Project Report

on

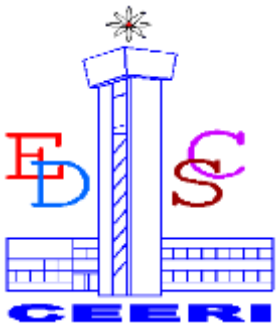
DESIGN AND IMPLEMENTATION OF IEEE 754 COMPATIBLE FLOATING POINT COPROCESSOR ON AN FPGA

BY

Kushagra Shah	2016A3PS0167G
Shreyas Ravishankar	2016AAPS0180H

Prepared in Partial Fulfilment of the Course Practice School -1 at

Central Electronics and Engineering Research Institute, Pilani



A Practice School Station of

Birla Institute of Technology and Science, Pilani

(23rd May – 13th July 2018)

Project Report

on

DESIGN AND IMPLEMENTATION OF IEEE 754 COMPATIBLE FLOATING POINT COPROCESSOR ON AN FPGA

BY

Name	ID No.	B.E. (Hons) in
Kushagra Shah	**2016A3PS0167G**	**Electrical and Electronics Engineering**
Shreyas Ravishankar	**2016AAPS0180H**	**Electronics and Communication Engineering**

Prepared in Partial Fulfilment of the Course Practice School -1 at

Central Electronics and Engineering Research Institute, Pilani

A Practice School Station of

Birla Institute of Technology and Science, Pilani

(23rd May – 13th July 2018)

ACKNOWLEDGEMENTS

We would like to use this opportunity to express our gratitude to everyone who supported us throughout the course of this project. We are thankful for their aspiring guidance, invaluably constructive criticism and friendly advice during the project work. We are sincerely grateful to them for sharing their truthful and illuminating views on the number of issues related to the project.

We would like to thank *Dr. Santanu Chaudhary*, Director of CEERI – Pilani for providing us opportunities to work in such an esteemed research institute. We also thank *Prof. N K Swami*, PS Coordinator and *Mr. Vinod Verma* for their co-operation. We would like to extend our heartfelt gratitude to our PS instructor *Prof. Pawan Sharma* and *Prof. Murugesan S.*, for his constant support, words of wisdom and timely suggestions. We would like to thank the PS Division of BITS Pilani University for taking such a brilliant and necessary initiative.

We would especially like to thank *Dr. Ravi Saini* for giving us this great opportunity of working under them on such an interesting topic.

We would also like to thank everyone else who was directly or indirectly involved to provide us with this opportunity.

Kushagra Shah

Shreyas Ravishankar

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE, PILANI (RAJASTHAN)

Practice School Division

Station……CSIR – CEERI, Pilani…… **Centre**……Pilani……

Duration……50 Days…… **Date of Start**.....May 23, 2018……

Date of Submission……July 13, 2018……..

Title of the Project: Design and Implementation of IEEE 754 Compatible Floating Point Coprocessor on an FPGA

Name, ID No., Discipline of the Students:

Name	ID No.	B.E. (Hons) in
Kushagra Shah	2016A3PS0167G	Electrical and Electronics Engineering
Shreyas Ravishankar	2016AAPS0180H	Electronics and Communication Engineering

Name and Designation of the Expert:

Dr. Ravi Saini, Senior Scientist, IC Design Group

Name of the PS Faculty:

Prof. Pawan Sharma, Prof. Murugesan S.

Key Words: Floating Point Arithmetic, FPGA, IEEE 754, Pipelined Architecture.

Project Areas: Computer Architecture

Abstract: This report gives an outline of the aim of the project and its applications. The aim is to design an optimized architecture of a Floating Point Unit/Coprocessor, and implement it on an FPGA Board. The IEEE 754 floating point standard and general algorithms of floating point operations (addition, subtraction, multiplication, division) will be discussed. Pipelining will also be implemented which gives a higher throughput for multiple inputs.

Signature(s) of Student(s) **Signature of PS Faculty**

Date: **Date:**

TABLE OF CONTENTS

1. INTRODUCTION

1.1. Introduction to CEERI

Organisation Overview:

Central Electronics Engineering Research Institute, Pilani (CEERI) is one of the premier research institutes in India which specialises in the field of Electronics. It was established in the year 1953, under the aegis of *Council of Scientific and Industrial Research* (CSIR). CEERI was established with the objective of nurturing the creative excellence of scientists and collaborating with industry for innovative product development. It envisions building innovative technology in electronics as well as allied sciences and engineering, with high social and strategic impact. Research and development takes place in mainly three fields:

1. Advanced Electronic Systems- Electronic Instrumentation, Industrial Control, Power Electronics, Robotics, Image Processing etc.
2. Advanced Semiconductor Electronics - Micro-Electro-Mechanical Systems, Micro sensors, Very Large Scale Integration Design (Digital, Analog, Mixed Signal), Photonic Devices etc.
3. Microwave Tubes - Klystron, Magnetron, Travelling Wave Tubes, Gyrotron, Plasma Tubes, Terahertz devices etc.

Since our work mainly involved FPGAs, it is appropriate to discuss more about the Integrated Systems Department (under the Cyber Physical Systems Group).

Integrated Systems:

Integrated Systems Group has been involved in the design and development of VLSI integrated circuits, spanning in scope from microprocessor design to text-to-speech synthesis and image/video processing applications. The group has experience with both FPGA and ASIC platforms.

The IP core based SOC design and implementation, based on open source IP cores has become a core interest of the group recently. Design and implementation is done using various EDA tools and the technology foundry libraries with academic licenses whereas the fabrication, assembly and packaging is executed through a MPW facility under academic prototyping scheme. The tie-up with complete range of IC Design infrastructure companies, namely, EDA tool vendors and foundries, has built a strong eco-system of chip based system development in the nation and provided the means of proliferating the same across academic institutes.

1.2. Introduction to the Project

A *Floating-Point Unit* is a part of a computer system specially designed to carry out operations on floating point numbers. The unit may be in-built in the computer system or it may be a coprocessor. Generally, the operations performed are addition, subtraction, multiplication, division, square root, and bit shifting.

The aim of this project is to implement a *floating point* coprocessor on an *FPGA* that can perform the operations: addition, subtraction, multiplication, and division.

A *Field-Programmable Gate Array* (FPGA) is an integrated circuit designed to be configured by a customer or a designer after manufacturing – hence "field-programmable". The FPGA configuration is generally specified using a *Hardware Description Language* (HDL). FPGAs contain an array of programmable logic blocks, and a hierarchy of reconfigurable interconnects that allow the blocks to be "wired together", like many logic gates that can be inter-wired in different configurations.

Verilog HDL is used to design the RTL (*Register Transfer Level*) of this project. The simulations are done on the software *ModelSim Altera* for the verification of results.

The floating point numbers are represented using the *single precision IEEE 754 standard* throughout the project. It is a technical standard for floating-point computation established in 1985 by the Institute of Electrical and Electronics Engineers (IEEE). The standard addressed many problems found in the diverse floating point implementations that made them difficult to use reliably and portably.

The basic architecture for *addition, subtraction, multiplication, and division* has been defined according to standard algorithms used worldwide. The *non-pipelined* as well as *pipelined* architectures have been developed.

In computing terminology, a *pipeline*, is a set of data processing elements connected in series, where the output of one element is the input of the next one. The elements of a pipeline are often executed in parallel or in time-sliced fashion. Some amount of buffer storage is often inserted between elements.

The software *Xilinx Vivado Design Suite* has been used for synthesis and implementation of the algorithms on an FPGA (Xilinx ZedBoard Zynq Evaluation and Development Kit).

Finally, all the algorithms have been combined under one roof to work as a *Floating Point Unit*. Also, a *comparison* is made between the non-pipelined and pipelined architectures.

2. BACKGROUND

2.1. Floating Point Unit

When a CPU executes a program that is calling for a floating-point (FP) operation, there are three ways by which it can carry out the operation:

1. It may call a *floating-point unit emulator* (which is a floating-point library) using a series of simple fixed-point arithmetic operations which can run on the integer ALU. These emulators can save the added hardware cost of a FPU but are significantly slow.
2. It may use *add-on FPUs* (coprocessor) that are entirely separate from the CPU, and are typically sold as an optional add-ons which are purchased only when they are needed to speed up math-intensive operations.
3. It may use an *integrated FPU* present in the system.

The FPU implemented in this project is a *single precision IEEE 754 compliant unit*. It can handle basic floating point operations like addition, subtraction, multiplication and division. This is then implemented on an FPGA.

2.2. Field-Programmable Gate Array (FPGA)

FPGA is a type of device that is widely used in the logic or digital electronic circuits. FPGAs are semiconductor devices that contain programmable logic and interconnections. The programmable logic components, or logic blocks as they are known, may consist of anything from logic gates, through to memory elements or blocks of memories, or almost any element. An FPGA has many sub parts which contribute to its flexibility (shown in the Figure-1).

Configurable Logic Blocks (CLB):

The logic blocks can be configured to perform different operations. The configuration of each logic block may be different which enable function execution in parallel. Through the I/O pins one can interface these applications with other devices. Ex- A Digital Signal Processor.

CLB, in general, at the fundamental level are made of Look up Tables (LUTs), Flip Flops and MUXes.

Figure-1

An FPGA Board

I/O Blocks: The I/O pins are very flexible and can be made to act as input, output and tristate. They can also contain flip flops and other complicated elements.

The FPGA has volatile memory. It cannot store programmable logic on its own. The config block on the FPGA (which contains the configuration logic) must be attached to an external memory. It can be flash memory. When an FPGA is turned on, the config block loads the configuration onto the FPGA.

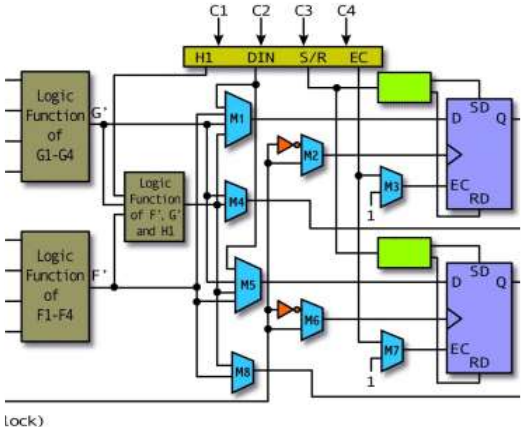


Figure-2

Block Diagram of an FPGA

Advantages of FPGAs:

1. They are capable of executing virtually any logical function.
2. They are fast compared to microprocessors.
3. They are field programmable unlike ASICs.
4. They are massively parallel.
5. High I/O count for efficient interfacing.

Disadvantages of FPGAs

1. They are expensive.
2. They consume high power.
3. They have volatile memory.
4. They require use of complicated tools.
5. HDLs are not easy to learn, and are non-intuitive.
6. They are hard to choose and compare for the application desired.

2.3. IEEE 754 Floating Point Standard

Single-precision Floating-Point format is a computer number format, usually occupying *32 bits* in computer memory. It represents a wide dynamic range of numeric values by using a floating radix point.

The IEEE 754 standard specifies a 32 bit binary number having:

- **Sign (1 bit) -** The sign bit represents the sign of the number, 0 for positive and 1 for negative.
- **Exponent Width (8 bits) -** The exponent bits are the biased exponent of the given floating point number when written in scientific notation. Biased exponent implies adding 127 to the exponent. This is done so that negative exponents can be stored as positive values in memory.
- **Significand (23 bits) -** The significand of the floating point number is a 24 bit value where the MSB is always 1 and hence not explicitly mentioned. It only contains the 23 bits which is the rounded mantissa of the number.

A floating point decimal number is first converted to its equivalent binary number using the standard method. This binary number is then represented in the scientific notation i.e.
Number = $(-1)^s$ x 1.M x $2^{(e - bias)}$
This is then converted to a 32 bit IEEE 754 format number as shown in Figure - 3.

The value of bias for an 8 bit exponent is:
[$2^{(8-1)}$ -1 = 127]

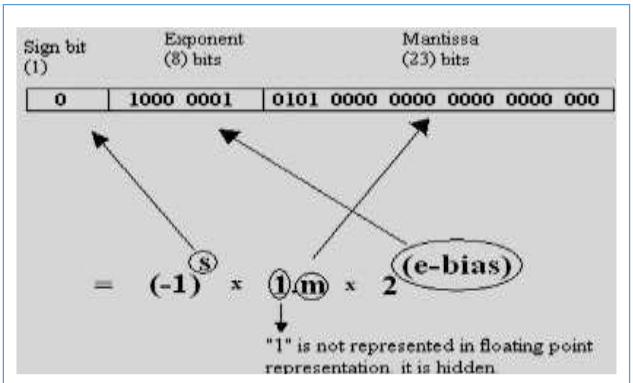


Figure - 3
Scientific to IEEE 754 notation

Ex- 286.75_{10} can be converted to the IEEE 754 format using the above steps:

0	1000 0111	0001 1110 1100 0000 0000 000
Sign bit (1)	Exponent (8) bits	Mantissa (23) bits

2.4. FPGA Design Flow

The following steps are followed for implementing any algorithm on an FPGA:

1. **High Level RTL Design**- Register-Transfer Level (RTL) is a design abstraction which models a synchronous digital circuit in terms of the flow of digital signals between hardware registers, and the logical operations performed on those signals. RTL abstraction is used to create high-level representations of a circuit from which lower-level representations and ultimately actual wiring can be derived.
2. **Functional Simulation**- RTL design is checked for functionality by forcing values to the inputs, clocks and other parameters. The output waveform of the design (obtained on *Modelsim Altera*) should match with the correct results. Otherwise, the RTL code is corrected for errors.
3. **Synthesis and Mapping**- Synthesis is a process by which an abstract form of desired circuit behaviour, typically at RTL, is turned into a design implementation in terms of logic gates by a synthesis tool (such as *Vivado design suite*).
4. **Placing and Routing**- The first process, *placing* involves deciding where to place all electronic components, and logic elements in a limited amount of space. This is followed by *routing*, which decides the exact design of all the wires needed to connect the placed components. This step must implement all the desired connections considering all the rules and limitations of the manufacturing process.
5. **Timing Analysis-** Static Timing Analysis is done to:
   a) Verify that timing requirements are met for all paths in the design.
   b) Analyse setup and hold performance for all constrained paths in the design.
   c) Verify that the operational frequencies are within component performance limits.

   Static Timing Analysis (STA) computes the expected timing of a digital circuit without requiring a simulation of the full circuit. Static timing analysis helps in measuring the circuit timing in a fast and reasonably accurate manner. The speedup comes from the use of simplified timing models and by mostly ignoring logical interactions in circuits.

6. **Bitstream Generation** and deploying it onto the FPGA- The term bitstream is used to describe the configuration data to be loaded into an FPGA. The detailed format of the bitstream for a particular FPGA chip is usually considered proprietary to the FPGA vendor.

3. FLOATING POINT ARITHMETIC

Floating point arithmetic differs from integer arithmetic because of a different representation of a floating point number in a computing system. Before discussing the algorithms for floating point arithmetic, we will discuss a **few issues:**

3.1. Rounding

One of the following standard methods can be used for rounding:

1. Round towards zero:
    a. Figure out how many bits (digits) are available. Take that many bits (digits) for the result and throw away the rest. This has the effect of making the value represented closer to 0.0
2. Round towards positive infinity:
    a. Regardless of the value, round towards +infinity.
3. Round towards negative infinity:
    a. Regardless of the value, round towards -infinity.
4. Round to nearest:
    a. Use representation NEAREST to the desired value. This works fine in all but 1 case, where the desired value is exactly half way between the two possible representations.
    b. The half way case: When "1000..." appears to the right of the number of digits to be kept, then use the representation that has zero as its least significant bit.

The rounding method used in this project is 'round to the nearest' technique.

3.2. Special Numbers

1. **Zero** - Zero can be represented in two ways in the IEEE 754 format with a positive or negative sign.
2. **Infinity**- Results caused during *overflows* or when carrying out division by zero can be infinite valued. Positive infinity is represented by 7FF0000000000000 and negative infinity is represented by FFF0000000000000.
3. **NAN** (not a number) - These represent results of operations that cannot be represented as a number. Some examples are $(\infty - \infty)$, $(-\infty + \infty)$, $(0 \times \infty)$, $(0 \div 0)$, $(\infty \div \infty)$, and imaginary numbers. When a NAN is encountered one can choose to raise an invalid operation flag or we can propagate the error without signalling an exception, which are called *signalling* and *quiet* NANs respectively. A signalling Nan is represented by any bit pattern between 7F800001 and 7FBFFFFF or between FF800001 and FFBFFFFF. A quiet Nan, on the other hand is represented by any bit pattern between 7FC00000 and 7FFFFFFF or between FFC00000 and FFFFFFFF.

4. **Normal Number** - Ordinary numbers that can be represented in IEEE 754 format.
5. **Subnormal Number**- Handling subnormal numbers is tricky as they pose some problems during *rounding*. Values which are very close to zero cannot be represented accurately. In the IEEE 754 standard, numbers less than 1.0×2^{Emin} are represented using significand less than 1. This is called *gradual underflow* as the numbers gradually lose their significance as the magnitude decreases. Ex- consider a number 'x' in base 10 with four significant figures: $x = 1.234 \times 10^{Emin}$. Then (x/10) will be rounded to 0.123×10^{Emin}, having lost a digit of precision. Similarly, (x/100) rounds to 0.012×10^{Emin}, and (x/1000) to 0.001×10^{Emin}. While (x/10000) is finally small enough to be rounded to 0.

3.3. Exceptions

IEEE specifies the following exceptions that can occur during any operation:

1. **Invalid Operation**- This exception is raised if the result is a *NAN*. It is considered to be invalid.
2. **Overflow**- This exception is raised when the result is too large to be represented by the given number of bits. The resultant exponent is checked whether it is greater than the maximum possible biased exponent (here, 127+127). If so, it is then *rounded off to infinity*.
3. **Divide By Zero**- This exception is raised when an attempt is made to divide a number by zero. The result is *rounded off to infinity* with a negative or positive sign based on the sign of the dividend and divisor.
4. **Underflow**- This exception is raised when the number is too small to be represented by the given number of bits i.e. *subnormal numbers*. If the exponent is less than the minimum possible biased exponent (here, 127-126), then there is an underflow.
5. **Inexact**- This exception is raised when the result cannot be exactly represented by floating point numbers.

4. FLOATING POINT OPERATIONS – ALGORITHMS

4.1. Addition Algorithm

The two floating point numbers to be added are: X1 and X2. And the result is stored as X3.
$X3 = X1 + X2 = [(-1)^{s1} \times M1 \times 2^{e1}] + [(-1)^{s2} \times M2 \times 2^{e2}]$
Where:

- s1, s2 are the sign bits of number X1 & X2.
- e1, e2 are the (biased) exponent bits of number X1 & X2.
- M1, M2 are the mantissa bits of number X1 & X2. (the implicit 1 is omitted here)

The **primary logic** here can be explained using a special case:
Suppose two positive numbers X1 and X2 are to be added, and suppose e1>e2. Then the result,
$X3 = [M1 \times 2^{e1}] + [M2 \times 2^{e2}] = [M1 + (M2 \times 2^{e1-e2})] \times 2^{e1}$

The significand is the sum of M1 and shifted M2. The exponent is e1. The sign can be computed separately. The necessary normalisations have to be done.

The **complete algorithm** is as follows:

1. If e1< e2, swap the operands. This ensures that the difference of the exponents satisfies d = e1 − e2 ≥ 0. Tentatively set the exponent of the result to e1.
2. If the signs of a1 and a2 differ, replace s2 by its two's complement.
3. Place s2 in a p-bit register and shift it d = e1 − e2 places to the right (shifting in 1's if s2 was complemented in the previous step). From the bits shifted out, set g to the most-significant bit, r to the next most-significant bit, and set sticky to the OR of the rest.
4. Compute a preliminary significand S = s1 + s2 by adding s1 to the p-bit register containing s2. If the signs of a1 and a2 are different, the most-significant bit of S is 1, and there was no carry-out, then S is negative. Replace S with its two's complement. This can only happen when d = 0.
5. Shift S as follows. If the signs of a1 and a2 are the same and there was a carryout in step 4, shift S right by one, filling in the high-order position with 1 (the carry-out). Otherwise shift it left until it is normalized. When left-shifting, on the first shift fill in the low-order position with the g bit. After that, shift in zeros. Adjust the exponent of the result accordingly. 6. Adjust r and s. If S was shifted right in step 5, set r := low-order bit of S before shifting and s := g OR r OR s. If there was no shift, set r := g, s := r OR s. If there was a single left shift, don't change r and s. If there were two or more left shifts, r := 0, s := 0.
6. Rounding is as shown in the table below (p0 is the LSB of the register):

Rounding mode	Sign of result>=0	Sign of the result<0
Nearest	+1 if (r and p0) or (R and S)	+1 if (r and p0) or (R and S)

7. The final sign is decided based on the table given below-

Swap(Step 1)	Complement(Step 4)	Sign1	Sign2	Sign result
Yes	Don't Care	+	-	-
Yes	Don't Care	-	+	+
Yes	No	+	-	+
No	No	-	+	-
No	Yes	+	-	-
No	Yes	-	+	+

4.2. Subtraction Algorithm

The two floating point numbers to be subtracted are: X1 and X2. And the result is stored as X3.
$X3 = X1 - X2 = [(-1)^{s1} \times M1 \times 2^{e1}] - [(-1)^{s2} \times M2 \times 2^{e2}]$

Where:

- s1, s2 are the sign bits of number X1 & X2.
- e1, e2 are the (biased) exponent bits of number X1 & X2.
- M1, M2 are the mantissa bits of number X1 & X2. (the implicit 1 is omitted here)

The **primary logic** here is the similar to the one used for the addition algorithm:

For performing the operation [X1 – X2], we write it as [X1 + (-X2)], i.e. we first invert the sign the second number and then add it to the first number using the above mentioned addition algorithm.

4.3. Multiplication Algorithm

The two floating point numbers to be multiplied are: X1 and X2. And the result is stored as X3.
$X3 = X1 \times X2 = [(-1)^{s1} \times M1 \times 2^{e1}] \times [(-1)^{s2} \times M2 \times 2^{e2}]$

Where:

- s1, s2 are the sign bits of number X1 & X2.
- e1, e2 are the (biased) exponent bits of number X1 & X2.

M1, M2 are the mantissa bits of number X1 & X2. (the implicit 1 is omitted here)

The **primary logic** here is:

$X3 = [(-1)^{s1} \times M1 \times 2^{e1}] \times [(-1)^{s2} \times M2 \times 2^{e2}] = (-1)^{[s1 \text{ xor } s2]} \times [M1 \times M2] \times 2^{[e1 + e2]}$

The significand, exponent and sign can be computed separately. The necessary normalisations have to be made.

The **complete algorithm** is as follows:

1) The final sign is decided as [S1 xor S2].
2) For the final exponent, the sum of the 2 exponents is computed and the bias is subtracted from it. [E1+E2-127]
3) The product computed directly can be at most 48 bits. For rounding the steps given below are followed:
   a) The most significant and the least significant 24 bits are stored in P and A registers respectively.
   b) Let s represent the sticky bit, g (for guard) the most-significant bit of A, and r (for round) the second most-significant bit of A. The sticky bit will be calculated by OR operation on the rest of the bits of A. The 2 cases which arise are:
      i) The high-order bit of P is 0. Shift P left 1 bit, shifting in the g bit from A. Shifting the rest of A is not necessary.
      ii) The high-order bit of P is 1. Set s := s ∨ r and r := g, and add 1 to the exponent.
   c) Then, rounding is done as shown in the figure below (p0 is the LSB of the register):

Rounding mode	Sign of result>=0	Sign of the result<0
Nearest	+1 if (r and p0) or (R and S)	+1 if (r and p0) or (R and S)

4.4. Division Algorithm

The two floating point numbers to be divided are: X1 and X2. And the result is stored as X3.
$X3 = X1 / X2 = [(-1)^{s1} \times M1 \times 2^{e1}] / [(-1)^{s2} \times M2 \times 2^{e2}]$

Where:

- s1, s2 are the sign bits of number X1 & X2.
- e1, e2 are the (biased) exponent bits of number X1 & X2.

M1, M2 are the mantissa bits of number X1 & X2. (the implicit 1 is omitted here)

The **primary logic** here is:

$X3 = [(-1)^{s1} \times M1 \times 2^{e1}] / [(-1)^{s2} \times M2 \times 2^{e2}] = (-1)^{[s1 \text{ xor } s2]} \times [M1 / M2] \times 2^{[e1 - e2]}$

The significand, exponent and sign can be computed separately. The necessary normalisations have to be made. For calculating (M1/M2), three methods were tried:

4.4.1. Using the Divide Operator

The result after the division operation has to be a floating point number, but the divide operator in *Verilog* gives an integer result.

To tackle this problem, the dividend is multiplied by an appropriate power of 2 (i.e. shifted left filling the lower bits with 0 by appropriate number of bits). The significand is a 24 bit value, hence a shift by 24 bits is made. Ex- 13/3=4, but 1300/3=433 for an accuracy of two decimal places.

4.4.2. Using the Non-Restoring Division Algorithm

The result after the division operation has to be a floating point number, but the *Non-Restoring Division* algorithm gives an integer result.

This problem is also tackled using the same methodology as before- the dividend is shifted by 24 bits. The Non-Restoring Division algorithm is as follows:

1. Let register A (of size N) store the dividend and register B store the divisor.
2. Consider a register P of size equal to N+1. Initialize it to zero.
3. Now perform the following steps N times:
    a. If P is negative, shift the register (P, A) one bit to the left and add the contents of B to P. Store the result back in P.
    b. If P is non negative, then shift (P, A) one bit to the left and subtract the contents of B from P. Store the result back in P.
    c. If P is negative, set lower order but A to 0 otherwise, set it to 1.
4. After N iterations, A is the quotient. If P is non negative, then P is the remainder, else (P+B) gives the value of the remainder.

4.4.3. Using the Newton-Raphson Division Algorithm

The result of the *Newton-Raphson algorithm* is a floating point number. So, there is no need of an initial shift operation. Suppose N is the dividend and D is the divisor in the following algorithm:

1. Scale N and D such that [0.5 <D < 1]. This is done by dividing N and D by $2^{(exp2 +1)}$.
2. Get the initial approximation for (1/D) given by the expression: (48/17) - (32/17)*D.
3. Subsequent approximations of D are given by the *recursive formula*:
   $X_{i+1} = X_i*(2 - D*X_i)$, where X = (1/D) and sub 'i' corresponds to the i^{th} approximation.
4. Calculate X_n, where n is chosen such that sufficient accuracy is obtained. Multiply N and X_n to obtain an approximated significand of the result.

*The **Non-Restoring Algorithm**'s approach has been followed in this project due to various factors such as simpler logic, lesser hardware, more accurate results etc.*

5. PIPELINED ARCHITECTURE

Pipelining is the process of accumulating instruction from the processor through a pipeline. It allows storing and executing instructions in an orderly process. It is also known as *pipeline processing*.

Pipelining is a technique where multiple instructions are overlapped during execution. Pipeline is divided into stages and these stages are connected with one another to form a pipe like structure. Instructions enter from one end and exit from another end. Pipelining increases the overall instruction throughput.

In a pipelined system, each segment consists of an input register followed by a combinational circuit. The register is used to hold data and combinational circuit performs operations on it. The output of combinational circuit is applied to the input register of the next segment.

Two important terms encountered while designing a pipelined architecture are:

- **Latency**- Time delay from when the input is established until the output associated with that input becomes valid.
- **Throughput** – Rate at which inputs are processed to give outputs.

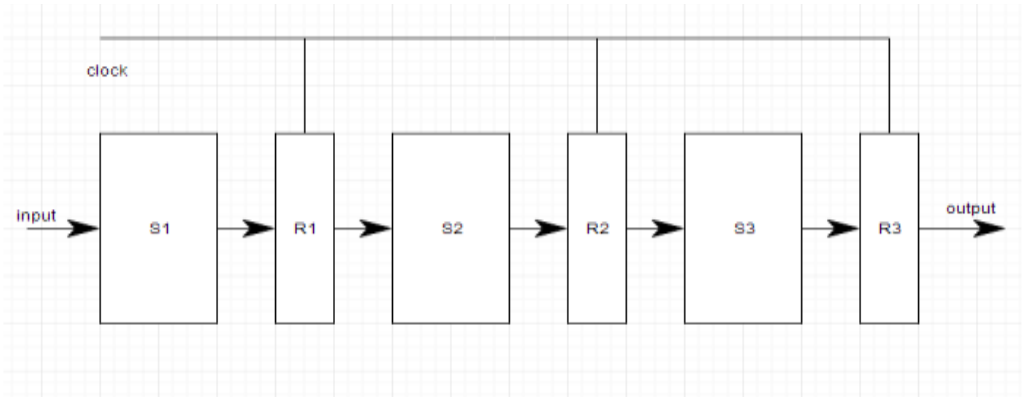


Figure - 4

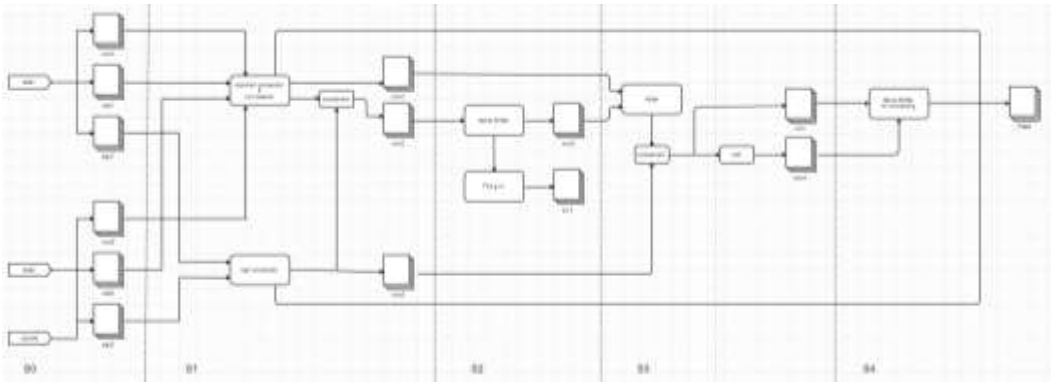
Pipelined Architecture

6. BLOCK DIAGRAMS OF THE ARCHITECTURE

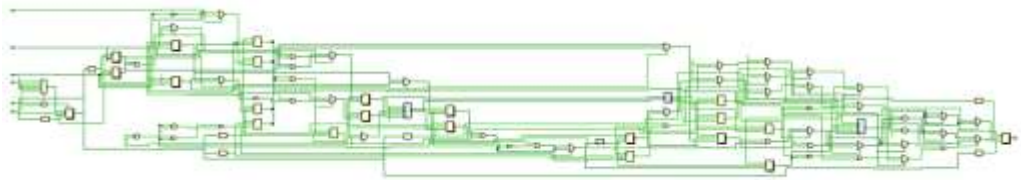
Before writing the Verilog codes for any algorithm, the architecture of the algorithm is designed. A block diagram is made which is then compared with the RTL schematic obtained from the synthesis tool (Vivado). This is done to verify if the desired architecture is implemented by the written Verilog code. It can be seen that the designed architectures match with the obtained RTL schematic on looking closely.

6.1. Non-Pipelined Addition/Subtraction

Designed Architecture

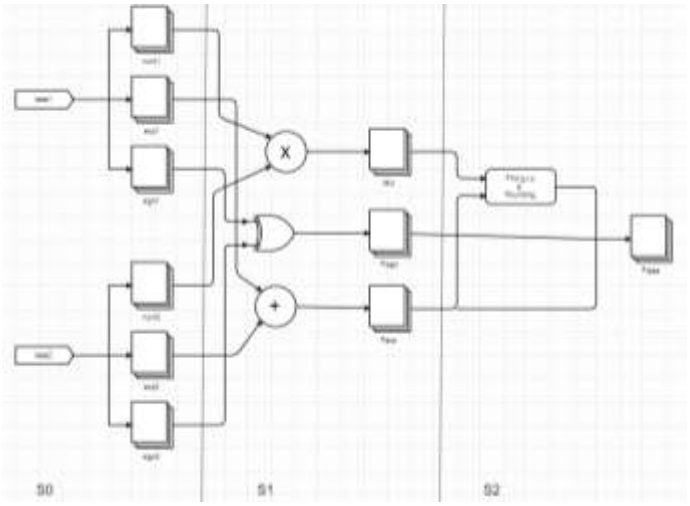


RTL Schematic

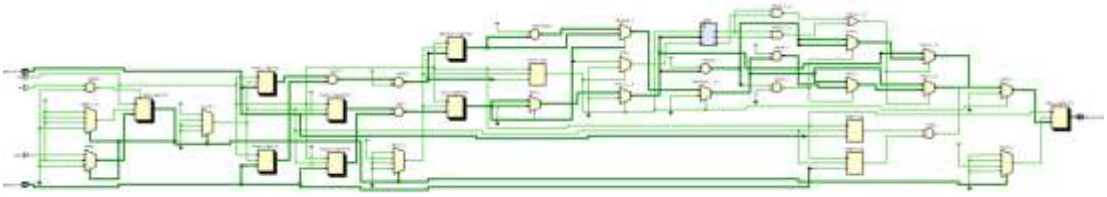


6.2. Non-Pipelined Multiplication

Designed Architecture

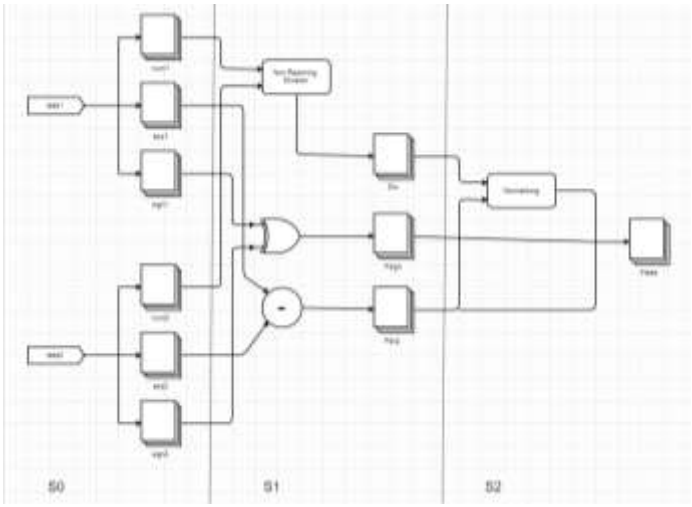


RTL Schematic

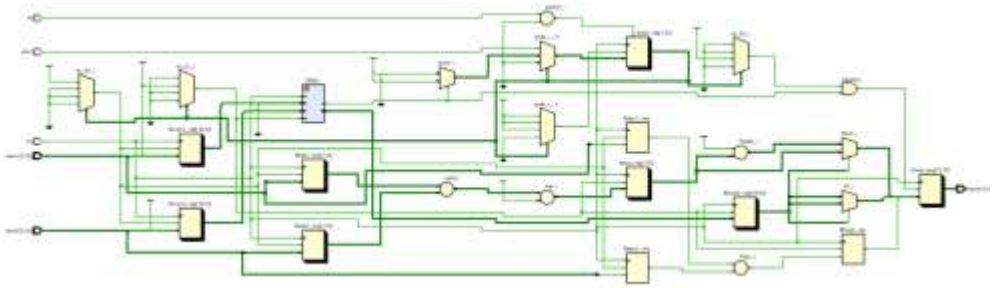


6.3. Non-Pipelined Division

Designed Architecture

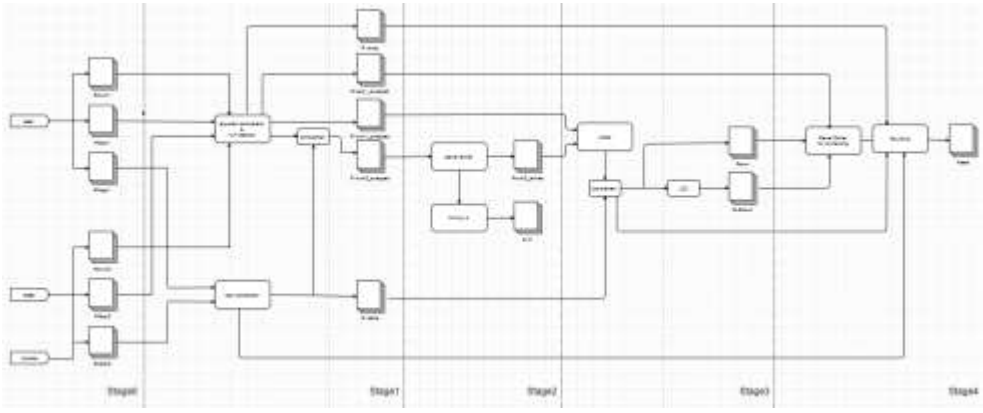


RTL Schematic

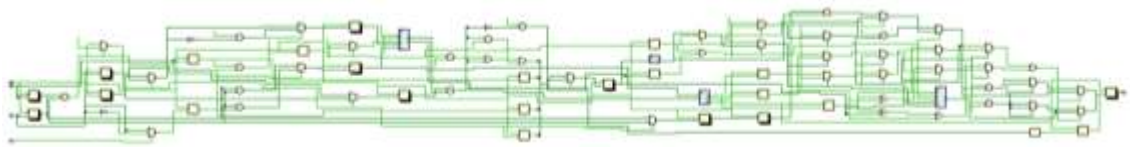


6.4. Pipelined Addition/Subtraction

Designed Architecture

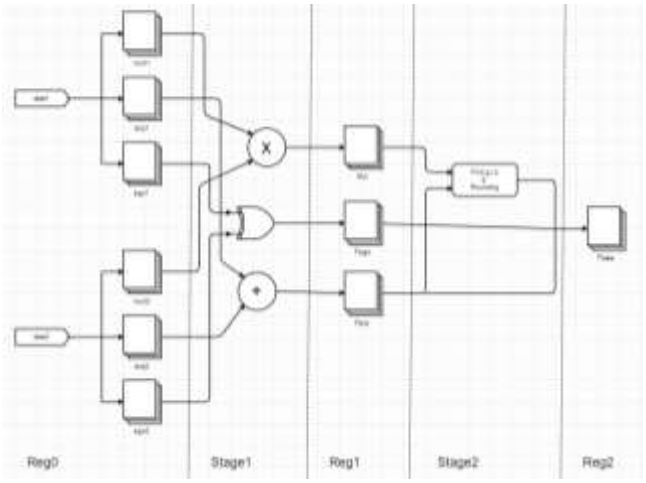


RTL Schematic

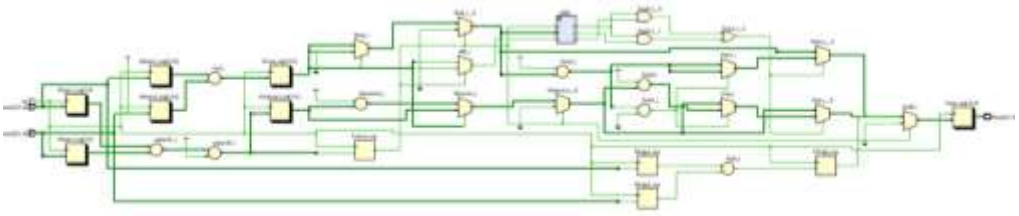


6.5. Pipelined Multiplication

Designed Architecture



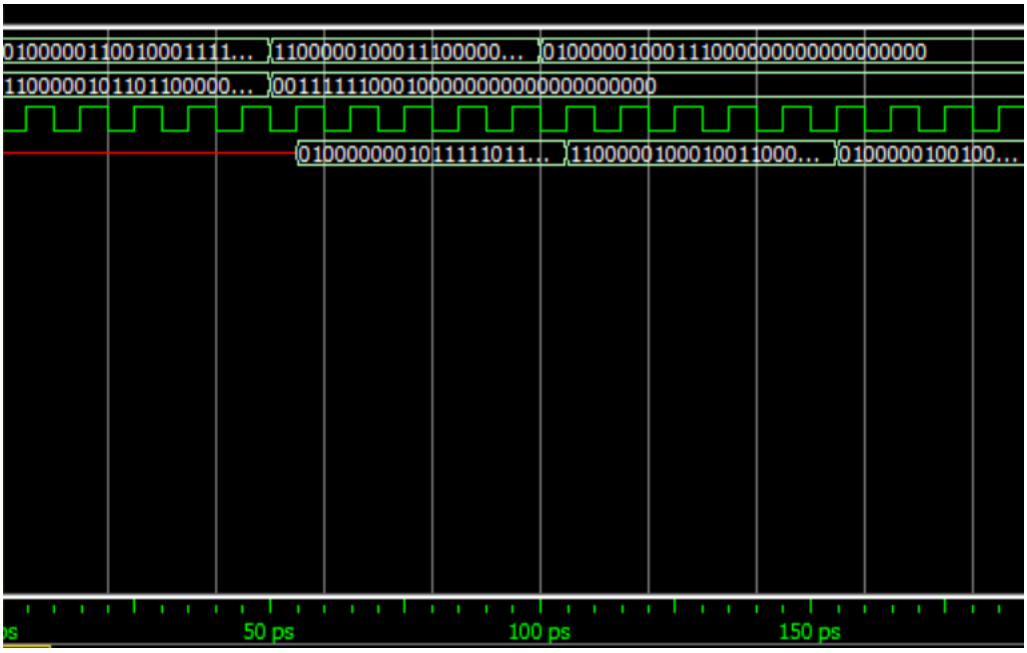
RTL Schematic



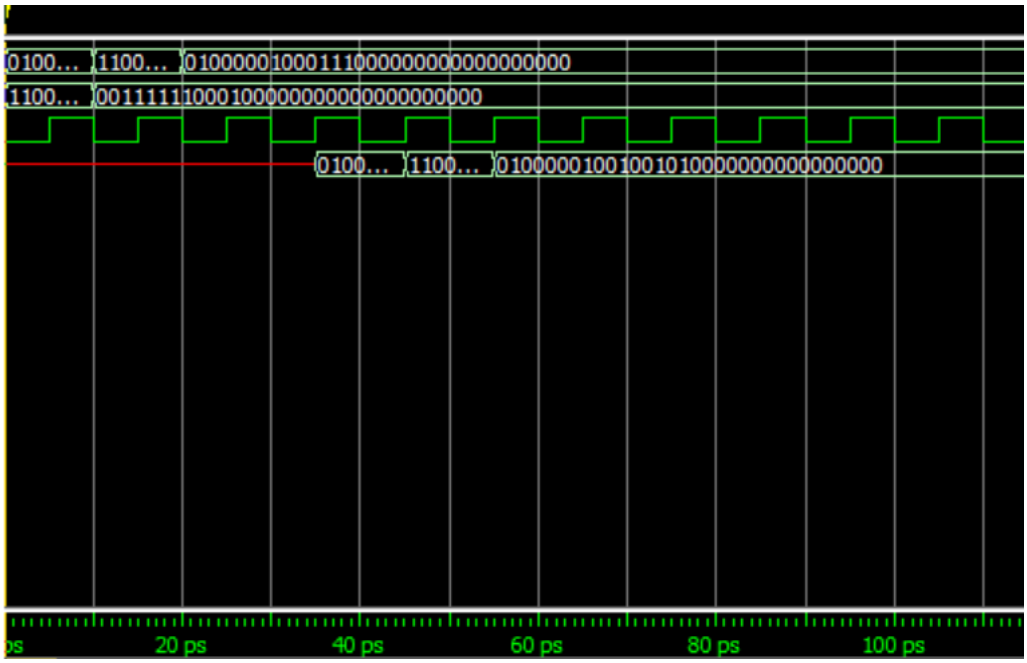
7. SIMULATION RESULTS

The simulations were performed on the software ModelSim Altera. The screenshots of the simulation results are shown in the figures below.

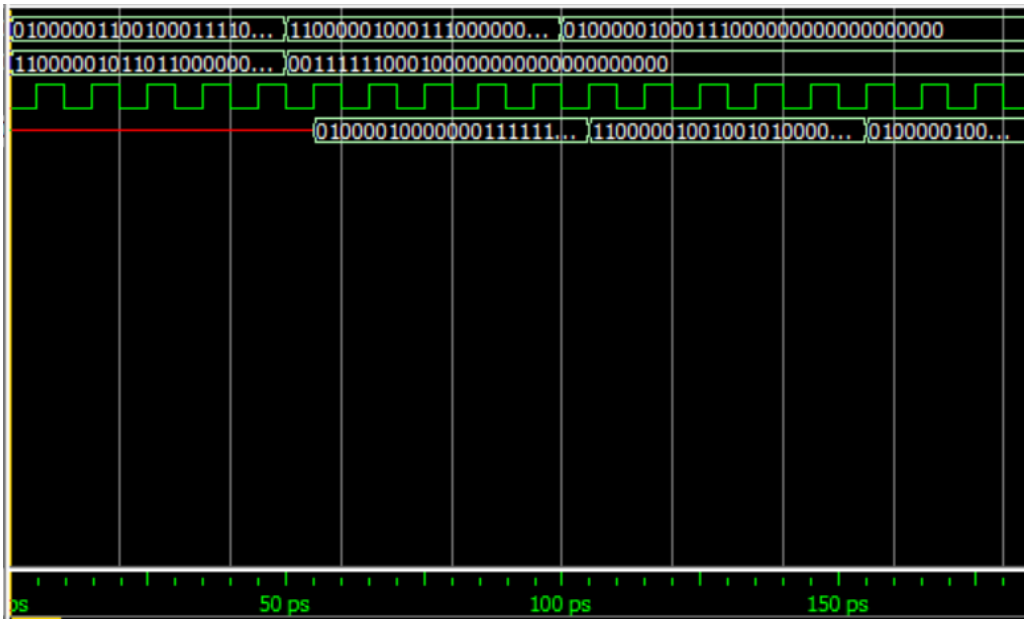
Non-Pipelined Addition – Latency = 5 cycles; Throughput = 1 per 5 cycles



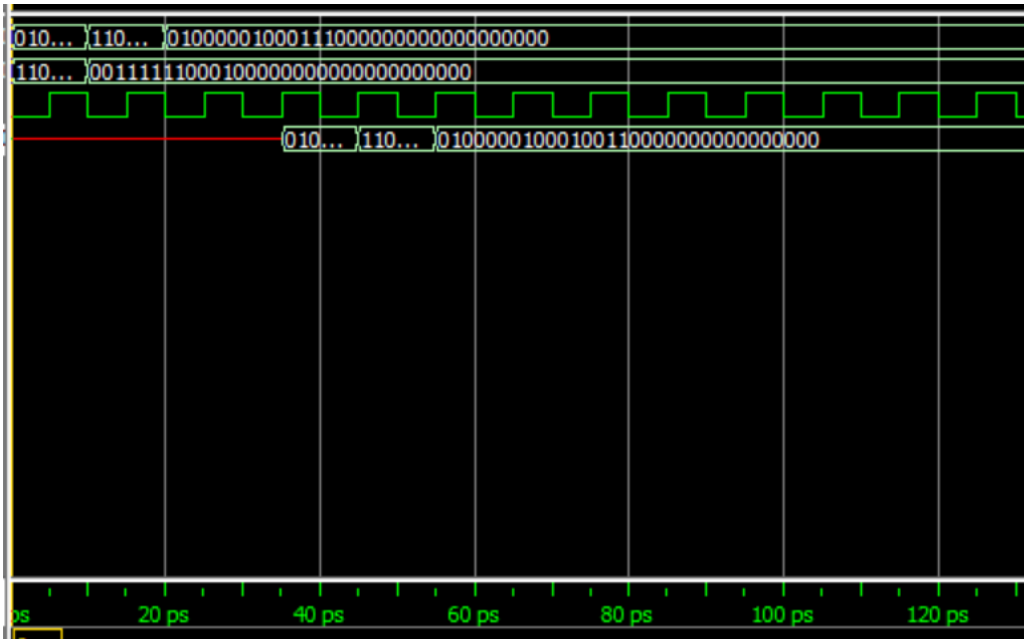
Pipelined Addition – Latency = 5 cycles; Throughput = 1 per 1 cycle



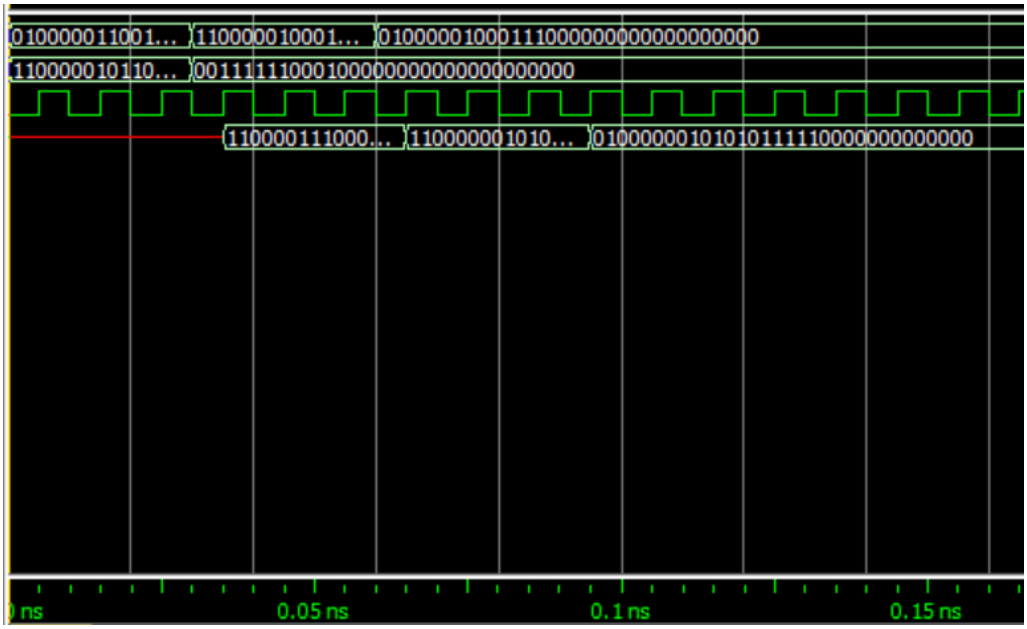
Non-Pipelined Subtraction – Latency = 5 cycles; Throughput = 1 per 5 cycles



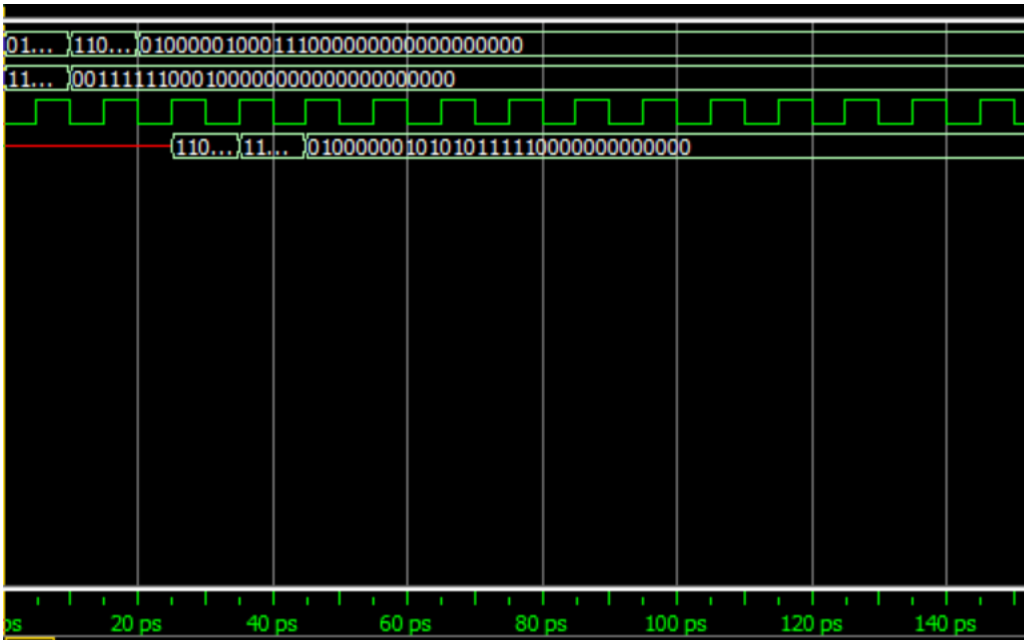
Pipelined Subtraction – Latency = 5 cycles; Throughput = 1 per 1 cycle



Non-Pipelined Multiplication Latency = 3 cycles; Throughput = 1 per 3 cycles

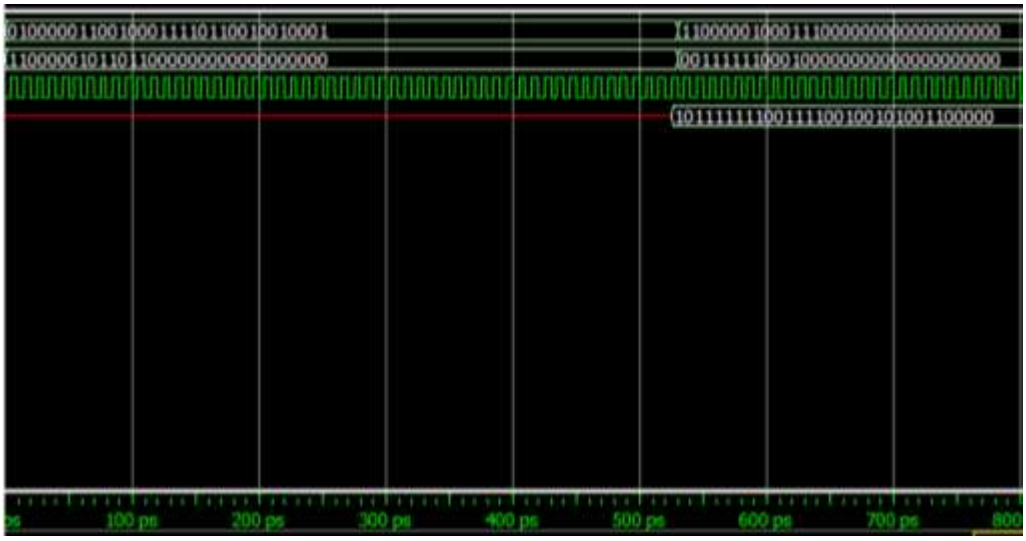


Pipelined Multiplication – Latency = 3 cycles; Throughput = 1 per 1 cycle



Non-Pipelined Division – Latency = 51 cycles; Throughput = 1 per 51 cycles

Zoomed image for one input set



8. ANALYSIS OF SIMULATION RESULTS

- *First thing* to be noticed in the simulation is the accuracy of the results. Considering a single input set for *Non-Pipelined Addition*, the observations are as follows:

The *numbers added* were:
X1 = $+18.241_{10}$ = 0 10000011 00100011110110010010001
X2 = -14.750_{10} = 1 10000010 11011000000000000000000
The *result* obtained was:
X3 = 0 10000000 10111110110011001000100
i.e. X3 = $+3.49051094055_{10}$
The result should ideally be = $+3.491_{10}$

It can be seen that the results do not exactly match with the ideal values. This is due to rounding off the numbers to 23 bits. The error due to rounding is more in multiplication and division as the resultant 46 bits are rounded off to 23 bits. Rounding leads to loss of data and thus, error.

Hence, it can be said that the results of the simulations are correct within the limitations of the rounding errors.

- *Another point* to be noted is that a pipelined architecture improves the efficiency of a design for multiple set of inputs. Pipelining results in a *significant increase in the throughput* keeping the *latency same* with the help of a few *additional registers*.
- It was *not practical to apply pipelining* for the Division Algorithm as it would need a lot of additional registers. This would lead to an increase in cost which is not compensated by the improvement in its efficiency.

9. SYNTHESIS AND IMPLEMENTATION RESULTS

The following are the components present on the FPGA:

1. **LUT** - Look Up Tables – It is a table that determines what the output is for any given input(s). In the context of combinational logic, it is the truth table.
2. **FF** – Flip Flops – It is the main component in an FPGA that is used to keep the track of a state inside of the chip. This way an FPGA knows what happened in the past, and it can keep track of counters, finite state machines, and status of things.
3. **DSP** – Digital Signal Processing Blocks – It is an architecture that has been optimized to implement various common DSP functions with maximum performance and minimum logic resource utilization.
4. **IO** – Input Output Banks – It is a group of I/O pins that share a common resource such as one power supply or one output current reference. It makes the FPGA easier to manufacture.
5. **BUFG** – Clock Buffers – It is an architecture-independent global buffer which distributes high fan-out clock signals throughout a target device. It is a buffered clock, and it's the normal way of using a clock in a design.

9.1. Utilization of the FPGA

Non-Pipelined and Pipelined Addition/Subtraction

RESOURCES	AVAILABLE	NON-PIPELINED		PIPELINED	
LUT	53200	538	(1.01%)	443	(0.83%)
FF	106400	221	(0.21%)	198	(0.19%)
DSP	220	0	(0.00%)	0	(0.00%)
IO	200	100	(50.00%)	98	(49.00%)
BUFG	32	1	(3.13%)	1	(3.13%)

Non-Pipelined and Pipelined Multiplication

RESOURCES	AVAILABLE	NON-PIPELINED		PIPELINED	
LUT	53200	98	(0.18%)	93	(0.17%)
FF	106400	78	(0.07%)	77	(0.07%)
DSP	220	2	(0.91%)	2	(0.91%)
IO	200	99	(49.50%)	97	(48.50%)
BUFG	32	1	(3.13%)	1	(3.13%)

Non-Pipelined and Pipelined Addition/Subtraction

RESOURCES	AVAILABLE	NON-PIPELINED	
LUT	53200	164	(0.31%)
FF	106400	332	(0.31%)
DSP	220	0	(0.00%)
IO	200	99	(49.50%)
BUFG	32	1	(3.13%)

9.2. Timing Analysis

The following table summarises the timing analysing carried out on the *Vivado Design Suite*:

ARCHITECTURE	MAX CLK FREQUENCY	MIN CLK PERIOD	LATENCY	THROUGHPUT
Non-Pipe Add/Sub	116.144 MHz	8.610 ns	43.05 ns (*5 cycles*)	1 per 5 cycles
Pipe Add/Sub	98.184 MHz	10.185 ns	50.925 ns (*5 cycles*)	1 per 1 cycle
Non-Pipe Mul	171.204 MHz	5.841 ns	17.523 ns (*3 cycles*)	1 per 3 cycles
Pipe Mul	171.204 MHz	5.841 ns	17.523 ns (*3 cycles*)	1 per 1 cycle
Non-Pipe Div	152.788 MHz	6.545 ns	333.795 ns (*51 cycles*)	1 per 51 cycles

The throughput has significantly increased with the help of pipelining:

- Addition/Subtraction – **3 vs 15 outputs** per 15 cycles (for non-pipelined vs pipelined)
- Multiplication – **5 vs 15 outputs** per 15 cycles (for non-pipelined vs pipelined)

10. CONCLUSION

The basic design of an adder, subtractor, multiplier and divider has been completed with sufficiently accurate results. Optimized codes for the non-pipelined and pipelined architectures have been written and synthesised successfully. These were also implemented on an FPGA. All the algorithms were combined into a single code to work as a single Floating Point Unit. This was all done after a detailed analysis of several research papers and sources.

This project was an excellent medium for us to explore software environments such as ModelSim Altera and Xilinx Vivado. It also introduced us to the basics of Floating Point Arithmetic, Computer Architecture, RTL Architecture Design, FPGA Implementation, and Pipelining. Implementation of our algorithms enabled us to look into different aspects (software and hardware) of Embedded Systems.

There is a scope for improvement in all the individual algorithms in terms of reduced logic and better accuracy. Reuse of hardware blocks can be properly implemented. Power consumption and timing can also be improved. The concepts used in this project gave us an understanding of IC Design which can be very helpful in our upcoming projects.

11. REFERENCES

1) **Books**
   a) Computer Architecture : A Quantitative Approach 5th Edition by J. L. Hennessy and Patterson
   b) Verilog HDL-A guide to Digital Design and Synthesis by Samir Palnitkar
2) **Research Papers**
   a) Purna Ramesh Addanki, "An FPGA Based High Speed IEEE - 754 Double Precision Floating Point Adder/Subtractor and Multiplier Using Verilog", International Journal of Advanced Science and Technology Vol. 52, March, 2013
   b) Atul Rahman, "Optimized Hardware Architecture for Implementing IEEE 754 Standard Double Precision Floating Point Adder/Subtractor", 17th International Conference on Computer and Information Technology (ICCIT), 2014
3) **Websites**
   a) MIT 6.004 -Computation structures: https://www.youtube.com/watch?v=uMvO1buogaE
   b) NPTEL-Digital Computer Organization by Prof.P.K. Biswas: https://www.youtube.com/watch?v=AXgfeV568c8
   c) synthesis and design flow: https://www.sologic.net/documents/knowledge/tutorial/Basic_FPGA_Tutorial_VHDL/sec_design_implementation.html
   d) FSM implementation in Verilog: https://inst.eecs.berkeley.edu/~cs150/sp12/resources/FSM.pdf
   e) Division algorithm: https://en.wikipedia.org/wiki/Division_algorithm