University of Tripoli

Computer and Electronics Engineering department

Design and Implementation of Single Precision Floating-point Arithmetic Logic Unit for RISC Processor on FPGA

FPGA Design for Embedded Systems
EC 582

By:
Noor Alhuda Adela (022152760)
Dania K. Alamen (022160629)

Supvr:
Dr. Mohamed M. Eljhani

March 2022

Summary

In this project report, we discuss the design of an arithmetic logic unit (ALU) and a floating-point unit (FPU) architecture that together performs all arithmetic and logical operations of computer processors and gives the flexibility of scalability up to 32-bits. The ALU and FPU were implemented in Verilog, simulated, and tested in ModelSim Altera.

Table of Contents

1. Introduction

This paper focuses on the design of the arithmetic logic unit (ALU) and floating-point unit (FPU), which perform integer mathematical and logic operations, and decimal mathematical operations, respectively. The end goal of this thesis is the design, Verilog implementation, and synthesis of the arithmetic logic unit and floating-point unit of a 32-bit reduced instruction set computer (RISC) processor.

The input consists of an instruction that contains an operation code (opcode), one or more operands. The operation code tells the ALU what operation to perform and the operands are used in the operation. (For example, two operands might be added together or compared logically.) The format may be combined with the opcode and tells, for example, whether this is a fixed-point or a floating-point instruction. The output consists of a result that is placed in a storage register and settings indicate whether the operation was performed successfully.

The remainder of the paper is divided into two parts - one each for the ALU and FPU. The first part describes the implementation of the functions of the arithmetic logic unit. Integer adders, multipliers, and dividers are discussed in detail, with the basic logic operations - AND, OR, XOR, etc. The second part of the paper describes the floating-point unit. Implementations of single-precision adder/subtractors, multipliers, and dividers that conform to the IEEE 754 standard are presented.

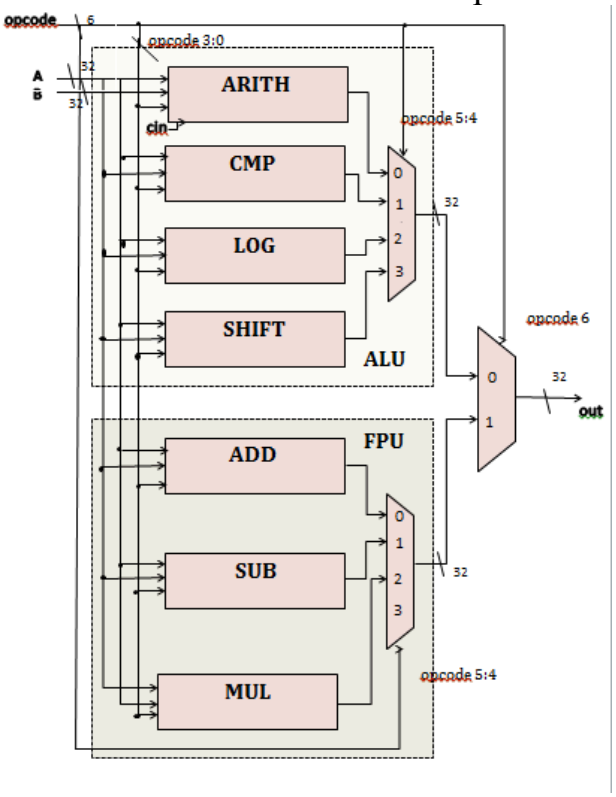

Figure 1. The general architecture of hybrid ALU

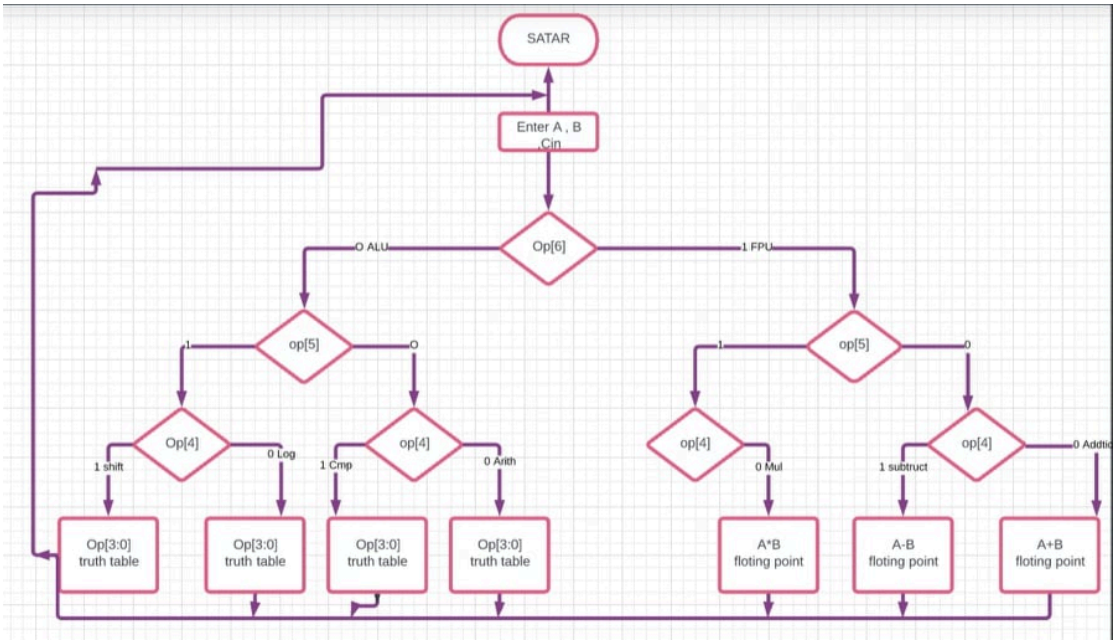

Figure 2: Flowchart

1.1 Project Timeline

We planned to follow a Bottom to up approach while designing this ALU. Firstly, we implemented simple logic blocks such as Comparator, Adder, Shifter, etc. in Verilog. Then after verifying their performance in the ModelSim simulator, with respect to generated signal patterns as inputs, we implemented the whole design.

Part I

2. Arithmetic Logic Unit Overview

An arithmetic logic unit (ALU) is an integral essential combinational circuit of any central processing unit (CPU). The processor uses the ALU to perform arithmetic operations such as addition, subtraction, multiplication, and division - as well as logical operations - OR, AND, XOR, inversion, and transformation operations. Additional operations such as data block comparisons. The ability to perform all these results of operations in the ALU is one of the most complex circuits in the CPU.

2.1. Design Block

Our design comprises 5 basic sections. We describe each one below

- Opcode Decoder
- Arithmetic Block
- Logical Block
- Comparator Block
- Shifter Block

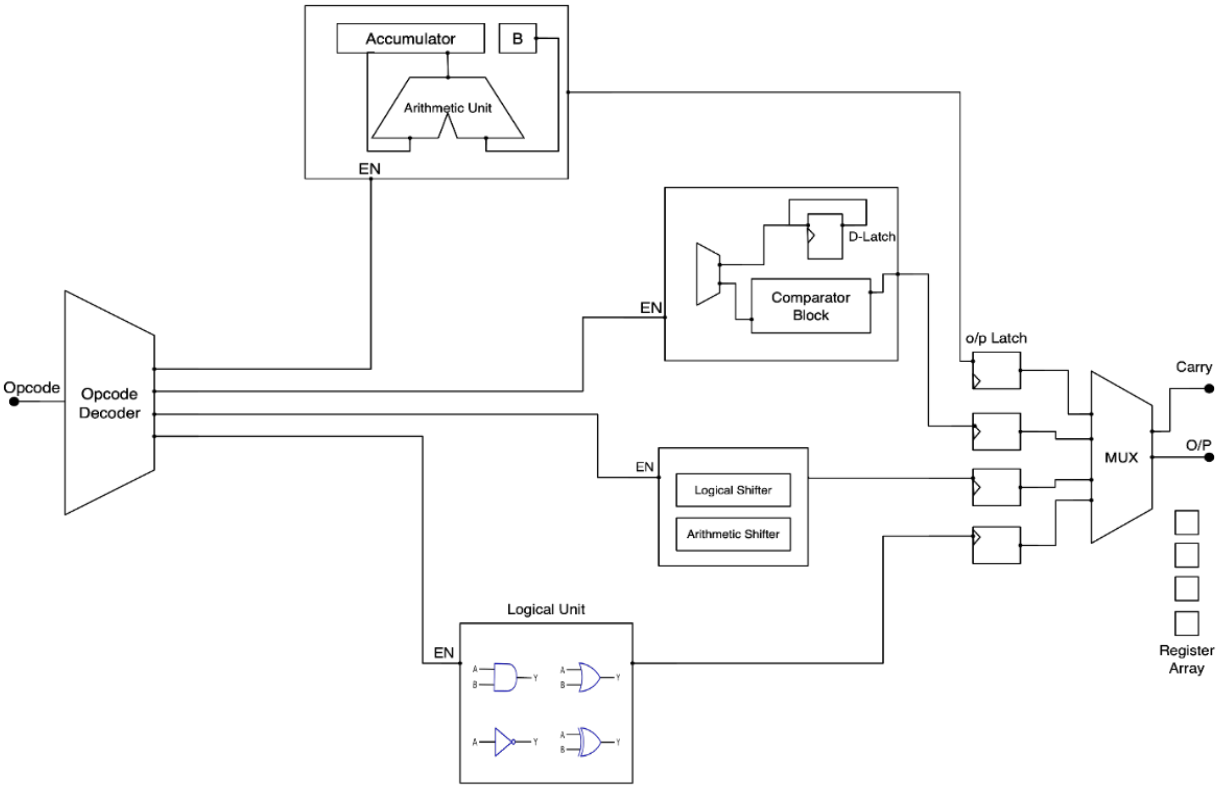

Figure 3 . ALU Block Diagram

2.1.1. Opcode Decoder:

The proposed design consists of a primary opcode decoder unit that activates the respective function block based on the instruction opcode performed by the processor and forwarded to the ALU block. The respective output lines will feed the enable lines of respective function blocks.

The decoder is a 6:43 unit i.e., it takes 6 bits opcode as input and in turn activates the respective function block to perform the desired operation.

Selected Instructions

The designed ALU takes a clock signal, two 32-bit operands, and a 6-bit opcode as inputs. A complete list of the implemented instructions can be seen in Table 1.

SELECT OPCODE				ACTIVE-HIGH DATA				
				ARITHMETIC S4 S5 =00		*COMPARRATOR* S4 S5 =01	*LOGIC* S4 S5 =10	*SHIFTING* S4 S5 =11
S0	S1	S2	S3	Cin=0	Cin=1			
0	0	0	0	OUT=A	OUT=A+1	OUT=A > B	OUT=~A	OUT= shl A
0	0	0	1	OUT=A+B	OUT=A+B+1	OUT=A < B	OUT=~(A^B)	OUT=shl B
0	0	1	0	OUT=A+`B	OUT=A+`B+1	OUT=A ≠ B	OUT=~A ⱽB	OUT=shr A
0	0	1	1	OUT=-1	OUT=0		OUT=0	OUT=shr B
0	1	0	0	OUT=B	OUT=B+1		OUT=~(AⱽB)	OUT=rol A
0	1	0	1	OUT=`B	OUT=`B+1		OUT=~B	OUT=rol B
0	1	1	0	OUT=A-B	OUT=A-B+1		OUT=A	OUT=ror A
0	1	1	1	OUT=B-A	OUT=B-A+1		OUT=A^~B	OUT=ror B
1	0	0	0	OUT=1	OUT=2		OUT=AⱽB	
1	0	0	1	OUT=0	OUT=1		OUT=~A^B	
1	0	1	0	OUT=A-1	OUT=A		OUT=B	
1	0	1	1	OUT=A+A	OUT=A+A+1		OUT=A^B	
1	1	0	0	OUT=B-1	OUT=B		OUT=1	
1	1	0	1				OUT=Aⱽ~B	

Table 1. ALU Operations

Using a Combination of These operations, any logic operation can be implemented.

2.1.2. Arithmetic Block:

This block is used to implement arithmetic operations such as Addition, Subtraction, Multiplication, and division. The Accumulator and the auxiliary b register feed as inputs to this block.

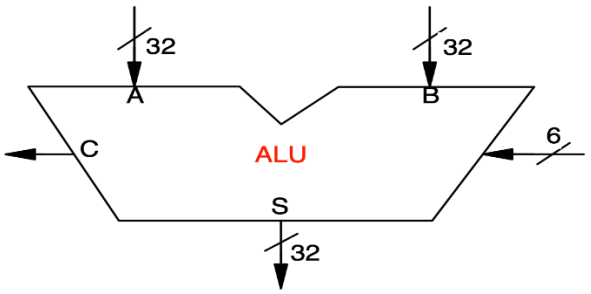

Figure 4. Arithmetic Operation

Integer Addition/Subtraction

Fast addition is extremely important in many digital systems. As an elementary school child knows, addition and subtraction are the same operations. Subtraction merely inverts the sign of the second operand. Using this knowledge, it becomes simple to implement integer subtraction using any type of integer adder.

To alter the adder to be able to handle both integer addition and subtraction, logic must be established to perform two's complement conversion on the B operand if subtraction is selected, and to leave B untouched if addition is selected. This can be done in one of two ways: using an inverter and a multiplexer on each bit, or an XOR gate with one bit tied to control, and the other tied to the corresponding bit of B.

Integer Multiplication

Multiplication, like addition, is a heavily used operation that figures prominently in many types of operations. Among many other uses, multiplication is used in signal processing and scientific applications. It is also a common basis for division.

Integer multipliers can be implemented in a variety of ways. Typical implementations are a shift and add. The multiplication operation produces a 64-bit output that utilizes both registers.

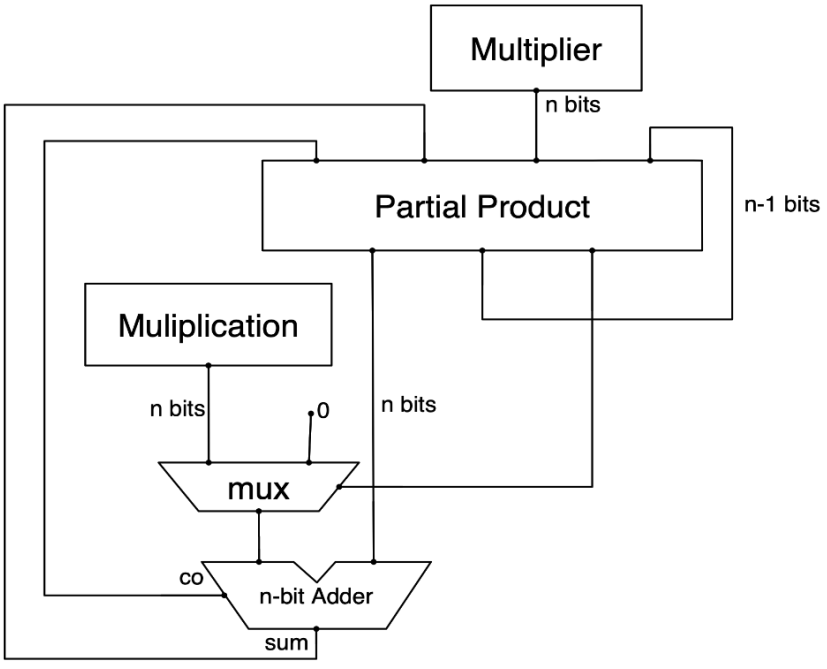

Figure 5. Shift and Add Multiplier Circuit

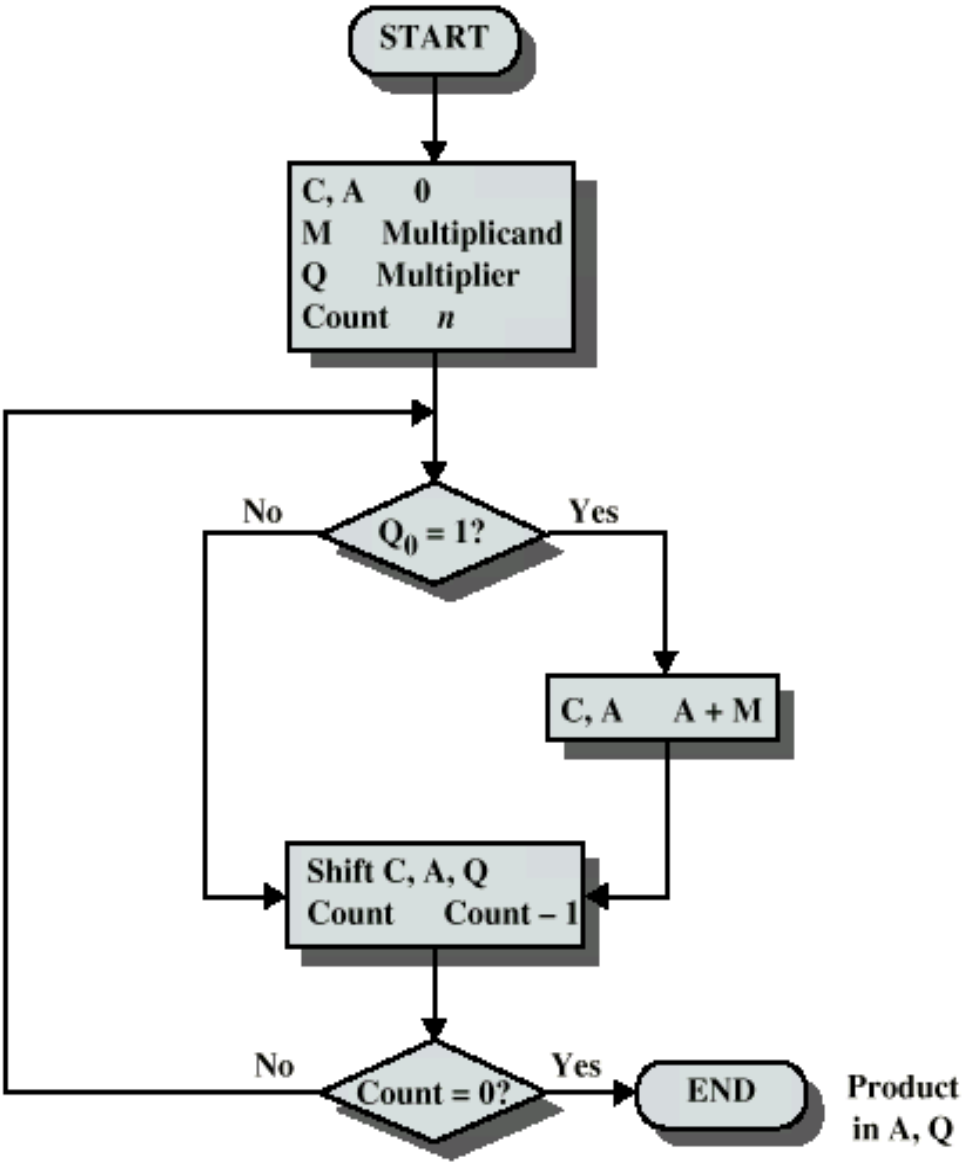

Figure 6: Unsigned Binary Multiplication Flowchart

Integer Division

The division is the least used of the four basic arithmetic operations. As such, it has been the least researched of the four operations and remains the most difficult operation to implement efficiently.

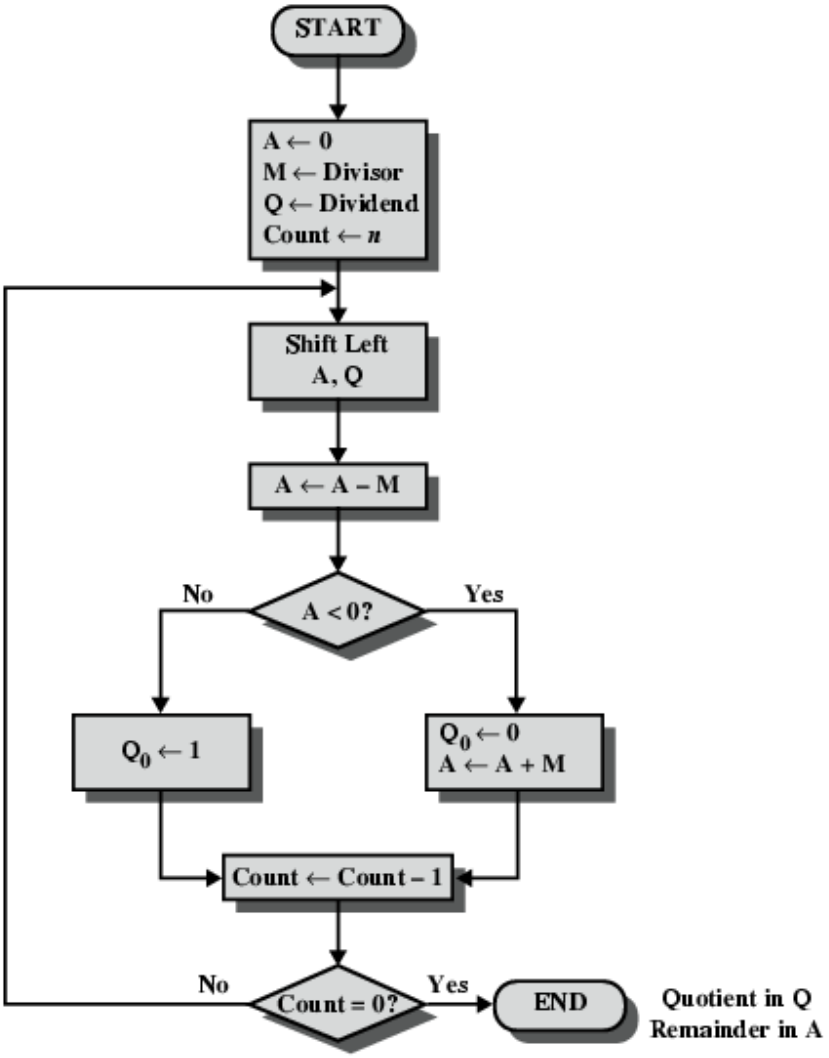

Figure 7: Unsigned Binary Division Flowchart

2.1.3. Comparator block:

This block consists of combinational circuit HDL code which performs bit matching and comparisons. Respective outputs may be used to branch instructions based on the comparison. Similarly, numerically larger, or smaller indications on respective operands may be used.

2.1.4. Logical block:

This block comprises basic logic gate units such as AND, OR, NOT, XOR, etc. Such operations on data operands are served by this block. Outputs are stored in respective latches. The logical operations implement a series of standard logic operations on the operands at the bit level. The AND operation produces a 1 at output bit i only if A_i and B_i are both equal to 1. The OR operation produces a 1 at output bit i if A_i or B_i is B_i is equal to 1, but not both. The NOR operation is the opposite of the OR operation. A 0 is inserted at the bit position if the operating conditions are not met. The ENV operation inverts each bit of both A and B.

2.1.5. Shifter/Rotate block:

This block consists of basic shifters such as Barrel Shifters and mechanisms for bit rotation. The right shift is implemented both arithmetically - where the operation is sign-extended as it is shifted – and logically - where 0's are inserted into the bit positions that are shifted out.

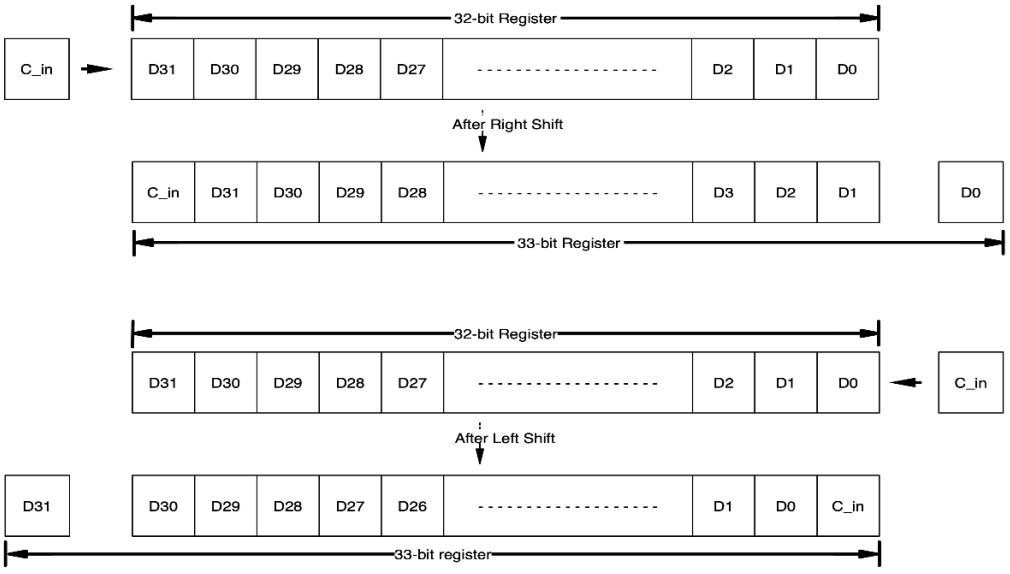

Figure 8. An example for a 32 bit scaled shifter block

Part II

3. Floating-Point Unit Overview

Floating-Point Units (FPU) are the hardware components that handle decimal mathematical operations in the CPU. Like the ALU, the FPU implements the four basic mathematical operations - addition, subtraction, multiplication, and division - the difference being the number representation scheme utilized. An ALU handles integer values, represented in binary numbers. This means that the entire 32-bits of the bit vector represents the portion of a number to the left of the decimal point. An FPU deals with both the integer and fraction portions of numbers. As there is no way to slide a decimal point into the bit vectors to tell the computer what is the integer and what is the fraction, the operands must be divided into sections representing the sign, exponent, and mantissa of the number.

3.1. The IEEE 754 Standard

The standard supports single-precision 32-bit numbers, and double-precision 64-bit numbers. As would be expected, double-precision offers a larger range (11 exponent bits compared to 8) and greater accuracy (52 fraction bits compared to 23) than single-precision. As they operate in the same manner and the focus of the work presented in this paper is on 32-bit inputs, only the single-precision format will be explored.

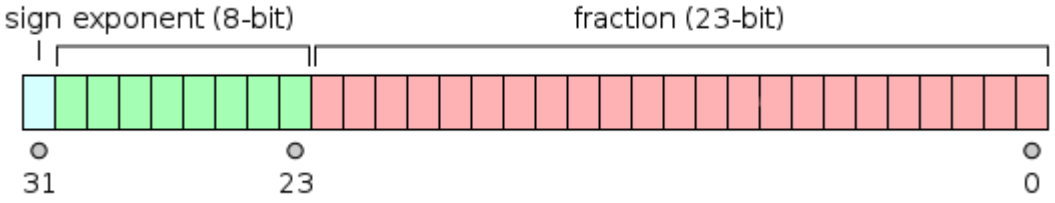

Figure 9: Single-Precision IEEE 754 Format

Table 2. Floating-Point: Special Cases

Number	Sign	Exponent (e)	Fraction (f)
0	X	00000000	00000000000000000000000
∞	0	11111111	00000000000000000000000
−∞	1	11111111	00000000000000000000000
NAN	X	11111111	nonzero

3.2. FPU Instructions

The instructions implemented for the floating-point unit are added, subtract, multiply and divide.

Table 3. FPU Operations

Operation	OP Code
Addition	00
Subtraction	01
Multiplication	10

3.2.1.Floating-Point Adder/Subtractor

Just as integer addition/subtraction is the most used ALU operation, FP addition/subtraction is the most utilized floating-point operation.

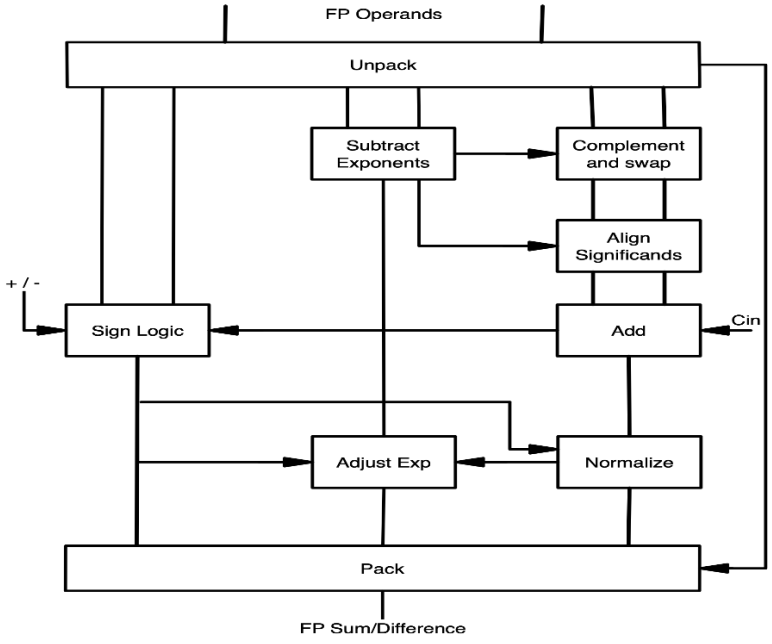

Figure 10. Block Diagram of Floating-Point Adder/Subtractor

The difference between the exponents is used to determine the amount of right shifting necessary to align the smaller operand with the larger operand.

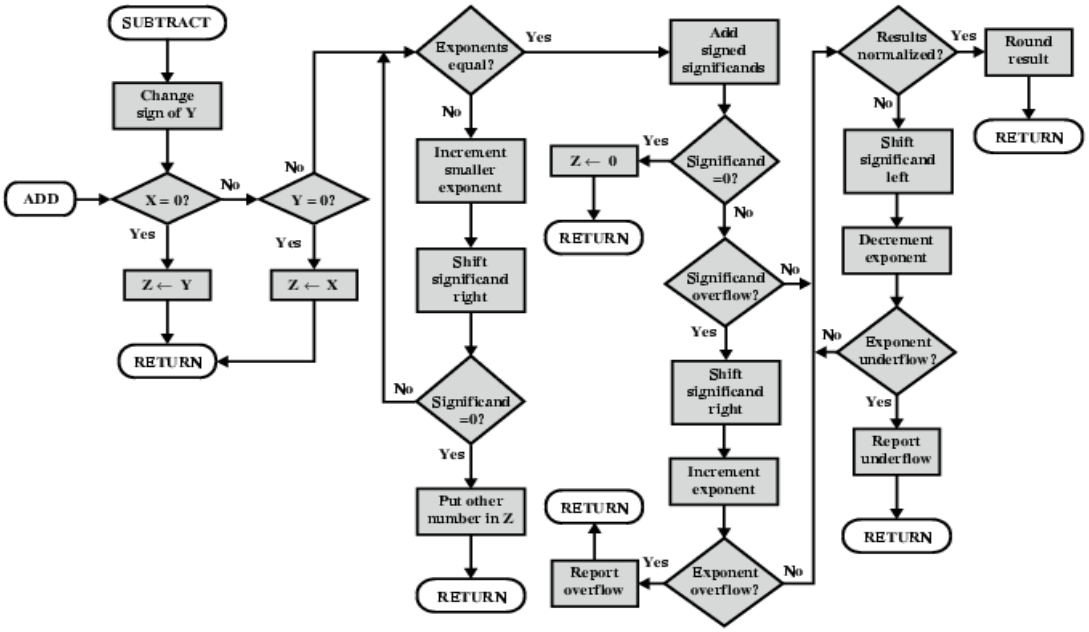

Figure 11: FP Addition & Subtraction Flowchart

3.2.2.Floating-Point Multiplication

Floating-point multiplication has nearly as many far-reaching applications as floating-point addition/subtraction. As a result, it is important to implement an efficient multiplier design.

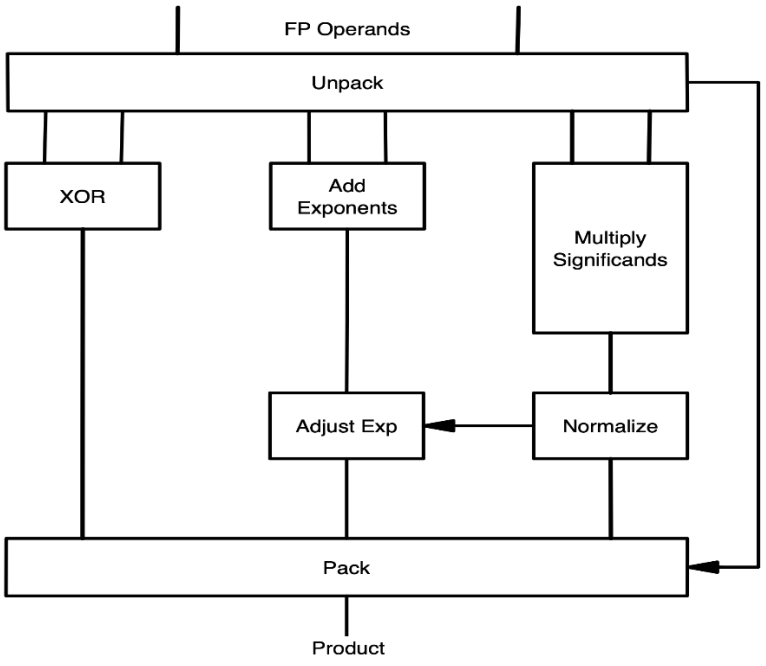

Figure 12.Generic FP Multiplier Block Diagram

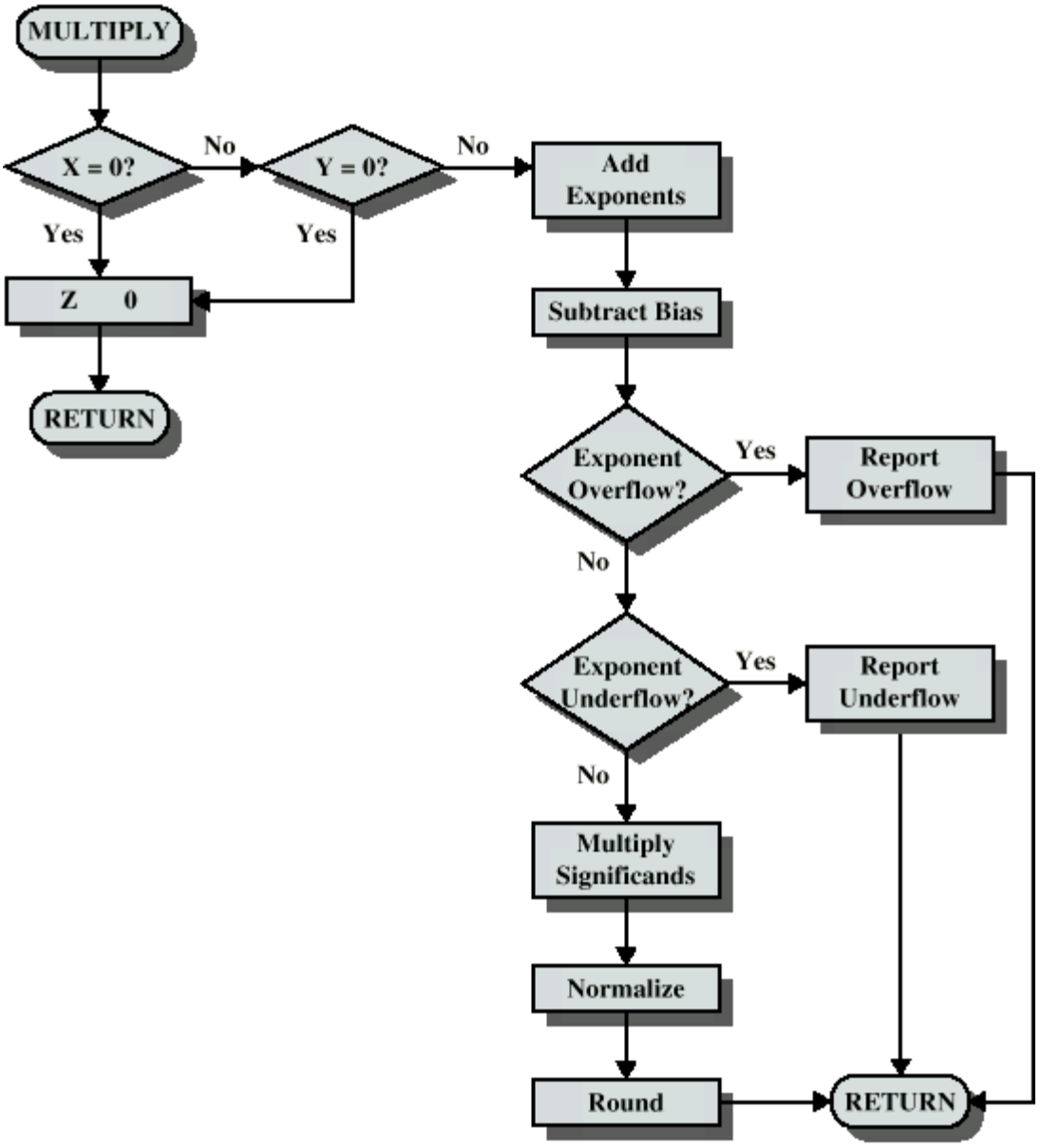

Figure 13: Floating Point Multiplication Flowchart

4. ALU & FPU Synthesis Results

Each of the arithmetic blocks and floating-point described previously were implemented in the Verilog hardware descriptive language. Test benches were developed in ModelSim, and executed in order to ensure that the functions performed as expected.

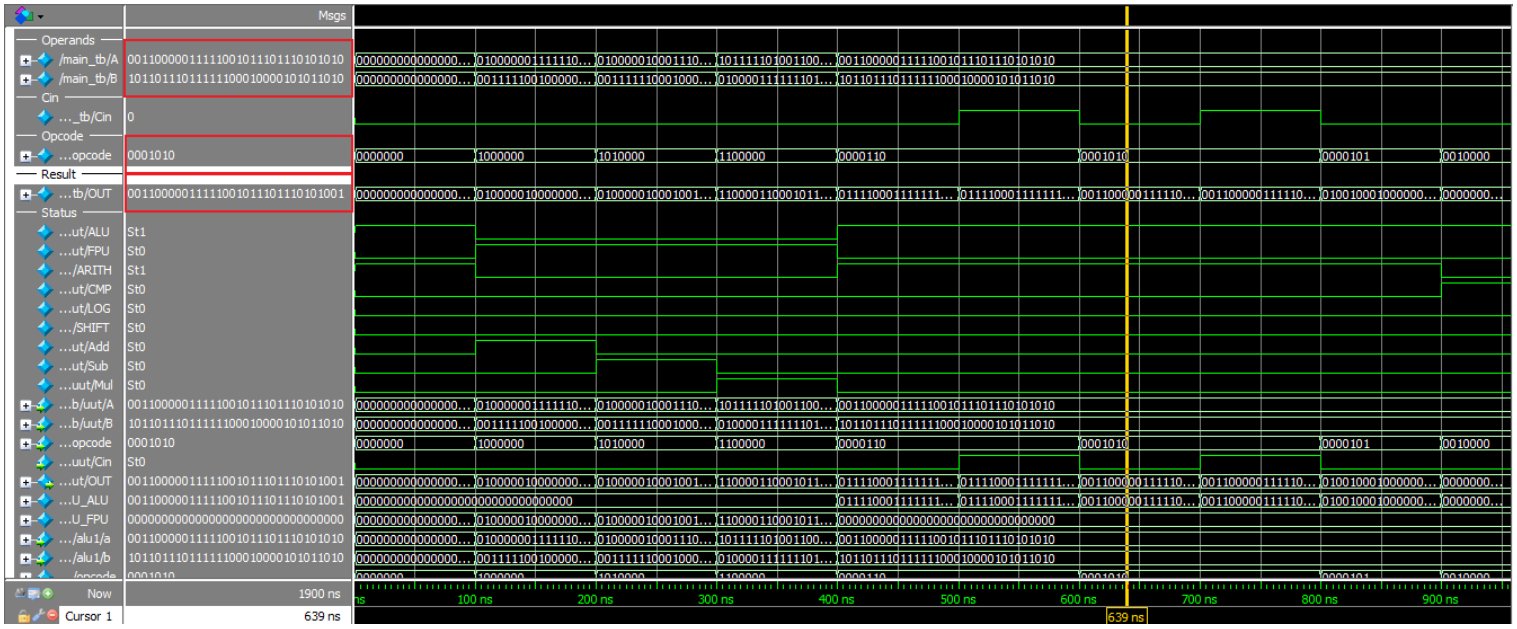

Figure 14: A – 1

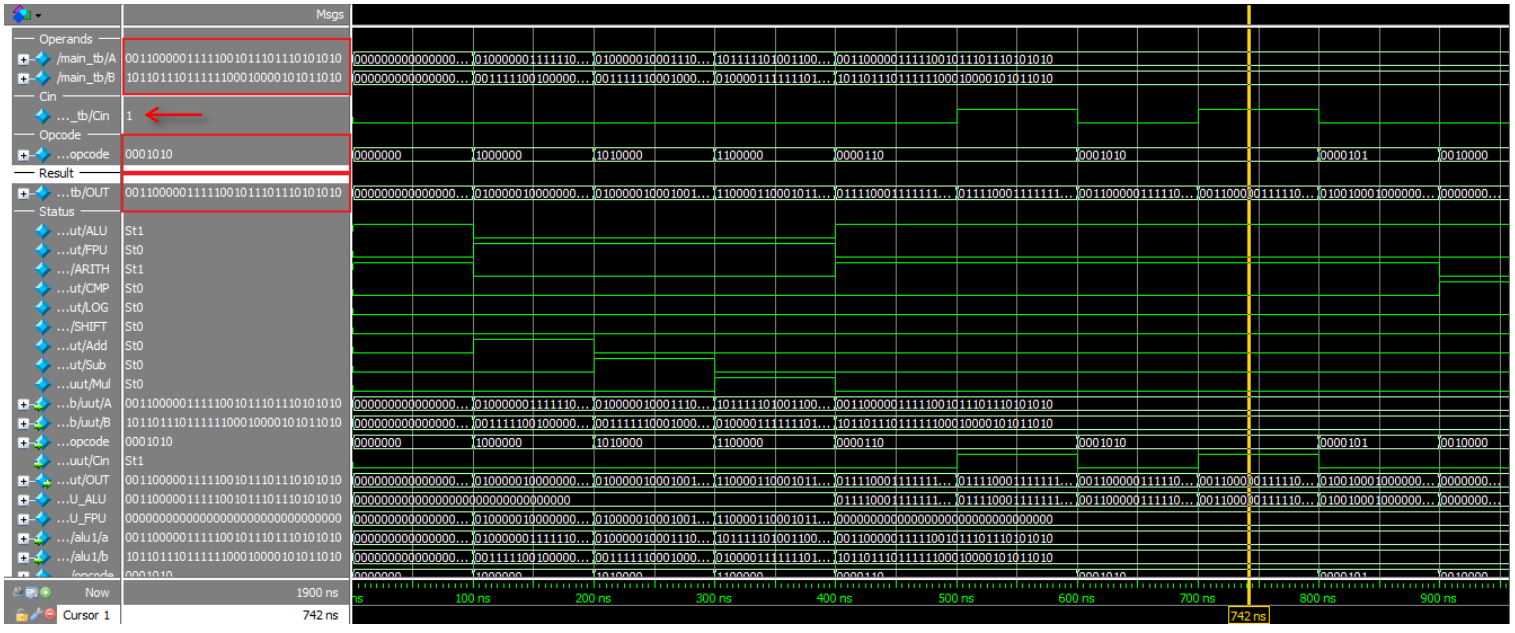

Figure 15: A

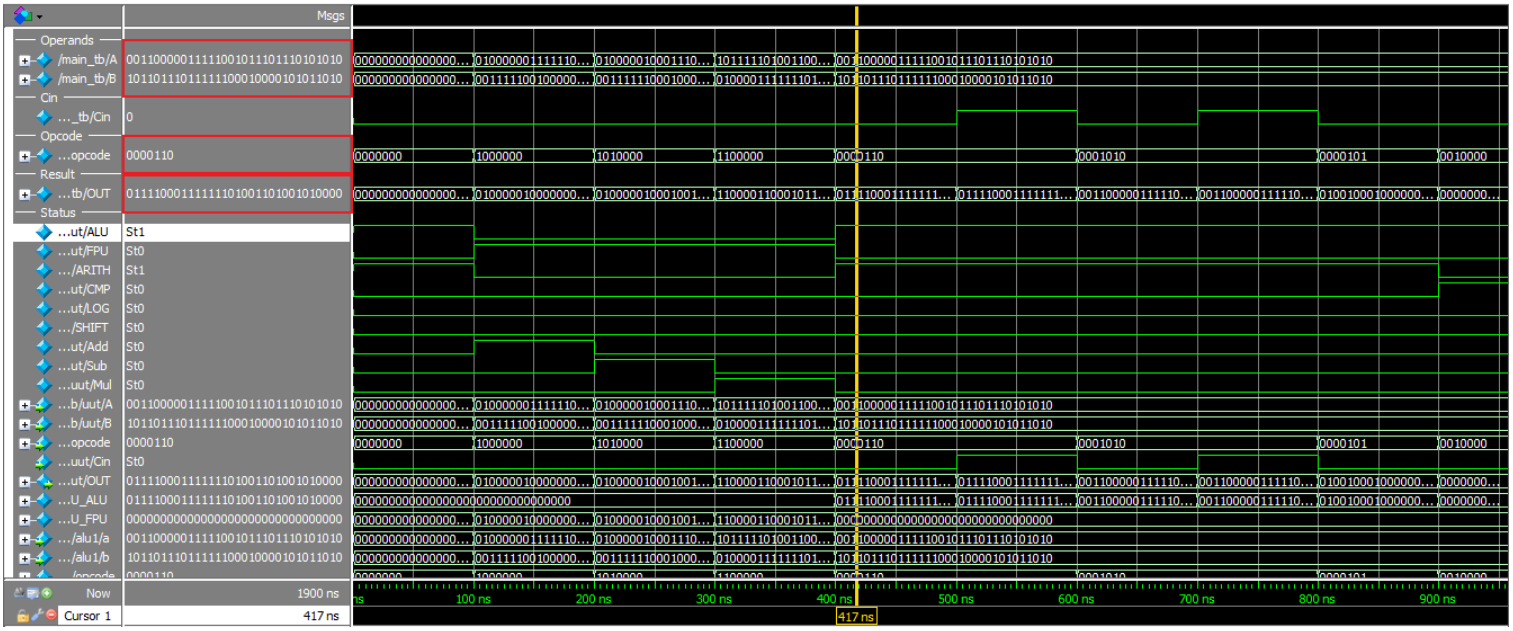

Figure 16: $A - B$

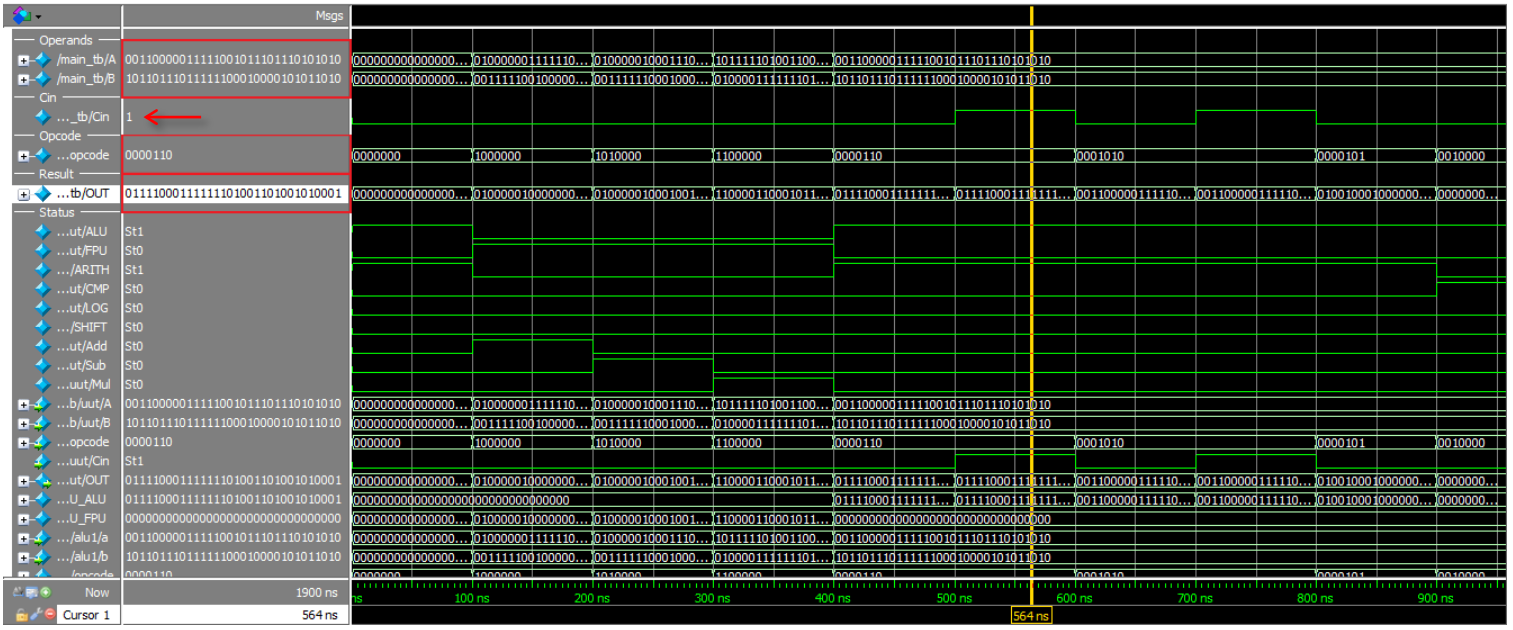

Figure 17: $A - B + 1$

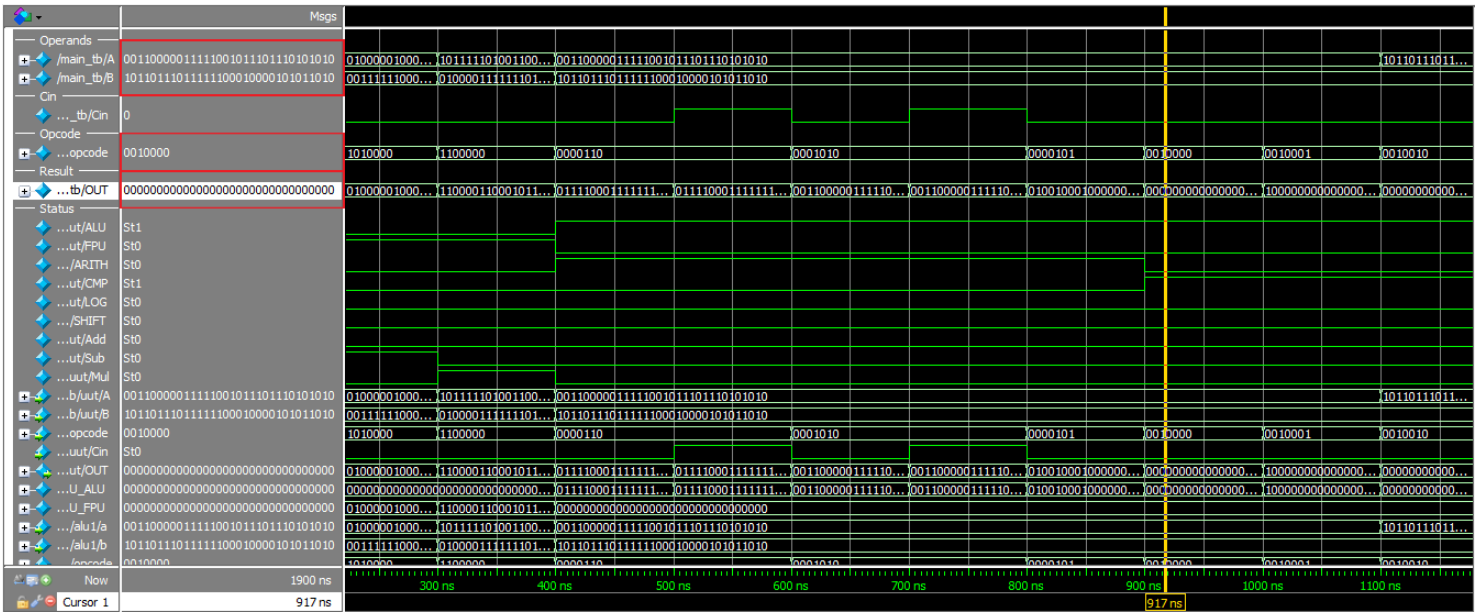

Figure 18: A > B

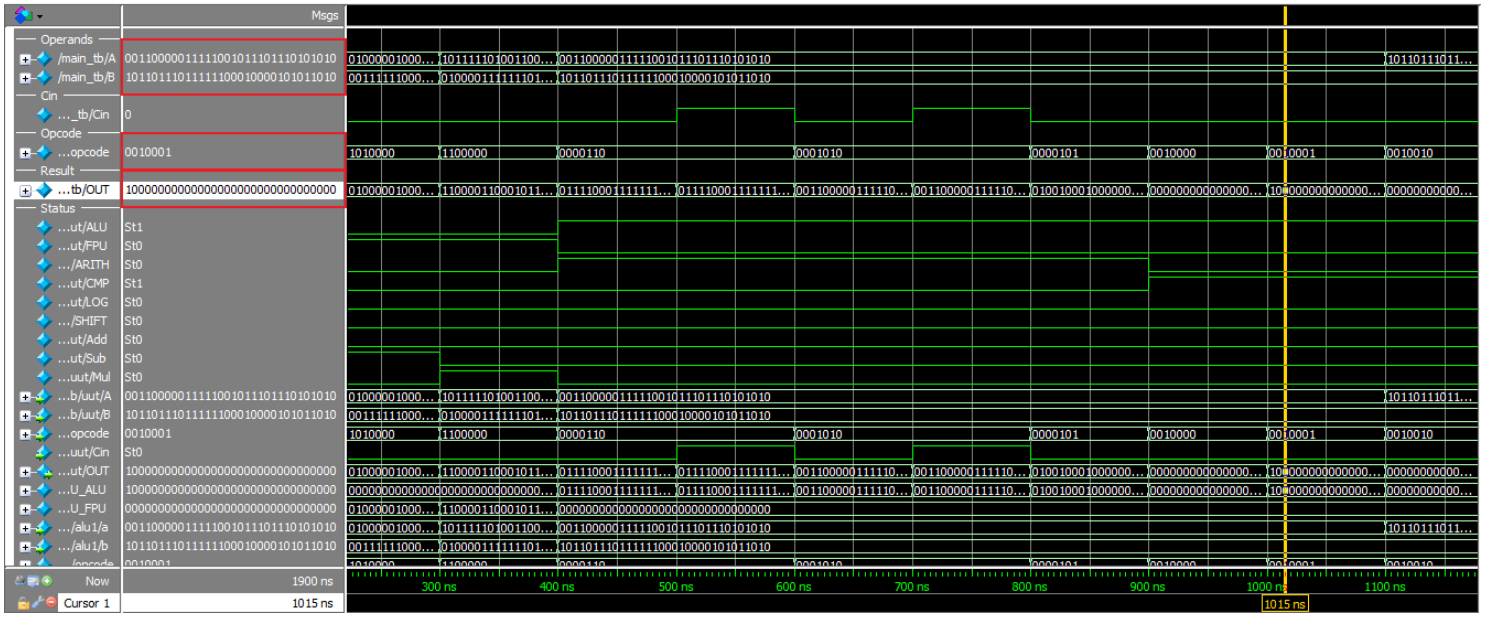

Figure 19: A < B

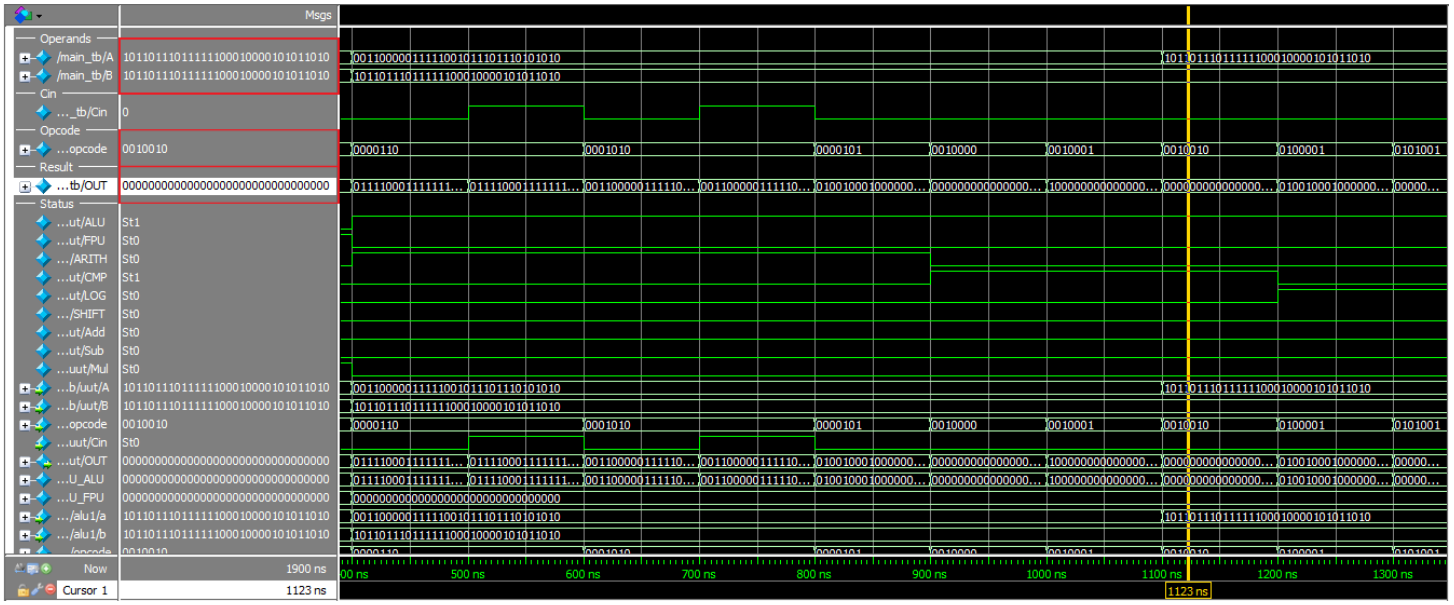

Figure 20: A ≠ B

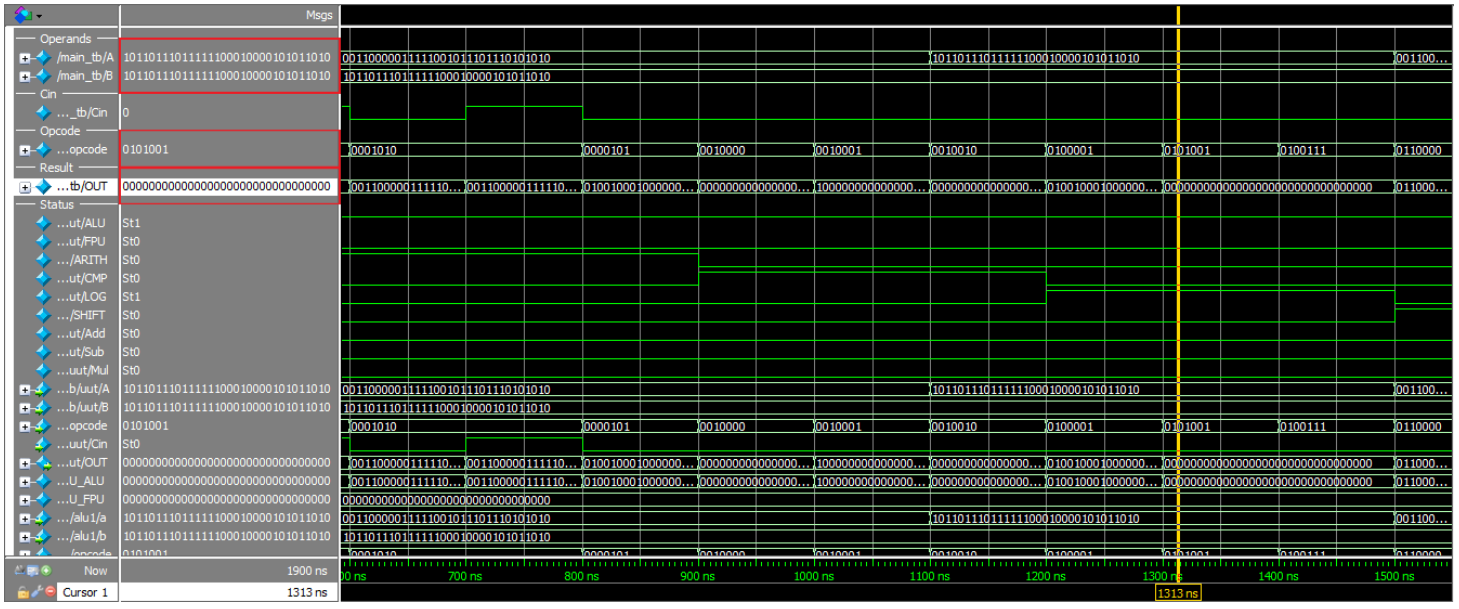

Figure 21: ~A&B

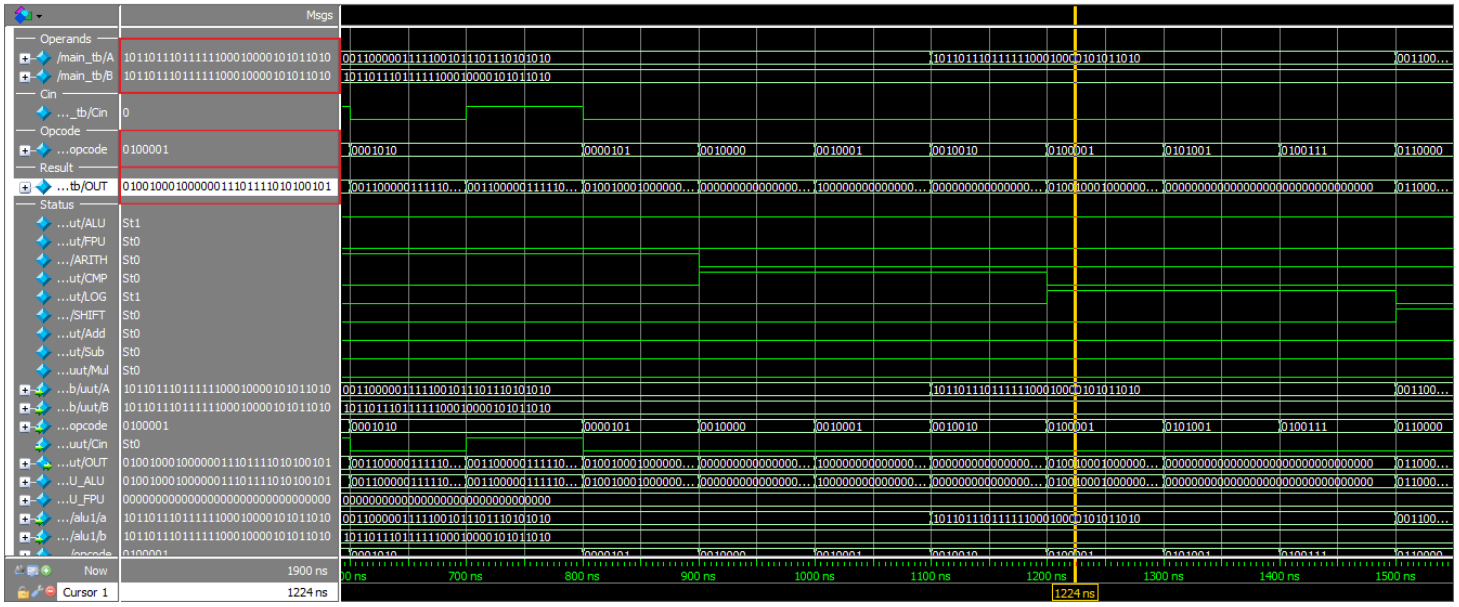

Figure 22: ~(A&B)

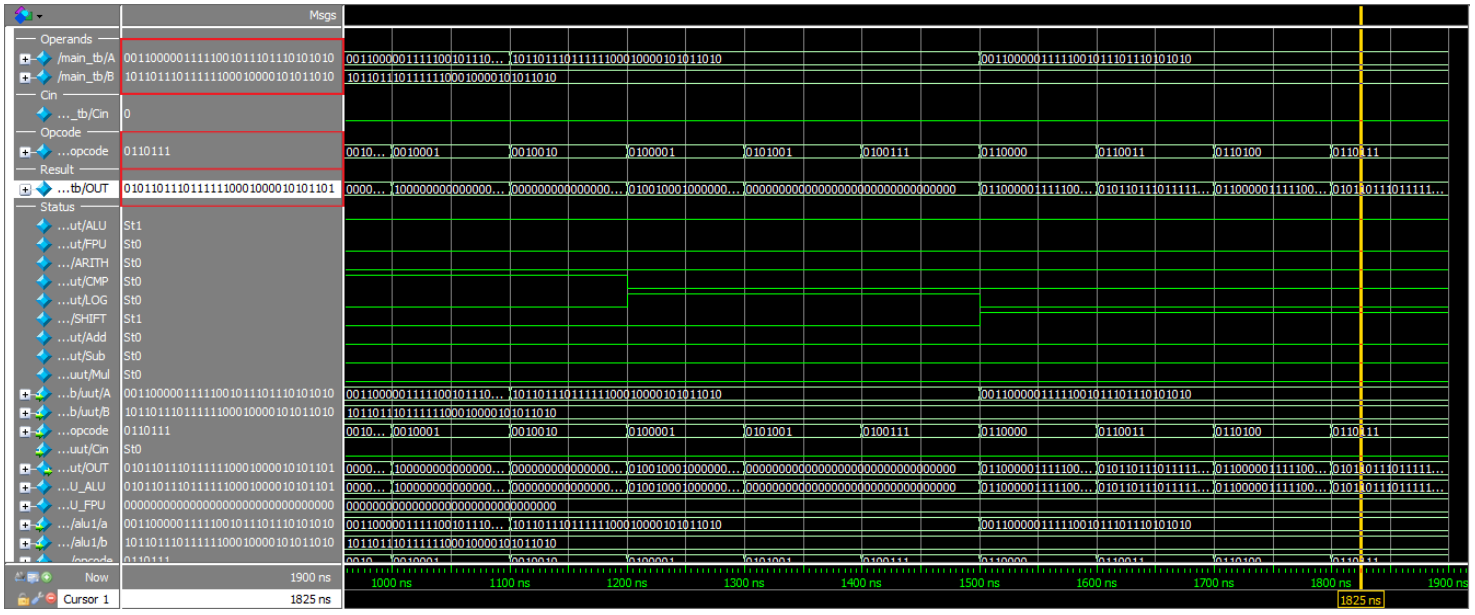

Figure 23: ror B

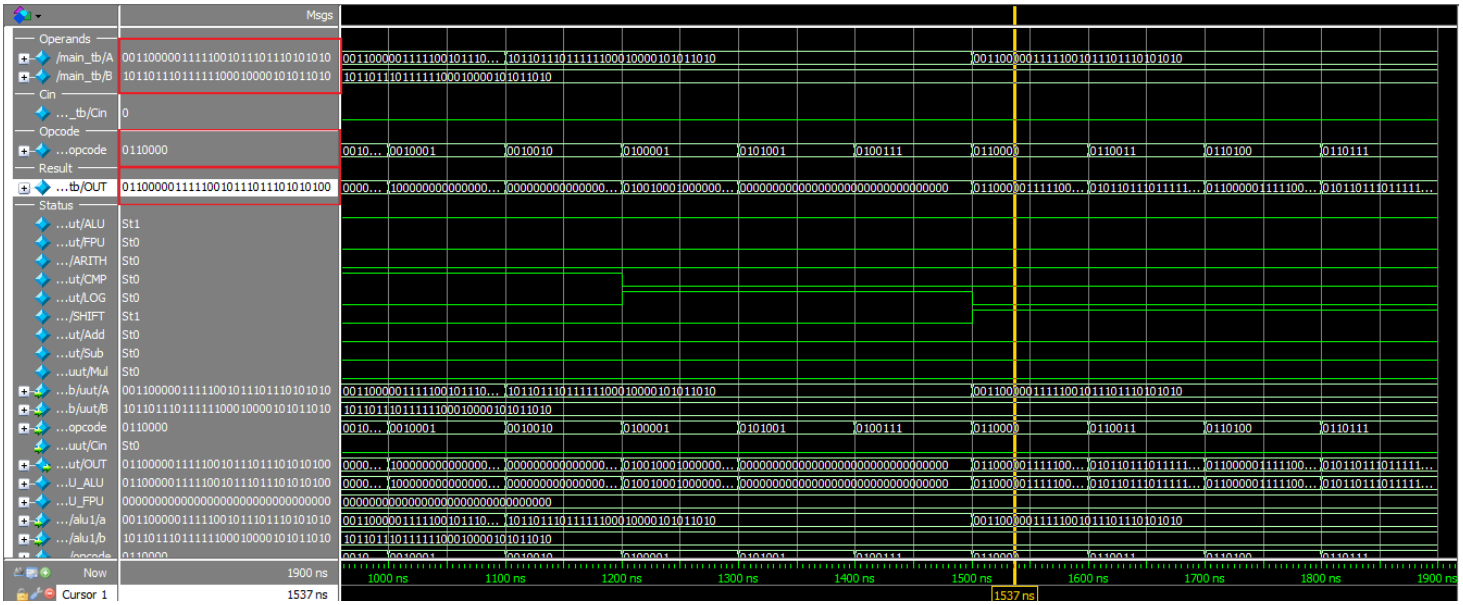

Figure 24: shl A

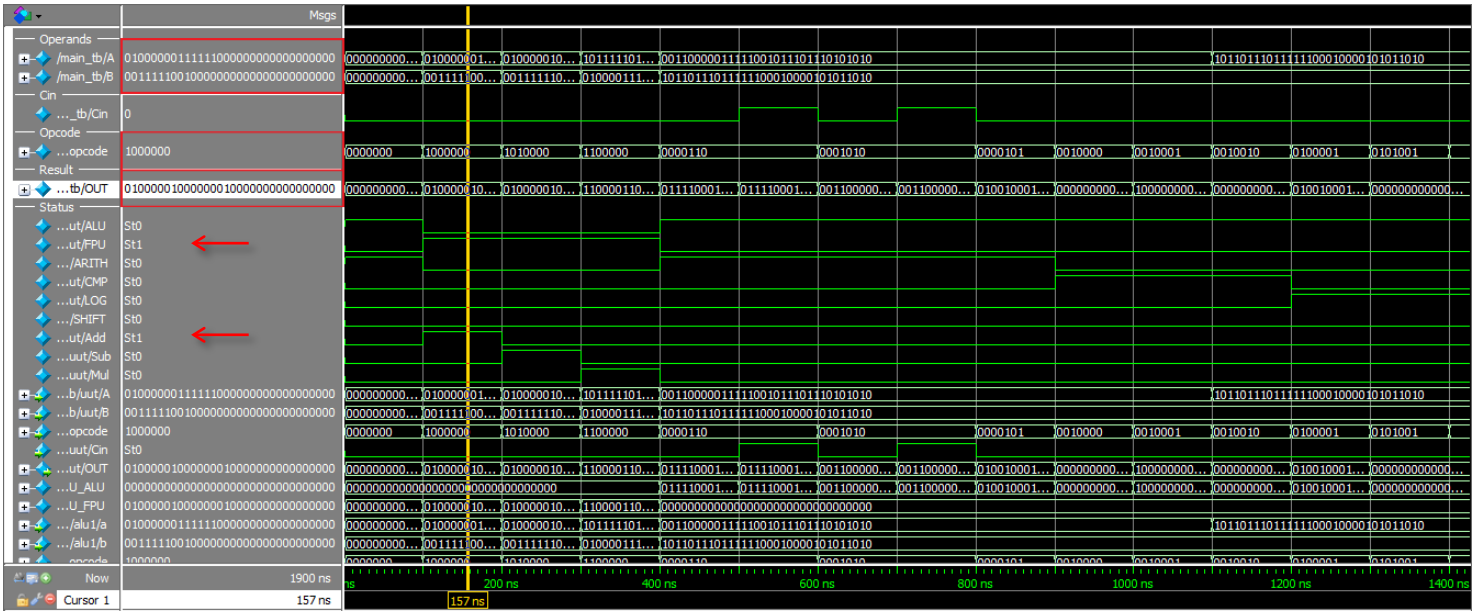

Figure 25: FPU Adder

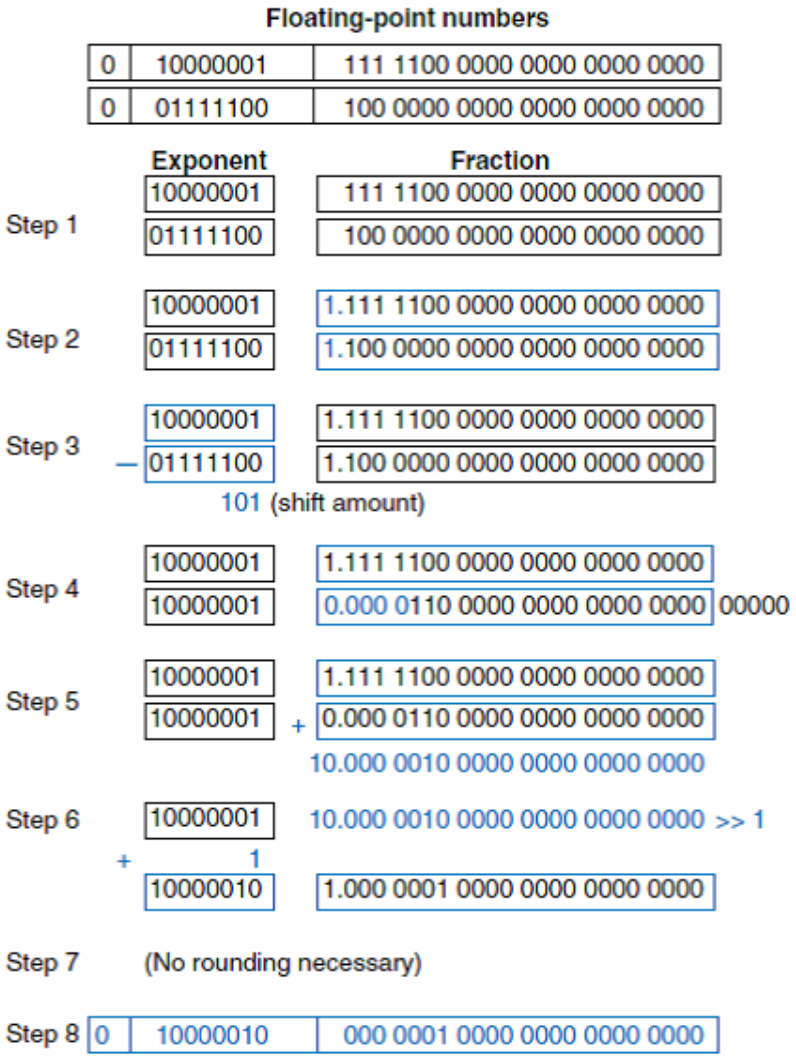

Figure 26: Floating-point Addition

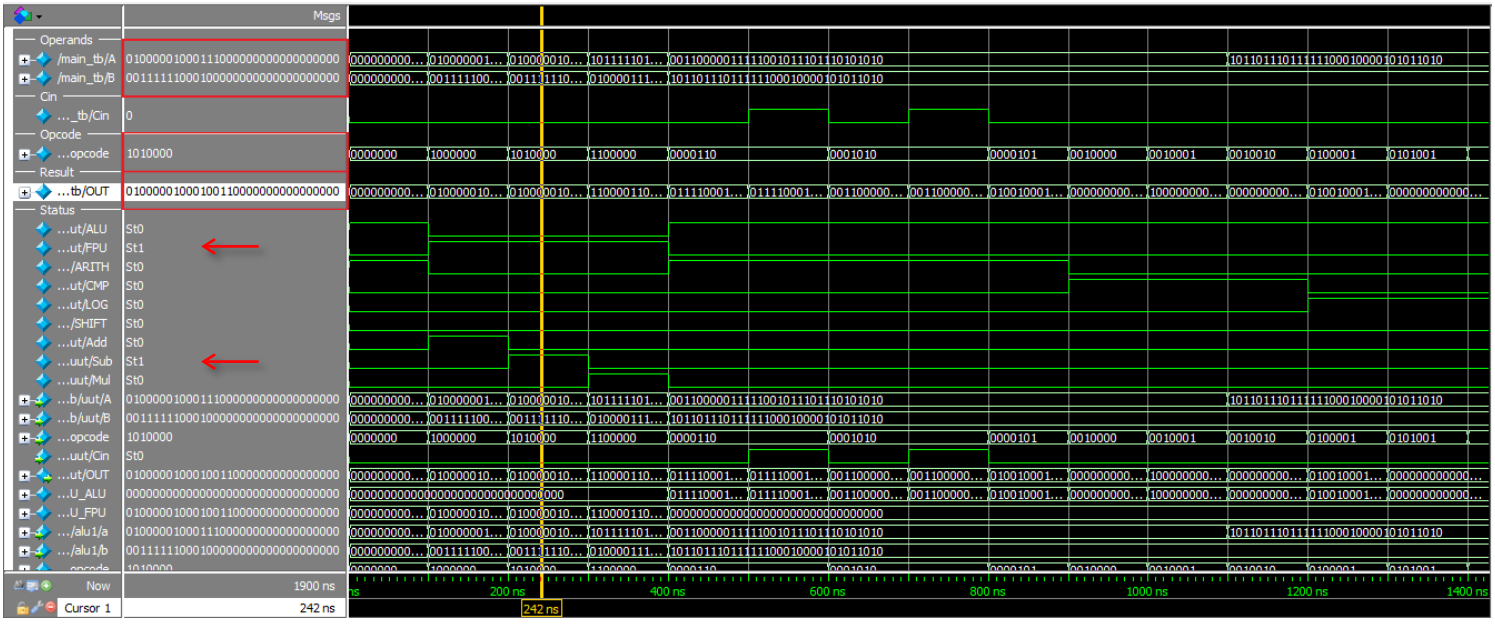

Figure 27: FPU Subtraction

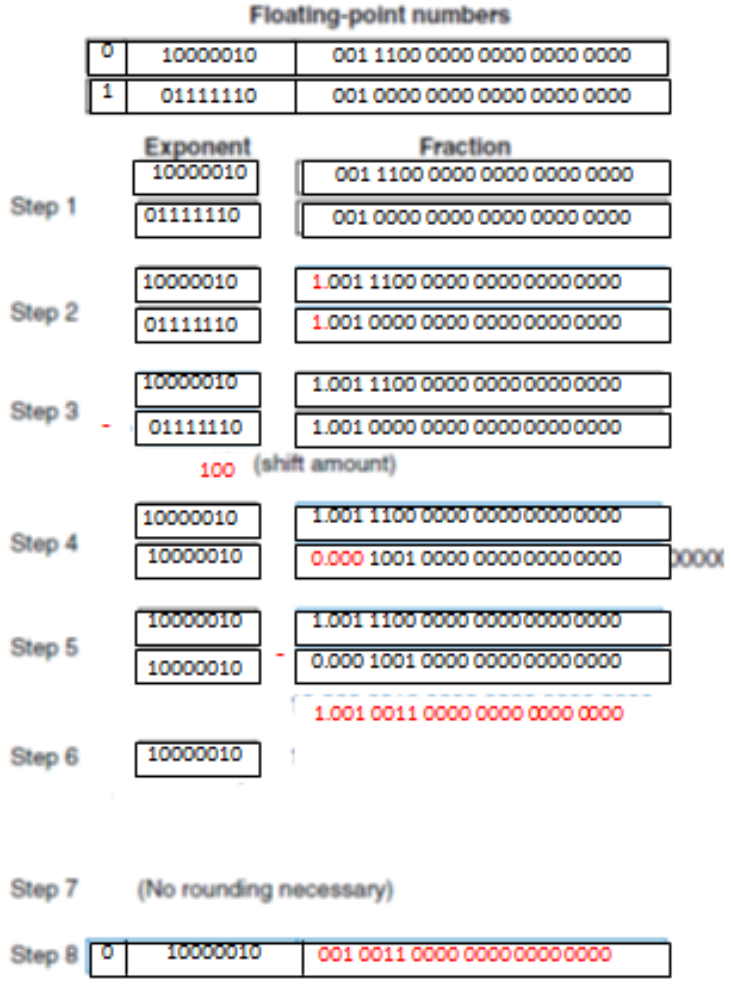

Figure 28: Floating-point Subtraction

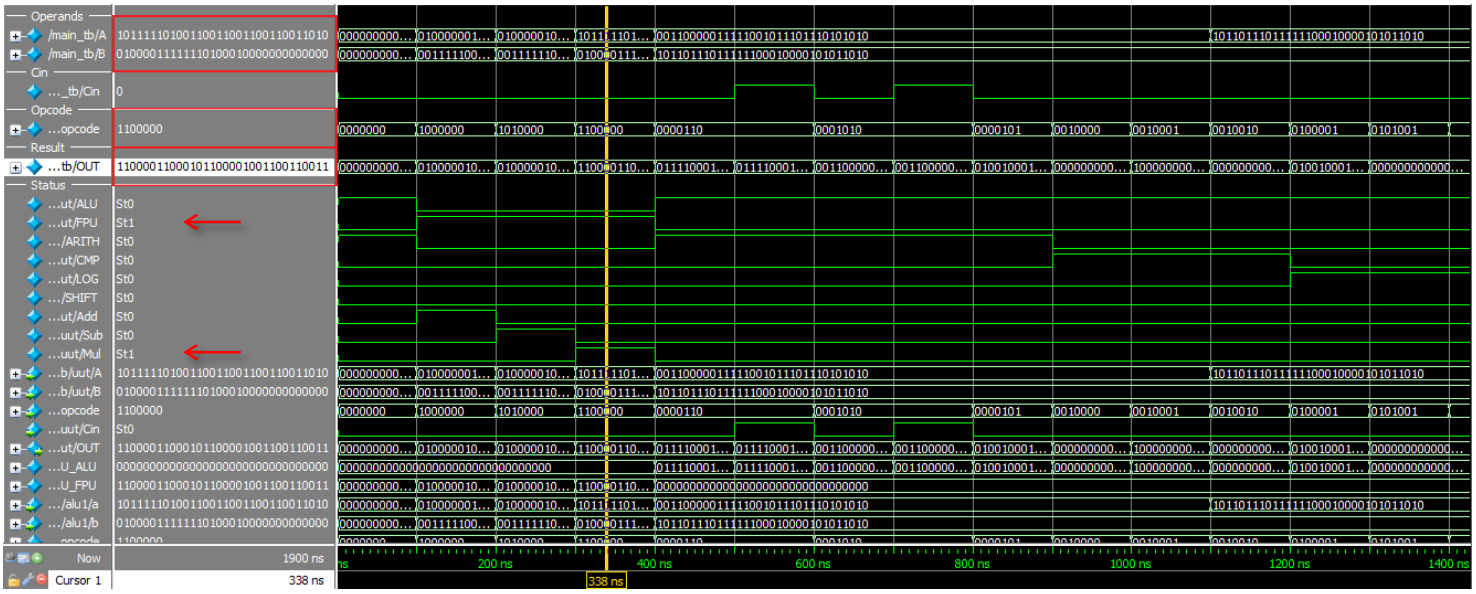

Figure 28: FPU Multiplication

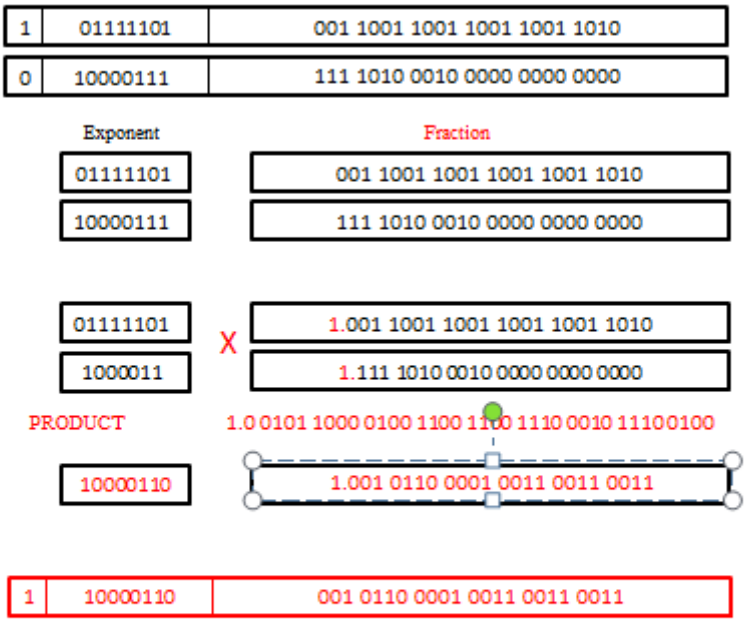

Figure 30: Floating-point multiplicat

5. Conclusion

While working on this project we sought to gain practical experience and knowledge in the field of computer engineering and computer organization. Tools such as ModelSim Simulator, used in this project offer a unique way of design testing and verification by enabling signal.

REFERENCES

1) Stallings, William, *Computer Organization & Architecture: Designing for Performance, 7th ed.* Upper Saddle River, NJ: Prentice-Hall, 2006.

2) Computer Architecture ALU Design: Division and Floating Point, http://www.ann.ece.ufl.edu/courses/eel4713_12spr/slides/Lec8-division.pdf