DISSERTATION

Parallel Pipeline Implementation of 64-bit FPU on Hardware

by

Ng Kiat Hong
1450
Bachelor of Engineering (Hons)
(Electrical and Electronic Engineering)

Supervised by:
Mr. Fawnizu Azmadi Hussin

Dissertation submitted in partial fulfilment of
the requirements for the course
EEB5034 Final Year Design Project

JUNE 2004

Universiti Teknologi PETRONAS
Bandar Seri Iskandar
31750 Tronoh
Perak Darul Ridzuan

CERTIFICATE OF APPROVAL

Parallel Pipeline Implementation of 64-bit FPU on Hardware

by

Ng Kiat Hong

A project dissertation submitted to the
Electrical and Electronics Engineering Programme
Universiti Teknologi PETRONAS
in partial fulfilment of the requirement for the
BACHELOR OF ENGINEERING (Hons)
(ELECTRICAL AND ELECTRONICS ENGINEERING)

Approved by,

[illegible]

(Mr. Fawnizu Azmadi Husin)

UNIVERSITI TEKNOLOGI PETRONAS

TRONOH, PERAK

JUNE 2004

CERTIFICATE OF ORIGINALITY

This is to certify that I am responsible for the work submitted in this project, that the original work is my own except as specified in the references and acknowledgements, and that the original work contained herein have not been undertaken or done by unspecified sources or persons.

[illegible]

NG KIAT HONG

ABSTRACT

This project is entitled "Parallel Pipelined Implementation of 64-bit FPU on Hardware". Most modern processors typically have two different logic units which handle the calculations required by the computer. One of them is the arithmetic-logic unit (ALU) which operates on integer operands while the other is the floating point unit (FPU) which operates on real operands. The aim of this project is therefore to create a FPU which complies with the IEEE-754 double precision standard (64-bit).

The project also aims to study the speed improvements offered by parallel and pipelined design. The project also requires application of advanced digital design techniques by using Verilog in a real world project. The designed FPU is targeted to be capable of performing floating point addition (FADD), subtraction (FSUB), multiplication (FMUL) and division (FDIV) operations equally as fast. The FPU must also demonstrate the performance rewards of the parallel and pipeline design. It is thus implied that the project would require an initial study on FP numbers and FP arithmetic. How FP arithmetic is actually implemented in hardware must also be know in-depth.

The section on methodology details each steps that is expected to be taken throughout the course of the project. The methodology would serve as a general guideline to execute the project and more details and other refinements may be made as further progress is made into the project. The project basically has two main phases, the first being software RTL coding to be completed in semester 1 while the second is hardware implementation and testing in FPGA.

The results available from the project thus far is incomplete, because of time constraints, the Verilog coding is not totally finished. Once the codes are done, RTL tests and simulation would need to be conducted, and then only will it be implemented on the FPGA.

ACKNOWLEDGEMENT

The present project work was carried out from August 2003 to May 2004 under the guaidance of Mr. Fawnizu Azmadi Husin from Electrical and Electronics Engineering Programme of Universiti Teknologi Petronas. The author would like to acknowledge his supervisors for providing valuable guaidance and advices needed throughout the course of the project.

The author also wishes to express his gratitude to Mr. Rudolf Usselmann, whose work at www.opencores.org provided a vital base for this project to get started.

Last but not least, the author wishes to thank every else directly or indirectly involved in the completion of the project. Among them is Ivan, a colleague also involved in a FPGA project, who has shared his know how's in Verilog coding. The author also appreciates all the lab technicians who have helped maintain the FPGA lab.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

1 INTRODUCTION

1.1 Background of Study

There are two general formats to represent binary numbers: fixed-point and floating-point. Fixed point notations are usually used to represent either positive or negative integers only. Real numbers cannot be fully represented in fixed point as it lacks the ability to store fractional components. An example of fixed point notation is the two's complement system. Real numbers usually require the floating point notation which represents number in binary scientific notation. The format is $\pm$ M x 2^E where $1 \leq M < 2$. The separate fields to the mantissa (M) and exponent (E) in floating point notation provide plenty of flexibility for representing extremely large or extremely small numbers.

Unlike fixed point numbers, floating numbers are considerably more complex, requiring dedicated hardware to manipulate on most computers. This aim of this project is to design and implement an arithmetic unit capable of manipulating floating-point numbers. This unit is called a Floating Point Unit (FPU) and the hardware implementation would be done on a Field Programmable Grid Array (FPGA).

1.2 Problem Statement

The challenge of this project would be to design an IEEE-754 compliant FPU using hardware description language, and then implement the design on hardware using field programmable grid arrays (FPGA). Additionally, the FPU must support the double-precision format (64-bit wide for every number) specified in the IEEE standard. An existing implementation of a single-precision FPU (32-bit wide) is already available as an open IP core by Rudolf Usselmann at Opencores. [1]

The FPU must also be built with performance enhancements in mind. Therefore, the FPU would be designed with multiple pipeline stages to increase the efficiency of its execution. The FPU, being an arithmetic execution unit would have an arithmetic pipeline. However, the FPU still needs to be controlled by a control unit, which must employ a basic instruction pipeline in order to keep the FPU busy all the time. The target of this project would then be how to design an FPU, with parallel and pipelined architecture. If possible, the performance of a pipelined design versus a non-pipelined design could be compared.

1.3 Objectives

1. Learn and apply digital design techniques to implement a crucial part of the microprocessor.
2. To keep up with industrial trends in digital design and learning large scale digital design using the Verilog language.
3. Study the architecture and design of a Floating Point Unit (FPU).
4. To make a worthy contribution to parallel and pipelined FPU design.
5. To produce a FPU that would be capable of performing all four arithmetic operations: ADD, SUB, MUL and DIV operation equally fast as each other.
6. Acquire the skills to manage a large scale long term project over two semesters.

1.4 Scope of Study

This project would require designing the FPU using a hardware description language (HDL). The language of choice here is Verilog for it is relatively easy to learn and is closer to hardware. The project also involves studying the architecture and specification of the FPU. The design and implementation of the FPU must be compliant with the IEEE-754 standard. Concepts such as parallel execution and pipelining are also explored in the project.

CHAPTER 2

LITERATURE REVIEW AND THEORY

2 LITERATURE REVIEW AND THEORY

2.1 Real Number System

The real number system consists of both rational and irrational numbers. Both can be represented on a line shown below:

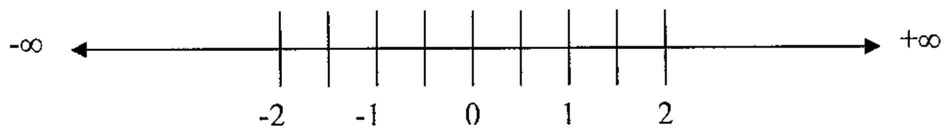


Figure 1: The Real Line

All real numbers stretch from -∞(negative infinity) to ∞(positive infinity). Infinities are not numbers themselves, but they represent the extreme ends of real numbers. Integers are rounded real numbers such as ...-2, -1, 0, 1, 2...

Integers are easily represented in computers in binary form. The only challenge to binary representation of integers would be the sign but this is easily solved using the "two's complement" number system*. Real numbers meanwhile require the use of the floating point number system, which is more challenging. In decimal form, a typical floating point number is written in the scientific format

$$\pm M \times 10^{E},\ 1 \leq M < 10$$

The number M (mantissa) always has a value in between 1 and 10, for example 1.8687 or 9.0974. Note that both numbers only have 1 single digit before the decimal point. The exponent (E) would determine the real position of the decimal point in M. Therefore, numbers such as 0.0025647 is represented as 2.5647×10^{-3} and 9998.2

* Other systems such as "one's complement", "sign-and-magnitude" are also used.

would be represented as 9.9982 x 10^3. The very act of having the decimal point move around back and forth gives rise to the name “floating point”.

2.2 Binary Floating Point Numbers

Binary digits could only carry two values, 0 and 1. Therefore, binary floating point numbers have the following format:

$$\pm M \times 2^E,\ 1 \leq M < 2$$

Therefore the binary expansion of the mantissa, M would be:

$$M = (b_0.b_1b_2b_3\ldots)_2 \text{ with } b_n = 0 \text{ or } 1$$

Normalizing binary floating point numbers means adjusting it so that the leading bit, b_0 has a value of 1. This would have great implications in hardware later. A few terms are normally used while referring to floating point number systems. They are:

Precision:

The number of bits used to store the mantissa, M

Machine epsilon, ε:

The gap between the number 1 and the next largest floating point value

Unit in the last place, ulp(x):

The gap between a floating point number x and the next larger floating point number (for $x > 0$) or next smaller floating point number (for $x < 0$).

At its inception, the format for floating point numbers was very loosely defined and each company’s solution differed from the other. This made porting software across to other platforms very difficult.

2.3 IEEE Floating Point Number Representation [2,3,7]

The IEEE FP number system was an industry wide initiative to standardize the representation and arithmetic of binary FP numbers across multiple platforms and fields of interest (both academics and industry). The IEEE 754-1985 defines three formats for binary representation of floating point numbers; single, double and extended. The basic difference is that each format successively uses more bits to store the mantissa, hence providing higher precision for each number. Both the single and double format does not store the leading bit of the mantissa (since it is always 1), however the Intel Extended format do. Therefore, the precision of single and double format is equal to the number of bits plus one. Other definitions in the IEEE standard include:

- bits used for both mantissa and exponent
- range of numbers that can be represented – normal range and subnormal range
- rounding operation and rounding modes
- exception conditions such as are invalid operation, division by zero, overflow, underflow and inexact. The results of these operations are usually set to "zero", "infinity" or "NaN[†]".
- arithmetic process; guard, round and sticky bits
- many more lower level details

Table 1: IEEE 754 Floating Point Formats

Format	Total Bits	Sign Bit	Exponent Bits	Mantissa Bits
Single	32	1	8	23 (+1 hidden)
Double	64	1	11	52 (+1 hidden)
Extended(Intel)	80	1	15	64

The format chosen for the project is the double precision. The precision of the double format is effectively 53 bits since the leading bit of the mantissa is not stored.

For the double format the exponent range is from Emin = -1022 to Emax = 1023. The range of representable numbers is thus from 2.2 x 10^{-308} to 1.8 x 10^{308}.

[†] Not A Number

2.4 IEEE – 754 Rounding Modes [2,3]

A design compliant with the IEEE-754 standard must implement four (4) rounding modes. They are:

- **Round to nearest:** The result is rounded to the nearest representable number
- **Round to zero:** The result is rounded to the number nearer to zero
- **Round up (to +∞):** The result is rounded to the number nearer to +∞
- **Round down (to -∞):** The result is rounded to the number nearer to -∞

The default mode is **round to nearest**. In this mode, the result (let it be x) would be rounded to the value that is nearest to it, in other words, one that gives the lowest rounding error. Let the two bounds of x be $x+$ and $x-$ ($x+$ and $x-$ are both representable numbers). In this mode, x will be rounded to $x+$ if it is nearer to it and vice versa. However, if both $x+$ and $x-$ are equally near to x, the one with its least significant bit 0 would be chosen.

The next mode is **round to zero**. This mode is the simplest mode of all as the result is obtained by simply truncating all the extra bits to the left of x. Though simple, the disadvantage of this mode is that a consistent bias towards zero will be developed over successive operations. This is a serious shortcoming as the result of an operation with multiple steps may have a large error.

The last two modes are **round up** and **round down**. Both require rounding the result either towards the number nearer to positive infinity or negative infinity. These rounding modes are useful in interval arithmetic as two values that correspond to the upper and lower endpoints of the real result can be obtained. Interval arithmetic can be used to monitor and control errors in floating point computations as two values can be produced for each result. If the desired endpoints themselves cannot be represented, they can be either rounded up or rounded down. The accuracy of the result therefore depends on the width of the interval. Most algorithms that implement such arithmetic are designed to produce narrow interval, the more narrow the interval, the more accurate is the result.

2.5 Addition and Subtraction of Sign and Magnitude Numbers Using 2's Complement Method [4]

The mantissa of a floating point number is basically a magnitude only component while the sign is stored on an additional bit. Taken together, the sign bit and the mantissa bits would form a sign-and-magnitude number. Operating on sign-and-magnitude data is slightly more complicated than the more common two's (2's) complement. Multiplication and division is actually easier on sign-and-magnitude data compared to 2's complement data, but the reverse is true for addition and subtraction.

The algorithm employed for addition and subtraction in this particular FPU is based on 2's complement method (Figure 12). For sign-and-magnitude numbers, the operation needs to be carried out on the magnitudes only, thus for addition the sum of the mantissa must be found while for subtraction, the difference between the mantissa is calculated. Therefore, it is imperative to find out what the true operation is on the magnitudes first before operating on them. For instance, an addition means adding both magnitudes together when both their signs match. If their signs do not match, the true task is to calculate the difference between the magnitudes.

For true addition operations, the mantissa is simply added to one another, without any special consideration. This is similar to 2's complement addition, except that the carry-out is not ignored. Subtraction is the operation that truly utilizes the 2's complement method. Let the operation be A – B. The method used is to take the 2's complement of B and add it to A. If the magnitude of A is larger than B, the carry out from the operation would be one (1). This carry out indicates that the answer is the correct one. However, if the carry out is zero (0), then B is larger than A, thus the result obtained is incorrect. To obtain the correct answer, the algorithm takes the two's complement of the original result and then invert the sign. The following equations prove the validity of the subtraction process:

Let the first operand = A

Let the second operand = B

2's complement of B = $2^n - B$

$A - B = A + (2^n - B)$ (when $A > B$)

$= A - B + 2^n$

$= A - B$ (2^n discarded)

Since the final carry-out of 2's complement addition is discarded, the 2^n term in the result can be ignored, leaving A – B as the result of the operation. However, this only happens when A > B. For cases where A < B, the 2^n term will not be generated (no end carry) because A would need to borrow a digit from 2n to be subtracted by B. Therefore, when A is smaller than B, the difference in magnitude between A and B is given by (B – A), as shown below:

$A - B = A + (2^n - B)$ (when $A < B$)

$= 2^n - (B - A)$

Taking 2's complement inverse of (A – B)

$2^n - (A - B) = 2^n - 2^n + (B - A) = (B - A)$

2n – (B – A) is actually the 2's complement of (B-A). Thus, by taking the 2's complement of the result, we would obtain the difference between A and B, that is (B – A). When a smaller number is subtracted by a larger number, the sign of the result has to be changed. Thus the sign of the result is the inverse of A.

Therefore, when an end-carry is not detected in the result of the subtraction, the correct output would be the 2's complement of the existing result and the sign of the output is the inverse of the first operand.

2.6 Pipelining (Arithmetic vs. Instruction) [4]

There is a small but very important difference between arithmetic and instruction pipelining. Arithmetic pipelining is applied only to arithmetic units inside a processor while instruction pipelining could be applied on any processor in general.

Arithmetic pipelining basically takes the arithmetic execution stage (starting from operand fetch to producing an output) and split it into a few pipeline stages. This is the type of pipelining that is going to be used for the FPU. Using the FPU designed for this project as an example, the process of floating point addition, subtraction, multiplication and division can be divided into three segments. Each segment then forms a single pipeline stage. Therefore, the execution of one floating point operation would overlap with the execution of another operation in an earlier pipeline stage. Further detail on arithmetic pipelining is explained on Section 4.3 (page 17)

Instruction pipelining is not currently being applied for the FPU project. Instruction pipelining splits the fetch, decode, execute and store instructions into a few pipeline stages and overlap their execution. Therefore, the execution of these four instructions would be the one that overlaps in an instruction pipeline. While the CPU is executes the decode instruction, a new fetch instruction is also being executed. The pipeline structure of a 4-stage instruction pipeline processor is shown below:

Table 2: Pipeline Structure of an Instruction Pipeline

Clock Transition	T1	T2	T3	T4	T5	T6
Instruction 1	Fetch	Decode	Execute	Store		
Instruction 2		Fetch	Decode	Execute	Store	
Instruction 3			Fetch	Decode	Execute	Store
Instruction 4				Fetch	Decode	Execute

2.7 Theoretical Speed Improvement of Pipelining [3]

This section attempts to show a simple mathematical proof of the potential speedup in a pipelined design. Let the cycle time, τ be the time it takes for an instruction to advance through a stage of the pipeline. This cycle time can be determined by:

$$\tau = \tau_m + d$$

where

τ_m	=	maximum stage delay (delay through the stage which experiences the largest delay
d	=	time delay of through a latch (register), needed to store date between stages

Usually, the delay d can be ignored as $\tau_m >> d$. Now, let n instructions be executed without branches through the pipeline. The total execution time, T_k is

$$T_k = [km + (n\text{-}1)]\ \tau, \textit{ where } k = \textit{number of pipeline stages}$$

Without pipelining, the value of k is 1, thus the total execution time for a non-pipelined design is

$$T1 = nk\tau$$

The speedup factor of a pipelined design is:

$$S_k = \frac{T_1}{T_k} = \frac{nk\tau}{[k + (n\text{-}1)\tau]} = \frac{nk}{[k + (n\text{-}1)]}$$

If we observe the behaviour of the equation above at the limit of n $\rightarrow \infty$, we will see that $S_k = k$. That means that the speed improvement of a pipelined design over a non-pipelined one is equal to the number of pipeline stages itself. Therefore, the larger the number of pipelines, the faster it is. However, this potential speedup occurs only in ideal cases where n instructions can be fed into the pipeline without branching. In real life, branches occur very often, reducing the potential speedup. Another factor that would reduce the speedup is latches that exist between two pipeline stages. They are also called as pipeline registers. All registers have a finite read and write delay to it, thus every signal propagating from one stage to another must encounter this delay. A single stage combinatorial circuit would not need such registers as all the signals are connected directly to each other. Hence, the speedup of a pipelined design over a non-pipelined one is actually less than k, the number of pipeline stages.

CHAPTER 3

METHODOLOGY AND PROJECT WORK

3 METHODOLOGY AND PROJECT WORK

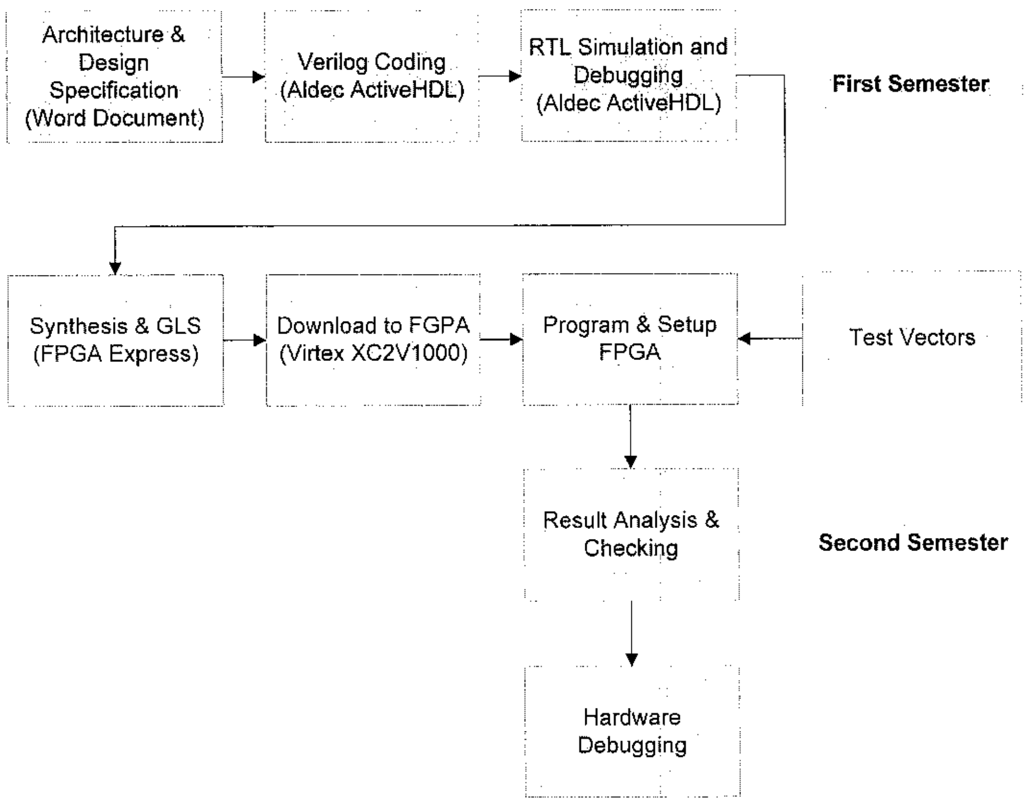

Architecture & Design Specification (Word Document)
Verilog Coding (Aldec ActiveHDL)
RTL Simulation and Debugging (Aldec ActiveHDL)
First Semester
Synthesis & GLS (FPGA Express)
Download to FGPA (Virtex XC2V1000)
Program & Setup FPGA
Test Vectors
Result Analysis & Checking
Second Semester
Hardware Debugging

Figure 2: Design Process Flow

From the flow chart above, the project could be largely divided into two stages to be completed over two semesters. The first semester would be concerned with purely RTL (Register Transfer Language) coding while the second semester would be implementing the design in FPGA and running all the hardware tests.

Before the design stage can start, literature research would be conducted first. Only after literature review is done, then the design process illustrated in Figure 2 could start.

3.1 Literature Research

Literature research was done on IEEE-754 compliant FP numbers and various FPU designs. The main concern here is to find the general architecture and requirements of a FPU. Studies were also done on the components that make up the FPU, such as binary adders, multipliers and dividers. Literatures are largely obtained from library texts and also sources on the internet.

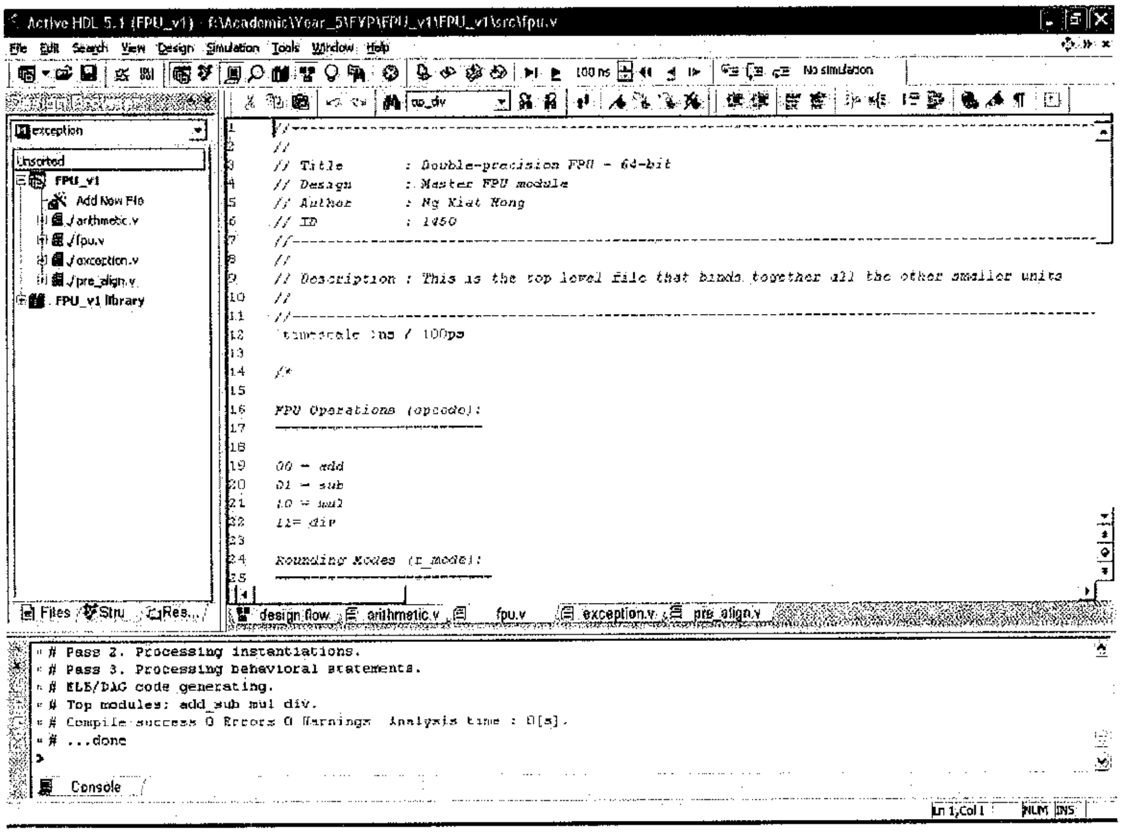
3.2 Architecture Design and Specification

Every part of the FPU is first documented in form of a MS Word document before being coded. The document serves as a design specification and guideline. The functions of each module along with specific implementation details are included in the document. Relevant flow charts are used to illustrate the algorithms used. ‡

3.3 Verilog Coding [6]

Verilog coding starts once the specification for any component of the FPU is completed. Verilog codes are entered using the Aldec ActiveHDL IDE. Ideally, each Verilog module is stored in separate source files. Different source files are then kept in a common design/workspace. This allows easy manageability of the codes while retaining the ability to call modules coded somewhere else. Written codes are then compiled to check for syntax errors.

‡ The design specification is attached as part of the appendix



```
//------------------------------------------------------------------------
//
// Title        : Double-precision FPU - 64-bit
// Design       : Master FPU module
// Author       : Ng Kiat Hong
// ID           : 1450
//------------------------------------------------------------------------
//
// Description : This is the top level file that binds together all the other smaller units
//
//------------------------------------------------------------------------
`timescale 1ns / 100ps

/*

FPU Operations (opcode):
------------------------


00 = add
01 = sub
10 = mul
11= div

Rounding Modes (r_mode):
------------------------
```

```
# Pass 2. Processing instantiations.
# Pass 3. Processing behavioral statements.
# ELB/DAG code generating.
# Top modules: add_sub mul div.
# Compile success 0 Errors 0 Warnings  Analysis time : 0[s].
# ...done
>
```

Figure 3: Aldec ActiveHDL Main Screen

3.4 RTL Simulation and Debugging

The completed RTL codes need to be tested in software first before being downloaded to the FPGA and implemented in hardware. RTL simulation is used to test the functionality of the FPU (or parts of the FPU) and to catch logic bugs. The tool used to simulate the Verilog design is also Aldec ActiveHDL.

A testbench is written to provide stimulus to the design. The design must respond with certain outputs for certain combinations of input. An incorrect output would indicate a logic bug that needs to be traced and fixed. The source of test vectors for the test bench could either be hard coded in the test bench itself or obtained from an external file.

After simulation, the result can be monitored and viewed in many different ways. One of the most useful is the waveform viewer which tracks the logic values of any desired signal. Using the waveform editor, both the states of the input and output could be easily observed on a timeline. This serves as a great debugging tool to root out problems in the code or design. The figure below shows an example of the waveform for a single bit full adder with Cin, x and y as the input and Cout and s as the output. If the output does not respond as predicted by the input, a bug may exist in the code.

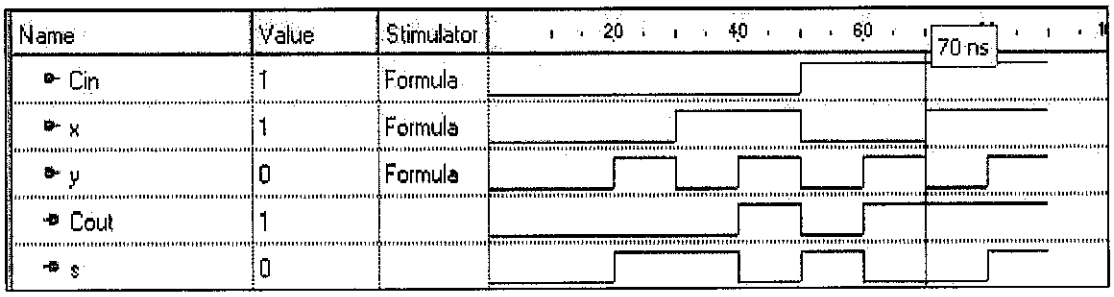
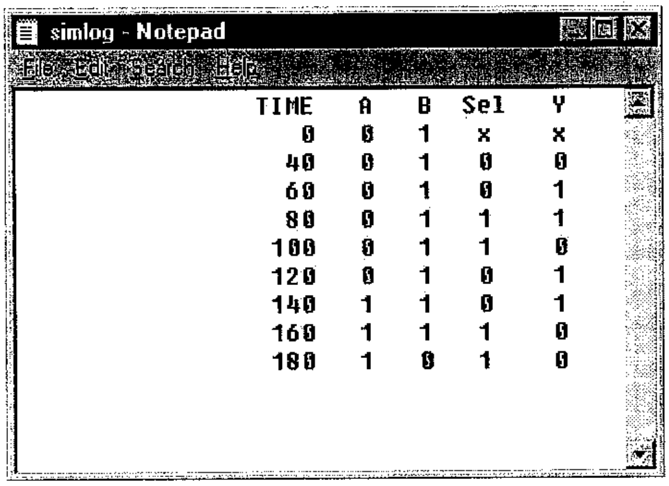


Figure 4: ActiveHDL Waveform Output

The function "fmonitor" in ActiveHDL also allows signals to be recorded in an external text file. An example of the output is shown in Figure 5. The text files could then be further processed using another custom program that checks the output file for any violations. A file-processing friendly language such as Perl could be used.



```
TIME   A   B   Sel   Y
   0   0   1    x    x
  40   0   1    0    0
  60   0   1    0    1
  80   0   1    1    1
 100   0   1    1    0
 120   0   1    0    1
 140   1   1    0    1
 160   1   1    1    0
 180   1   0    1    0
```

Figure 5: Signal changes recorded on an external text file

Any bugs found at this point would require going back to the RTL coding stage and fixing the bug at the Verilog code. All previously written testbench and test vectors must then be reran on the newly fixed code to check if the bug has been truly fixed and if any new bugs were introduced by modifying the code.

3.5 Synthesis and Gate Level Simulation (GLS)

The finished RTL code could be synthesized using Synopsys FPGA Express, an industrial strength Verilog synthesis tool. The Verilog codes would be loaded into FPGA express and it would check for any syntax errors before proceeding. After synthesis, an EDF netlist file would be created.

This file could again be loaded into ActiveHDL for GLS. GLS operates on the netlist, meaning all the gates generated from the RTL codes. Bugs could still be found at gate level which did not exist at RTL level because of the uncertainty in the synthesis process. Coding styles, compiler directives and custom libraries are among the few factors that may change how the software synthesizes the design. Therefore, the generated circuit may not turn out to be as expected during RTL coding. Bugs found here should also be fixed at the Verilog code.

3.6 Download to FPGA (Virtex-II XC2V1000)

After or during GLS, the design could be downloaded to the FPGA chip for implementation on hardware. The preferred chip would be the Xilinx Virtex-II XC2V1000 as it contains the most number of gates (1 Million gates) of all the chips that are available in the university laboratory.

The Virtex-II chip resides on a reference board which contains plenty of I/O and memory functions. Some examples are the PCI interface, DDR memories, RS232 port, 7 segment displays, LED and much more. All these features would greatly speed up hardware testing and implementation as all the I/O to the FPGA chip do not need to be manually wired. PROMs are available on board for any programs to be loaded and executed.

3.7 Program and Setup FPGA

Once the FPGA has been programmed, it would need to be set up using Xilinx own tool. Configuring the FPGA is done through the JTAG interface on the reference board. Test programs that should be executed, along with the test vectors are then loaded into the on board RAM for hardware testing. The output from the program could then be stored in memory or output to another device for manual observation.

Hardware implementation is potentially the most difficult of all as subtle boundary conditions that may not be accounted for in RTL could cause bugs in hardware. Electrical loading conditions and signal quality are also other things that are out of control during RTL but must be taken care of when the hardware implementation is obtained.

3.8 Result Analysis & Checking and Hardware Debugging

The results from hardware tests need to be stored somewhere for analysis later. Since all the latter stages are schedules for the second semester, hardware testing plans are not yet ready.

CHAPTER 4

RESULTS AND DISCUSSION

4 RESULTS AND DISCUSSION

Sections 4.1 to 4.4 would discuss the general features of the FPU. Section 4.5 onwards meanwhile would present the design of each part of the FPU in detail.

4.1 General Architecture

The general architecture of the FPU is shown in Figure 6 below:

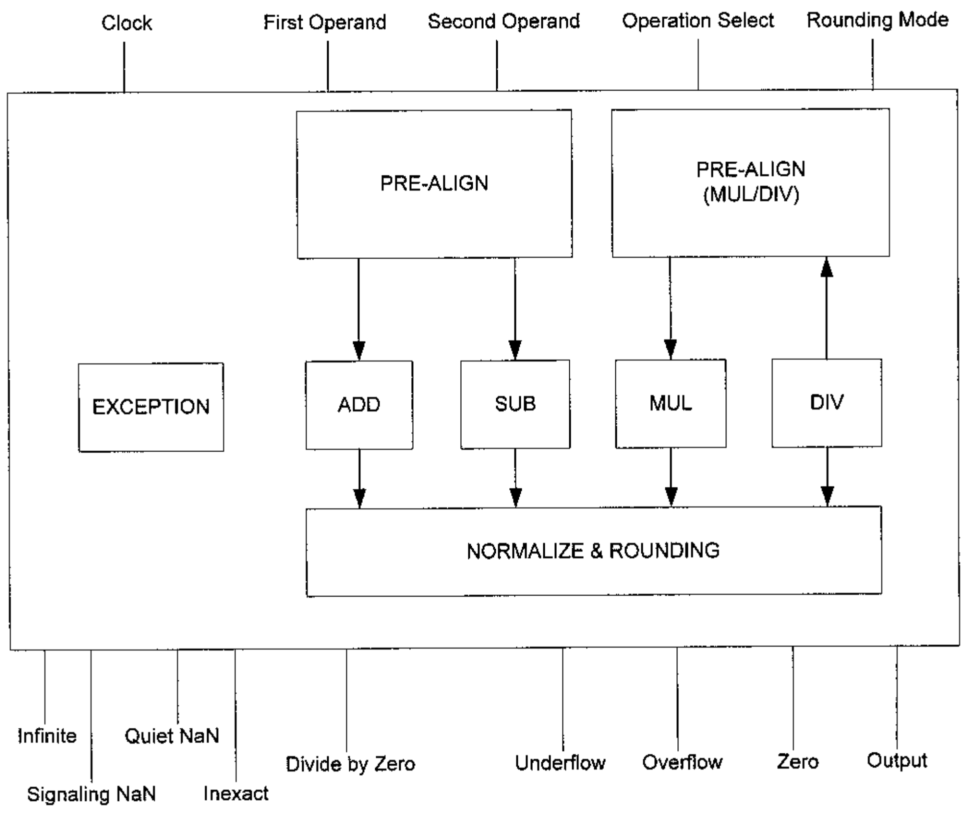


Figure 6: General Architecture of the FPU

The design in Figure 6 is the final top-level architecture of the FPU pending any late changes. Design issues and other performance enhancements may require a redefinition of the architecture. Do note that the architecture above do not explicitly show the pipeline structure of the FPU.

There are 5 inputs, 9 outputs and 3 major stages in the FPU. The first major stage consists of the pre-align units. The second contains all the arithmetic units operating on mantissas and the third contains the normalizing and rounding units along with any output logic (not shown). Seven (7) major sub-units are contained inside the FPU, each of them coded as a separate Verilog "module". The full design specifications are available in the appendix. The basic function of each unit is listed in the table below:

Table 3: Functions of FPU Sub-Units

Sub-Units	*Function(s)*
Pre-Align	Aligns the exponent and mantissa of each operand before addition and subtraction. Determines the true operation (Add or Sub) to be performed on the mantissas
Pre-Align (Mul/Div)	Performs dividend alignment for division operations. Add or Sub exponents on Mul or Div operations. Determines the sign of the result
Addition/Subtraction	Adds and subtracts pre-aligned mantissas
Multiply	Obtain the product of two mantissas from multiplication
Divide	Obtain the quotient and remainder by dividing the dividend mantissa with the divisor mantissa
Normalize/Rounding	Normalize the result from the Add/Sub, Mul and Div units. After normalizing, the result would be rounded according to the selected rounding mode
Exception	Checks for exceptions in the operands and then flags them. These signals may be used at later pipelines stages and also required to set flags in the status register of full CPU.

Table 4 lists the definition of each input to the FPU. Both input operands must be in normalized form or in any other special form as defined by the IEEE-754 standard. The FPU is never expected to handle non-floating point numbers. In real practice, the operands already exist in IEEE-754 compliant forms inside the main system memory before being fetched and operated upon by the FPU. Higher level software must handle all the relevant format conversions before the operands are sent stored in memory for operation.

Table 4: Functions of Inputs to FPU

Input	*Function*
Clock	External clock input. This signal should come from the clock generator and is used as the common clock that synchronizes every operation inside the FPU
First Operand	First input operand (IEEE-754 Compliant FP Number)
Second Operand	Second input operand (IEEE-754 Compliant FP Number)
Operation Select	Selects between Add, Subtract, Multiply and Divide operation
Rounding Mode	Selects between the four (4) different rounding modes: i. Round-to-nearest ii. Round-to-zero iii. Round-to-positive infinity (up) iv. Round-to-negative infinity (down)

Table 6 meanwhile lists the functions of the outputs from the FPU:

Table 5: FPU Outputs and their functions

Output	*Function*
Output	The floating point output as a result from the operation performed by the FPU on the input operands.
Infinite	Output operand in infinite
Ine	Output operand is not exact, instead it has been rounded
Signaling NaN	Signalling NaN; a type of NaN that causes an exception. This feature is not implemented and is aliased to Quiet NaN.
Quiet NaN	Quiet NaN; a non-exception type of NaN.
Divide by Zero	Asserted when an operand is divided by zero
Underflow	Result underflow; result is lower than the smallest possible number that can be represented
Overflow	Result overflow; result is higher than the highest possible number that can be represented
Zero	The result is zero

4.2 Clocking Strategy

Most of the FPU would operate in one clock domain. In fact, every module would operate on a global clock signal sourced from the input signal called "clk" except for the divide unit. In hardware, the "clk" input would be connected to the clock generator circuit on board the FPGA test board.

In such circumstances, each stage of the FPU must be able to complete its operation within a single clock cycle. This is required in order to fulfill the pipelined design requirement of having one complete result at every clock cycle. To properly implement the clocking scheme, each sub unit is designed as a combinatorial circuit, but with its final output registered on a clock-synched D flip-flop or latch. Figure 7 illustrates such design.

The divide module may be required to operate with a clock signal up to 53 times faster than the global "clk" signal. This is due to the division algorithm which would require 53 iterations to complete.

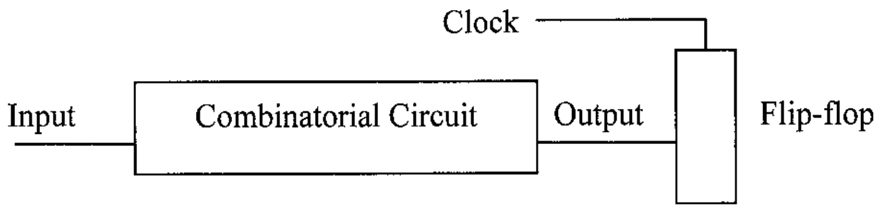


Figure 7: Registering Combinatorial Output Using Delay Flip-Flops/Latches

4.3 Pipelining Strategy

The architecture of the FPU can be split into three (3) pipeline stages. Figure 8 illustrates all the three stages clearly. The top half is the data path that the operands would follow for addition or subtraction operations while the bottom half is the execution path for multiplication and division operations. Two separate paths were designed as addition/subtraction does not share much hardware in common with multiplication/division. Both groups use fairly distinct methods to produce the results.

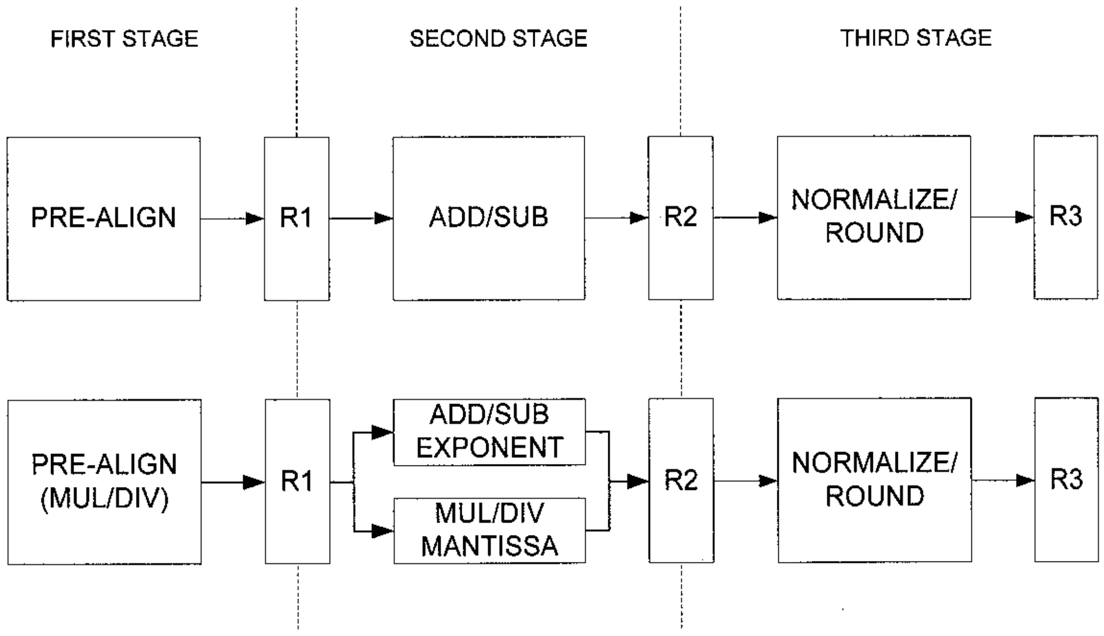


Figure 8: Pipeline and Dataflow Structure of FPU

This arithmetic pipeline design would then produce one floating point result at the output after every clock cycle (beyond the initial latency). Using the ADD/SUB datapath as an example, if each operation to be performed is called OpX, where X is an integer that denotes every successive operation, the flow of execution on the pipeline would be as illustrated in Table 6.

Table 6: Pipeline Structure of the FPU

Clock Transition	T1	T2	T3	T4	T5	T6
Pre-align	Op1	Op2	Op3	Op4	Op5	Op6
Add/Sub		Op1	Op2	Op3	Op4	Op5
Normalize/Round			Op1	Op2	Op3	Op4

From Table 6, it can be seen that at the third clock transition, the first operation has already passed through all the pipeline stages and its results made available at the output. Beyond this point (the initial latency period), a new output for each successive operation would be available after every clock. Although the example uses the Add/Sub datapath, the same result would have been obtained from the Mul/Div datapath.

4.3.1 Pipeline Registers

Referring to Figure 8, three groups of registers would need to be designed to correspond with all three pipeline stages. All three of them must also be synched on the global clock input.

Each pipeline registers R1, R2 and R3 would store the intermediate result from each pipeline stage before it could be operated by the next stage. R1, R2 and R3 are not single registers though as they represent multiple registers that exist at each pipeline stage. Depending on the individual pipeline stage, registers may be needed to hold one or more outputs from that stage. As an example, the pre-align unit has both pre-aligned operands and the true operation select as the output. Registers must be made to hold all those.

4.3.2 Special Notes on the Exception Unit

All the pipelining discussion so far has not touched on the exception unit of the FPU. In actual fact, this unit is implemented to sit along with both pre-align units in the first stage of the pipeline. The exception unit analyzes the inputs to look for exceptions, thus their results must be made available for later pipeline stages to process. Therefore, it has to execute in the first stage of the pipeline and also conform to the single clock execution time constraint.

4.4 Built in Adder versus Custom Coded Adder

The design of the arithmetic unit stage relies heavily on the synthesizer choice of what type of adders and multipliers to use. Supported operators are Add, Subtract and Multiply. Therefore, all these three operations are performed using simple Verilog commands: "+", "-" and "*". Only the divider would require manual hand coding as synthesizers support only the divide operator ("/") for power of 2 divisors.

The justification for such design choice was because the Virtex-II chip where this FPU is going to be implemented has its own specific arithmetic optimizations. From the Virtex-II data sheet:

Fast Lookahead Carry Logic
Dedicated carry logic provides fast arithmetic addition and subtraction. The Virtex-II CLB has two separate carry chains. The height of the carry chains is two bits per slice. The carry chain in the Virtex-II device is running upward.

Arithmetic Logic
The arithmetic logic includes an XOR gate that allows a 2-bit full adder to be implemented within a slice. In addition, a dedicated AND (MULT_AND) gate improves the efficiency of multiplier implementation.

What the Virtex-II does is that it tries to solve the fundamental problem of binary addition, which is carry generation and propagation by using dedicated circuitry to quickly calculate the carry results. The Virtex-II also has its own dedicated 2-bit full adder in every slice. Every CLB (Configurable Logic Block) of the Virtex-II contains 4 slices; therefore plenty of hardware is available to implement arithmetic operations.

The advantage of using the dedicated arithmetic resources inside the FPGA is that they are really fast and they already there to be used. If a designer wishes to custom code the arithmetic units (adders and multipliers especially), the FPGA implementation may end up being slower. The reason for this is that the custom code implementation would need complicated routing inside the FPGA, negating all the speed advantage of its architecture. Custom codes also would take up plenty of area to implement on the FPGA, wasting the logic resources.

From "Real World FPGA Design" by Ken Coffman, a custom coded 8-bit ripple carry adder is twice as slow and also consumes twice the area compared to a synthesizer chosen adder. The synthesized comparison for the ripple carry adder versus the synthesis-tool version is shown in the table below:

Table 7: Custom coded RCA versus synthesizer-tool chosen adder

Type of Adder (8 bit)	*Custom-coded ripple carry*	*Synthesizer-tool version*
Resources:		
IOs	27	27
FG Function Generators	16	8
H Function Generators	0	0
CLB Flip Flops	7	0
Clock	77.6 MHz	135.1 MHz

4.5 The Exception Unit

The exception unit reads the input operands to the FPU and checks for four (4) types of exceptions. The exceptions are listed in Table 8.

Table 8: Exception Types Handled by the Exception Unit

Exception Type	*Description*
Infinity	Checks if one or both operands are infinite values
NaN	Checks if one or both operands are invalid numbers or values
Zero	Checks if one or both operands have a value of zero
Subnormal	Checks if one or both operands have a denormalized value

This unit takes three (3) inputs: clock, operand A and operand B while generating eight outputs as listed below:

- Opa_nan
- Opb_nan
- Opa_inf
- Opb_inf
- Opa_zero
- Opb_zero
- Opa_dn
- Opb_dn

4.6 The Pre-Align Unit for Addition and Subtraction Operations

This unit is only activated when addition or subtraction operations are performed. The functions of this unit are:

- Align the mantissas for addition and subtraction operations when their exponents do not match.
- Restore the implicit leading bit and also the guard, round and sticky (GRS) bits into the mantissa.
- Determine the true operation (addition or subtraction) that needs to be performed on both input mantissas.

The pre-align unit has three (3) inputs and four (4) outputs, both described in Table 9 and Table 10. The pre-align unit would take both input operands supplied by the user and operates on them as shown in Figure 9.

Table 9: Inputs to Pre-Align (ADD/SUB)

Inputs	*Description*
op_addsub	Indicates the operation requested by the user.
opA	First operand entered by the user (Operand A)
opB	Second operand entered by the user (Operand B)

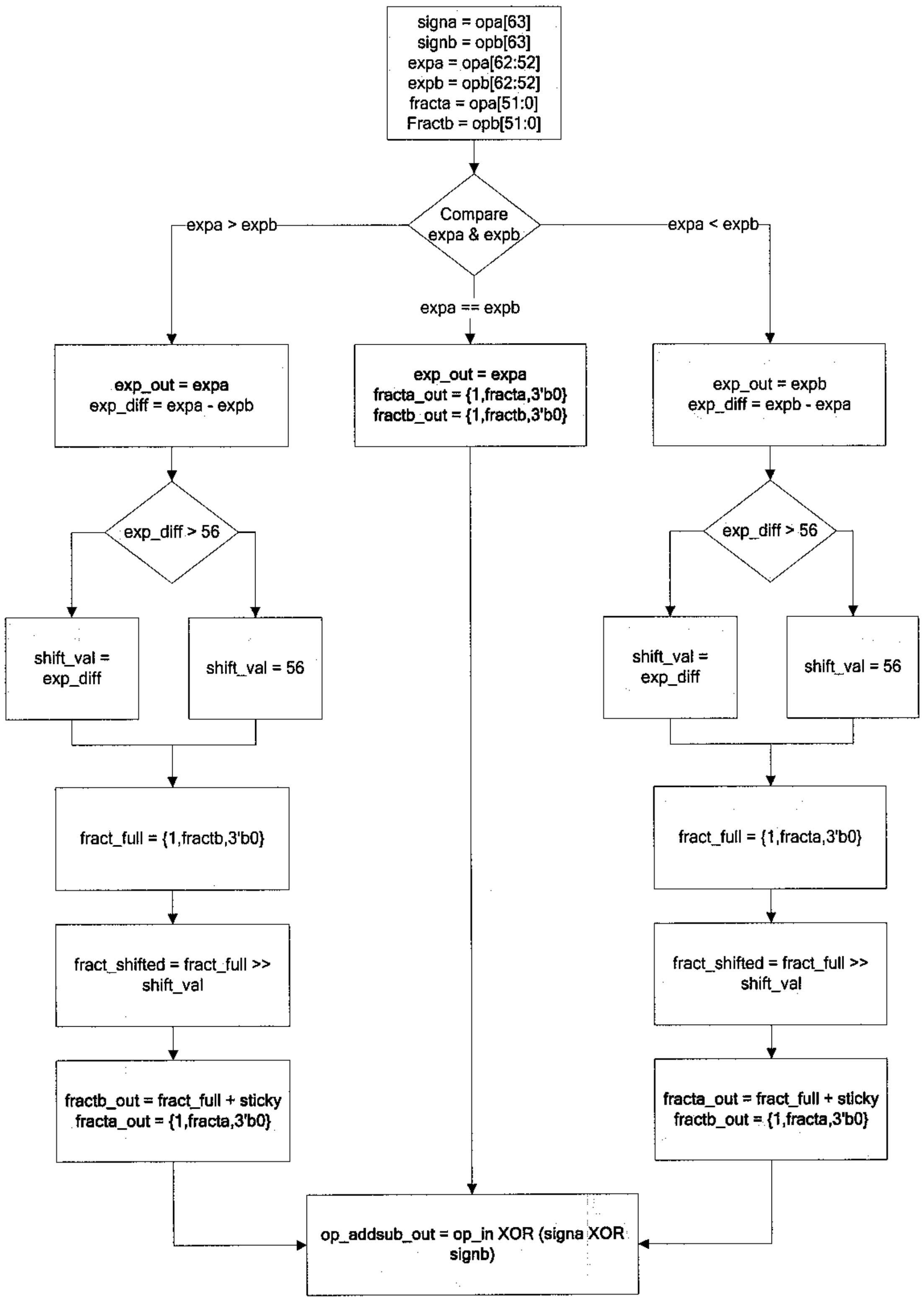
Table 10: Outputs from Pre-Align (ADD/SUB)

Outputs	*Description*
op_addsub_out	True operation to be performed on mantissas
fracta_out	First mantissa aligned
fractb_out	Second mantissa aligned
exp_out	Exponent for the result

Let A and B be the input operands. From Figure 9, the pre-align unit first splits both A and B into their sign, exponent and fraction (mantissa) components. The exponents of A and B are then compared. The next step would depend on the outcome of this comparison. If both exponents are equal, there is no need for mantissa alignment, thus the outputs (fracta_out, fractb_out, exp_out) are simply assigned to the inputs. The exponent of A is arbitrarily chosen as the output exponent. The exponent of B can equally be used as the output exponent.

If the exponent of A is larger than B, the fraction of B would need to be shifted to the right until both exponents are equal. The right shift is chosen to maintain the integrity of the B. If B's mantissa were to be shifted to the left, it would lose its most significant bits. This would unacceptably alter the numbers. The difference of the exponents would be used as the shift value. However, since the mantissa part is only 56 bits long, the maximum shift value is capped at 56. Shifting beyond 56 would only result in all zeros.

Since the IEEE-754 format specifies an implicit leading bit, this bit must be restored to the fraction before any shifting can take place. Besides the leading bit, three more bits must also be appended to the right hand end of the fraction. These bits are commonly known as the guard, round and sticky bits.



signa = opa[63]
signb = opb[63]
expa = opa[62:52]
expb = opb[62:52]
fracta = opa[51:0]
Fractb = opb[51:0]
Compare expa & expb
expa > expb
expa < expb
expa == expb
exp_out = expa
exp_diff = expa - expb
exp_out = expa
fracta_out = {1,fracta,3'b0}
fractb_out = {1,fractb,3'b0}
exp_out = expb
exp_diff = expb - expa
exp_diff > 56
exp_diff > 56
shift_val = exp_diff
shift_val = 56
shift_val = exp_diff
shift_val = 56
fract_full = {1,fractb,3'b0}
fract_full = {1,fracta,3'b0}
fract_shifted = fract_full >> shift_val
fract_shifted = fract_full >> shift_val
fractb_out = fract_full + sticky
fracta_out = {1,fracta,3'b0}
fracta_out = fract_full + sticky
fractb_out = {1,fracta,3'b0}
op_addsub_out = op_in XOR (signa XOR signb)

Figure 9: Pre-Align for Addition and Subtraction Algorithm

The variable called fract_full would be assigned to the restored version of B's mantissa. It is this variable which would be shifted. A similar operation would be performed if the exponent of A is smaller than B, with the mantissa of A being shifted instead of B. The shifted mantissa would then be sent as an output along with the fully restored version of the other input mantissa.

One last operation that the pre-align unit performs is to calculate the true operation to be carried out on the mantissas. The algorithm first checks the sign bits of both operands to determine which operation needs to be carried out on the mantissas. An XOR operation is performed on the sign bits of both operand and the result XOR-ed again with the operation selected. For an ADD operation, the same sign on both operands means that their mantissas should be added. If it is different, then they should be subtracted. For a SUB operation, the same sign on both operands would mean that their mantissas must be subtracted from each other vice-versa.

4.7 The Pre-Align Unit for Multiplication and Division Operations

This unit is only required for multiplication or division operations. Its functions are:

- Align the dividend for divide operations (dividend alignment) [4]
- Restore the implicit leading bit to the mantissa of both operands
- Perform addition or subtraction on the exponents
- Determine the sign of the multiplication or division result

This unit has four (4) inputs and four outputs, each defined in the tables that follow:

Table 11: Inputs to Pre-Align (MUL/DIV)

Inputs	*Description*
clk	Clock input to synchronize operation in the unit
op_muldiv	Indicates either multiply or divide operation
opA	First operand (Multiplicand or Dividend)
opB	Second operand (Multiplier or Divisor)

Table 12: Outputs from Pre-Align (MUL/DIV)

Outputs	*Description*
sign_out	Resultant sign of the multiplication or division
fracta_out	106 bit long dividend or multiplicand
fractb_out	53 bit long divisor or multiplier
exp_out	Exponent result of the multiplication or division

The pre-align unit execution is split into two stages. First is the dividend alignment stage and second is addition/subtraction of aligned exponents. Figure 10 on the next page shows the flowchart for the dividend alignment operations by the pre-align unit.

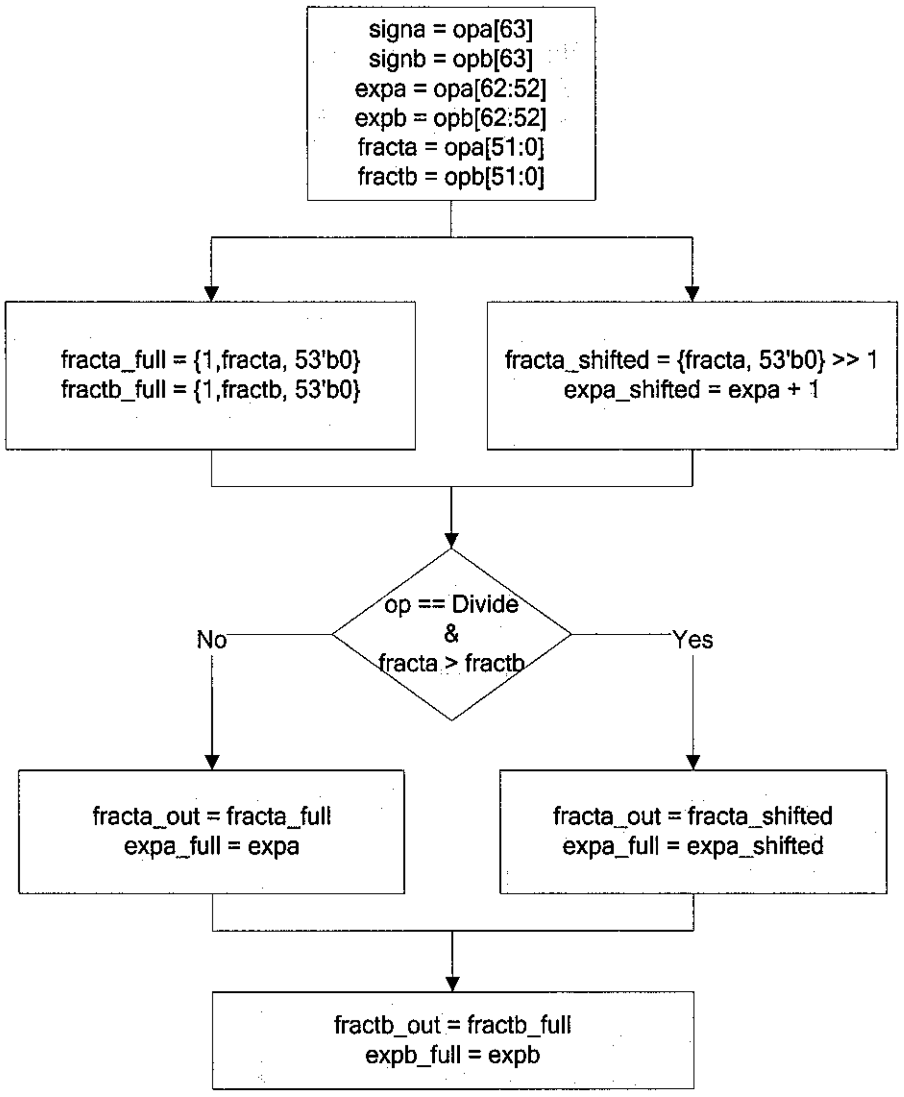


Figure 10: Dividend Alignment in Pre-Align Unit for Multiplication and Division Operations

The process starts by extracting the sign, exponent and mantissa components of both operands. Let both operands be A and B. The implicit leading bits for the mantissa of A and B is then restored. Both the aligned and non-aligned dividend (A) is first formed. If the operation is a divide and the fraction of A (dividend) is larger than B (divisor), the output dividend will be assigned to the shifted dividend. Thus, the dividend alignment process is completed. Do note that the aligned length of A is padded up to 106 bits to prepare it for division operations. For multiply operations, the alignment would not be done and the extra 53 bits padded to its least significant positions would be ignored.

Figure 11 meanwhile shows the exponent addition and subtraction operation which takes place after the dividend alignment. Multiplication operations would require addition of the exponents while division operations would require subtraction. Since the exponents are stored in biased representation, the bias value must be subtracted or added from the result to give to restore the proper bias to it. [3]

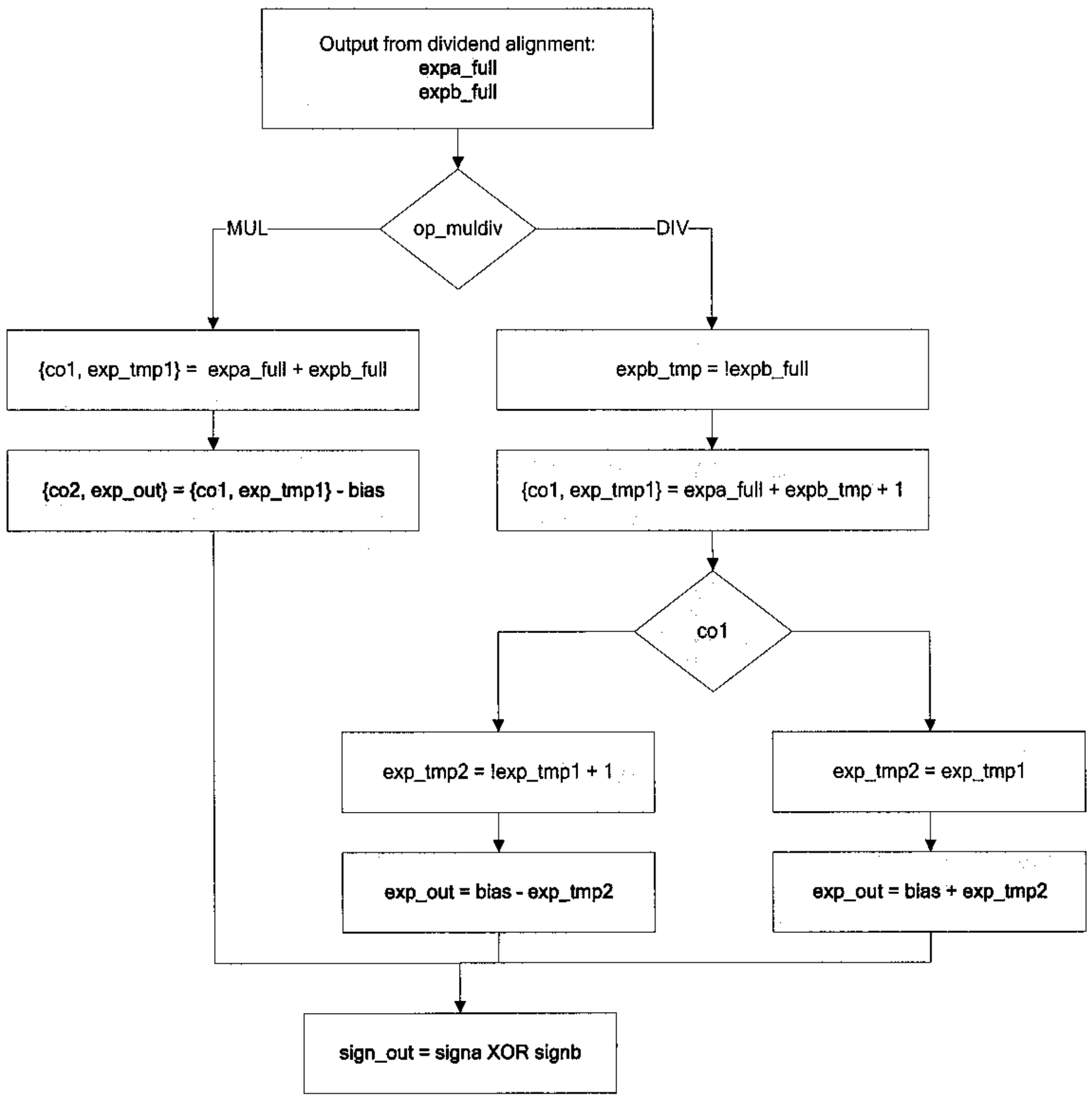

Output from dividend alignment:
expa_full
expb_full
op_muldiv
MUL
DIV
{co1, exp_tmp1} = expa_full + expb_full
{co2, exp_out} = {co1, exp_tmp1} - bias
expb_tmp = !expb_full
{co1, exp_tmp1} = expa_full + expb_tmp + 1
co1
exp_tmp2 = !exp_tmp1 + 1
exp_out = bias - exp_tmp2
exp_tmp2 = exp_tmp1
exp_out = bias + exp_tmp2
sign_out = signa XOR signb

Figure 11: Addition and Subtraction of Exponents for Multiplication and Division Operations

The bias value for the 64-bit double-precision is 1023 in decimal. Therefore, for the addition of biased exponents, the bias value of 1023 must be subtracted the results. Conversely, the value of 1023 must be added to the subtraction result of biased exponents. The operation here is very similar to what is done in the addition and subtraction unit for mantissas, the only exception being the restoring of the bias value that needs to be carried out by this unit. After the addition or subtraction of biased exponents, the sign of the result must also be determined. Since the operation involves only multiplication or division, the resultant sign is simply the XOR of the sign for both input operands.

4.8 Addition and Subtraction Unit [4]

This particular unit (referred to as Add/Sub Unit from this point onwards) performs addition and subtraction on only the mantissa of input operands, The resultant exponent is obtained from the pre-align unit, thus only the resultant mantissa needs to be calculated here. The inputs and outputs of this unit are described in Tables 13 and 14 below.

Table 13: Inputs to Add/Sub Unit

Inputs	*Description*
clk	Clock input to synchronize operation within the unit
op_addsub	True operation to be performed on the mantissas. Obtained from the output of Pre-Align (Add/Sub) Unit.
fracta	Mantissa of first operand; aligned and restored to 56 bits length.
fractb	Mantissa of second operand; aligned and restored to 56 bit length
signa	The sign of the first input operand, needed for the resultant sign

Table 14: Outputs from Add/SubUnit

Outputs	*Description*
sum	Sum or difference of both input mantissas
co	Carry out from the addition or subtraction operation
sign_sum	The sign of the result.

The execution of the Add/Sub unit is illustrated in Figure 12. The unit first loads all its inputs into the appropriate wires. Since the mantissas at the input are already aligned, the addition or subtraction can be done directly. Both operations are carried out in two's complement style[§]. The op_addsub input wire is used to invert all the bits of the second mantissa (fractb) and also as a first bit carry-in input at the full adder. The effect is that during subtraction operations, the inverse of fractb (in two's complement) is obtained and then added to the first operand (fracta) for the final result. After each subtraction, the carry-out will be examined to see if it is one (1) or zero (0). If it is zero that means the magnitude of A is smaller than B, thus the correct result is given by the two's complement of the result, and the sign inverted. Else, the sign of the result is the same as the sign of the first operand.

[§] Please refer to Literature Review (Section 2.5) for proof that this can be done.

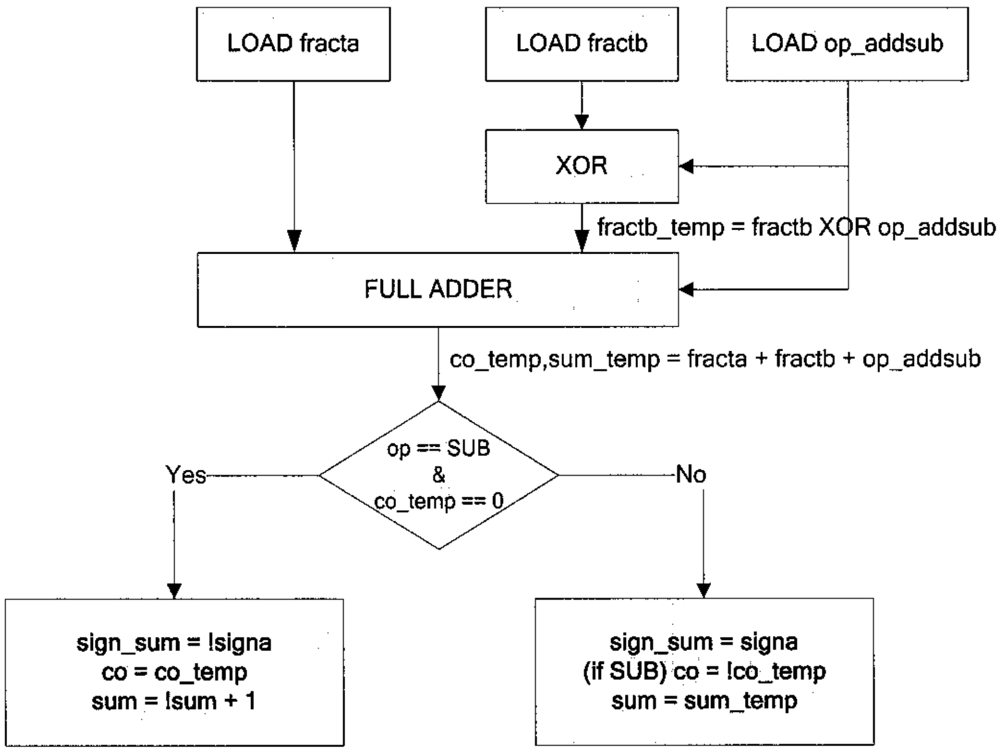


Figure 12: Add/Sub Unit Operation Flowchart

4.9 Multiplication Unit

This unit also operates on mantissas only. The task of adding the exponents during multiplication is already relegated to the Pre-Align (MUL/DIV) unit. Its input and outputs are as follows

Table 15: Inputs to Multiplication Unit

Inputs	*Description*
clk	Clock input to synchronize operation within the unit
fracta	53 bit long mantissa of the first operand (the multiplicand)
fractb	53 bit long mantissa of the second operand (the multiplier)

Table 16: Outputs from Multiplier Unit

Outputs	*Description*
product	106 bit long product of fracta and fractb

Both input mantissas are only 53 bits long since the guard bits do not need to be used. Both fracta and fractb is connected to the output of the Pre-Align (MUL/DIV) unit (which restores the implicit leading bit to the mantissas). The product is formed by using the Verilog multiply operator to obtain an optimized synthesized logic.

4.10 Division Unit

Like the multiplication unit, the division unit operates only on the mantissas. Its function is to accept two input mantissas and divides them to produce a quotient and a remainder string. The inputs and outputs of this unit are shown in Table 17 and Table 18.

Table 17: Inputs to Division Unit

Inputs	*Description*
clk	Clock input to synchronize operation within the unit
dividend	106 bit long dividend mantissa
divisor	53 bit long divisor mantissa

Table 18: Outputs from Division Unit

Outputs	*Description*
quotient	The quotient result from the division
remainder	The remainder result from the division

The division is performed using the algorithm for unsigned binary division. This algorithm requires that the dividend be twice as long as the divisor to execute the division. To generate the 106 bit long dividend, the mantissa is padded with zeros to the right (least significant position). This is accomplished by the Pre-Align Unit (MUL/DIV). Padding zeros to the right is merely adding zeros to the right of the radix point, thus the magnitude of the number is not affected at all. The dividend must also be aligned so that its first 53 bits are smaller than the divisor. Otherwise, divide overflow may occur, where there are insufficient bits to store the resulting quotient. [4]

Because of the algorithm employed, the division unit must run in a separate clock domain. More specifically, the division unit requires a clock that is 53 times faster than the external clock as 53 shifts and subtractions are required to compute the quotient and the remainder. However, this is still a work in progress, thus the divisor implemented in the current design is still a behavioral style "/" operator which is unsynthesizable.

4.11 The Normalization Unit

The output from the arithmetic stage of the FPU is usually a non-normalized result. Therefore, the function of the normalization unit would be to normalize the arithmetic result before it is sent as an output from the FPU. In the actual implementation, one large unit is used to contain both this normalization unit and the rounding unit. Both operate within one pipeline stage. However, it is more convenient to discuss them as separate entities. The inputs and outputs used by the normalization unit only are listed in the tables below:

Table 19: Inputs to Normalization Unit

Inputs	*Description*
clk	Clock input to synchronize operation within the unit
fract_in	Result mantissa from the arithmetic stage
exp_in	Result exponent from the arithmetic stage

Table 20: Outputs from Normalization Unit

Outputs	*Description*
fract_shifted	The normalized mantissa
exp_shifted	The exponent after normalization
denormalized	Indicates that a denormalized number is formed
overflow	Indicates that an overflow happened

Normalizing a mantissa would require shifting it to the left or right until its most significant bit is a one (1). Figure 13 illustrates the process. The first step of normalizing would be to count the number of leading zeros in the mantissa. This "counter" is implemented as a fast and wide multiplexer (mux) that selects a number ranging from 0 to 105 at its input to be sent to its output, depending on the number of leading zeros in the mantissa. This concept is shown in Figure 14. The direction to shift is then determined using the number of leading zeros. Because of the way the input is defined, the right shift is only necessary when the there are no leading zeros. The exponent is then checked to see if it is already at the maximum value of 11111111110. If it is, a right shift cannot be done as it would cause an overflow. Therefore, a combination of right shift and maximum exponent will result in an overflow and the corresponding flag must be set to indicate this condition.

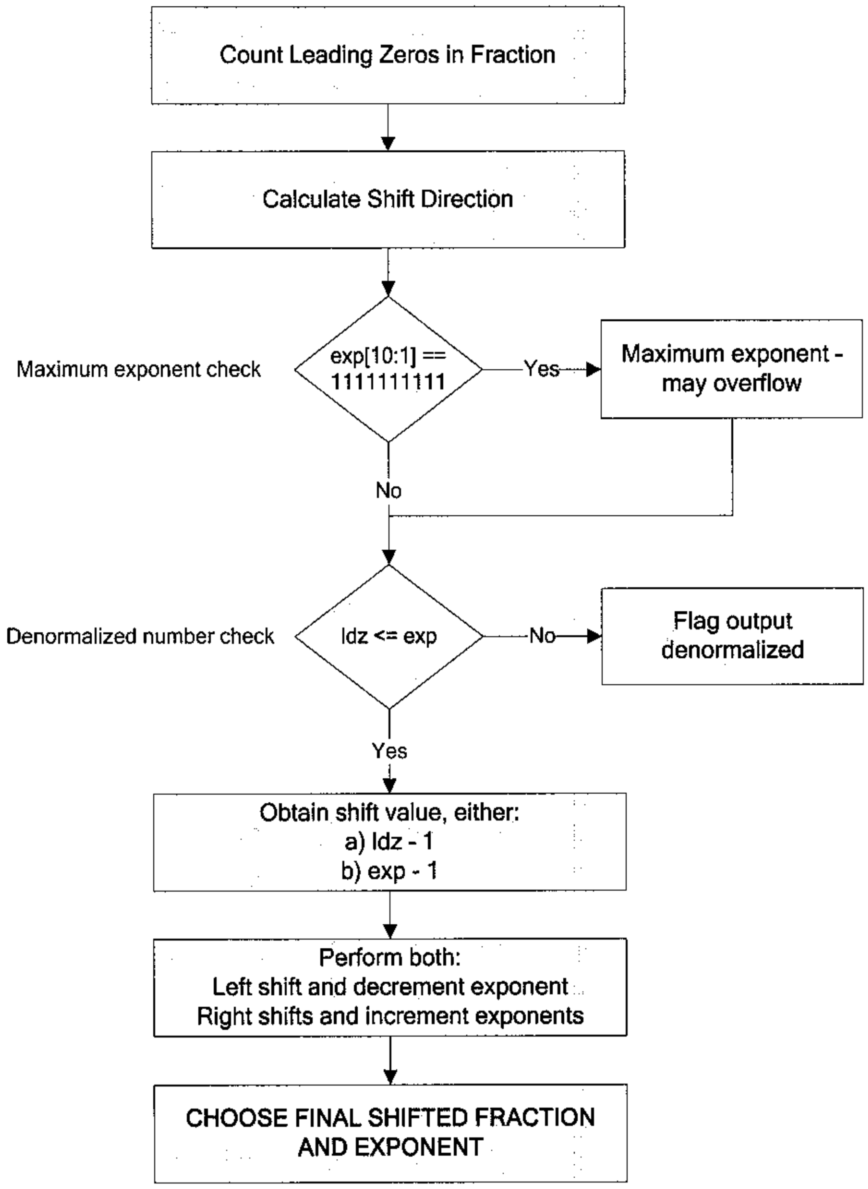

Count Leading Zeros in Fraction
Calculate Shift Direction
Maximum exponent check
exp[10:1] == 1111111111
Yes
Maximum exponent - may overflow
No
Denormalized number check
ldz <= exp
No
Flag output denormalized
Yes
Obtain shift value, either: a) ldz - 1 b) exp - 1
Perform both: Left shift and decrement exponent Right shifts and increment exponents
CHOOSE FINAL SHIFTED FRACTION AND EXPONENT

Figure 13: Normalization Flow

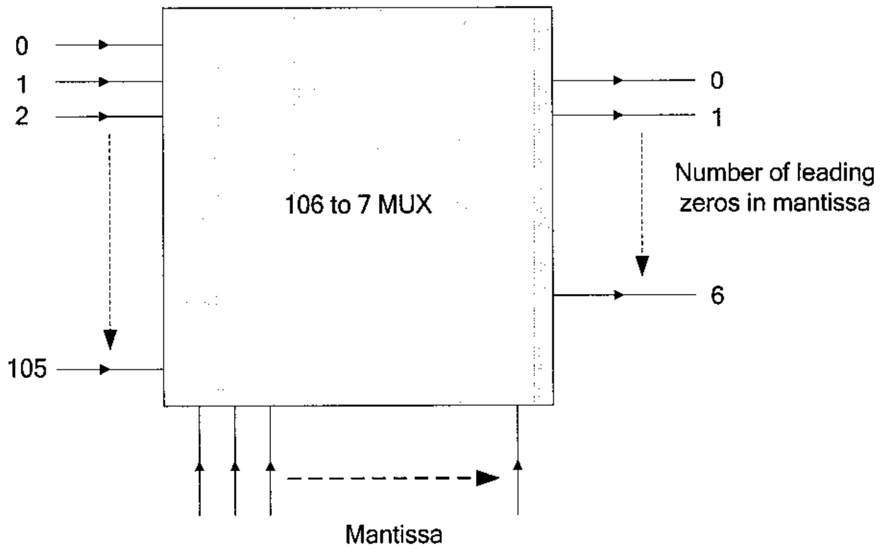


Figure 14: Mux to Check the Number of Leading Zeros

A check for conditions leading to denormalized number is also done. Each left shift would require the exponent to be decremented. Thus, if the number of left shifts exceeds the value of the exponent, a denormalized number will result. A flag is also set to indicate such condition. Since the exponent cannot be decremented beyond its minimum value of 00000000001; there is a limit to how many left shifts can be performed to the mantissa. Effectively, the number of left shifts is either equal to the number of leading zeros minus one (normalized result) or the exponent minus one (denormalized result). After all calculations are done, both the left and right shift is done on the mantissa. The correct result is then chosen using another multiplexing logic depending on the direction of shift (determined earlier).

4.12 The Rounding Unit

The rounding unit forms the second half of the larger normalization and rounding unit. Its function is to round the normalized mantissa into IEEE-754 compliant format. The IEEE-754 specification provides for four (4) rounding modes: round to nearest, round to zero, round up (positive infinity) and round down (negative infinity). The inputs and outputs of the rounding unit are summarized in Table 21 and 22 below:

Table 21: Inputs to Rounding Unit

Inputs	*Description*
clk	Clock input to synchronize operation within the unit
fract_shifted	Normalized mantissa from the normalization unit
exp_shifted	Normalized exponent from the normalization unit
sign	The sign of the result
r_mode	Rounding mode selected

Table 22: Outputs from Rounding Unit

Outputs	*Description*
fract_out	Rounded mantissa
exp_out	Rounded exponent
inexact	Inexact – indicates mantissa is truncated during rounding

The operation of the rounding unit is shown as a flow chart in Figure 15. Numbers could either be rounded into a lower magnitude (called X- in Figure 15) or a higher magnitude (X+ in Figure 15). The value of X- is simply the normalized mantissa, with all the extra bits to the right truncated. Since the mantissa has 53 bits, every bit starting from the 54th bit will be discarded to form X-. The value of X+ is obtained by adding a 1 to the least significant position in X-. Both X- and X+ therefore form the two end products of the rounding unit.

$$X- = (1.b1b2b3.....b52)2 \times 2E$$

$$X+ = [(1.b1b2b3.....b52) + (0.000.....1)]2 \times 2E$$

Depending on the rounding mode, either one of these will be chosen as the final rounded output. The choosing logic forms the largest part of this unit. The rounding logic for all modes is relatively simple, with the exception of round to nearest.

One notable catch of forming X+ is that adding "1" at the least significant position may generate a carry that propagates all the way to the most significant bit. This will be a significant issue if X+ were to be chosen as the final rounded output. This condition can be detected using the carry-out from the most significant stage of X+. To compensate, X+ must be shifted to the right and the exponent incremented by 1. Incrementing the exponent presents yet another problem if the exponent is already at maximum value. More logic will have to be spent to detect this condition and set the rounded outputs properly according to the IEEE specification.

For round to zero, the rounded output is always X-. For round up (positive infinity), x+ would be chosen if the number is positive, while x- would be chosen if the number is negative. Round down (negative infinity) is the reverse of round up X- would be chosen for a negative number while X+ is chosen for a positive number.

Round to nearest is the most complicated of all rounding modes. The IEEE- 754 specification requires this mode to be the default rounding mode if no rounding modes are specified. The FPU however, must rely on the software to set the default rounding mode as it does not include the logic to set it by default. This makes sense since it is much easier to implement in software rather than hardware and no additional hardware cost is incurred. In this mode, the first truncated bit (refereed to as last bit in Figure 15) is checked to see if it is zero. If this bit is zero (0), then the number is closer to the lower magnitude thus X- would be chosen. If it is a one (1), then a tie condition is checked. A tie condition would happen if every bit to the right of the first truncated bit is a zero. If one of those bits is not zero, then the number is closer to the larger magnitude and X+ would be chosen. In case of a tie, the number is equally spaced between X- and X+, therefore a tie-breaker is used to decide on the final result. The tie-breaker is the least significant bit (LSB) in X- and X+ (they are mutually exclusive). The final output would be equal to X- if its final bit is zero, else it will be X+. [2,3,5] The logic used to determine the result for the round to nearest mode is based on the truth table below:

Table 23: Truth Table Showing the Select Logic for Round to Nearest

First Truncated Bit	*Rest of Truncated Bit*	*LSB of X-*	*Result (0 for X-, 1 for X+)*
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

The first three columns of Table 23 represents the inputs to consider for choosing the result while the last column shows the result itself as a function of the inputs. A zero (0) on the result means X- is chosen while a one (1) means X+ is chosen. Using the Karnaugh Map minimization technique, the logic used to select the result is reduced to:

Result = (First Truncated Bit) OR (Rest of Truncated Bit AND LSB of X-)

Figure 15 does not shown the overflow handling inside the rounding unit. The IEEE-754 standard has specifically defined the rounded output of a floating point number whenever an overflow occurs. In case of overflow, the number would either be rounded to infinity or to the largest number that can be represented in the format. Figure 16 provides an alternative view of the algorithm applied in this unit.

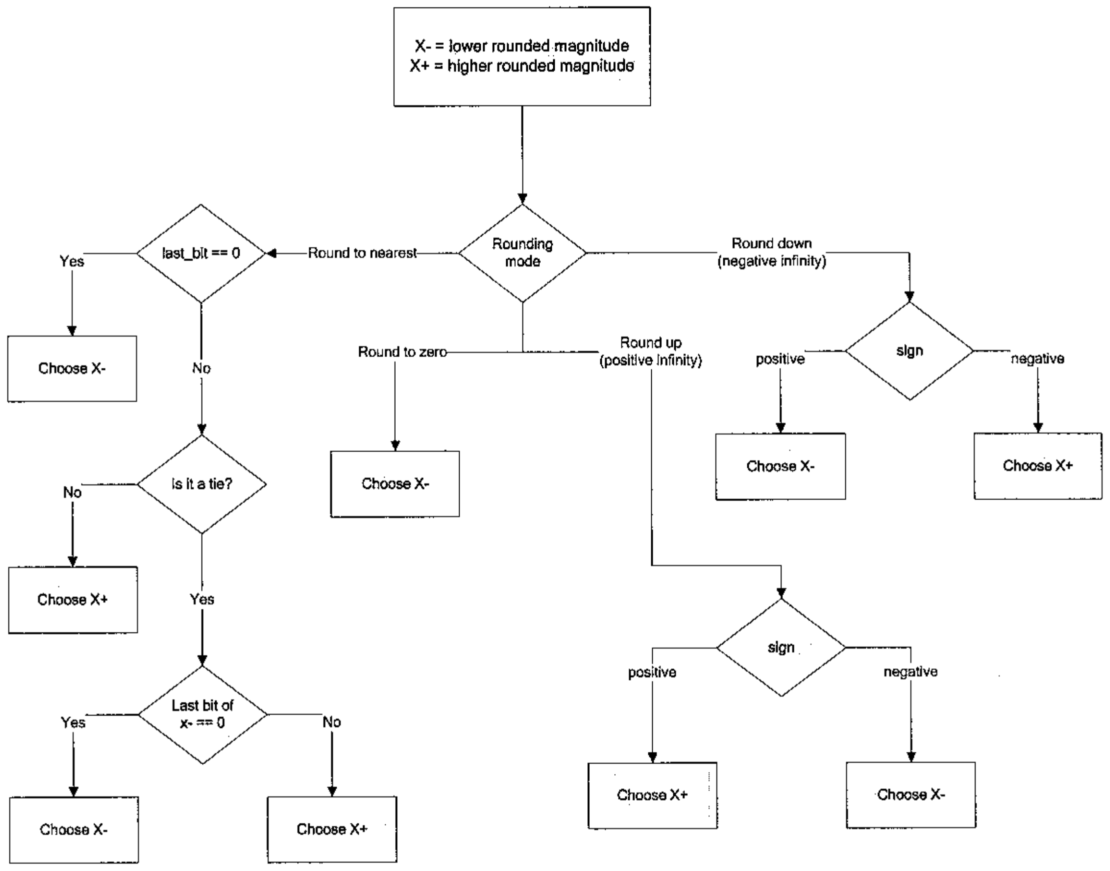


Figure 15: Rounding Unit Operation Flow

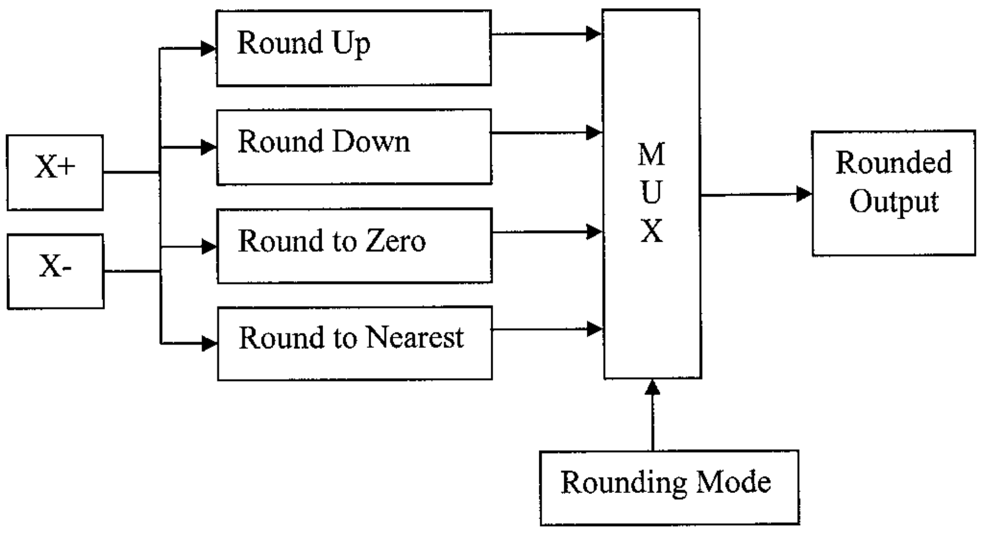


Figure 16: Alternative View of the Rounding Algorithm

CHAPTER 5

CONCLUSION AND RECOMMENDATION

5 CONCLUSION AND RECOMMENDATION

5.1 Further Improving FPU Performance

Performance wise, there are still a few areas that can be improved in the design. These improvements could come from addition of more pipeline stages to the current design. More specifically, two additional pipeline stages could be gained by splitting the first and last pipeline stages into two individual stages.

Both pre-align units actually perform two separates operations. For the ADD/SUB pre-align unit, the first operation is to compare and obtain the difference in exponents while the second operation is right shifting of the smaller mantissa. Since the first operation involves a potentially large and complex subtraction (to obtain the difference), doing the subtraction in one stage of the pipeline and then the shifting in the next stage of the pipeline could result in considerable speed improvement (since each pipeline stage is now shorter and takes lesser time).

For the MUL/DIV pre-align unit, some reorganization would be needed to split the stage into two. The dividend alignment operation is relatively simple, thus it does not need to be in a separate stage. The addition and subtraction of exponent operation however is big and two stages can be identified here. First would be the addition or subtraction of the exponents and second would be the addition or subtraction with the bias value needed to restore the proper bias to the exponents. Therefore, the first add-subtract operation can be done along with the dividend alignment while the second add-subtract placed in a later pipeline stage.

The third stage also consists of two separate operations, normalizing and rounding. If these operations were to be split into two stages, the bottleneck at stage 3 of the current pipeline could also be reduced.

Both optimizations above, coupled with the suggested optimization below, would result in a faster FPU. The pipeline structure of the FPU with a 5 stage pipeline is illustrated in Figure 17 below:

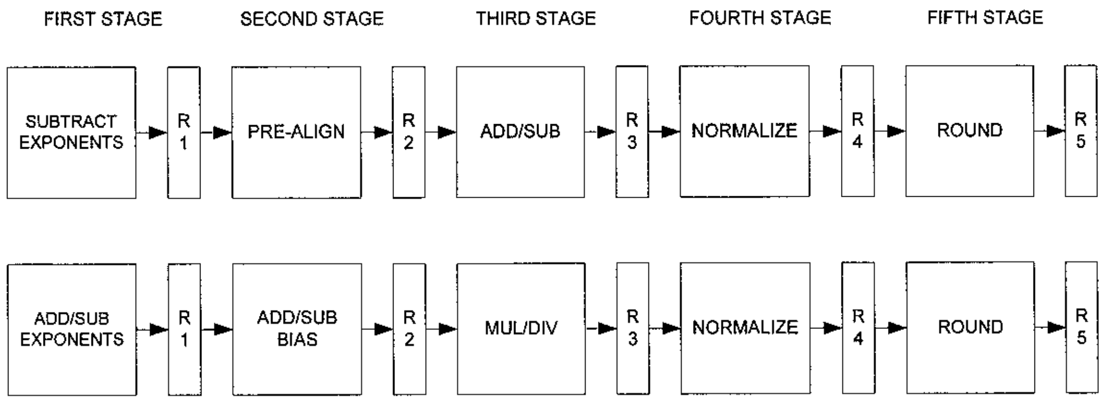


Figure 17: Pipeline Stages and Registers (5 levels)

5.2 Using more Advanced Adder and Multiplier Algorithm

Besides adding more pipeline stages as suggested above, the arithmetic calculation stage must also be fast enough to keep up. If not, every other pipeline stage would be limited by the clock speed of the arithmetic stage.

Even though it has been mentioned before that using the synthesizer optimized blocks would be the best option, there is a third option which may yield better results. That method would be to custom code the adders and multipliers, but with compiler directives and vendor specific libraries (if available) so that the custom codes would be implemented using the dedicated arithmetic resources on the FPGA. This approach offers the best of both worlds as more advanced arithmetic implementation could be used with the fast arithmetic resources. The only problem with this approach is to find out how compiler directives should be used and what each directive would mean. Vendor specific libraries are installed on most RTL IDEs (such as ActiveHDL) but problems remain on their exact usage.

For example, a custom coded adder could be a cascaded structure consisting of 4-bit wide carry-lookahead logic. 4 bit is generally the optimal width for the synthesizer to improve area and speed performance. Longer adders (56 bit long for the FPU) would need to cascade the 4 bit adder cells using carry lookahead logic too. Other interesting arithmetic algorithms could also be explored, such as carry select adders, carry save adders and other unsigned multiplier algorithms.

5.3 Recommendations for Future Works

Another recommendation is that more studies will be made on how to increase the level of "parallelism" of the FPU. This would require much more work on the control unit.

The current design used here actually has three different execution units, one addition/subtraction, one for multiplication and one for division. Therefore, three separate instructions could potentially run in parallel. Although some units along the datapath are shared, such as the normalization/rounding unit, these could easily be duplicated for every operation running in parallel. To implement such scheme however, would require an advanced scheduling algorithm that would recognize ahead of time; which instruction are non-dependent and can be executed in parallel.

Another area of improvement would be in logic minimization. Certain codes used in the design is still behavioral based and may generate excessive logic. The codes could be optimized in order to synthesize a more efficient logic, one that could perform the same operation using fewer gates. This would also lower the area utilization on the FPGA, allowing even more functions to be added to the chip.

5.4 Unsupported Features

The design currently does not have a synthesizable division algorithm. Full support involving denormalized numbers is also not yet available. Divisions especially will encounter serious problems if a denormalized input was used. The checks for underflow and overflow are also not fully functional for multiplication and division operations.

REFERENCES

[1] Rudolf Usselmann, Open Floating Point Unit, Free IP Cores Project, www.opencores.org (last accessed 25/8/2003)

[2] Michael L. Overton, 2001, "Numerical Computing with IEEE Floating Point Arithmetic", Society for Industrial and Applied Mathematics (SIAM)

[3] William Stallings, 2003, "Computer Organization & Architecture: Designing for Performance", Prentice Hall

[4] M. Morris Mano, 1993, "Computer System Architecture, Third Edition" Prentice Hall

[5] Theorem 5 by D.Goldberg, What every computer scientist should know about floating point arithmetic, ACM Computer Survey, 1991

[6] Weng Fook Lee, 2003, "Verilog Coding for Logic Synthesis", John Wiley & Sons

[7] Guy Even & Wolfgang J. Paul, May 2000, "On the design of IEEE FPU", IEEE Transactions on Computers, vol. 49

APPENDIX A1

SOURCE CODE FOR FPU (TOP-LEVEL ARCHITECTURE)

```
//---------------------------------------------------------------------------------------------------
//
// Title      : Double-precision FPU - 64-bit
// Design     : Master FPU module
// Author     : Ng Kiat Hong
// ID                        : 1450
//---------------------------------------------------------------------------------------------------
//
// Description : This is the top level file that binds together all the other smaller units
//
//---------------------------------------------------------------------------------------------------
`timescale 1ns / 100ps

/*

FPU Operations (opcode):
========================

00 = add
01 = sub
10 = mul
11= div

Rounding Modes (r_mode):
========================
00 = round to nearest
01 = round to zero
10 = round up (+inf)
11 = round down (-inf)

*/

module fpu ( clk, reset, opcode, r_mode, opa, opb, out,
                                        snan, qnan,  div_by_zero, overflow, inf, ine, zero);

//main ports
input clk, reset;
input [1:0] opcode;
input [1:0] r_mode;
input [63:0] opa, opb;
output [63:0] out;

reg [63:0] out;

//exception outputs
output snan, qnan;
output div_by_zero;
output overflow;
//output underflow;
output inf, ine, zero;

//Reg to double-sync inputs
reg [1:0] opcode_sync1, opcode_sync2;
reg [1:0] r_mode_sync1, r_mode_sync2;
reg [63:0] opa_sync1, opa_sync2;
reg [63:0] opb_sync1, opb_sync2;

//Asynchronous RESET and double-sync of all external inputs
//Double-sync is helpful in avoiding metastability problems - metastable - state between 0 & 1
//Reason is: external inputs may not conform to setup & hold times
//Inputs that do not meet setup&hold times may drive latches/FFlops to metastable states
//Double synching would force a metastable input to the correct stable state by driving another signal
//that matches the setup and hold times
```

```
always @(posedge clk or posedge reset)
        begin
                if (reset)
                        begin
                                opcode_sync1 <= 2'b0;
                                opcode_sync2 <= 2'b0;
                                r_mode_sync1 <= 2'b0;
                                r_mode_sync2 <= 2'b0;
                                opa_sync1               <= 64'b0;
                                opa_sync2               <= 64'b0;
                                opb_sync1               <= 64'b0;
                                opb_sync2               <= 64'b0;
                        end
                else
                        begin
                                opcode_sync1 <= opcode;
                                opcode_sync2 <= opcode_sync1;
                                r_mode_sync1 <= r_mode;
                                r_mode_sync2 <= r_mode_sync1;
                                opa_sync1               <= opa;
                                opa_sync2               <= opa_sync1;
                                opb_sync1               <= opb;
                                opb_sync2               <= opb_sync1;
                        end
        end


//----------------------------------------//
// ALL MODULE INSTANTIATION STARTS HERE //
//----------------------------------------//


//clock delayed signals to propagate signals to the proper stage
reg signa_r2, signa_r3, signa_out;
reg [1:0] opcode_r2, opcode_r3, opcode_out; //opcode_out is needed at output logic
reg [1:0] r_mode_r2, r_mode_r3;
reg sign_muldiv_r3;  //clock delay coded before normalization block
reg [10:0] exp_addsub_r3;

always @(posedge clk)
        signa_r2 <= opa_sync2[63];

always @(posedge clk)
        signa_r3 <= signa_r2;

always @(posedge clk)
        signa_out <= signa_r3;

always @(posedge clk)
        opcode_r2 <= opcode_sync2;

always @(posedge clk)
        opcode_r3 <= opcode_r2;

always @(posedge clk)
        opcode_out <= opcode_r3;

always @(posedge clk)
        r_mode_r2 <= r_mode_sync2;

always @(posedge clk)
        r_mode_r3 <= r_mode_r2;



//------------------------------------------------------------------------------------------
//Exception unit instantiation
//------------------------------------------------------------------------------------------

//Wire declaration
wire opa_nan, opb_nan;
wire opa_inf, opb_inf;
wire opa_zero, opb_zero;
wire opa_dn, opb_dn;
```

```
exception unit1 ( clk, opa_sync2[62:0], opb_sync2[62:0],
                                        opa_nan, opb_nan,
                                        opa_zero, opb_zero,
                                        opa_inf, opb_inf,
                                        opa_dn, opb_dn );



//-------------------------------------------------------------------------------------
//Pre-Align (ADD/SUB) unit instantiation
//-------------------------------------------------------------------------------------

//Wire declaration
wire [55:0] fracta_addsub, fractb_addsub;
wire [10:0] exp_addsub;
wire op_addsub;

pre_align unit2 ( clk, opcode_sync2[0], opa_sync2, opb_sync2,
                                        fracta_addsub, fractb_addsub, exp_addsub, op_addsub );



//-------------------------------------------------------------------------------------
//Pre-Align (MUL/DIV) unit instantiation
//-------------------------------------------------------------------------------------

//Wire declaration
wire sign_muldiv;
wire [105:0] fracta_muldiv;
wire [52:0] fractb_muldiv;
wire [10:0] exp_muldiv;

pre_align_mul unit3 ( clk, opcode_sync2[0], opa_sync2, opb_sync2,
                                              sign_muldiv, fracta_muldiv, fractb_muldiv, exp_muldiv );



//-------------------------------------------------------------------------------------
//ADD/SUB unit instantiation
//-------------------------------------------------------------------------------------

//Wire declaration
wire [55:0] sum;
wire co;
wire sign_sum;

add_sub unit4 ( clk, op_addsub, fracta_addsub, fractb_addsub, signa_r2,
                                        sum, sign_sum, co );



//-------------------------------------------------------------------------------------
//MUL unit instantiation
//-------------------------------------------------------------------------------------

//Wire declaration
wire [105:0] product;

mul unit5 ( clk, fracta_muldiv[105:53] , fractb_muldiv, product );



//-------------------------------------------------------------------------------------
//DIV unit instantiation
//-------------------------------------------------------------------------------------

//Wire declaration
wire [52:0] quotient, remainder;

div unit6 ( clk, fracta_muldiv, fractb_muldiv, quotient, remainder);
```

```
//-----------------------------------------------------------------------------------------
//Normalization & Rounding unit instantiation
//-----------------------------------------------------------------------------------------

//Wire/Reg declaration for inputs to norm_round - reg bcoz case statements had 2 be used
reg [105:0] fract_dn;
reg [10:0] exp_dn;
reg sign_dn;

wire [105:0] fract_sum_r3;
wire [105:0] fract_quo_r3;
wire fract_addsub_zero;                    //Unrelated - Used only in output stage - indicates add/sub output is zero

//Wire declaration for outputs
wire [52:0] fract_out;
wire [10:0] exp_out;
wire dn_out, ine_out, overflow_out, norm_inf;

//clk delay sign_muldiv signal from first stage to third stage
always @(posedge clk)
        sign_muldiv_r3 <= sign_muldiv;

always @(posedge clk)
        exp_addsub_r3 <= exp_addsub;

//Padding input fraction to proper length
assign fract_sum_r3 = {co, sum, {49{1'b0}}};
assign fract_quo_r3 = {quotient, {53{1'b0}}};

//Check for zero in ADD/SUB result
assign fract_addsub_zero = ~(|fract_sum_r3[105:49]);


//choose correct fraction input to norm/round stage
always @(opcode_r3 or fract_sum_r3 or product or fract_quo_r3)
        begin
                case (opcode_r3)
                        00,01: fract_dn = fract_sum_r3;
                        10: fract_dn = product;
                        11: fract_dn = fract_quo_r3;
                endcase
        end

//choose the correct exponent input
always @(opcode_r3[1] or exp_addsub_r3 or exp_muldiv)

        begin
                case (opcode_r3[1])
                        1'b0: exp_dn = exp_addsub_r3;
                        1'b1: exp_dn = exp_muldiv;
                endcase
        end

//choose the correct sign input
always @(opcode_r3[1] or sign_sum or sign_muldiv_r3)
        begin
                case (opcode_r3[1])
                        0: sign_dn = sign_sum;
                        1: sign_dn = sign_muldiv_r3;
                endcase
        end

norm_rnd unit7 (clk, fract_dn, exp_dn, sign_dn, opcode_r3, remainder, r_mode_r3,
                                        fract_out, exp_out, dn_out, ine_out, overflow_out, norm_inf );
```

```
//-----------------------------------------------------------------------------------------
//FPU output logic
//-----------------------------------------------------------------------------------------


/*-----------------------------------------------------------------------------------------
List of signals that needs to be clock propagated here (3rd stage output)
1 - sign for result - get from normalization unit input
2 - all exception unit signals: opa/b_nan/zero/inf/dn
3 - true op for addition/subtraction
4 - input opcode - already done on top!
-----------------------------------------------------------------------------------------*/

//Reg to clock propagate signals
reg sign_out;
reg opa_nan_r3, opa_nan_out, opb_nan_r3, opb_nan_out;
reg opa_inf_r3, opa_inf_out, opb_inf_r3, opb_inf_out;
reg opa_zero_r3, opa_zero_out, opb_zero_r3, opb_zero_out;
reg opa_dn_r3, opa_dn_out, opb_dn_r3, opb_dn_out;
reg op_addsub_r3, op_addsub_out;
reg fract_addsub_zero_out;

//Clock delay the signals
always @(posedge clk)
		begin
				sign_out <= sign_dn;

				opa_nan_r3 <= opa_nan;
				opa_nan_out <= opa_nan_r3;

				opb_nan_r3 <= opb_nan;
				opb_nan_out <= opb_nan_r3;

				opa_inf_r3 <= opa_inf;
				opa_inf_out <= opa_inf_r3;

				opb_inf_r3 <= opb_inf;
				opb_inf_out <= opb_inf_r3;

				opa_zero_r3 <= opa_zero;
				opa_zero_out <= opa_zero_r3;

				opb_zero_r3 <= opb_zero;
				opb_zero_out <= opb_zero_r3;

				opa_dn_r3 <= opa_dn;
				opa_dn_out <= opa_dn_r3;

				opb_dn_r3 <= opb_dn;
				opb_dn_out <= opb_dn_r3;

				op_addsub_r3 <= op_addsub;
				op_addsub_out <= op_addsub_r3;

				fract_addsub_zero_out <= fract_addsub_zero;
		end


//-----------------------------------------------------------------------------------------
//Check for exceptions
//-----------------------------------------------------------------------------------------

//Check 1: NaN
wire addsub_nan, mul_nan, div_nan;

//NaN happens only for inf - inf
assign addsub_nan = ~opcode_out[1] & op_addsub_out & opa_inf_out & opb_inf_out;

//NaN results for 0 x inf and inf x 0
assign mul_nan = opcode_out[1] & ~opcode_out[0] & ((opa_zero_out & opb_inf_out) | (opa_inf_out & opb_zero_out));

//NaN results for 0/0 and inf/inf
assign div_nan = &opcode_out & ((opa_zero_out & opb_zero_out) | (opa_inf_out & opb_inf_out));
```

```
//Assert Qnan for all these NaN conditions - assign Snan to Qnan
assign qnan = opa_nan_out | opb_nan_out | addsub_nan | mul_nan | div_nan;
assign snan = qnan;


//Check 2: INF
wire add_inf, sub_inf, mul_inf, div_inf, res_inf;

//For ADD: Result is INF if either input operands are inf
assign add_inf = ~opcode_out[1] & ~op_addsub_out & ( opa_inf_out | opb_inf_out );

//For SUB: Result is INF only if one of the inputs are INF - cannot be both (that's a NaN)
assign sub_inf = ~opcode_out[1] & op_addsub_out & (opa_inf_out ^ opb_inf_out);

//For MUL: Result is INF if one input is a INF while the other is NON-ZERO
assign mul_inf = opcode_out[1] & ~opcode_out[0] & ((opa_inf_out & ~opb_zero_out) | (~opa_zero_out & opb_inf_out));

//For DIV: Result is INF if its a NON-ZERO/ZERO or INF/NON-INF
assign div_inf = &opcode_out & ((~opa_zero_out & opb_zero_out) | (opa_inf_out & ~opb_inf_out));

//Assert Infinity for all the conditions above
assign inf = norm_inf | add_inf | sub_inf | mul_inf | div_inf;

//Check 3: Divide by zero - simple one
assign div_by_zero = &opcode_out & opb_zero_out;

//Check 4: Inexact output
assign ine = ine_out;

//Check 5: Zero
wire addsub_zero, mul_zero, div_zero;

//For ADD/SUB, ZERO is when the fractional output itself is already zero
//Had to hack the design and code a workaround to support the case when two equal numbers are SUBBed.
assign addsub_zero = ~opcode_out[1] & fract_addsub_zero_out;

//For MUL, result is ZERO if is NON-INFxZERO or ZEROxNON-INF
assign mul_zero = opcode_out[1] & ~opcode_out[0] & ((~opa_inf_out & opb_zero_out) | (opa_zero_out & ~opb_inf_out));

//For DIV, result is ZERO if its ZERO/NON-ZERO
assign div_zero = &opcode & opa_zero_out & ~opb_zero_out;

//Assert zero for all conditions above
assign zero = addsub_zero | mul_zero | div_zero;

//check 6: Overflow - works for addition/subtraction only
assign overflow = overflow_out;

//Check 7: Underflow - check not supported

//Calculate sign for inf result
wire sign_inf;

assign sign_inf = (norm_inf | mul_inf | div_inf) ? sign_out :
                                                      (add_inf) ? signa_out :
                                                      (sub_inf) ? ((opa_inf_out) ? signa_out : ~signa_out) :
                                                      sign_out;


always @(qnan or inf or sign_out or sign_inf or dn_out or zero or fract_out or exp_out)
        begin
                if (qnan)
                        begin
                                out[51:0] = {52{1'b1}};
                                out[62:52] = {0{1'b1}};
                                out[63] = sign_out;
                        end
                else if (inf)
                        begin
                                out[51:0] = {52{1'b0}};
                                out[62:52] = {10{1'b1}};
                                out[63] = sign_inf;
                        end
                else if (dn_out)
```

```
                    begin
                              out[51:0] = fract_out[51:0];
                              out[62:52] = {10{1'b0}};
                              out[63] = sign_out;
                    end
          else if (zero)
                    begin
                              out[51:0] = {52{1'b0}};
                              out[62:52] = {10{1'b0}};
                              out[63] = sign_out;
                    end
          else
                    begin
                              out[51:0] = fract_out[51:0];
                              out[62:52] = exp_out;
                              out[63] = sign_out;
                    end
end




endmodule
```

APPENDIX A2

RTL TESTBENCH AND WAVEFORM FOR FPU (TOP-LEVEL)

```
//This is the top level testbench used to validate the functionality of the FPU

`timescale 1ns / 100ps

/*
The FPU has 6 inputs and 9 outputs
Inputs:
            clock
            reset
            [1:0] opcode
            [1:0] r_mode
            [63:0] opa
            [63:0] opb
Outputs:
            [63:0] out
            snan
            qnan
            div_by_zero
            overflow
            underflow
            inf
            ine
            zero
*/

module FPU_TB;

parameter clk = 20;

integer seed;

//reg inputs
reg clock, reset;
reg [1:0] opcode;
reg [1:0] r_mode;
reg [63:0] opa, opb;

//wire outputs
wire [63:0] out;
wire snan, qnan;
wire div_by_zero;
wire overflow, underflow;
wire inf, ine, zero;

//Local variable
reg signa, signb;
reg [10:0] expa, expb;
reg [51:0] fracta, fractb;


//clock signal
always
            #10 clock <= !clock;

initial
            begin
                        clock <= 1'b0;
                        reset <= 1'b0;
                        opcode <= 2'b0;
                        r_mode <= 2'b0;

                        signa <= 0;
                        expa <= 11'h0;
                        fracta <= 52'b0;
                        signb <= 0;
                        expb <= 11'h0;
                        fractb <= 52'b0;
```

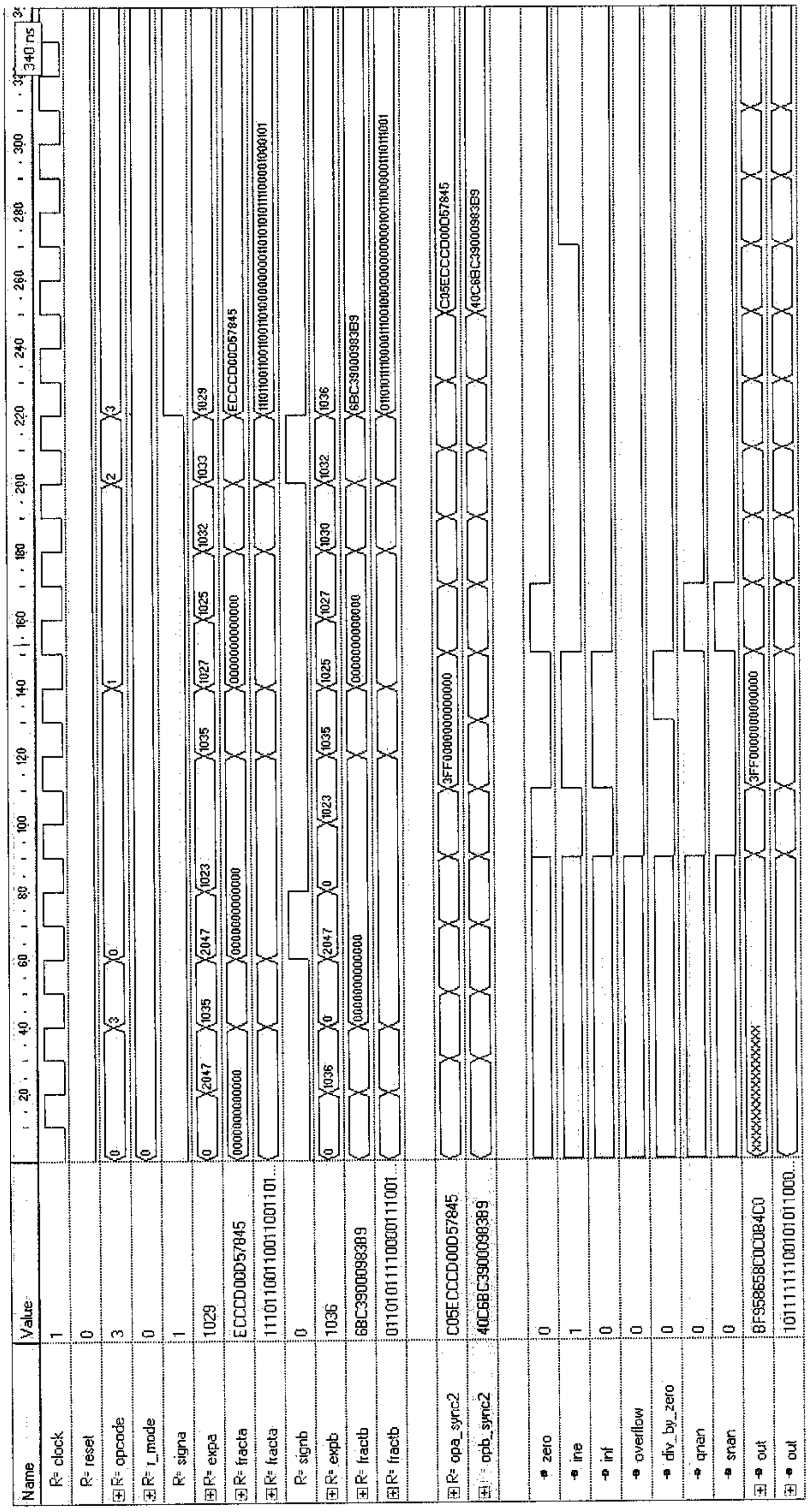
```
/*
Scenario: Inputs that would cause exception flags to be asserted
*/


//Case 1: Infinite - when one infinite number is added to a normal number
#clk; //110

opcode <= 2'b00;
//fracta is inf
expa <= 11'h7FF;
fracta <= 52'b0;

expb <= $dist_uniform(seed, 1023, 1046);
fractb <= {$dist_uniform(seed, 1, 15), $dist_uniform(seed, 1, 16777215), $dist_uniform(seed, 1,
16777215)};

//Case 2: Infinite - by divide to zero
#clk;       //130

opcode <= 2'b11;

expa <= $dist_uniform(seed, 1023, 1046);
fracta <= {$dist_uniform(seed, 1, 15), $dist_uniform(seed, 1, 16777215), $dist_uniform(seed, 1,
16777215)};

expb <= 11'b0;
fractb <= 52'b0;

//Case 3: from +inf + -inf
#clk;       //150

opcode <= 2'b00;

signa <= 0;
expa <= 11'h7FF;
fracta <= 52'b0;

signb <= 1;
expb <= 11'h7FF;
fractb <= 52'b0;


/*
Scenario: ADD
*/

//Case 1: Add 1 and zero
#clk;       //170
opcode <= 2'b00;

signa <= 0;
expa <= 11'd1023;
fracta <= 52'b0;

signb <= 0;
expb <= 11'h000;
fractb <= 52'b0;

//Case 2: Add 1 with 1
#clk;       //190
expb <= 11'd1023;

//Case 3: Add two random numbers
#clk;       //210
signa <= 0;
signb <= 0;

expa <= $dist_uniform(seed, 1023, 1046);
expb <= $dist_uniform(seed, 1023, 1046);

fracta <= {$dist_uniform(seed, 1, 15), $dist_uniform(seed, 1, 16777215), $dist_uniform(seed, 1,
16777215)};
```

```
		fractb <= {$dist_uniform(seed, 1, 15), $dist_uniform(seed, 1, 16777215), $dist_uniform(seed, 1,
16777215)};

		/*
		Scenario: SUB
		*/

		//Case 1: take 16 - 4
		#clk;		//230
		opcode <= 2'b01;

		signa <= 0;
		signb <= 0;

		expa <= 11'd1027;
		expb <= 11'd1025;
		fracta <= 52'b0;
		fractb <= 52'b0;

		//Case 2: take 4 - 16
		#clk;		//250

		expb <= 11'd1027;
		expa <= 11'd1025;
		fractb <= 52'b0;
		fracta <= 52'b0;

		//Case 3: take random number
		#clk;		//270

		expa <= $dist_uniform(seed, 1023, 1046);
		expb <= $dist_uniform(seed, 1023, 1046);

		fracta <= {$dist_uniform(seed, 1, 15), $dist_uniform(seed, 1, 16777215), $dist_uniform(seed, 1,
16777215)};
		fractb <= {$dist_uniform(seed, 1, 15), $dist_uniform(seed, 1, 16777215), $dist_uniform(seed, 1,
16777215)};

		/*
		Scenario: MUL
		*/

		//Case 1: MUL of two random numbers
		#clk;		//290
		opcode <= 2'b10;

		signa <= 0;
		signb <= 1;

		expa <= $dist_uniform(seed, 1023, 1046);
		expb <= $dist_uniform(seed, 1023, 1046);

		fracta <= {$dist_uniform(seed, 1, 15), $dist_uniform(seed, 1, 16777215), $dist_uniform(seed, 1,
16777215)};
		fractb <= {$dist_uniform(seed, 1, 15), $dist_uniform(seed, 1, 16777215), $dist_uniform(seed, 1,
16777215)};

		/*
		Scenario: DIV
		*/

		//Case 1: Div of two random numbers
		#clk;		//310
		opcode <= 2'b11;

		signa <= 1;
		signb <= 0;

		expa <= $dist_uniform(seed, 1023, 1046);
		expb <= $dist_uniform(seed, 1023, 1046);

		fracta <= {$dist_uniform(seed, 1, 15), $dist_uniform(seed, 1, 16777215), $dist_uniform(seed, 1,
16777215)};
		fractb <= {$dist_uniform(seed, 1, 15), $dist_uniform(seed, 1, 16777215), $dist_uniform(seed, 1,
16777215)};
```

```
            end

always @(signa or signb or expa or expb or fracta or fractb)
            begin
                        opa <= {signa, expa, fracta};
                        opb <= {signb, expb, fractb};
            end

initial
            #340 $finish(2);

//Unit Instantiation
fpu fpu_unit1 (
.clk(clock),
.reset(reset),
.opcode(opcode),
.r_mode(r_mode),
.opa(opa),
.opb(opb),
.out(out),
.snan(snan),
.qnan(qnan),
.div_by_zero(div_by_zero),
.overflow(overflow),
.inf(inf),
.ine(ine),
.zero(zero)

);


 Endmodule
```

APPENDIX A3

RTL SIMULATION WAVEFORM FOR FPU (TOP-LEVEL)



APPENDIX B

SOURCE CODE EXCEPTION UNIT

```
/*
This file contains only the Exception module which checks for
            - nan
            - inf
            - zero
            - denormalized
conditions of each input operands
*/

`timescale 1ns / 100ps

module exception( clk, opa, opb, opa_nan, opb_nan,opa_zero, opb_zero, opa_inf, opb_inf, opa_dn, opb_dn);

input clk;
input [62:0] opa, opb;

output opa_nan, opb_nan;
output opa_inf, opb_inf;
output opa_zero, opb_zero;
output opa_dn, opb_dn;

//Register all outputs
reg         opa_nan, opb_nan;
reg         opa_zero, opb_zero;
reg         opa_inf, opb_inf;
reg         opa_dn, opb_dn;

//Local wires and reg
wire [10:0] expa, expb;                     //stores exponent of opX
wire [51:0] fracta, fractb;         //stores significant of opX

wire expa_ones, expa_zeros, fracta_zeros;   //expa_ones represent all "1" in expa
wire expb_ones, expb_zeros, fractb_zeros;   //expb_ones represent all "1" in expb

//Assigning the exp and mantissa field to seperate wires
assign expa = opa[62:52];
assign fracta = opa[51:0];
assign expb = opb[62:52];
assign fractb = opb[51:0];

//------------------------------------------------------------------------//
//
// Major block - checking whether the input operands are inf, zero or NaN
//

//Do every check as a combinatorial assignment - then latch the output using a register

//Check for all 1's in exp of A & B - indicates either an infinite number or a NaN
assign expa_ones = &expa;
assign expb_ones = &expb;

//Check for all 0's in exp of A & B = indicate either Zero or denormalized number
assign expa_zeros = ~(|expa);
assign expb_zeros = ~(|expb);

//Check for all 0's in fraction of A & B = used to check for NaN inputs
assign fracta_zeros = ~(|fracta);
assign fractb_zeros = ~(|fractb);

//Check for NaN on both operand A & B
always @(posedge clk)
            begin
                        opa_nan <= expa_ones & (~fracta_zeros);
                        opb_nan <= expb_ones & (~fractb_zeros);
            end

//Check for INF on both operand A & B
```

```
always @(posedge clk)
        begin
                opa_inf <= expa_ones & (fracta_zeros);
                opb_inf <= expb_ones & (fractb_zeros);
        end

//Check for ZERO on operand A & B
always @(posedge clk)
        begin
                opa_zero <= expa_zeros & (fracta_zeros);
                opb_zero <= expb_zeros & (fractb_zeros);
        end

//Check for Denormalized on operand A & B
always @(posedge clk)
        begin
                opa_dn <= expa_zeros & (~fracta_zeros);
                opb_dn <= expb_zeros & (~fractb_zeros);
        end

endmodule
```

APPENDIX C

SOURCE CODE FOR PRE-ALIGN UNIT (ADD/SUB)

```
/*

Pre-align module:
        Function is to align the mantissa for addition and subtraction
        Also checks for the real operation to be performed, either add or sub

*/

`timescale 1ns / 100ps

module pre_align ( clk, op_addsub, opa, opb,
                                        fracta_out, fractb_out, exp_out, op_addsub_out );

//Input/Output Declaration
input clk;
input op_addsub;
input [63:0] opa, opb;

output [55:0] fracta_out, fractb_out;
output [10:0] exp_out;
output op_addsub_out;

//Register all outputs
reg [55:0] fracta_out, fractb_out;
reg [10:0] exp_out;
reg op_addsub_out;

//Local Wires and Reg
wire opa_lead, opb_lead;
reg signa, signb;
reg [10:0] expa, expb;
reg [51:0] fracta, fractb;

reg [10:0] exp_diff, exp_diff_1;
reg [55:0] fract_full, fract_full_shr;
reg [55:0] fracta_out_1, fractb_out_1;
reg [10:0] exp_out_1;
reg sticky;

reg op_addsub_out_1;

//Get implicit leading "1" bit - if exponent is non-zero, leading bit is 1
assign opa_lead = |expa;
assign opb_lead = |expb;

always @(opa or opb or opa_lead or opb_lead or expa or expb or fracta or fractb or exp_diff_1 or exp_diff
        or fract_full or fract_full_shr or sticky)
        begin
        //seperate the sign, exponent and mantissa field into seperate reg
        signa <= opa[63];
        signb <= opb[63];
        expa <= opa[62:52];
        expb <= opb[62:52];
        fracta <= opa[51:0];
        fractb <= opb[51:0];

        if (expa == expb)
                begin
                exp_out_1 <= expa;
                fracta_out_1 <= {opa_lead, fracta, 3'b0};
                fractb_out_1 <= {opb_lead, fractb, 3'b0};
                //The following assignments are done to keep the values below from floating
                exp_diff_1 <= 11'b0;
                exp_diff <= exp_diff_1;
                fract_full <= 56'b0;
                fract_full_shr <= fract_full;
                end //if expa == expb
```

```
		else if (expa > expb)	//if expa is larger than expb
				begin
				exp_out_1 <= expa;
				exp_diff_1 <= expa - expb;
				//The fraction can only be shifted by a max of d56 as the mantissa field
				//is only 56 bits wide with the hidden, guard and sticky bits
				exp_diff <= (exp_diff_1 > 11'd56) ? 11'd56 : exp_diff_1;
				//Recover the hidden bit and the 2 guard bit
				fract_full <= {opb_lead, fractb, 3'b0};
				//Right shift the mantissa
				fract_full_shr <= fract_full >> exp_diff;
				//Join the shifted bit with the sticky bit
				fractb_out_1 <= {fract_full_shr[55:1], fract_full_shr[0] | sticky};
				fracta_out_1 <= {opa_lead, fracta, 3'b0};	//fracta is not affected
				end //if expa > expb
		else								//if expb is larger than expa
				begin
				exp_out_1 <= expb;
				exp_diff_1 <= expb - expa;
				//The fraction can only be shifted by a max of d56 as the mantissa field
				//is only 56 bits wide with the hidden, guard and sticky bits
				exp_diff <= exp_diff_1 > 11'd56 ? 11'd56 : exp_diff_1;
				//Recover the hidden bit and the 2 guard bit
				fract_full <= {opa_lead, fracta, 3'b0};
				//Right shift the mantissa
				fract_full_shr <= fract_full >> exp_diff;
				//Join the shifted bit with the sticky bit
				fracta_out_1 <= {fract_full_shr[55:1],fract_full_shr[0] | sticky};
				fractb_out_1 <= {opb_lead, fractb, 3'b0};	//fractb is not affected
				end	//else
		end // always

always @(posedge clk)
		begin
				exp_out <= exp_out_1;
				fracta_out <= fracta_out_1;
				fractb_out <= fractb_out_1;
		end

always @(fract_full or exp_diff)
		begin
		//The case statement is used to obtain the value of the sticky bit
		//This method is brute force as it would synthesize a giant single level MUX - but potentially is fastest
		case(exp_diff)		//synopsys full_case parallel_case
				00: sticky = 1'b0;
				01: sticky = fract_full[0];
				02: sticky = |fract_full[01:0];
				03: sticky = |fract_full[02:0];
				04: sticky = |fract_full[03:0];
				05: sticky = |fract_full[04:0];
				06: sticky = |fract_full[05:0];
				07: sticky = |fract_full[06:0];
				08: sticky = |fract_full[07:0];
				09: sticky = |fract_full[08:0];
				10: sticky = |fract_full[09:0];
				11: sticky = |fract_full[10:0];
				12: sticky = |fract_full[11:0];
				13: sticky = |fract_full[12:0];
				14: sticky = |fract_full[13:0];
				15: sticky = |fract_full[14:0];
				16: sticky = |fract_full[15:0];
				17: sticky = |fract_full[16:0];
				18: sticky = |fract_full[17:0];
				19: sticky = |fract_full[18:0];
				20: sticky = |fract_full[19:0];
				21: sticky = |fract_full[20:0];
				22: sticky = |fract_full[21:0];
				23: sticky = |fract_full[22:0];
				24: sticky = |fract_full[23:0];
				25: sticky = |fract_full[24:0];
				26: sticky = |fract_full[25:0];
				27: sticky = |fract_full[26:0];
				28: sticky = |fract_full[27:0];
				29: sticky = |fract_full[28:0];
				30: sticky = |fract_full[29:0];
```

```
                        31: sticky = |fract_full[30:0];
                        32: sticky = |fract_full[31:0];
                        33: sticky = |fract_full[32:0];
                        34: sticky = |fract_full[33:0];
                        35: sticky = |fract_full[34:0];
                        36: sticky = |fract_full[35:0];
                        37: sticky = |fract_full[36:0];
                        38: sticky = |fract_full[37:0];
                        39: sticky = |fract_full[38:0];
                        40: sticky = |fract_full[39:0];
                        41: sticky = |fract_full[40:0];
                        42: sticky = |fract_full[41:0];
                        43: sticky = |fract_full[42:0];
                        44: sticky = |fract_full[43:0];
                        45: sticky = |fract_full[44:0];
                        46: sticky = |fract_full[45:0];
                        47: sticky = |fract_full[46:0];
                        48: sticky = |fract_full[47:0];
                        49: sticky = |fract_full[48:0];
                        50: sticky = |fract_full[49:0];
                        51: sticky = |fract_full[50:0];
                        52: sticky = |fract_full[51:0];
                        53: sticky = |fract_full[52:0];
                        54: sticky = |fract_full[53:0];
                        55: sticky = |fract_full[54:0];
                        56: sticky = |fract_full[55:0];
            endcase
            end

//Final block, check for the actual operation to be carried out
always @(op_addsub or signa or signb)
            begin
                        op_addsub_out_1 = op_addsub ^ (signa^signb);
            end

always @(posedge clk)
            op_addsub_out = op_addsub_out_1;

endmodule
```

APPENDIX D

SOURCE CODE FOR PRE-ALIGN UNIT (MUL/DIV)

```
/*

Pre-align module for MUL & DIV operation:
            Functions:
            1. Perform dividend alignment to overcome divide overflow
            2. Restore implicit bit in fraction A and B
            3. ADD/SUB the exponent on MUL & DIV operations
            4. Calculate the sign of MUL & DIV operations

*/

`timescale 1ns / 100ps

module pre_align_mul ( clk, op_muldiv, opa, opb, sign_out, fracta_out, fractb_out, exp_out );

//Port Declaration
input clk;
input op_muldiv;
input [63:0] opa, opb;

output sign_out;
output [105:0] fracta_out;
output [52:0] fractb_out;
output [10:0] exp_out;

//Register outputs
reg sign_out;
reg [105:0] fracta_out;
reg [52:0] fractb_out;
reg [10:0] exp_out;
reg overflow;
//reg underflow;

//Local Wires & Reg
wire [105:0] fracta_shifted;
wire [10:0] expa_shifted;

wire [105:0] fracta_out_temp;
wire [10:0] expa_full_temp;
wire [105:0] fracta_full;
wire [52:0]fractb_full;
reg [10:0] expa_full;
reg [10:0] expb_full;
reg op_muldiv_2;


wire [10:0]expb_full_tmp;
wire co1, co_d;
wire [10:0] exp_tmp1, exp_tmp2, exp_tmp3;


//Seperate sign, exp & fract bits
wire signa = opa[63];
wire signb = opb[63];
wire [10:0] expa = opa[62:52];
wire [10:0] expb = opb[62:52];
wire [51:0] fracta = opa[51:0];
wire [51:0] fractb = opb[51:0];


//Obtain implicit leading bit
wire opa_lead = |expa;
wire opb_lead = |expb;

//-----------------------------------------------------------------------------------------
//Function 1: Perform dividend alignment if fracta is larger than fractb on DIV operation
```

```
//Obtain shifted values of fracta & expa
assign fracta_shifted = {fracta_full} >> 1;
assign expa_shifted = expa + 1;

//-----------------------------------------------------------------------------------------
//Function 2: Restore implicit bit of both fractions


assign fracta_full = {opa_lead, fracta, {53{1'b0}}};
assign fractb_full = {opb_lead, fractb};

assign fracta_out_temp = (op_muldiv & (fracta >= fractb)) ? fracta_shifted : fracta_full;
assign expa_full_temp = (op_muldiv & (fracta >= fractb)) ? expa_shifted : expa;

//Choose the shifted values if the operation is DIV & fracta is larger than fractb
always @(posedge clk)
        begin
                fracta_out <= fracta_out_temp;
                expa_full <= expa_full_temp;

                //Register B outputs
                fractb_out <= fractb_full;
                expb_full <= expb;
        end


//---------------------------------------------------------------------------
//Function 3: Add or Subtract Exponent - Biased Representation!!

//assign {co1,exp_tmp1} = op_muldiv ? (expa_full - expb_full) : (expa_full + expb_full);
//assign {co2,exp_tmp2} = op_muldiv ? ({co1,exp_tmp1} + 12'd1023) : ({co1,exp_tmp1} - 12'd1023); //Add/Sub bias value of
1023

//Use algo from ADD/SUB unit to perform addition/subtraction of biased exponents
//Biased Exponents are basically magnitude only - range from 1 - 2046
//After add/sub operation, the bias value must be sub/add to restore the correct value to the biased exponent result


//Delay op_muldiv signal by one clock
always @(posedge clk)
        op_muldiv_2 <= op_muldiv;

        //invert expb_full if operation is SUB - XOR with 11 bits of op_muldiv
        assign expb_full_tmp  = expb_full ^ {11{op_muldiv_2}};

        //perform addition as usual with op_addsub as initial carry-in to the adder
        assign {co1, exp_tmp1} = expa_full + expb_full_tmp + op_muldiv_2;

        assign {exp_tmp2} = (op_muldiv_2 & !co1) ? (~exp_tmp1 + 1) : exp_tmp1 ;

        assign {co_d, exp_tmp3} = (op_muldiv_2) ?
                                                                        ((co1) ? ( 12'd1023 + exp_tmp2 ) : ( 12'd1023 -
exp_tmp2)) :
                                                                        ({co1, exp_tmp2} - 12'd1023) ;

always @(posedge clk)
        begin
                exp_out <= exp_tmp3;
                overflow <= ~op_muldiv & co_d;
                //underflow <= ???
        end

//---------------------------------------------------------------------------
//Function 4: Calculate the sign of the result
always @(posedge clk)
        sign_out <= signa ^ signb;


endmodule
```

APPENDIX E

SOURCE CODE FOR ARITHMETIC EXECUTION UNIT

```
// This file contains all modules of the execution core
// All pure ADD/SUB/MUL/DIV operation is done here

`timescale 1ns / 100ps

/*

-----ADD/SUBTRACT MODULE-----
56 bit long add/sub module - 53 for mantinssa
                                                    - 2 for guard bit
                                                    - 1 for sticky bit

*/

module add_sub (clk, op_addsub, fracta, fractb, signa, sum, sign_sum, co);

        input clk, op_addsub, signa;        //only the sign of opA is required
        input [55:0] fracta, fractb;
        output [55:0] sum;
        output co, sign_sum;

        //register outputs
        reg [55:0] sum;
        reg co, sign_sum;

        //local variables
        wire [55:0] fractb_temp;
        wire co_temp;
        wire [55:0] sum_temp;

        //op_addsub == 0 is ADD & op_addsub == 1 is SUB
        //SUB operation is carried out in 2's complement

        //invert fractb if operation is SUB - extend op_addsub to 56 bits to XOR every bit in fractb
        assign fractb_temp  = fractb ^ {56{op_addsub}};

        //perform addition as usual with op_addsub as initial carry-in to the adder
        assign {co_temp, sum_temp} = fracta + fractb_temp + op_addsub;

        //if the carry out from SUB operation is 0, then fracta < fractb
        //therefore, when co_temp == 0, get the 2's complement of the result and invert the sign
        always @(posedge clk)
                begin
                        if (op_addsub & !co_temp)
                                begin
                                        sign_sum <= !signa;
                                        co <= co_temp;
                                        sum <= ~sum_temp + 1;
                                end
                        else
                                begin
                                        sign_sum <= signa;
                                        co <= op_addsub ? !co_temp : co_temp;
                                        sum <= sum_temp;
                                end
                end

endmodule


/*

-----MUL MODULE-----
2 inputs - 53 bit long operands
1 output - 106 bit long product

NOTE: This multiplication unit will only multiply the fraction/mantissa part of the
```

```
		number. The addition of the exponents will be performed using another module that
		could run simultaneously with the add/sub pre-align.
		The calculation for the sign of the result is also left to the other module

*/

module mul (clk, fracta, fractb, product);

	input clk;
	input [52:0] fracta, fractb;		//Since guard bits are of no use, 53 bits are sufficient
	output [105:0] product;

	reg [105:0] product;

	always @(posedge clk)
		begin
			product <= fracta * fractb;
		end

endmodule


/*

-----DIV MODULE-----
Inputs - OpA - Dividend- 106bits
	   - OpB - Divisor - 53 bits
output - Quotient - 53 bit long
		  Remainder- 53 bit long

NOTE: The division unit takes a divident that is padded with zeros on the right hand side
	and then divides it with the divisor.
	The task of padding the zeros is left to another unit - either the master FPU or using
	another module responsible for subtracting the exponents.
	The task of divident alignment to avoid divide overflow is also done by the other module

*/

module div (clk, dividend, divisor, quo, rem);

	input clk;
	input [105:0] dividend;
	input [52:0] divisor;
	output [52:0] quo, rem;

	reg [52:0] quo, rem;

	//local wires
	wire [105:0] quo_temp;
	wire [105:0] rem_temp;

	//Both These are not synthesizable - Code the algorithm YOURSELF!!!
	assign quo_temp = dividend / divisor;
	assign rem_temp = dividend % divisor;

	always @(posedge clk)
		begin
			quo <= quo_temp[52:0];
			rem <= rem_temp[105:53];
		end

endmodule
```

APPENDIX F

SOURCE CODE FOR NORMALIZATION AND ROUNDING UNIT

```
//This source file would contain modules for 2 functions
//Function 1 is Normalizing
//Function 2 is Rounding

`timescale 1ns / 100ps

/*

Normalizing unit:
            inputs: 112 bit long mantissa
                                    11 bit long exponent
                                    1 bit sign
                                    2 bit opcode

*/

module norm_rnd (clk, fract_in, exp_in, sign, opcode, remainder, r_mode,
                                    fract_out, exp_out, dn_out, ine_out, overflow_out, infinite_out );

input clk;
input [105:0] fract_in;
input [10:0] exp_in;
input sign; //Rounding only
input [1:0] opcode;    //Rounding only
input [52:0] remainder;         //Rounding only
input [1:0] r_mode;    //Rounding only

output [52:0] fract_out;
output [10:0] exp_out;
output dn_out;
output ine_out;
output overflow_out;
output infinite_out;

//Reg all outputs
reg [52:0] fract_out;
reg [10:0] exp_out;
reg dn_out;
reg ine_out;
reg overflow_out;
reg infinite_out;

//Local Wires & Registers
reg [6:0] fract_ldz;    /*reg needed to hold the value constantly - register should not be clocked as it is fed by
                                                    combi circuit - up to 111 leading zeros may be present, thus
a 7 bit register is needed*/
wire shift_dir;                 //Direction of shift
wire exp_max;                   //Exponent is max
wire [6:0] shift_val;  //Magnitude of shift
wire ldz_less_exp;     //Exponent is larger than no of left shifts required
wire [6:0] fract_ldz_mi1;
wire [10:0] exp_in_mi1;
wire out_dn;                    //indicates that the output will be denormalized
wire out_overflow;     //indicates that output overflow occured - exponent & mantissa cannot store result
wire [105:0] fract_sh_R, fract_sh_L;
wire [10:0] exp_sh_R, exp_sh_L;
wire [105:0] fract_shifted;
wire [10:0] exp_shifted;

wire [52:0] fract_down, fract_up_temp, fract_up;        //wire to store rounded down & up fractions
wire cout;                      //wire to store carry out from fract_up
wire [10:0] exp_up; //wire to store exponent for fract_up - needed becoz fract_up may have carry-out
wire fract_trunc;      //indicates that some value have been truncated
wire rem_not_zero;     //indicates that the division has a non-zero remainder - used to signal inexact
```

```
reg  [52:0] fract_round;
reg  [10:0] exp_round;
wire [52:0] fract_near_temp, fract_round_nearest, fract_round_zero;             //stores result for every rounding mode
reg  [52:0] fract_round_up, fract_round_down;           //stores result for every rounding mode
wire [10:0]exp_near_temp, exp_round_nearest, exp_round_zero;
reg  [10:0] exp_round_up, exp_round_down;
wire last_bit, trunc, r_near_sel, fract_up_overflow;
wire [1:0] r_up_sel, r_down_sel;

//-----------------------------------------------------------------------------------------
//Normalization unit starts here
//-----------------------------------------------------------------------------------------

//Count Leading Zeros in input fraction/mantissa
always @ (fract_in)
        casex(fract_in)         // synopsys full_case parallel_case
        106'b1???????????????????????????????????????????????????????????????????????????????????????????????????
?????? : fract_ldz =  0;
        106'b01??????????????????????????????????????????????????????????????????????????????????????????????????
?????? : fract_ldz =  1;
        106'b001?????????????????????????????????????????????????????????????????????????????????????????????????
?????? : fract_ldz =  2;
        106'b0001???????????????????????????????????????????????????????????????????????????????????????????????
??????? : fract_ldz =  3;
        106'b00001??????????????????????????????????????????????????????????????????????????????????????????????
??????? : fract_ldz =  4;
        106'b000001?????????????????????????????????????????????????????????????????????????????????????????????
??????? : fract_ldz =  5;
        106'b0000001????????????????????????????????????????????????????????????????????????????????????????????
??????? : fract_ldz =  6;
        106'b00000001???????????????????????????????????????????????????????????????????????????????????????????
??????? : fract_ldz =  7;
        106'b000000001??????????????????????????????????????????????????????????????????????????????????????????
??????? : fract_ldz =  8;
        106'b0000000001?????????????????????????????????????????????????????????????????????????????????????????
??????? : fract_ldz =  9;

        106'b00000000001????????????????????????????????????????????????????????????????????????????????????????
??????? : fract_ldz =  10;
        106'b000000000001??????????????????????????????????????????????????????????????????????????????????????
???????? : fract_ldz =  11;
        106'b0000000000001?????????????????????????????????????????????????????????????????????????????????????
???????? : fract_ldz =  12;
        106'b00000000000001????????????????????????????????????????????????????????????????????????????????????
???????? : fract_ldz =  13;
        106'b000000000000001???????????????????????????????????????????????????????????????????????????????????
???????? : fract_ldz =  14;
        106'b0000000000000001??????????????????????????????????????????????????????????????????????????????????
???????? : fract_ldz =  15;
        106'b00000000000000001?????????????????????????????????????????????????????????????????????????????????
???????? : fract_ldz =  16;
        106'b000000000000000001????????????????????????????????????????????????????????????????????????????????
???????? : fract_ldz =  17;
        106'b0000000000000000001???????????????????????????????????????????????????????????????????????????????
???????? : fract_ldz =  18;
        106'b00000000000000000001?????????????????????????????????????????????????????????????????????????????
????????? : fract_ldz =  19;

        106'b000000000000000000001????????????????????????????????????????????????????????????????????????????
????????? : fract_ldz =  20;
        106'b0000000000000000000001???????????????????????????????????????????????????????????????????????????
????????? : fract_ldz =  21;
        106'b00000000000000000000001??????????????????????????????????????????????????????????????????????????
????????? : fract_ldz =  22;
        106'b000000000000000000000001?????????????????????????????????????????????????????????????????????????
????????? : fract_ldz =  23;
        106'b0000000000000000000000001????????????????????????????????????????????????????????????????????????
????????? : fract_ldz =  24;
        106'b00000000000000000000000001???????????????????????????????????????????????????????????????????????
????????? : fract_ldz =  25;
        106'b000000000000000000000000001??????????????????????????????????????????????????????????????????????
????????? : fract_ldz =  26;
        106'b0000000000000000000000000001????????????????????????????????????????????????????????????????????
?????????? : fract_ldz =  27;
```

```
        106'b00000000000000000000000000001???????????????????????????????????????????????????????????????????
?????????? : fract_ldz = 28;
        106'b000000000000000000000000000001??????????????????????????????????????????????????????????????????
?????????? : fract_ldz = 29;

        106'b0000000000000000000000000000001?????????????????????????????????????????????????????????????????
?????????? : fract_ldz = 30;
        106'b00000000000000000000000000000001????????????????????????????????????????????????????????????????
?????????? : fract_ldz = 31;
        106'b000000000000000000000000000000001???????????????????????????????????????????????????????????????
?????????? : fract_ldz = 32;
        106'b0000000000000000000000000000000001??????????????????????????????????????????????????????????????
?????????? : fract_ldz = 33;
        106'b00000000000000000000000000000000001?????????????????????????????????????????????????????????????
?????????? : fract_ldz = 34;
        106'b000000000000000000000000000000000001???????????????????????????????????????????????????????????
??????????? : fract_ldz = 35;
        106'b0000000000000000000000000000000000001??????????????????????????????????????????????????????????
??????????? : fract_ldz = 36;
        106'b00000000000000000000000000000000000001?????????????????????????????????????????????????????????
??????????? : fract_ldz = 37;
        106'b000000000000000000000000000000000000001????????????????????????????????????????????????????????
??????????? : fract_ldz = 38;
        106'b0000000000000000000000000000000000000001???????????????????????????????????????????????????????
??????????? : fract_ldz = 39;

        106'b00000000000000000000000000000000000000001??????????????????????????????????????????????????????
??????????? : fract_ldz = 40;
        106'b000000000000000000000000000000000000000001?????????????????????????????????????????????????????
??????????? : fract_ldz = 41;
        106'b0000000000000000000000000000000000000000001???????????????????????????????????????????????????
???????????? : fract_ldz = 42;
        106'b00000000000000000000000000000000000000000001??????????????????????????????????????????????????
???????????? : fract_ldz = 43;
        106'b000000000000000000000000000000000000000000001?????????????????????????????????????????????????
???????????? : fract_ldz = 44;
        106'b0000000000000000000000000000000000000000000001????????????????????????????????????????????????
???????????? : fract_ldz = 45;
        106'b00000000000000000000000000000000000000000000001???????????????????????????????????????????????
???????????? : fract_ldz = 46;
        106'b000000000000000000000000000000000000000000000001??????????????????????????????????????????????
???????????? : fract_ldz = 47;
        106'b0000000000000000000000000000000000000000000000001?????????????????????????????????????????????
???????????? : fract_ldz = 48;
        106'b00000000000000000000000000000000000000000000000001????????????????????????????????????????????
???????????? : fract_ldz = 49;

        106'b000000000000000000000000000000000000000000000000001??????????????????????????????????????????
????????????? : fract_ldz = 50;
        106'b0000000000000000000000000000000000000000000000000001?????????????????????????????????????????
????????????? : fract_ldz = 51;
        106'b00000000000000000000000000000000000000000000000000001????????????????????????????????????????
????????????? : fract_ldz = 52;
        106'b000000000000000000000000000000000000000000000000000001???????????????????????????????????????
????????????? : fract_ldz = 53;
        106'b0000000000000000000000000000000000000000000000000000001??????????????????????????????????????
????????????? : fract_ldz = 54;
        106'b00000000000000000000000000000000000000000000000000000001?????????????????????????????????????
????????????? : fract_ldz = 55;
        106'b000000000000000000000000000000000000000000000000000000001????????????????????????????????????
????????????? : fract_ldz = 56;
        106'b0000000000000000000000000000000000000000000000000000000001???????????????????????????????????
????????????? : fract_ldz = 57;
        106'b00000000000000000000000000000000000000000000000000000000001?????????????????????????????????
?????????????? : fract_ldz = 58;
        106'b000000000000000000000000000000000000000000000000000000000001????????????????????????????????
?????????????? : fract_ldz = 59;

        106'b0000000000000000000000000000000000000000000000000000000000001???????????????????????????????
?????????????? : fract_ldz = 60;
        106'b00000000000000000000000000000000000000000000000000000000000001??????????????????????????????
?????????????? : fract_ldz = 61;
        106'b000000000000000000000000000000000000000000000000000000000000001?????????????????????????????
?????????????? : fract_ldz = 62;
```

```
        106'b0000000000000000000000000000000000000000000000000000000000000001?????????????????????????????
?????????????  : fract_ldz =  63;
        106'b00000000000000000000000000000000000000000000000000000000000000001????????????????????????????
?????????????  : fract_ldz =  64;
        106'b000000000000000000000000000000000000000000000000000000000000000001???????????????????????????
?????????????  : fract_ldz =  65;
        106'b0000000000000000000000000000000000000000000000000000000000000000001?????????????????????????
??????????????  : fract_ldz =  66;
        106'b00000000000000000000000000000000000000000000000000000000000000000001????????????????????????
??????????????  : fract_ldz =  67;
        106'b000000000000000000000000000000000000000000000000000000000000000000001???????????????????????
??????????????  : fract_ldz =  68;
        106'b0000000000000000000000000000000000000000000000000000000000000000000001??????????????????????
??????????????  : fract_ldz =  69;

        106'b00000000000000000000000000000000000000000000000000000000000000000000001?????????????????????
??????????????  : fract_ldz =  70;
        106'b000000000000000000000000000000000000000000000000000000000000000000000001????????????????????
??????????????  : fract_ldz =  71;
        106'b0000000000000000000000000000000000000000000000000000000000000000000000001???????????????????
??????????????  : fract_ldz =  72;
        106'b00000000000000000000000000000000000000000000000000000000000000000000000001??????????????????
??????????????  : fract_ldz =  73;
        106'b000000000000000000000000000000000000000000000000000000000000000000000000001????????????????
???????????????  : fract_ldz =  74;
        106'b0000000000000000000000000000000000000000000000000000000000000000000000000001???????????????
???????????????  : fract_ldz =  75;
        106'b00000000000000000000000000000000000000000000000000000000000000000000000000001??????????????
???????????????  : fract_ldz =  76;
        106'b000000000000000000000000000000000000000000000000000000000000000000000000000001?????????????
???????????????  : fract_ldz =  77;
        106'b0000000000000000000000000000000000000000000000000000000000000000000000000000001????????????
???????????????  : fract_ldz =  78;
        106'b00000000000000000000000000000000000000000000000000000000000000000000000000000001???????????
???????????????  : fract_ldz =  79;

        106'b000000000000000000000000000000000000000000000000000000000000000000000000000000001??????????
???????????????  : fract_ldz =  80;
        106'b0000000000000000000000000000000000000000000000000000000000000000000000000000000001?????????
???????????????  : fract_ldz =  81;
        106'b0000000000000000000000000000000000000000000000000000000000000000000000000000000001??????
?????????????????  : fract_ldz =  82;
        106'b00000000000000000000000000000000000000000000000000000000000000000000000000000000001?????
?????????????????  : fract_ldz =  83;
        106'b000000000000000000000000000000000000000000000000000000000000000000000000000000000001????
?????????????????  : fract_ldz =  84;
        106'b0000000000000000000000000000000000000000000000000000000000000000000000000000000000001???
?????????????????  : fract_ldz =  85;
        106'b00000000000000000000000000000000000000000000000000000000000000000000000000000000000001??
?????????????????  : fract_ldz =  86;
        106'b000000000000000000000000000000000000000000000000000000000000000000000000000000000000001?
?????????????????  : fract_ldz =  87;
        106'b0000000000000000000000000000000000000000000000000000000000000000000000000000000000000001
?????????????????  : fract_ldz =  88;
        106'b00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
1????????????????  : fract_ldz =  89;

        106'b00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
01???????????????  : fract_ldz =  90;
        106'b00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
001??????????????  : fract_ldz =  91;
        106'b00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0001?????????????  : fract_ldz =  92;
        106'b00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00001????????????  : fract_ldz =  93;
        106'b00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000001???????????  : fract_ldz =  94;
        106'b00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000001??????????  : fract_ldz =  95;
        106'b00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000001?????????  : fract_ldz =  96;
        106'b00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000001????????  : fract_ldz =  97;
        106'b00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000001???????  : fract_ldz =  98;
```

```
		106'b00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000001??????: fract_ldz = 99;

		106'b00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000001?????: fract_ldz = 100;
		106'b00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000001????: fract_ldz = 101;
		106'b00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000001???: fract_ldz = 102;
		106'b00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000001??: fract_ldz = 103;
		106'b00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000001?: fract_ldz = 104;
		106'b00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000?: fract_ldz = 105;
		endcase


//Calculate Shift Direction & shift value]
//Shift Right only when the number of leading zero is 0, otherwise shift Left
//assign shift_dir = |fract_ldz ? 1'b0 : 1'b1;
assign shift_dir = ~(|fract_ldz);	//shift_dir = 0 means left, shift_dir = 1 means right

//Check for exp = 1111111111? - It means that exp is already at max value or INF - no further increment is alowed
//assign exp_max = &exp_in[10:1] ? 1'b1 : 1'b0;
assign exp_max = &exp_in[10:1];

//Check that the number of leading zero is less than or equal to the exponent
assign ldz_less_exp = (fract_ldz <= exp_in[6:0]) ? 1'b1 : |exp_in[10:7];
		//Comparison done for the first 7 bits of the exponent
		//This is to reduce the bit length of the synthesized part
		//If either of the front 4 bit of of exp_in is 1, then exp_in would be larger than fract_ldz
		//Alternative line of code - seems more complex, plus it synthesizes an additional OR gate
		//assign ldz_less_exp = |exp_in[10:7] || fract_ldz <= exp_in[6:0] ? 1'b1 : 1'b0;

//Get both Left shift values first - choose using a mux later
assign fract_ldz_mi1 = fract_ldz - 1;
assign exp_in_mi1 = exp_in - 1;

//If no of ldz is less than exponent, shift by number of ldz - 1 else shift by exp - 1 (results in minimum exp & denormalized no)
assign shift_val = ldz_less_exp ? fract_ldz_mi1 : exp_in_mi1;

//Do actual shifting - obtain both shifted values: Right & Left
assign fract_sh_R = fract_in >> 1'b1;
assign exp_sh_R = exp_in + 1;

assign fract_sh_L = fract_in << shift_val;
assign exp_sh_L = exp_in - shift_val;

//Select between different shift values - no need to care about overflowed numbers - will be handled by rounding side
assign fract_shifted = (fract_ldz == 1) ? fract_in : (shift_dir) ? fract_sh_R : fract_sh_L;

assign exp_shifted = (fract_ldz == 1) ? (exp_in) : ((shift_dir) ? exp_sh_R : exp_sh_L);


//If the no of leading zero is more than the exponent, a denormalized number would result - thus it is the inverse of
ldz_less_exp
assign out_dn = !ldz_less_exp;			//signal will begin used to set output exponent to 0000...0

//If the shift direction is right (only when there is no ldz) and the exponent is already maximum - flag overflow
//Late bug fix - overflow can also occur if	the product of
assign out_overflow = (exp_max & shift_dir);			//used to set the output of rounding unit

//------------------------------------------------------------------------------------
//Rounding unit starts here
//------------------------------------------------------------------------------------

/*Rounding mode selectors
-------------------------
0 = round to nearest
1 = round to zero
2 = round up (+inf)
3 = round down (-inf)

Value of Nmax:
```

```
            fract_max = 53'h1FFFFFFFFFFFFF
            exp_max = 11'h7FE

Value of Ninf:
            fract_inf = 53'b0;
            exp_inf = 11'h7FF;

*/


//Get both rounded values
assign fract_down = fract_shifted[104:52];
assign {cout, fract_up_temp} = fract_down + {53'b0, 1'b1};          /*cout is needed in case the addition results in

                                                        a train of carries until the most significant bit*/

assign fract_up = (cout) ? {cout, fract_up_temp} >> 1 : fract_up_temp;      //shift result by 1 to the right if cout == 1

assign exp_up = exp_shifted + {10'b0,cout};

assign fract_up_overflow = cout & (&exp_shifted[10:1]);              //indicates that overflow occured while trying to
form X+

//Round to nearest calculation

//assign fract_near_temp = (tie) ?
//                      ((fract_down[0]) ? fract_up : fract_down) :
//                      ((last_bit) ? fract_up : fract_down) ;

/*Use the signal r_near_sel to select fract_up or down

last_bit is MSB of the truncated part
trunc is the OR of the all the other bits in the truncated part
fract_down[0] is LSB of X-
r_near_sel == 0 is selecting fract_down

K-map

last_bit    trunc       fract_down[0]       r_near_sel
0                           0                   0                       0
0                           0                   1                       0
0                           1                   0                       0
0                           1                   1                       0
1                           0                   0                       0
1                           0                   1                       1
1                           1                   0                       1
1                           1                   1                       1

*/

assign last_bit = fract_shifted[51];

assign trunc = |fract_shifted[50:0];

assign r_near_sel = last_bit & ( trunc | fract_down[0]);  //logic derived from K-map above

assign fract_near_temp = (r_near_sel) ? fract_up : fract_down;

assign exp_near_temp = (r_near_sel) ? exp_up : exp_shifted;

assign fract_round_nearest = (out_overflow | (r_near_sel & fract_up_overflow) ) ? 53'b0 : fract_near_temp;

assign exp_round_nearest = (out_overflow | (r_near_sel & fract_up_overflow) ) ? 11'h7FF : exp_near_temp;

//Round to zero calculation
assign fract_round_zero = (out_overflow) ? 53'h1FFFFFFFFFFFFF : fract_down;

assign exp_round_zero = (out_overflow) ? 11'h7FE : exp_shifted;

//Round up calculation
assign r_up_sel[0] = sign;
assign r_up_sel[1] = out_overflow | (~sign & fract_up_overflow);

always @(r_up_sel or fract_up or fract_down)
            case (r_up_sel)         // synopsys full_case parallel_case
```

```
                    2'b00 :     fract_round_up = fract_up;
                    2'b01 :     fract_round_up = fract_down;
                    2'b10 :     fract_round_up = 53'b0;
                    2'b11 :     fract_round_up = 53'h1FFFFFFFFFFFFF;
          endcase


always @(r_up_sel or exp_up or exp_shifted)
          case (r_up_sel)         // synopsys full_case parallel_case
                    2'b00 :     exp_round_up = exp_up;
                    2'b01 :     exp_round_up = exp_shifted;
                    2'b10 :     exp_round_up = 11'h7FF;
                    2'b11 :     exp_round_up = 11'h7FE;
          endcase

//Round down calcuation
assign r_down_sel[0] = ~sign;
assign r_down_sel[1] = out_overflow | (sign & fract_up_overflow);

always @(r_down_sel or fract_up or fract_down)
          case (r_down_sel)     // synopsys full_case parallel_case
                    2'b00 :     fract_round_down = fract_up;
                    2'b01 :     fract_round_down = fract_down;
                    2'b10 :     fract_round_down = 53'b0;
                    2'b11 :     fract_round_down = 53'h1FFFFFFFFFFFFF;
          endcase


always @(r_down_sel or exp_up or exp_shifted)
          case (r_down_sel)     // synopsys full_case parallel_case
                    2'b00 :     exp_round_down = exp_up;
                    2'b01 :     exp_round_down = exp_shifted;
                    2'b10 :     exp_round_down = 11'h7FF;
                    2'b11 :     exp_round_down = 11'h7FE;
          endcase

//Final case to select result fraction
always @ (r_mode or fract_round_nearest or fract_round_zero or fract_round_up or fract_round_down) //still got some more 2
list
          case (r_mode)          // synopsys full_case parallel_case
                    2'b00 : fract_round = fract_round_nearest;
                    2'b01 : fract_round = fract_round_zero;
                    2'b10 : fract_round = fract_round_up;
                    2'b11 : fract_round = fract_round_down;
          endcase

//Final case to select result exponent
always @ (r_mode or exp_round_nearest or exp_round_zero or exp_round_up or exp_round_down) //still got some more 2 list
          case (r_mode)          // synopsys full_case parallel_case
                    2'b00 : exp_round = exp_round_nearest;
                    2'b01 : exp_round = exp_round_zero;
                    2'b10 : exp_round = exp_round_up;
                    2'b11 : exp_round = exp_round_down;
          endcase

//This is to check for inexact conditions
//Signal if any values are truncated
assign fract_trunc = last_bit | trunc;
assign rem_not_zero = (&opcode) & (|remainder);

//Register all final outputs
always @(posedge clk)
          begin
                    fract_out <= fract_round;
                    exp_out <= exp_round;
                    dn_out <= out_dn;
                    ine_out <= fract_trunc | rem_not_zero;
                    overflow_out <= out_overflow;
                    infinite_out <= out_overflow & ( (|r_mode) | ( r_mode[1] & ~(r_mode[0]^sign)) );
          end


endmodule
```