _____	
CSE [xxxx] Digital Logic	Fall Semester 2024
Lab Number 9 – Floating Point Divider	
Perform [Month] [Date], [Year]	
This lab is performed on the DE10-Lite.	

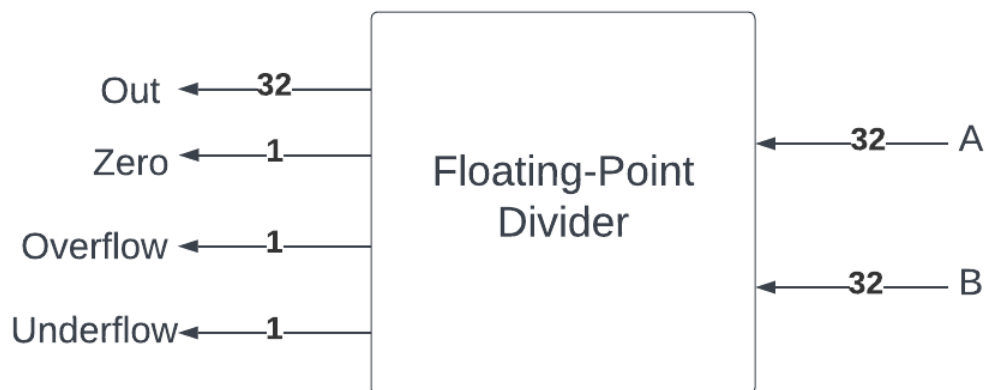
CSE 3441

LABORATORY ASSIGNMENT 9

FALL 2024

Floating Point Divider**(100 POINTS)****PURPOSE/OUTCOMES**

Your purpose in this lab is to design a 32-bit single precision floating point divider that can perform 32-bit floating point division and that produces exception, underflow, overflow, zero, and the final output. See Figure 1 for the input/output diagram of the floating point divider. The divider component of your floating point division can use any fixed point divider module of your choice. You will code your design in System Verilog, simulate to verify its correctness, and test its functionality using testbenches and ModelSim on Quartus. After completing this lab, you will have demonstrated an ability to design a floating point arithmetic unit, to write Verilog models of the floating point divider, to capture and verify your designs using Model-Sim on Quartus Prime.

**Figure 1 – Floating Point Divider**

BACKGROUND

In this lab you will use the instantiations created in all the previous labs from Lab 1 to Lab 5.

The modules you need for this lab are:

1. Lab 1: 8-bit Modular Exponent Subtractor
2. Lab 1: 24-bit Two-to-One Multiplexer
3. Lab 1: 8-bit Two-to-One Multiplexer
4. Lab 2: 24-bit Barrel Right Shifter
5. Lab 3: 8-bit Controlled Decrementer
6. Lab 4: 9-bit Carry Look Ahead Adder
6. Lab 8: Mantissa Append
7. Lab 8: Mantissa Rounding
8. Lab 8: Data Classifier Module

You will use the instantiations for the above mentioned module and connect them together using wires in a top level type module design.

DESIGN REQUIREMENT

1. Write a System Verilog model for a 32-bit single precision floating point divider. In addition to the floating point division, provide functionality for overflow, underflow, zero, & exception outputs. Use the architecture or block diagram shown in Figure 2 to understand how the modules are interconnected.

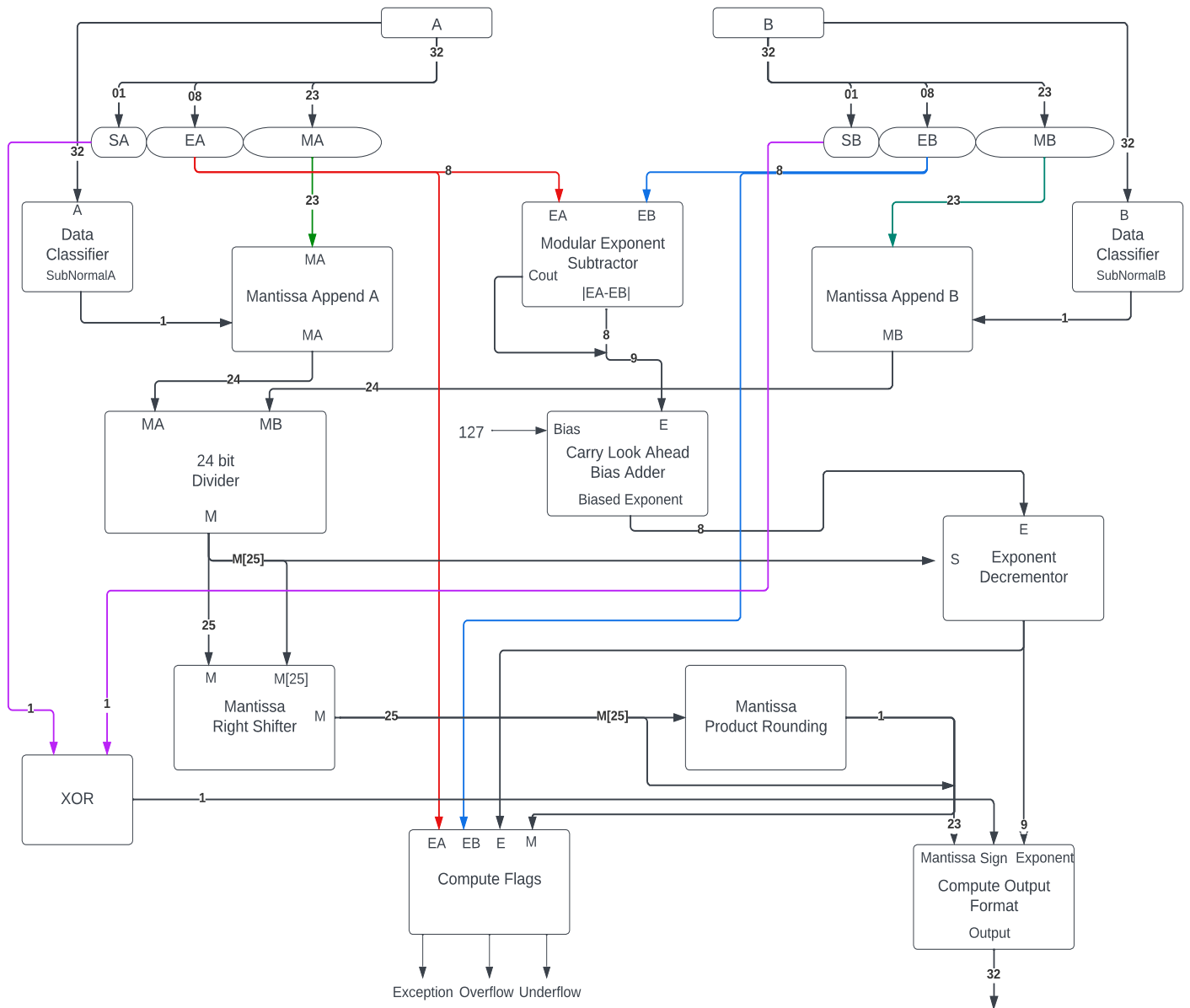


Figure 2 – Floating Point Divider Architecture

2. Add logic to compute the final sign bit output depending on the two different input values and the sign bit of each input value.
3. Add logic to compute the various error flags for this module like overflow, underflow etc.
4. Create an instantiation template for this module for future use.

DESIGN VERIFICATION

1. Simulate your design using waveforms to verify its correctness. Use the following values of A and B for your simulation inputs.

$$(a) A = 32'h4400_1CCD // 512.45$$

$$B = 32'h4322_570A // 162.34$$

$$(b) A = 32'h43E7_9EB8 // 463.24$$

$$B = 32'hC263_3333 // -56.80$$

$$(c) A = 32'hC288_999A // -68.3$$

$$B = 32'hC1C7_3333 // -24.9$$

$$(d) A = 32'h4455_9AE1 // 854.42$$

$$B = 32'h4419_E99A // 615.65$$

2. Include screen shots of your simulation waveform in your report.
3. Record the simulation results in the table below for your report.

A	B	R = A / B	Underflow	Overflow	Zero
512.45	162.34				
463.24	-56.80				
-68.3	-24.9				
854.42	615.65				

TESTBENCH VERIFICATION

1. Write a test bench for this 32-bit floating point divider module using test cases above.
2. Simulate the test bench using ModelSim on Quartus.
3. Screenshot your output from the tcl console.
4. Screenshot your ModelSim simulation results.

CHECK YOUR UNDERSTANDING

1. Which fixed arithmetic divider did you use and why?
2. How does the XOR gate compute the sign of the operation?

7.10 Lab 10

Name: _____ ID# _____	
Date Submitted: _____ Lab Section # _____	
CSE [xxxx] Digital Logic	Fall Semester 2024
Lab Number 10 – Floating Point Unit	
Perform [Month] [Date], [Year]	
This lab is performed on the DE10-Lite.	

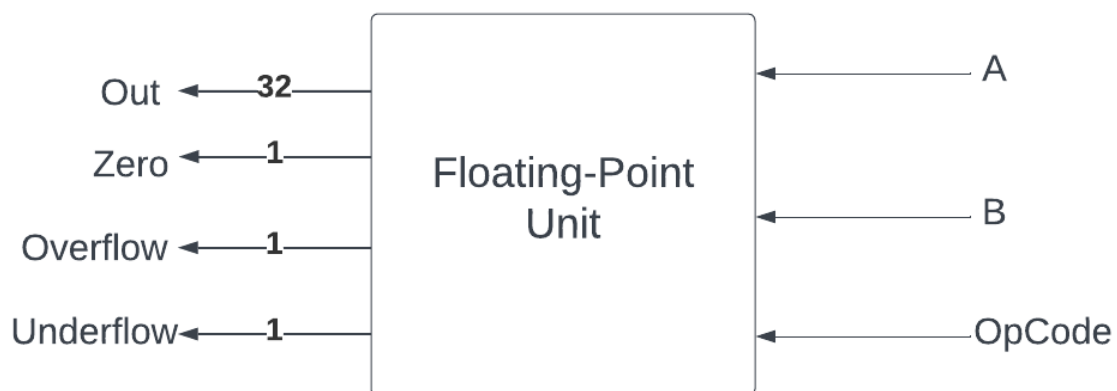
CSE 3441

LABORATORY ASSIGNMENT 10

FALL 2024

Floating Point Unit**(100 POINTS)****PURPOSE/OUTCOMES**

Your purpose in this lab is to design a 32-bit single precision floating point unit that can perform 32-bit floating point addition, subtraction, multiplication, and division based on a given Op code, and that produces exception, underflow, overflow, zero, and the final output. See Figure 1 for the input/output diagram of the floating point unit. The floating point unit will require multiplexers in order to figure out which operation needs to be done based on given opcode. You will code your design in System Verilog, simulate to verify its correctness, and test its functionality using testbenches and ModelSim on Quartus. After completing this lab, you will have demonstrated an ability to design a floating point arithmetic unit, to write Verilog models of the floating point unit, to capture and verify your designs using Model-Sim on Quartus Prime.

**Figure 1 – Floating Point Unit**

BACKGROUND

In this lab you will use the instantiations created in some of the previous labs:

1. Lab 5: Floating Point Adder
2. Lab 6: Floating Point Subtractor
3. Lab 8: Floating Point Multiplier
4. Lab 9: Floating Point Divider

DESIGN REQUIREMENT

Write a System Verilog model for a 32-bit single precision floating point Unit. Use the shown in Figure 2 to understand how the modules are interconnected.

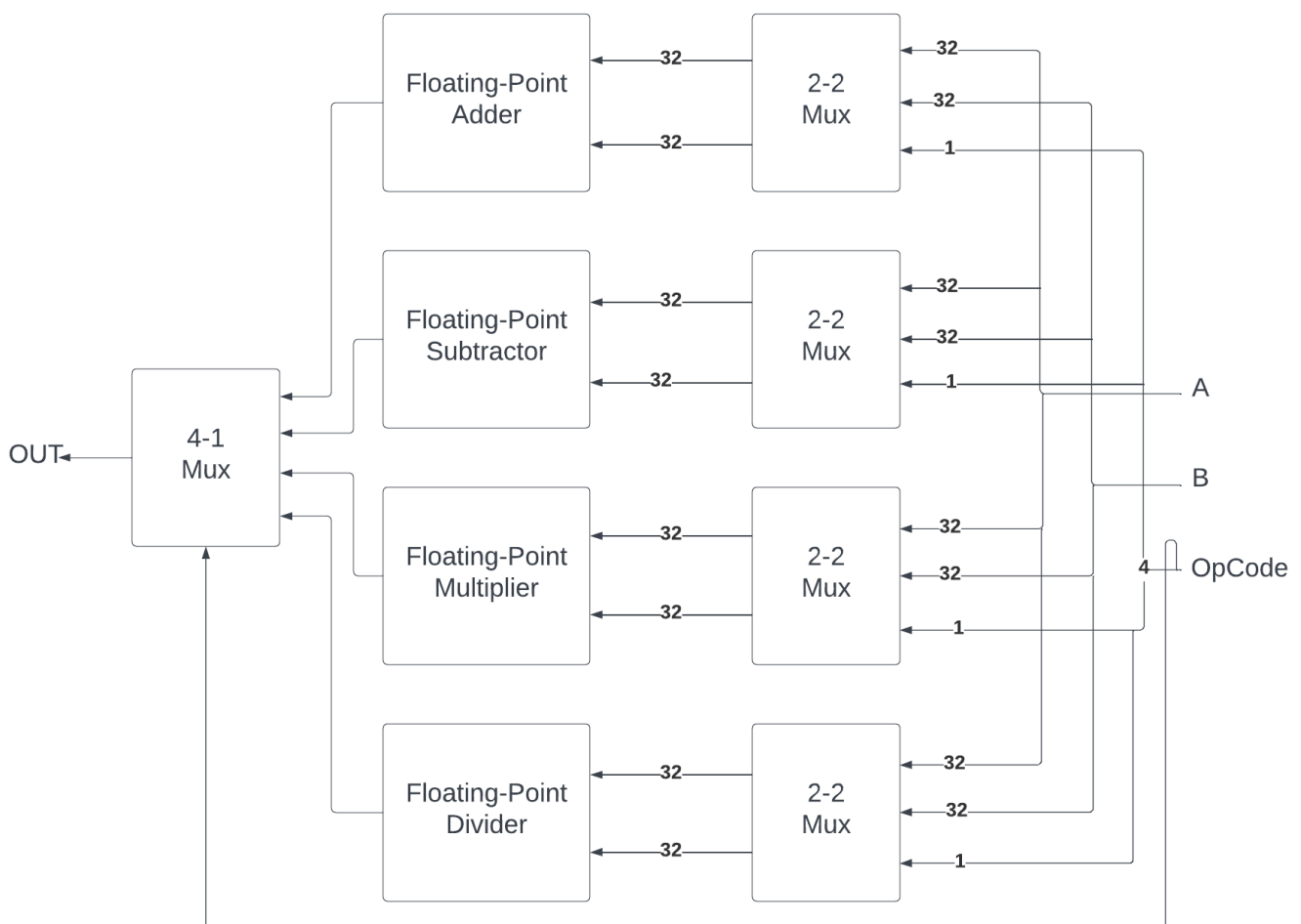


Figure 2 – Floating Point Unit Architecture

DESIGN VERIFICATION

1. Simulate your design using waveforms to verify its correctness. Use the following values of A and B for your simulation inputs.

(a) $A = 32'h4400_1CCD // 512.45$

$B = 32'h4322_570A // 162.34$

Opcode = 1000 // Division

(b) $A = 32'h4234_851F // 45.13$

$B = 32'h427C_851F // 63.13$

Opcode = 0100 // Multiplication

(c) $A = 32'h424D_0000 // 51.25$

$B = 32'h4174_CCCD // 15.3$

Opcode = 0010 // Subtraction

(d) $A = 32'h426F_EB85 // 59.979$

$B = 32'h40D0_0000 // 6.5$

Opcode = 0001 // Addition

2. Include screen shots of your simulation waveform in your report.

3. Record the simulation results in a table for your report.

TESTBENCH VERIFICATION

1. Write a test bench for this 32-bit floating point unit module using test cases above.

2. Simulate the test bench using ModelSim on Quartus.

3. Screenshot your output from the tcl console.

4. Screenshot your ModelSim simulation results.

Conclusion

This chapter of the thesis covered ten different laboratory assignments that convert the first five chapters of the thesis into an education module. These ten lab assignments are ready to be deployed in a Digital Logic & Design course and will equip students with all the necessary information and details to design a floating point unit just like the one implemented in this thesis. Additionally, the assignments also checks for the student's understanding after every lab section to make sure they understand the material being taught and the modules being implemented in a thorough fashion.

Chapter 8: Conclusion

This thesis paper consists of a total of eight chapters starting from the introduction of the subject matter in form of floating point number and floating point arithmetic. These eight chapters are mainly divided into four sections.

The first section consisted of just the first chapter that introduced the readers with all the basic concepts of floating point number, floating point number representation, and floating point arithmetic that is required to understand the consequent chapters of this thesis in an easy and secure manner.

The next section of the thesis dove deeper into what a floating point processor looks like and how this processor works. This section ranged from chapter two to chapter five which attempts to explain to the readers on how each module of a floating point unit is designed, constructed in hardware description language, and then tested using test benches and simulation tools in accordance with IEEE 754 online calculator.

The third section of the thesis acts as an education model. This section of the thesis converts the design, construction, and testing performed in the first chapter into an education model that makes it accessible for future students of this subject to build and test their own floating point unit and perhaps improve on the design in section two. This education model turns the entire floating point unit into a total of ten lab assignments with each lab working on some component of the unit.

And finally, the last and fourth section consists of the final conclusion and future scope of work, which is this chapter, along with chapter eight and nine which illustrates the references used throughout this thesis paper and a section about the author.

8.1 Future Scope of Work

The Verilog code written for complete 32-bit floating point arithmetic unit has been implemented and tested on ModelSim. Once this entire model has been created on the Verilog code as shown in this paper, the same can be optimized using system Verilog or VHDL and can be regenerated with optimized results.

Furthermore, an extension of this project can be construction of a piece of hardware that can facilitate the synthesis of this floating point unit on the DE-1 SoC Cyclone V board. The suggested piece of hardware will need to be equipped to take in 32-bit inputs and should be able to display the 32-bit output using an LCD screen or equivalent.

Bibliography

- [1] Russinoff, David M. "Floating-Point Numbers." *Formal Verification of Floating-Point Hardware Design*, 2021, pp. 47–54., https://doi.org/10.1007/978-3-030-87181-9_4.
- [2] Goldberg, David. "What Every Computer Scientist Should Know about Floating-Point Arithmetic." *ACM Computing Surveys*, vol. 23, no. 1, 1991, pp. 5–48., <https://doi.org/10.1145/103162.103163>.
- [3] Edwards, Eddie. "Floating Point Numbers." *Floating Point Numbers*, <https://www.doc.ic.ac.uk/~eedwards/compsys/float/>.
- [4] Brown, W. S. "A Simple but Realistic Model of Floating-Point Computation." *ACM Transactions on Mathematical Software*, vol. 7, no. 4, 1981, pp. 445–480., <https://doi.org/10.1145/355972.355975>.
- [5] "Floating-Point Basics and the IEEE-754 Standard." *Documentation – Arm Developer*, <https://developer.arm.com/documentation/den0042/a/Floating-Point/Floating-point-basics-and-the-IEEE-754-standard>.
- [6] "IEEE Annals of the History of Computing." *IEEE Annals of the History of Computing*, vol. 26, no. 2, 2004, pp. 01–01., <https://doi.org/10.1109/mahc.2004.1299651>.
- [7] "Machine Numbers and the IEEE 754 Floating-Point Standard." *Introduction to Scientific and Technical Computing*, 2016, pp. 31–37., <https://doi.org/10.1201/9781315382395-3>.
- [8] Presuhn, R. "Textual Conventions for the Representation of Floating-Point Numbers." 2011, <https://doi.org/10.17487/rfc6340>.
- [9] Schwarz M. Eric, Trong Dao Son "Introduction to denormalized numbers" Hardware Implementation of denormalized numbers.
- [10] Venners, Under the Hood By Bill, and Bill Venners. "Floating-Point Arithmetic." *InfoWorld, JavaWorld*, 1 Oct. 1996, <https://www.infoworld.com/article/2077257/floating-point-arithmetic.html>.
- [11] Stallings, William. "Floating Point Arithmetic." *Computer Organization and Architecture: Designing for Performance*, Pearson, New York, NY, 2022.
- [12] Stallings, William. "Floating Point Arithmetic." *Computer Organization and Architecture: Designing for Performance*, Pearson, New York, NY, 2022.
- [13] Linhart, Jean Marie. "Mata Matters: Overflow, Underflow and the IEEE Floating-Point Format." *The Stata Journal: Promoting Communications on Statistics and Stata*, vol. 8, no. 2, 2008, pp. 255–268., <https://doi.org/10.1177/1536867x0800800207>.

- [14] "5. Rounding." *Numerical Computing with IEEE Floating Point Arithmetic*, 2001, pp. 25–29., <https://doi.org/10.1137/1.9780898718072.ch5>.
- [15] Hettiarachchi, Don Lahiru, et al. "Integer vs. Floating-Point Processing on Modern FPGA Technology." *2020 10th Annual Computing and Communication Workshop and Conference (CCWC)*, 2020, <https://doi.org/10.1109/ccwc47524.2020.9031118>.
- [16] Cavanagh, Joseph. "Floating-Point Addition." *Computer Arithmetic and Verilog HDL Fundamentals*, 2017, pp. 551–570., <https://doi.org/10.1201/b12751-12>.
- [17] Stallings, William. "Floating Point Arithmetic." *Computer Organization and Architecture: Designing for Performance*, Pearson, New York, NY, 2022.
- [18] Doyen, Laurent, et al. "Robustness of Sequential Circuits." *2010 10th International Conference on Application of Concurrency to System Design*, 2010, <https://doi.org/10.1109/acsd.2010.26>.
- [19] Carroll Bill. "Twos Complement Convertor." *CSE 5357 Output Unit Lecture*.
- [20] "Systemverilog Study Notes. Barrel Shifter RTL Combinational Circuit." *element14 Community*, <https://community.element14.com/technologies/fpga-group/b/blog/posts/systemverilog-study-notes-barrel-shifter-rtl-combinational-circuit>.
- [21] Admin. "8 Bit Barrel Shifter Verilog." *VLSI GYAN*, 22 Jan. 2022, <http://vlsigyan.com/barrel-shifter-verilog/>.
- [22] Johri, Raj, et al. "High Performance 8 Bit Cascaded Carry Look Ahead Adder with Precise Power Consumption." *International Journal of Communication Systems*, vol. 28, no. 8, 2014, pp. 1475–1483., <https://doi.org/10.1002/dac.2727>.
- [23] Carroll Bill. "Carry Look Ahead Adder block Diagram" *CSE 5357 Carry Look Ahead Adder lecture presentation*. UTA CSE 5357
- [24] H., Schmidt. "Tools & Thoughts." *IEEE-754 Floating Point Converter*, <https://www.h-schmidt.net/FloatConverter/IEEE754.html>.
- [25] Boldo, Sylvie, and Marc Daumas. "Properties of the Subtraction Valid for Any Floating Point System." *Electronic Notes in Theoretical Computer Science*, vol. 66, no. 2, 2002, pp. 132–144., [https://doi.org/10.1016/s1571-0661\(04\)80408-0](https://doi.org/10.1016/s1571-0661(04)80408-0).
- [26] Stallings, William. "Floating Point Arithmetic." *Computer Organization and Architecture: Designing for Performance*, Pearson, New York, NY, 2022.
- [27] Roy, Dr. Shirshendu. "Floating Point Architectures." *Digital System Design*, 13 Mar. 2021, <https://digitalsystemdesign.in/floating-point-architectures/>.
- [28] *Barrel Shifter*, <https://esrd2014.blogspot.com/p/barrel-shifter.html>.

- [29] *Barrel Shifter*, <https://esrd2014.blogspot.com/p/barrel-shifter.html>.
- [30] H., Schmidt. "Tools & Thoughts." *IEEE-754 Floating Point Converter*, <https://www.h-schmidt.net/FloatConverter/IEEE754.html>.
- [31] Al-Ashrafy, Mohamed, et al. "An Efficient Implementation of Floating Point Multiplier." *2011 Saudi International Electronics, Communications and Photonics Conference (SIEPC)*, 2011, <https://doi.org/10.1109/siepc.2011.5876905>.
- [32] Roy, Dr. Shirshendu. "Floating Point Multiplication." *Digital System Design*, 13 Mar. 2021, <https://digitalsystemdesign.in/floating-point-multiplication/>.
- [33] Stallings, William. "Floating Point Arithmetic." *Computer Organization and Architecture: Designing for Performance*, Pearson, New York, NY, 2022.
- [34] Ganssle, Jack. "IEEE 754 Floating Point Numbers." *The Firmware Handbook*, 2004, pp. 203–205., <https://doi.org/10.1016/b978-075067606-9/50019-9>.
- [35] "Design and Analysis of FIR Filters Using Wallace Tree Multiplier and Carry Select Adder." *International Journal of Recent Trends in Engineering and Research*, 2018, pp. 151–153., <https://doi.org/10.23883/ijrter.conf.02180328.024.l2uiw>.
- [36] Manzoor Qasim, Syed, et al. "Towards Optimised FPGA Realisation of Microprogrammed Control Unit Based FIR Filters." *Control Theory in Engineering [Working Title]*, 2019, <https://doi.org/10.5772/intechopen.90662>.
- [37] Vlsiverify. "Wallace Tree Multiplier." *VLSI Verify*, 11 Dec. 2022, <https://vlsiverify.com/verilog/verilog-codes/wallace-tree-multiplier>.
- [38] Aswani, T.S., and B. Premanand. "Area Efficient Floating Point Addition Unit with Error Detection Logic." *Procedia Technology*, vol. 24, 2016, pp. 1149–1154., <https://doi.org/10.1016/j.protcy.2016.05.068>.
- [39] Maclaren, Nick. "IEEE 754 Error Handling and Programming Languages." *IEEE*, Mar. 2000, <https://doi.org/10.3403/01786371u>.
- [40] H., Schmidt. "Tools & Thoughts." *IEEE-754 Floating Point Converter*, <https://www.h-schmidt.net/FloatConverter/IEEE754.html>.
- [41] Grover, Naresh, and M.K. Soni. "Design of FPGA Based 32-Bit Floating Point Arithmetic Unit and Verification of Its VHDL Code Using MATLAB." *International Journal of Information Engineering and Electronic Business*, vol. 6, no. 1, 2014, pp. 1–14., <https://doi.org/10.5815/ijieeb.2014.01.01>.
- [42] Roy, Dr. Shirshendu. "Floating Point Division." *Digital System Design*, 13 Mar. 2021, <https://digitalsystemdesign.in/floating-point-division/>.

- [43] Stallings, William. "Floating Point Arithmetic." *Computer Organization and Architecture: Designing for Performance*, Pearson, New York, NY, 2022.
- [44] Green, Will. "Division in Verilog." *Project F*, 1 Mar. 2023, <https://projectf.io/posts/division-in-verilog/>.
- [45] H., Schmidt. "Tools & Thoughts." *IEEE-754 Floating Point Converter*, <https://www.h-schmidt.net/FloatConverter/IEEE754.html>.
- [46] *Design of Single Precision Float Adder (32-Bit Numbers) According to ...*
<https://upcommons.upc.edu/bitstream/handle/2099.1/15467/32BitFloatingPointAdder.pdf?sequence=4>.

About the Author



Kartikey Sharan, born in Bihar, India, is a Master's student at University of Texas at Arlington. Kartikey moved to United States at the age of 17 to attend University of Texas at Arlington BS in Computer Engineering. He has 4 years of experience writing Verilog modules and has worked on projects such as designing and coding TRISC processor, and

designing and coding a SPI IP Core with Linux Device Drivers. He has a keen interest in the field of FPGA based digital system designs. Presently, he is working as a graduate teaching assistant under Dr. Bill Carroll for the Advanced Digital Logic & Design course at UTA.